T/AW087/21
Support and Coordination

2057699-TN-02-04

Developing an Exascale-Ready Fusion Simulation
Revision 4.0

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

University of York

Gihan Mudalige, Ben McMillan, and Tom Goffrey

University of Warwick

August 26, 2022

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

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.

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.

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 [1]. Many of the systems likely to be available in the next decade will employ hierarchical parallelism, delivered by a diverse set of architectures [2, 3]. 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 [4] 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 [5]. 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 alternative 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

In this report we evaluate the performance portability of these applications using the metric introduced by Pennycook et al. [6], and use the visualisation techniques outlined by Sewall et al. [7]. The Pennycook metric allows us to calculate the *performance portability* of an application according to Equation (1).

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

In the equation, the performance portability (P) of an application a, solving problem p, on a given set of platforms H, is calculated by finding the harmonic mean of an application's performance efficiency ($e_i(a,p)$). The performance efficiency for each platform can be calculated by comparing achieved performance against the best recorded (possibly non-portable) performance on each individual target platform (i.e. *the application efficiency*, or by comparing the achieved performance against the theoretical maximum performance achievable on each individual platform (i.e. *the architectural efficiency*). Should the application fail to run on one of the target platforms, a performance portability score of 0 is awarded.

While Equation (1) provides a formal definition for performance portability, this single value metric may not answer all questions a developer might have about their application. In recognising this, in this report we use two visualisation techniques introduced by Sewall et al. [7]. These visualisations are best explained with an example.

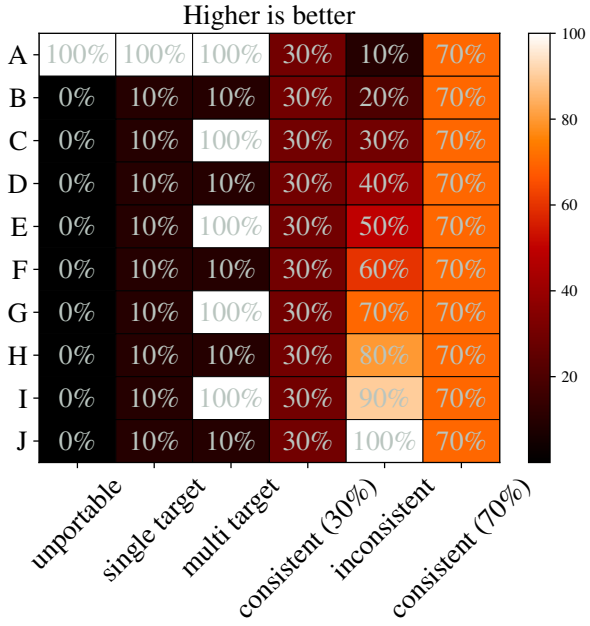
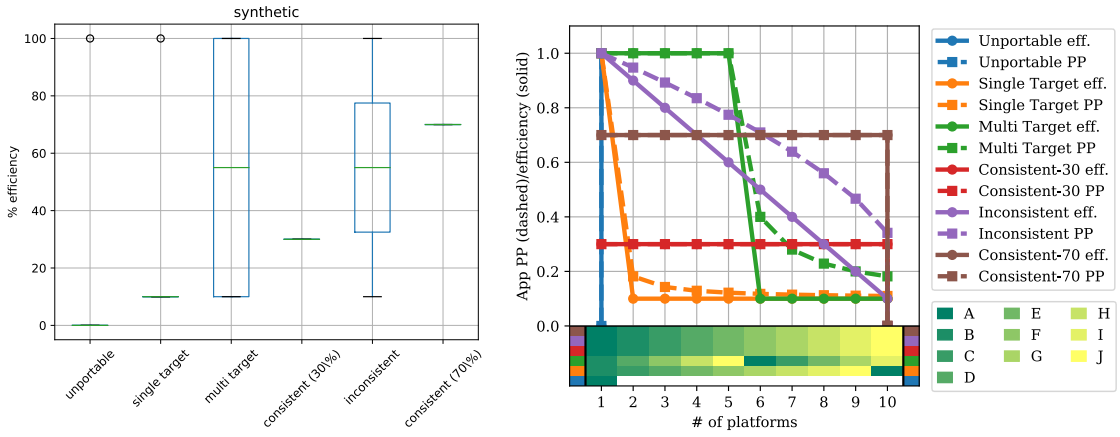


Figure 1: Synthetic data set for six implementations running across 10 platforms taken from Sewall et al. [7]

Figure 1 presents a simple synthetic data set for six implementations of an application running across 10 platforms. These implementations are: **unportable** with high performance on a single platform, but not portable to any other platform; **single target** with high performance on a single platform, but low performance on all others; **multi target** achieving high performance on some platforms, and low performance on others; **inconsistent** showing a range of performance across all platforms; and **consistent** showing consistent low (30%) or high (70%) performance across all platforms.

We could simply apply the performance portability metric in Equation (1) to this synthetic data but this may mean that we lose some information about how the performance portability is spread across platforms, and how the metric changes as we add and remove platforms from the evaluation set.

Figure 2(a) addresses this first concern, showing not only the median efficiency of an application, but also the spread of efficiencies (and any outliers). The second concern is addressed in the cascade plot in Figure 2(b), where the applications performance portability and efficiency are plotted as platforms are added to the evaluation set in descending order of efficiency. A more in-depth analysis of these visualisation techniques can be found in Sewall et al. [7].



(a) Box plot

(b) Cascade plot

Figure 2: Example plots for the synthetic data provided in Figure 1

We will use these plots to analyse the performance portability of various approaches to developing future-proofed software throughout this report. In some cases, where only a single implementation is available, we will use *architectural efficiency* rather than *application efficiency*. In these constrained cases, we will augment our analysis with Roofline models [8].

3 Approaches to Exascale Application Development

Considering the systems that are likely to be available in the next 5-10 years, it is clear that heterogeneity is likely to be a key feature, particularly with the efforts to build Exascale systems. With the exception of Fugaku, all announced pre- and post-Exascale systems make use of a CPU architecture coupled with GPU accelerators. As such, achieving high performance on such systems requires exploitation of hierarchical parallelism.

On heterogeneous platforms, a significant proportion of the available performance comes from the accelerators, with the host CPU primarily providing problem setup, synchronisation, and I/O operations. Each of the major GPU manufacturers provide a different programming model to interact with their accelerators and so application developers must consider their approach when targeting a heterogeneous system. Further consideration must also be given to vendor-supported approaches that may lead to vendor lock-in.

In this section, we outline the programming languages, models and libraries that provide abstractions for developers at various levels to develop applications targeting these systems. Our survey follows much of the findings from [4] together with specific considerations for algorithms of interest for the fusion domain. Some code examples are provided in Appendix A.

3.1 General Purpose Programming Languages

In this class we consider traditional programming languages with long history of usage and support in scientific computing. These languages typically allow fine control over every aspect of an algorithms implementation.

Scientific computing is dominated by the **Fortran**, **C** and **C++** programming languages. On ARCHER, the UK's recently retired Tier-1 resource, Fortran applications accounted for 69.3% of the machine's core hours, while C and C++ applications made up 6.3% and 7.4%, respectively [9]. This skew towards Fortran is in part due to a number of mature applications with large user bases, such as CASTEP and VASP, and its longevity in HPC, meaning that it benefits from mature compiler support more than most other languages.

Although usage of Fortran-based applications currently dwarfs C/C++ applications in HPC, there are signs that this is changing, likely as a result of the levels of support for C/C++ in new programming models and libraries. Of particular note are those that make extensive use of templates. These programming models encourage portability across different hardware – a key motivation as HPC becomes more heterogeneous.

Another language growing in popularity in HPC is **Python**. While not traditionally a "high performance" language, it provides interfaces to many external libraries, often written using languages such as C and Fortran. This has meant that Python can provide an easy interface for developers to write their applications at a high-level, leaving the implementation and execution to optimised libraries (see Section 3.5). Due to Python's use in a wide range of fields, by large corporations such as Alphabet, the community has invested significant effort into improving the performance of pure Python. The flexibility of the language and dynamic type system limits opportunities for static analysis and optimisation; instead Just-In-Time (JIT) compilers

have been developed, both as libraries to target particular code hotspots (Numba), and whole programs (PyPy). However, threading within Python, and thus its parallel performance, remains poor, limited by the Global Interpreter Lock (GIL) present in the reference CPython implementation, PyPy and Stackless Python. Removing this lock has proven difficult, limiting Python's use in HPC to primarily a "glue" language, coordinating work done in components implemented in higher-performance languages.

There is a long history of research and development of languages for scientific and high performance computing, including those such as Chapel, Fortress and X10 (DARPA 2002) which target parallel computation. These have tended to remain niche languages and have not been widely adopted. A promising language which is general purpose but designed in particular for scientific computing, is the **Julia** language[1]. This has a syntax which is familiar to Matlab or Fortran programmers, but is built on a sophisticated type system and language design, and uses LLVM to perform JIT compilation for CPU and GPU hardware. It is a relatively new language (version 1.0 was released in August 2018), but is seeing rapid adoption in scientific and machine-learning communities, and already has some libraries which are recognised as best in class (e.g. DifferentialEquations.jl, [10]). It aims to combine the flexibility and high productivity of Python, with high performance.

Developing applications in these general purpose programming languages present a number of challenges:

1. The languages are very prescriptive, and optimising an application for one system may harm performance on another system. In fact optimising for one architecture can obfuscate the science source so much so that future maintenance and addition of new features becomes difficult.
2. Applications developed with multiple code paths may provide portable performance, but requires duplicated effort keeping each code path up to date.
3. Parallelism must be explicitly written into the application, almost always using parallel programming extensions to the languages (as discussed in the next section), significantly increasing the complexity of development.

3.2 Parallel Programming Models

In this class we consider the programming models that extend from traditional general purpose programming languages to provide parallelisation both on-node (e.g. vectorisation, threading) and off-node (e.g. message passing). We also consider programming models that are designed specifically for heterogeneous computation with accelerator devices.

The parallelism available on modern supercomputers is hierarchical in nature. Vector operations (in the form of **SSE** and **AVX**) provide parallelism within a core, while threading (or Symmetric Multithreading, SMT) provides parallelism within a node. Parallelism across a system is usually provided in the form of message passing or shared global memory techniques.

[1] `https://julialang.org/`

Vectorised code can be achieved during the compilation phase, if there are no data dependencies present in the code. All modern compilers attempt to generate vectorised code through auto-vectorisation, usually when higher optimisation levels are specified (e.g. with compiler flags such as `-O2` and above). However, the compiler will only produce vectorised code when it is absolutely certain that no dependencies exist. In almost all non-trivial (especially real-world) codes a conclusive determination cannot be made and auto-vectorisation fails.

A developer can aid the compiler with the use of **compiler directives** or **vector intrinsics**. SIMD compiler directives, such as `#pragma omp simd`, were added to the OpenMP 4.0 standard, and should be supported in any compliant compiler. The pragmas allow a developer to indicate that an assumed dependency can be ignored, potentially resulting in the compiler generating vectorisable code that is portable across architectures. However, the compiler may still believe there is a dependency present; in this case, the developer must use a lower-level API (e.g. Intel Intrinsics) to directly manipulate the vector registers. This is likely to result in higher performance at the expense of both portability and productivity [11].

Distributing execution across all cores in a node can be achieved through threading and shared memory, or through message passing. Some message passing approaches, also allow parallelisation to read beyond a node.

In HPC applications threading is often achieved through **OpenMP** [12], while message passing is usually implemented using the **Message Passing Interface (MPI)** [13]. In rare cases, both threading and message passing can be achieved through the POSIX Threads (pthreads) library.

In OpenMP, parallelism is achieved by annotating loop structures with compiler directives (i.e. `#pragma`), such that the compiler can thread each iteration for execution in parallel. In MPI, parallelisation must be implemented explicitly with send/receive/scatter/gather operations.

Parallelisation beyond a single node requires inter-node communications. The de facto standard in HPC is MPI. MPI provides a number of functions for distributed computation, including point-to-point communications, one-sided communications, collective operations and reduction operations. In an MPI-parallelised program, each process operates on its own data, and communicates edge values to surrounding processes where a dependency exists.

There are also a number of programming models that treat the distributed memory space as a single homogeneous block. This partitioned global address space (PGAS) approach is taken by **Coarray Fortran** and **Unified Parallel C**, among others. In this model, communications are hidden to the application developer, but are typically implemented using MPI in the backend library.

3.2.1 Accelerator Extensions

For heterogeneous systems, host code is typically written using the programming languages mentioned previously to coordinate between compute nodes, however, the accelerators themselves usually require a different approach. This is a consequence of the significant differences in the accelerator architectures compared to

traditional CPUs.

Each vendor offers their own platform-specific programming model, such as **CUDA** from NVIDIA and **ROCm** from AMD. However, these approaches are typically not portable between vendors and algorithms often require significant re-engineering. Although proprietary, CUDA has been the most dominant accelerator programming extension and has maintained a high level of adoption in HPC given the widespread use of NVIDIA GPU hardware and the maturity and support that NVIDIA put into the numerical solver libraries based on CUDA. It follows a Single Instruction Multiple Data (SIMD) programming model where large number of threads are executed in lock-step on different data. **OpenCL** largely mirrors the SIMD model of CUDA, having a one-to-one equivalent API, but is developed as an open standard. With CUDA and OpenCL the programmer is given the opportunity to write explicit computational kernels for devices, with significant control over the orchestration of parallelism. OpenCL is supported by all major vendors (Intel, AMD, NVIDIA) and has been promoted as a vendor agnostic model. However the same OpenCL application will not necessarily give the best performance on all architectures, where some level of device specific optimisations are required to obtain best performance.

While offering much less control, **OpenACC** directives can be used to indicate/instruct a compiler what code can be executed on an accelerator. OpenACC also provides directives to indicate whether memory should be allocated on the host or the device, and when to move data between the two. Memory management, such as when data is moved to/from the device, and how often, are key considerations to achieving good performance. If not handled correctly, directives can lead to frequent data movement to/from a device and lead to significant slowdowns. Currently OpenACC is provided in commercial compilers from PGI and Cray, with the latter only supporting Cray-supplied hardware. GCC also offers nearly complete support for OpenACC 2.5, targeting both NVIDIA and AMD devices.

OpenMP added support similar to OpenACC for offloading computation to accelerators in version 4.0 of the standard. Similar to its counterpart, data locality is controlled through compiler directives, with parallelisable loops being specified using the `#pragma omp target` directive. OpenMP 4.0 is a good example of standards attempting to catch up with evolving hardware, where support for accelerator directives (which were introduced as a proprietary solution first in 2011 with OpenACC with the adoption of NVIDIA GPUs in HPC) were only added to the OpenMP standard in 2013. Even then OpenMP supporting compilers took several years more to fully implement the standard for working code.

Support for the OpenMP 4.0 and above can be found in commercial compilers from Intel, IBM, AMD and Cray, with a variety of target architectures. Support also exists in the Clang/LLVM [14] and GCC open-source compilers, with support for accelerators from NVIDIA, AMD and Intel[2].

While the explicit device control provided by the CUDA and OpenCL programming model may be more powerful than directive-based approaches, it may also significantly increase developer effort. More recently, the Khronos Group released **SYCL**, a new high-level cross-platform abstraction layer, which can be viewed as a data-parallel version of C++ based on OpenCL. Much of the concepts remain the same, but the significant amount of "boiler-plate" code required to setup parallelism in OpenCL applications is now not required

[2] `https://www.openmp.org/resources/openmp-compilers-tools/`

where SYCL uses a heavily templated C++ API.

In SYCL, there is typically a queue that work items can be submitted to. Parallelisation is achieved using constructs such as the `parallel_for` function.

Building on SYCL, Intel announced their new programming model, **OneAPI**, in 2018. OneAPI is a unified programming model, that combines several libraries (e.g. the Math Kernel Library), with Thread Building Blocks (TBB) and **Data Parallel C++** (DPC++). DPC++ is a cross-architecture language built upon the C++ and SYCL standard, providing some extensions to SYCL. Support for SYCL and DPC++ is provided in a number of compilers from vendors such as AMD, Intel, Codeplay and Xilinx, and can target a number of device types directly, or via existing OpenCL targets[3]. In the case of the Intel and Xilinx compilers, it is even possible to use SYCL to target FPGA devices. However, the question of whether one code written in SYCL is able to obtain the best performance on all supported hardware remains to be answered [15, 16, 17].

Parallelisation based on OpenMP and MPI have a long history in HPC application development. CUDA also now has about a decade of development, with OpenACC, and OpenCL following close behind. SYCL/DPC++ is the latest addition to the parallel programming extensions available. While CUDA, OpenMP, OpenACC all support C/C++ and Fortran, OpenCL and SYCL only support C/C++. If indeed C/C++ based extensions and frameworks dominate the parallel programming landscape for emerging hardware, there could well be a need for porting existing Fortran-based applications to C/C++.

The key considerations and challenges when using the above programming models and extensions to general purpose languages include:

1. Open standards lagging hardware development – especially when the standard is developed by a large number of organisations.
2. The *complete* implementation of these standards into many compilers can be slow.
3. Some of these programming models offer low-level fine control over parallelism and therefore may lead to overly complex code. In some cases different code-paths are required to get the best performance on different architectures [16], for example to handle the different memory layouts required to optimise for CPUs vs GPUs.

3.3 Software Libraries

In this class we consider classical software libraries that target scientific application development, implementing a diverse set of numerical algorithms.

Beyond the programming models mentioned previously, portability can also be achieved using kernel libraries provided by various vendors. These software libraries typically provide common mathematical functions and are often highly optimised for particular architectures.

[3] `https://www.khronos.org/sycl/`

The basis of many of these libraries is **BLAS** (Basic Linear Algebra Subprograms), first developed in 1979. BLAS provides vector operations, matrix-vector operations and matrix-matrix operations. **LAPACK** (Linear Algebra Package) builds on BLAS and provides routines for solving systems of linear equations. The **FFTW** library provides functions for computing discrete Fourier transforms, and is known to be the fastest free software implementation of the FFT.

Architecture-tuned implementations of BLAS, LAPACK and FFTW are often available, with notable examples being **AMD Optimized CPU Libraries**, **ARM Performance Libraries**, **Intel Math Kernel Library**, **cuBLAS**, **clBLAS**, **OpenBLAS**, and **Boost.uBLAS**. Similarly, **MAGMA** provides dense linear algebra kernels for multicore and accelerator architectures [18].

The **Portable, Extensible Toolkit for Scientific Computation (PETSc)** provides a number of data structures and routines for solving PDEs. It was developed by Argonne National Laboratory and employs MPI for distributing algorithms across an HPC system. Recently PETSc has implemented an abstraction layer for scalable communications over MPI and between host and GPU devices, PetscSF [19].

Similarly, **HYPRE** is a library of data structures, preconditioners and solvers developed at Lawrence Livermore National Laboratory. It can be built with support for GPU devices through CUDA, OpenMP offload, or using RAJA or Kokkos.

Trilinos is an extensive collection of open-source libraries that can be used to build scientific software, developed by Sandia National Laboratories. It provides a large number of packages for solving linear systems, preconditioning, using sparse graphs and matrices, among many others. It supports distributed memory computation through MPI and also provides shared memory computation through its own Kokkos package. Trilinos is included on Cray supercomputers as part of the **Cray Scientific and Math Libraries**.

The **CoPA Cabana** library provides a number of data structures, algorithms and utilities specifically for particle-based simulations [20]. Parallel execution of particle kernels is achieved through Kokkos for on-node parallelism (see Section 3.4) and MPI for off-node communication. Each of these libraries can be used to abstract away some of the mathematical operations and data storage requirements needed by scientific applications.

Using these libraries introduces a number of key considerations and challenges:

1. While the standard interfaces to these libraries may restrict their usefulness to some applications, it does encourage vendors to produce optimised *and portable* versions of performance critical functions.
2. Library functions often operate in lock-step, meaning operations cannot typically be fused. This may necessitate a number of unnecessary CPU-GPU transfers.

3.4 C++ Template Libraries

For this class we consider libraries that facilitate scheduling and execution of data parallel or task-parallel algorithms in general, but themselves do not implement numerical algorithms.

An approach, exclusive to C++ is the use of template libraries, which enables developers to write a generic "template" to express the operation such as a parallel-loop iteration, but at compile time select a specific implementation of a method or function (known as static dispatch). This allows users to express algorithms as a sequence of parallel primitives executing user-defined code at each iteration, e.g., providing a loop-level abstraction. These libraries follow the design philosophy of the C++ Standard Template Library [21] – indeed, their specification and implementation is often considered as a precursor towards inclusion in the C++ STL. The largest such projects are **Boost** [22], **Eigen** [23], and **HPX** [24]. While there are countless such libraries, here we focus on ones that also target performance portability in HPC.

Kokkos [25] is a C++ performance portability layer that provides data containers, data accessors, and a number of parallel execution patterns. It supports execution on shared-memory parallel platforms, such as CPUs and GPUs; it does not consider distributed memory parallelism, rather it is designed to be used in conjunction with MPI. Kokkos is a package within Trilinos, and is used to parallelise many of its solver libraries, but it can also be used as a standalone tool. Its data structures can describe where data should be stored (CPU memory, GPU memory, non-volatile, etc.), how memory should be laid out (row/column-major, etc.), and how it should be accessed. Similarly, one can specify where algorithms should be executed (CPU/GPU), what algorithmic pattern should be used (parallel for, reduction, tasks), and how parallelism is to be organised. It is a highly versatile and general tool capable of addressing a wide set of needs, but as a result is more restricted in what types of optimisations it can apply compared to a tool that focuses on a narrower application domain. Kokkos is able to target CUDA, OpenMP, pthreads, HIP or SYCL, meaning it can target all of the post-Exascale platforms currently deployed or in development.

RAJA is a similar abstraction developed by Lawrence Livermore National Laboratory [26]. It is in many respects similar to Kokkos but offers more flexibility for manipulating loop scheduling, particularly for complex nested loops. It also supports CPUs (with OpenMP and TBB), as well as NVIDIA GPUs with CUDA.

Both Kokkos and RAJA were designed by US DoE labs to help move existing software to new heterogeneous hardware, and this very much is apparent in their design and capabilities – they can be used in an iterative process to port an application, loop-by-loop, to support shared-memory parallelism. Of course, for practical applications, one needs to convert a substantial chunk of an application; on the CPU that is because non-multithreaded parts of the application can become a bottleneck, and on the GPU because of the cost of moving data to/from the device. Kokkos and RAJA are used heavily within the Exascale Computing Project (ECP) [27], and due to their reliance on template meta programming, can be used alongside almost any modern C++ compiler.

Using C++ template libraries comes with the following considerations:

1. Development time may be high due to the compilation times associated with heavily templated code.
2. Applications are restricted to being developed in modern C++.
3. Debugging heavily templated code can be difficult, with errors obfuscated by numerous templates. This can be particularly problematic for novice physicist programmers.
4. Platform specific code can be easily integrated into templated code to achieve higher performance on some platforms, provided that the abstraction used is carefully designed and at a sufficiently high level.

3.5 Domain Specific Languages

In this category we consider a wide range of languages and libraries – the key commonality is that their scope is limited to a particular application or algorithmic domain.

Domain Specific Languages (DSLs) and approaches by definition restrict their scope to a narrower problem domain, set of algorithms, or computation/communication patterns. By sacrificing generality, it becomes feasible to attempt and address challenges in gaining all three of performance, portability and productivity. A wide range of approaches exist, at different levels of abstractions starting from libraries focusing on specific numerical methods (e.g. Finite Element method) to low-level parallel computation patterns and loop abstractions. Some are embedded in general purpose languages (eDSLs) such as C/C++/Fortran or Python allowing them to make use of the compiler and development infrastructure (debuggers and profilers) of these languages. Others develop an entirely new language of their own.

Restricting to a specific domain allows DSLs to apply more powerful optimisations to help deliver performance as well as portability. The key reason being that a lot of assumptions are already built into the programming interface (the domain specific API). As such, explicit description of the problem need not occur when programming with DSLs, significantly improving productivity. Conversely, the key deficiency of DSLs then is their limited applicability – if they cannot develop a considerable userbase, they will lack the support required to maintain them. As such two key challenges to building a successful DSL or framework are:

1. An abstraction that is wide enough to cover a range of interesting applications, but narrow enough so that powerful optimisations can be applied.
2. An approach to long-term support. A feasible model would be to follow the maintenance pattern of classical libraries.

DSLs can be categorised based on their level of abstraction. At a low level, a DSL might provide abstractions for sequences of basic algorithmic primitives, such as parallel for-each loops, reduction, scan operations etc. Kokkos and RAJA can be thought of as such loop-level abstractions supporting a small set of computation-communication "patterns".

3.5.1 DSLs for Stencil Computations

At a higher-level we could consider DSLs for stencil computations, providing abstractions for structured or unstructured stencil-based algorithms. This class of DSLs are for the most part oblivious to the numerical methods being implemented, which in turn allows them to be used for a wider range of algorithms, e.g., finite differences, finite volumes, or finite elements. The key goal here is to create an abstraction that allows the description of parallel computations over either structured or unstructured meshes (or hybrid meshes), with neighbourhood-based access patterns. Similar DSLs can be constructed for domains such as molecular dynamics that help express N-body interactions.

There are a number of notable and currently active DSLs at this level of abstractions. **Halide** [28] is a DSL intended for image processing pipelines, but generic enough to target structured-mesh computations [29], it has its own language, but is also embedded into C++ – it targets both CPUs and GPUs, as well as distributed memory systems. **YASK** [30] is a C++ library for automating advanced optimisations in stencil computations, such as cache blocking and vector folding. It targets CPU vector units, multiple cores with OpenMP, as well as distributed-memory parallelism with MPI. **OPS** [31] is a multi-block structured mesh DSL embedded in both Fortran and C/C++, targeting CPUs, GPUs and clusters of CPUs/GPUs – it uses a source-to-source translation strategy to generate code for a variety of parallelisations. **ExaSlang** [32] is part of a larger European project, ExaStencils, which allows the description of PDE computations at many levels – including at the level of structured-mesh stencil algorithms. It is embedded in Scala, and targets MPI and CPUs, with limited GPU support. Another DSL for stencil computations, **Bricks** [33] gives transparent access to advanced data layouts using C++, which are particularly optimised for wide stencils, and is available on both CPUs, and GPUs.

OP2 [34] and its Python derivative, **PyOP2** [35], give an abstraction to describe neighbourhood computations for unstructured meshes. They are embedded in C/Fortran and Python respectively, and can target CPUs, GPUs, and distributed memory systems. Unlike the structured-mesh motif (which uses a regular stencil), unstructured mesh computations are based on explicit connectivity information between mesh elements, leading to indirect increments. Indirect increments need to be carefully handled when parallelising, given the existence of data dependencies, and as such need different code-paths to obtain the best performance on different architectures [16]. OP2 generates parallel code targeting CPU and GPU clusters making use of a range of parallel programming models (SIMD, OpenMP, CUDA, SYCL etc. and their combinations with MPI). For mixed mesh-particle, and particle methods, **OpenFPM** [36], embedded in C++, provides a comprehensive library that targets CPUs, GPUs, and supercomputers.

A number of DSLs have emerged from the weather prediction domain such as **STELLA** [37] and **PSyclone** [38]. STELLA is a C++ template library for stencil computations, that is used in the COSMO dynamical core [39], and supports structured mesh stencil computations on CPUs and GPUs. PSyclone is part of the effort in modernising the UK Met Office's Unified Model weather code and uses automatic code generation. It currently uses only OpenACC for executing on GPUs. A very different approach is taken by the **CLAW-DSL** [40], used for the ICON model [41], which is targeting Fortran applications, and generates CPU and GPU parallelisations – mainly for structured mesh codes, but it is a more generic tool based on

source-to-source translation using preprocessor directives. It is worth noting that these DSLs are closely tied to a larger software project (weather models in this case), developed by state-funded entities, greatly helping their long-term survival. At the same time, it is unclear if there are any other applications using these DSLs.

3.5.2 Higher-Level DSLs

Domain specificity can be at an even higher level, where the DSL focuses on the declaration and solution of particular numerical problems. The most widely implemented DSLs at such a high level are frameworks for the solution of PDEs. The problem is specified starting at the symbolic expression of the problem (e.g. in Einstein notation). An intepretter or a compiler then (semi-) automatically discretises the problem and generates a solution. Most are focused on a particular set of equations and discretisation methods, and they can offer excellent productivity – assuming the problem to be solved matches the focus of the library.

Many of these libraries, particularly ones where portability is important, are built with a layered abstractions approach; the high-level symbolic expressions are transformed, and then passed to a layer that maps them to a discretisation, then this is given to a layer that arranges parallel execution – the exact layering of course depends on the library. This approach allows the developers to work on well-defined and well-separated layers, without having to gain a deeper understanding of the whole system. These libraries are most commonly embedded in the Python language, which has the most commonly used tools for symbolic manipulation in this field – although functional languages are arguably better suited for this, they still have little use in HPC. Due to the poor performance of interpreted Python, these libraries ultimately generate low-level C/C++/Fortran code to deliver high performance.

One of the most established such libraries is **FEniCS** [42], which targets the Finite Element Method. However it only supports CPUs and distributed memory cluster execution with MPI. **Firedrake** [43] is a similar project with a different feature set, which also only supports CPUs – it uses the aforementioned PyOP2 library for parallelising and executing generated code. A feature of Firedrake is that it generates code at runtime to exploit further optimisation opportunities, for example based on the mesh being available/input at runtime. The **ExaStencils** project [44] uses four layers of abstraction to create code running on CPUs or GPUs from the continuous description of the problem – its particular focus is structured meshes and multigrid. **OpenSBLI** [45] is a DSL embedded in Python, focused on resolving shock-boundary layer interactions and uses finite differences and structured meshes – it generates C code using the OPS library which provides the stencil abstraction. As noted before, OPS then generates parallel code targeting distributed memory machines with both CPUs and GPUs. **Devito** [46] is a DSL embedded in Python which allows the symbolic description of PDEs, and focuses on high-order finite difference methods, with the key target being seismic inversion applications. Devito also supports CPU and GPU parallelisation, where GPU acceleration is obtained by generating OpenACC directives.

In fusion research, the **BOUT++** framework has been developed as a flexible toolbox for solving a wide range of PDEs [47, 48]. Its design was in large part driven by the need for physicist users to modify and customise the model equations being solved. BOUT++ therefore uses C++ features to implement models in a way which closely mimics their mathematical form.

The BOUT++ framework then solves these equations, and allows the user runtime control over the finite difference methods and stencils used, as well as time integration solver, Laplacian inversions, and so on.

BOUT++'s physics model implementation language is an example of a eDSL, in this case C++ is the host language. eDSLs have the advantage of the user/developer being able to easily "break out" of the DSL and write generic code for situations not handled by the DSL, for example to handle complicated boundary conditions. The cost of this approach is that certain transformations of the code are harder to achieve. For example, each physics and arithmetic operator in BOUT++ contains a loop over the whole domain for its own kernel. To achieve the full performance with OpenMP or accelerators requires merging these loops into a single loop. This in turn necessitates rewriting the top-level set of equations to include this loop explicitly, or to use something akin to expression templates (as is done in libraries such as Eigen or Blitz++), which have their own downsides.

In addition to the above eDSL for implementing physics models, BOUT++ has a second DSL to specify the inputs and initial conditions for the simulations. This started from a simple `.ini` input format, but has developed over time into a Turing-complete language of its own, with a custom interpreter included in BOUT++. This gradual increase in complexity has been driven by the needs of physics studies, improving ease of use (reducing or eliminating pre-processing steps), and to facilitate testing with complex analytical expressions using the Method of Manufactured Solutions (MMS).

This flexibility in the input has proven to be extremely useful to users, and as a DSL the format is well suited to its specialised task of providing input expressions to BOUT++ simulations. Because of how it has gradually evolved in BOUT++, it is however a DSL with a very limited number of users, with all the disadvantages which come with this discussed previously. BOUT++ currently only supports execution on CPUs with OpenMP for multi-threading and MPI for distributed memory execution. Experimental branches exists with ongoing development to support GPU execution. These include (1) using Hypre [49] with GPU support for the Laplacian inversion parts of the problem (which in practice can take about half the total time) and (2) with RAJA for putting the user physics model on GPUs, with Umpire [50] to handle memory. This requires modifying the physics DSL to enable operations to be fused together, reducing the number of separate kernels which need to be launched.

Similar to BOUT++, the **Unified Form Language** (UFL), used in FEniCS and Firedrake provides a high-level language to describe variational forms. The problem to be solved is specified at a high level, which corresponds closely to the mathematical form.

Firedrake uses the FEniCS Form Compiler (FFC) to convert UFL to an intermediate representation, and then uses PyOP2 to generate code for target architectures, aiming to be performance portable on both CPUs and GPUs.

The most common challenges when using DSLs include:

1. Difficulties in debugging due to the extra hidden layers of software between user code and code executing on the hardware. However, DSLs generating low-level C/C++/Fortran codes can use standard

debuggers or profilers.

2. Extensibility – implementing algorithms that fall slightly outside of the abstraction defined by the DSL can be an issue.

3. Customisability – it is often difficult to modify the implementation of high-level constructs generated automatically.

To mitigate some of these issues, systems can be provided with "escape hatches", which provide ways for users to implement components of the problem which cannot be expressed in the high-level DSL. An example is custom flux-limiters, which cannot currently be expressed in UFL; instead a user needs to be able to implement their own kernels, and integrate these into the remainder of the system in an elegant way. Firedrake provides such escape hatches for direct access to linear algebra operators (PETSc), and allows implementation of custom PyOP2 kernels. However it should be noted that such modifications may not deliver the best performance on all hardware and should be used only sparingly or for prototyping. As it is the case with many complex performance issues there is no silver-bullet to solve all cases.

3.6 Summary

The increasingly diverse range of hardware being used in modern day HPC systems is making programming for these systems much more difficult. While most vendors provide hardware-specific programming models for dealing with heterogeneous parallelism, these are typically not portable between competing architectures and therefore may require significant redevelopment for any new hardware platforms.

Instead, a number of *performance portable* approaches have been proposed and developed. These approaches range from lightweight directive-based approaches, instructing a compiler to parallelise code effectively, to kernelising code specifically for execution on an accelerator.

Achieving high performance on the today's largest HPC systems requires application developers to deal with hierarchical parallelism. For many new applications, this will likely require a mix of programming languages and programming models (e.g. so called "MPI+X"). Additionally, this may require multiple levels of DSL, e.g., a DSL that allows users (domain scientists) to express the equations required, while a lower-level DSL generates efficient application code for execution on a wide-range of hardware. Certainly combining the expertise of DSL developers at these different levels, optimising for a multi-layered solution seems to be the most feasible and performant. Additionally, such an approach appears to provide the best future-ready option with transparent layers aiding in maintenance and extensibility.

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 is using Nektar++ to explore the performance of spectral elements, T/NA083/20 is focused on building a fluid referent model in both Bout++ and Nektar++, and T/NA079/20 is exploring the use of particle methods with the EPOCH particle-in-cell (PIC) code. It is therefore likely that the resultant NEPTUNE software stack will be a fluid model, based on the output of T/NA078/20 and T/NA083/20, coupled with a particle model, based on the output of T/NA079/20.

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 will 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 [51], UK-MAC [52], the NAS Parallel Benchmarks [53], the ECP Proxy Apps [27] and the SPEC benchmarks [54]. In this section, we will discuss the applications we have identified from these benchmark suites, that may be relevant to our performance investigations.

4.1 Fluid Models

As previously noted, the fluid modelling aspects of the NEPTUNE project are largely focused on the use of **Bout++** [47, 55] and **Nektar++** [56]. 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++ [57].

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[4].

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 [58, 59, 60, 61], and is available implemented using CUDA, HYPRE, OpenCL, PETSc and Trilinos[5].

miniFE

miniFE is a finite element mini-app, and part of the Mantevo benchmark suite [62, 8, 63, 64]. It implements an unstructured implicit finite element method and is available implemented using CUDA, Kokkos, OpenMP and OpenMP with offload[6].

Laghos

Laghos is a mini-app that is part of the ECP Proxy Applications suite [65, 66, 64]. 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[7].

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[8].

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[9].

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 [67] to provide portability to OpenMP, CUDA and HIP[10].

[4] `https://github.com/UoB-HPC/heat_sycl`
[5] `http://uk-mac.github.io/TeaLeaf/`
[6] `https://github.com/Mantevo/miniFE`
[7] `https://github.com/CEED/Laghos`
[8] `https://github.com/zer011b/fdtd3d`
[9] `https://mfem.org/electromagnetics/`
[10] `https://github.com/paranumal/hipBone`

4.2 Particle Methods

The optimal use of particles in NEPTUNE is currently being explored using the EPOCH particle-in-cell code [68], and its associated mini-app, minEPOCH [69][11]. 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. However, there are a number of 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 [64]. 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 [70]. VPIC is parallelised on-core using vector intrinsics, on-node through pthreads or OpenMP, and off-node using MPI. VPIC 2.0 [71] 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 [72]. 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 [73, 74].

Each of the three 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 all three 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:

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].

[11] `https://github.com/ExCALIBUR-NEPTUNE/minepoch`
[12] `https://github.com/ECP-copa/CabanaPIC`
[13] `https://github.com/lanl/vpic`
[14] `https://github.com/zjin-lcf/HeCBench/tree/master/sheath-cuda`

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++ [55]) to high-order finite element or spectral element (like Nektar++ [56]). Similarly, the particle-methods mini-apps all implement the particle-in-cell method (like EPOCH [68]).

The data presented in this section, and the applications are available on github, through the linking repository :`https://github.com/swright87/NEPTUNE-Miniapps`.

5.1 Heat

Heat is a benchmark from 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.

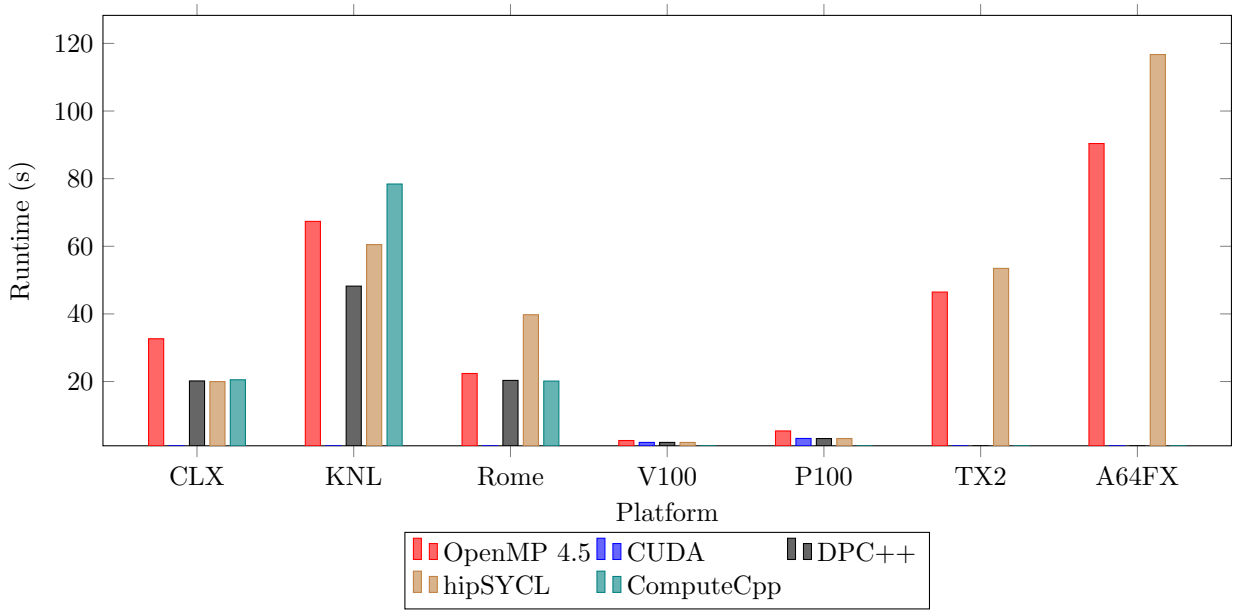


Figure 3: Heat runtime data

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, 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 3. 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 the only compiler that has been able to target every architecture, 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.

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 reasonable 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.

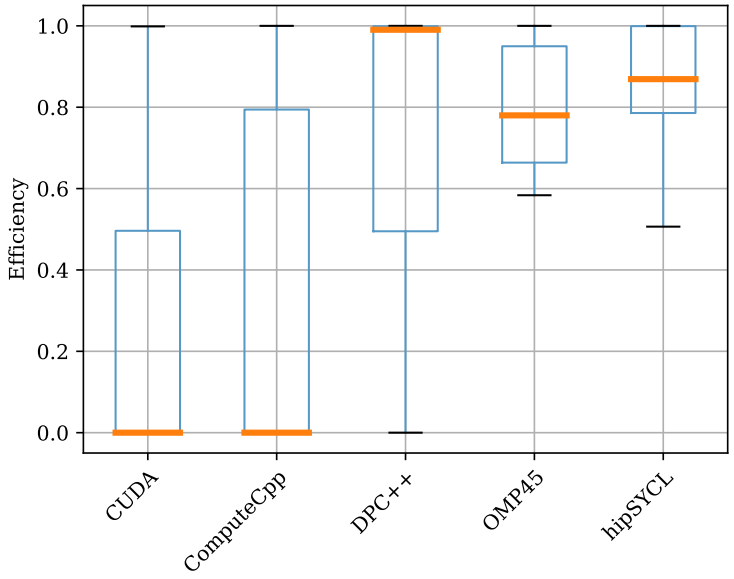


Figure 4: Box plot visualisation of performance portability of Heat

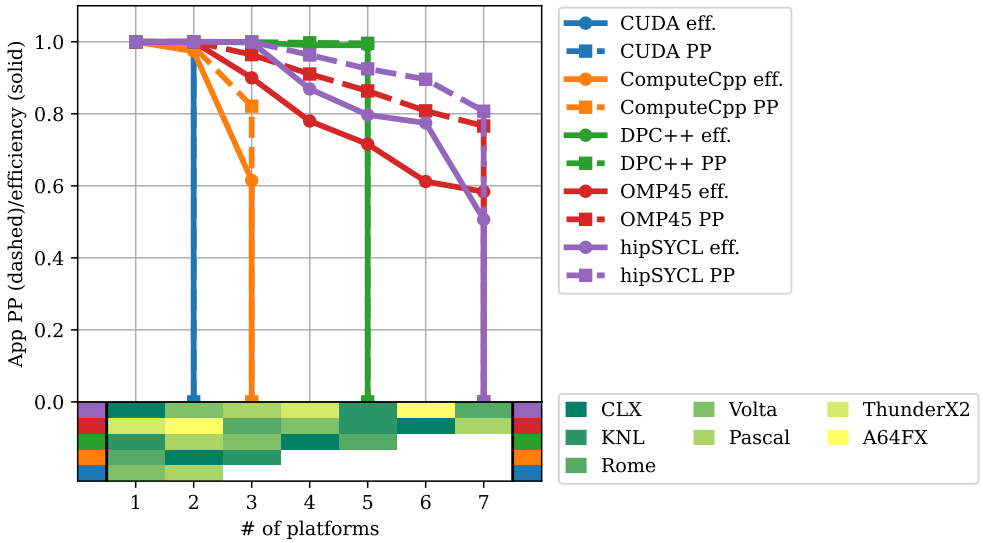


Figure 5: Cascade visualisation of performance portability of Heat

5.1.2 Portability

Figures 4 and 5 show the performance portability of the Heat application.

The best *performance portability* is achieved by OpenMP with offload and the SYCL port, compiled by hipSYCL. These two variants can be executed on every platform in the evaluation set. Figure 5 additionally shows that as platforms are added to the evaluation set, hipSYCL achieves ∼80% efficiency or more until the final platform (A64FX in this case) is added.

Conversely, CUDA and the SYCL port, as compiled by the ComputeCpp compiler, show the lowest portability. As expected the CUDA port only runs on the GPU platforms, while the ComputeCpp compiler relies heavily on the availability of a compliant OpenCL driver.

For the Rome and KNL platforms, DPC++ provides better performance than hipSYCL from the same codebase, 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 [58, 59, 60, 61], 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. [59] and one by Deakin et al. [58].

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 Intel Broadwell system, an Intel KNL system and an NVIDIA P100 system. The runtime for each implementation/configuration is presented in Figure 6. 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 Intel Skylake system, both NVIDIA P100 and V100 systems, AMDs Naples CPU, and the Arm-based ThunderX2 platform. Runtime results are provided in Figure 7.

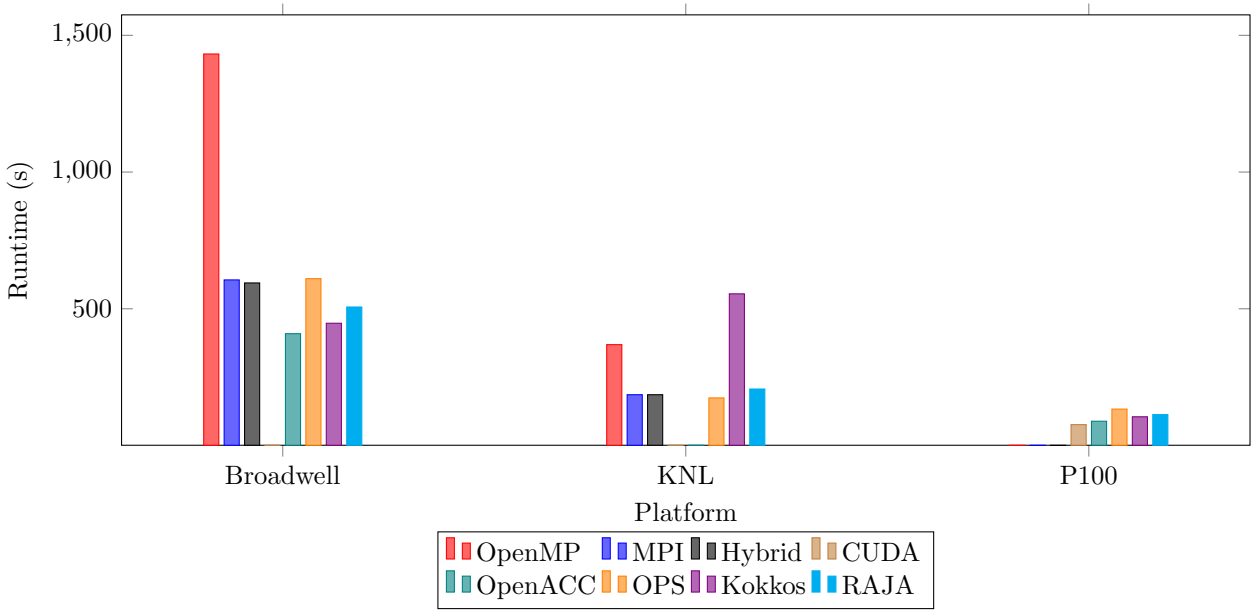


Figure 6: TeaLeaf runtime data from Kirk et al. [59]

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

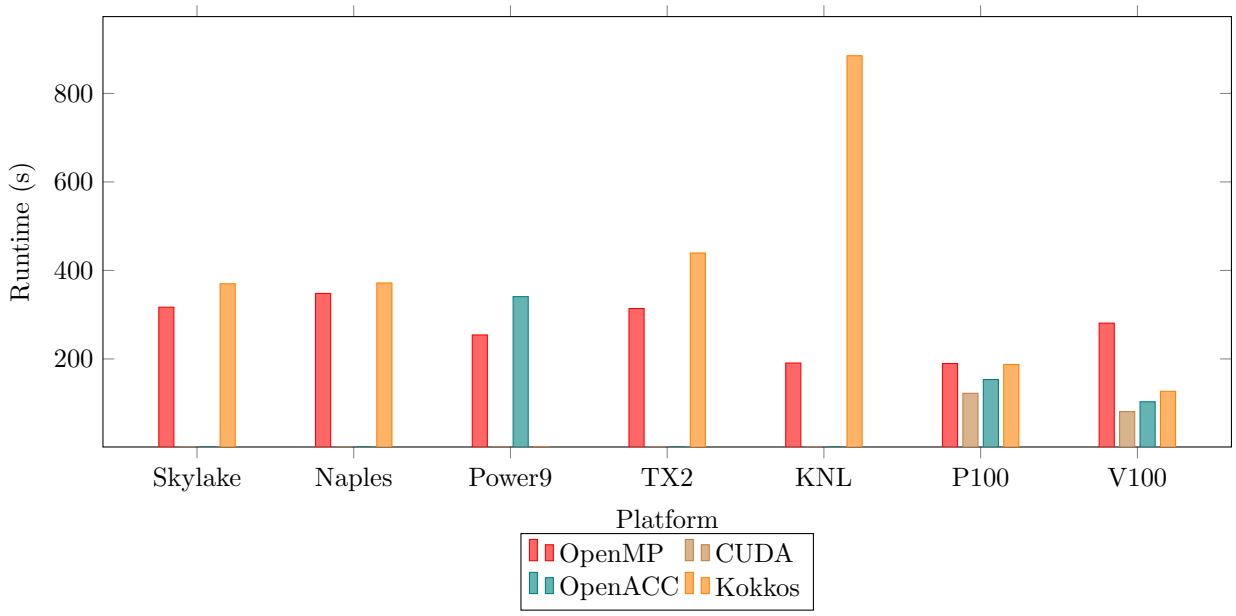


Figure 7: TeaLeaf runtime data from Deakin et al. [58]

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.

For study by Kirk et al. [59], Figures 8 and 9 allow us to visualise the performance portability of each approach to application development. Figure 8 shows a clear divide between the non-portable approaches (CUDA, OpenMP, MPI, Hybrid and OpenACC), and the portable approaches (Kokkos, OPS and RAJA), whereby each of the non-portable approaches span the full range from 0.0 efficiency up to 1.0 efficiency, while the three portable approaches each span a much smaller range of efficiencies.

Figure 9 better shows how the performance portability of each implementation changes as new platforms are added to the evaluation set. 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 6, 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.

Figures 10 and 11 show the same visualisations for the data from Deakin et al. [58]. Again, the non-portable programming model (CUDA) achieves the highest performance on its target architecture. 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.

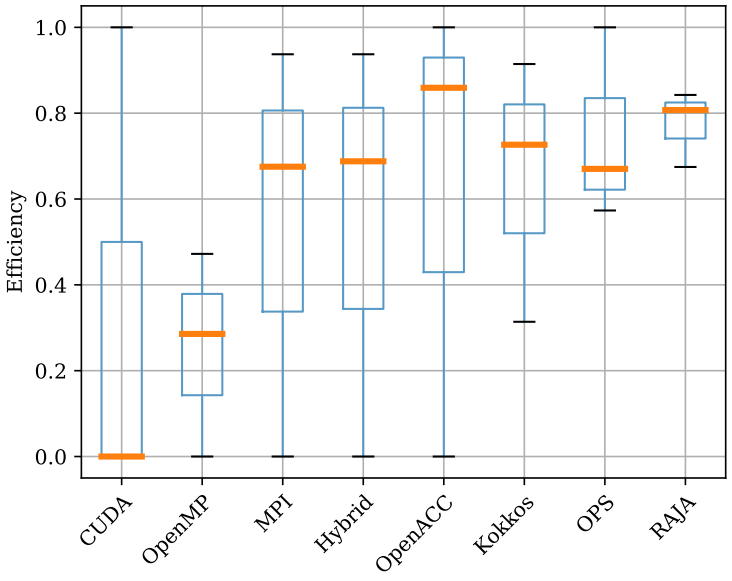


Figure 8: Box plot visualisation of performance portability from Kirk et al. [59]

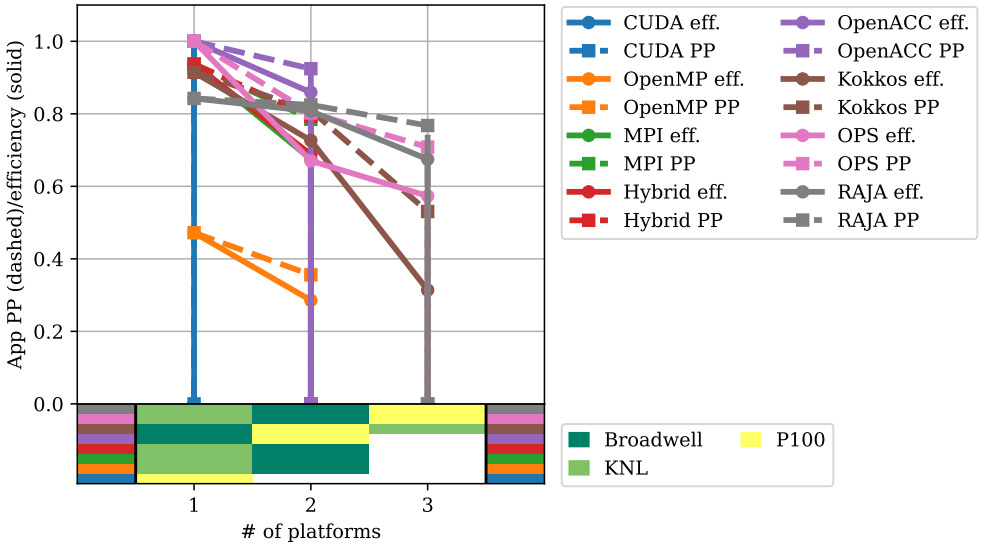


Figure 9: Cascade visualisation of performance portability from Kirk et al. [59]

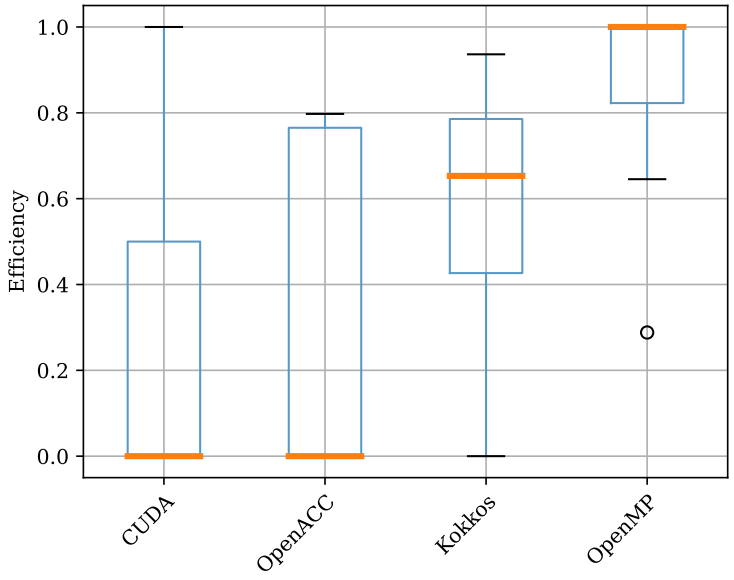


Figure 10: Box plot visualisation of performance portability from Deakin et al. [58]

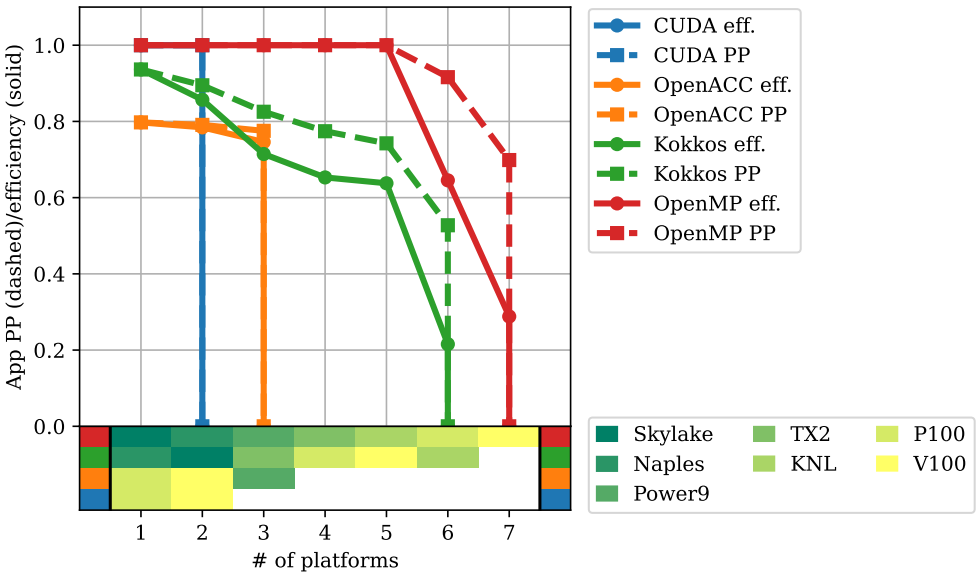


Figure 11: Cascade visualisation of performance portability from Deakin et al. [58]

5.3 miniFE

miniFE is a finite element mini-app, and part of the Mantevo benchmark suite [62, 8, 63, 64]. It implements an unstructured implicit finite element method and has versions available in CUDA, Kokkos, OpenMP (3.0+ and 4.5+) and SYCL[17].

While there are a number of data sources for miniFE data, many of these are limited in scope, and so to ensure consistency, all data presented in this section has been newly gathered. 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.

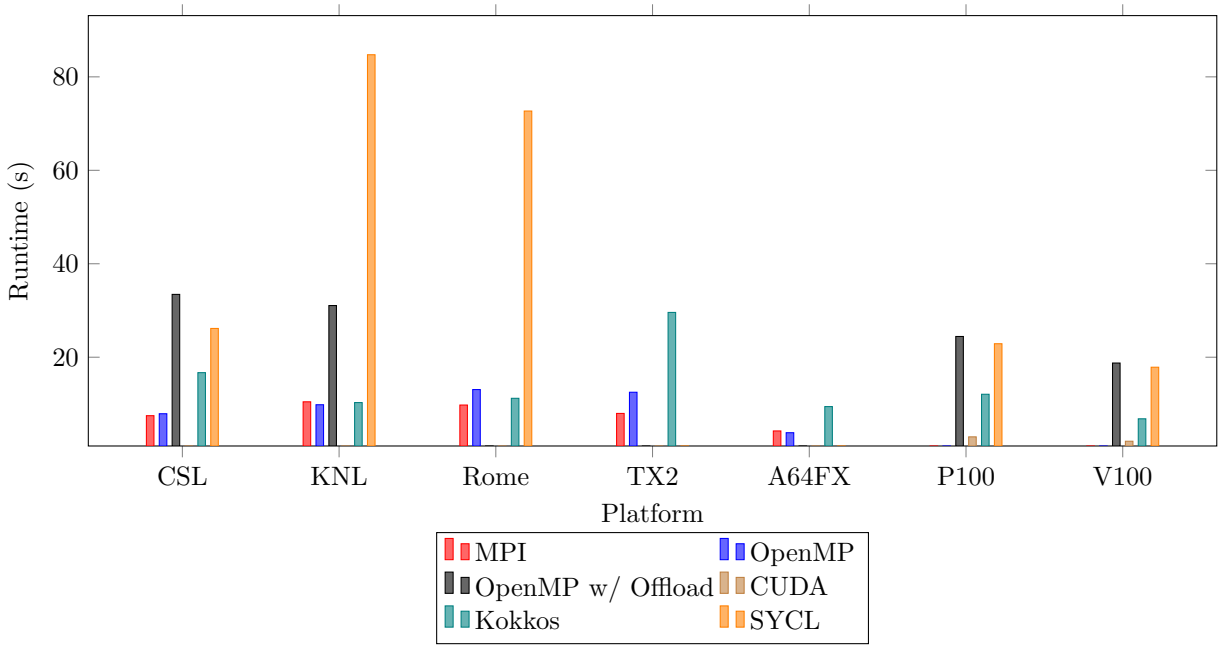


Figure 12: miniFE runtime data

5.3.1 Performance

The raw runtime results for these runs can be seen in Figure 12. In many of the miniFE ports available, only the conjugate solver has been parallelised effectively, so the results presented here 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[18]; this port has been generated using Intel's DPC++ Compatibility tool, which translates CUDA to DPC++, and is compiled using hipSYCL and GCC. Results have not yet been collected for the ARM-based system with SYCL, due to the unavailability of an appropriate compiler. The OpenMP with offload variant of miniFE runs successfully on both AMD Rome and Cavium

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

ThunderX2 platforms, but the runtimes are several orders of magnitude greater than all other platforms (likely due to an bug in the compiled code), and so have been removed.

Figure 12 shows that SYCL performance on the KNL and Rome platforms is far in excess of any other execution (with the exception of OpenMP with Offload on Rome which is not shown), and on the GPU platforms the SYCL runtime is on par with OpenMP with offload. This is likely due to the hipSYCL compiler generating OpenMP with Offload syntax for the SYCL code, and so it is unsurprising that performance is similar. Otherwise, the fastest performance on most CPU-based platforms comes from the native MPI variant of miniFE, and the fastest performance on the GPU-based platforms comes from CUDA.

5.3.2 Portability

Figures 13 and 14 present visualisations of the performance portability of miniFE, through various approaches.

The highest median performance comes from the non-portable MPI approach, since it is the best (or near best) performing implementation on all of the CPU platforms; however, it is not portable to the two GPU systems. Conversely, Figure 13 shows that CUDA has the worst lowest median performance, because it only runs on the two GPU systems, but is the best performing on each. The boxplots for both OpenMP with Offload and SYCL are very similar, and this is likely an artefact of SYCL being translated to OpenMP with Offload at compile-time by hipSYCL. The only programming model to run across all platforms currently is Kokkos, but on some platforms this may mean sacrificing a significant proportion of performance; whether this is sacrifice is acceptable likely requires further analysis considering the runtime cost alongside other associated costs, such as FLOP/s per Watt and FLOP/s per dollar.

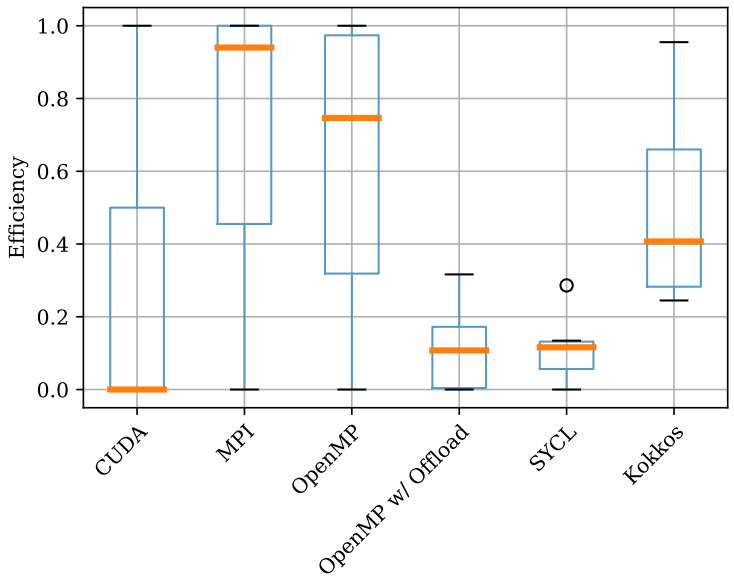


Figure 13: Box plot visualisation of performance portability of miniFE

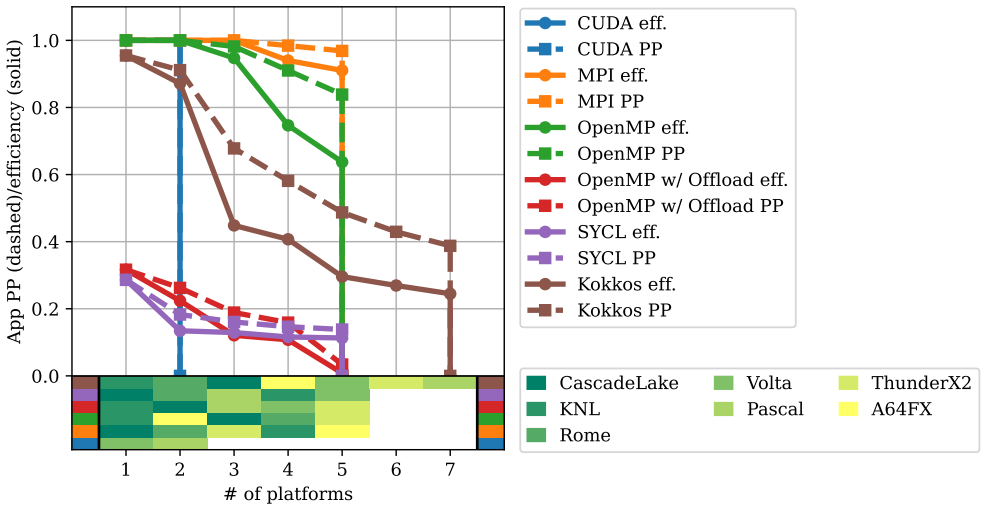


Figure 14: Cascade visualisation of performance portability of miniFE

Figure 14 better shows how the performance portability of miniFE evolves as more platforms are added for each programming model.

For CUDA, MPI, OpenMP and Kokkos, there are at least two platforms where they achieve over 80% efficiency, and in the case of MPI and OpenMP, this efficiency holds up until we reach the GPU platforms, while CUDA does the inverse of showing the best efficiency on the GPU platforms. SYCL and OpenMP with offload offer poor performance in our current data, and hence achieve less than 40% of peak application performance across all platforms; this is likely due to the use of the hipSYCL compiler and lack of platform specific optimisations[19]. As Kokkos is the only programming model we have full data for, it is the only programming model that spans all platforms; however, the performance efficiency decreases as more platforms are added to the evaluation set. While it is clearly a portable approach, it is not clear whether it is *performance portable* at this time.

5.4 Laghos

Laghos is a mini-app that is part of the ECP Proxy Applications suite [65, 66, 64]. 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 from the Isambard platform.

[19]Note: new data is in the process of being gathered using DPC++ and the newest release of the hipSYCL compiler. This data will be included in a future iteration of this report.

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

5.4.1 Performance

Figure 15 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.

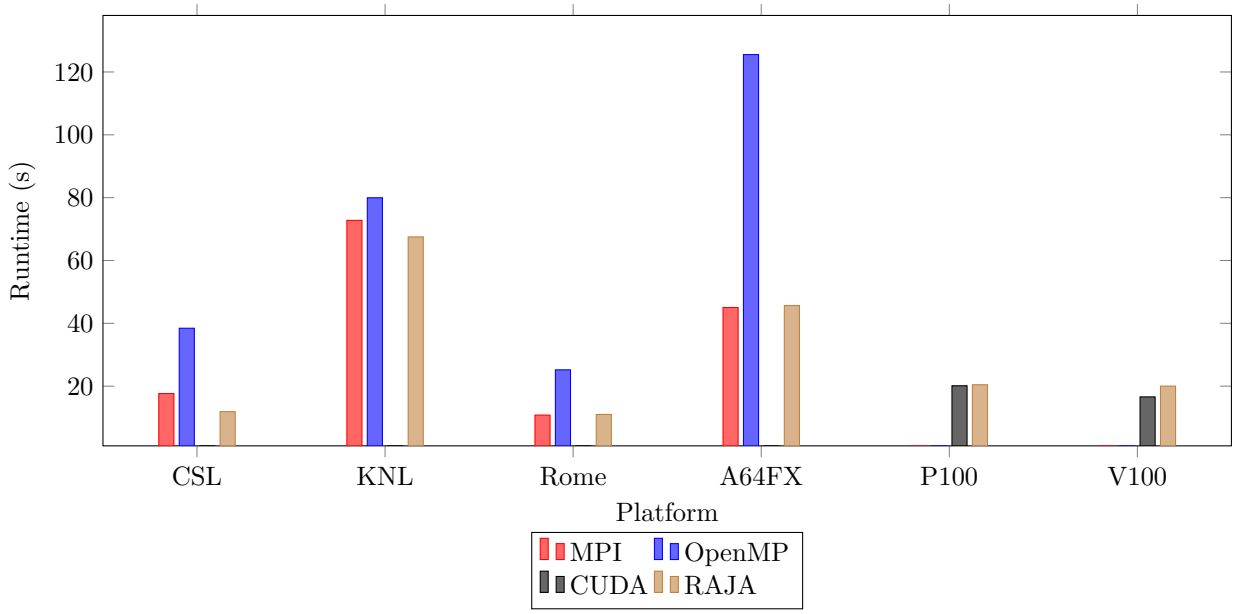


Figure 15: Laghos runtime data

5.4.2 Portability

Portability visualisations of each implementation of Laghos are provided in Figures 16 and 17.

Figure 16 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.

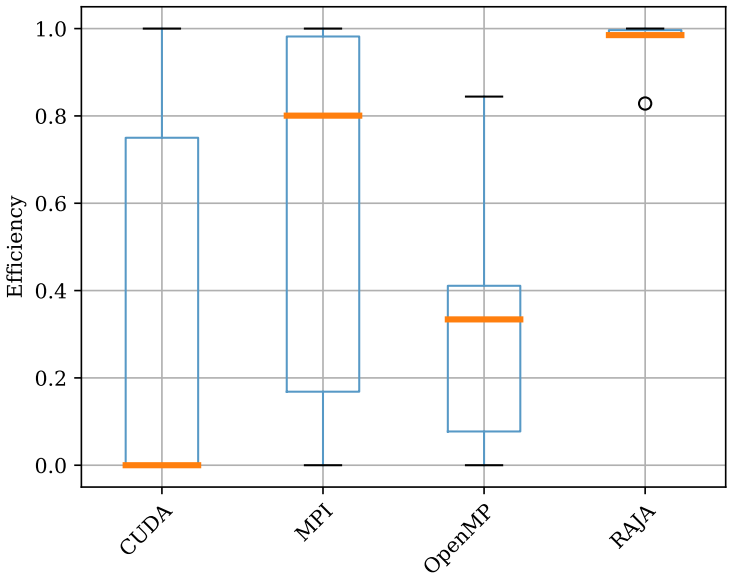


Figure 16: Box plot visualisation of performance portability of Laghos

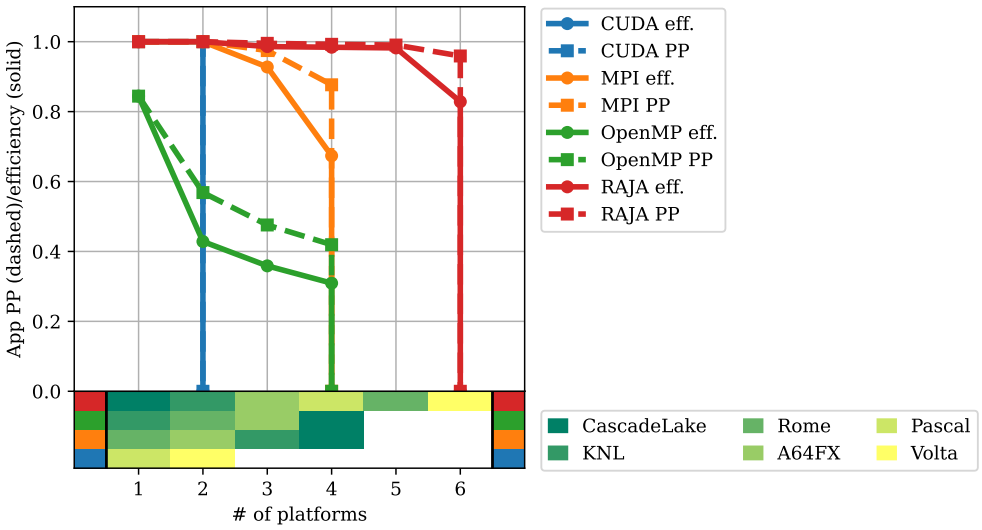


Figure 17: Cascade visualisation of performance portability of Laghos

5.5 CabanaPIC

CabanaPIC is a structured PIC demonstrator application built using the CoPA/Cabana library for particle-based simulations [64]. 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].

5.5.1 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 18 shows the achieved runtime for CabanaPIC across four of Isambard's platforms, running a simple 1D 2-stream problem with 6.4 million particles.

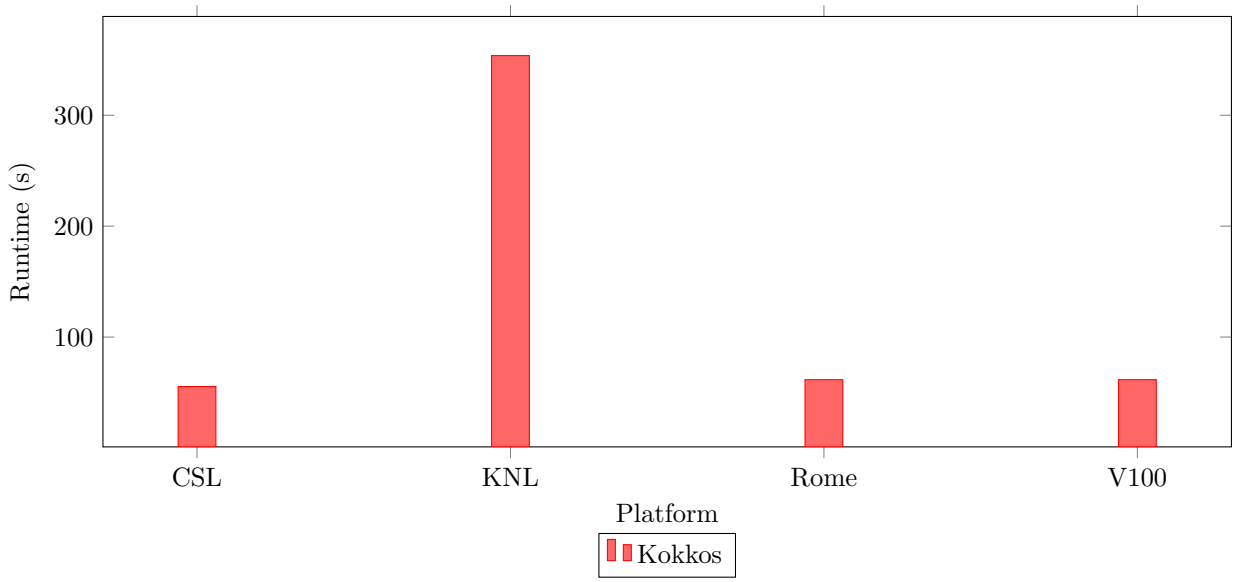


Figure 18: CabanaPIC data

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.

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

5.6 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 [70]. 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 [71] code has been developed to add support for heterogeneity using Kokkos to optimise the data layout and allow execution on accelerator devices.

5.6.1 Performance

Figure 19 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.

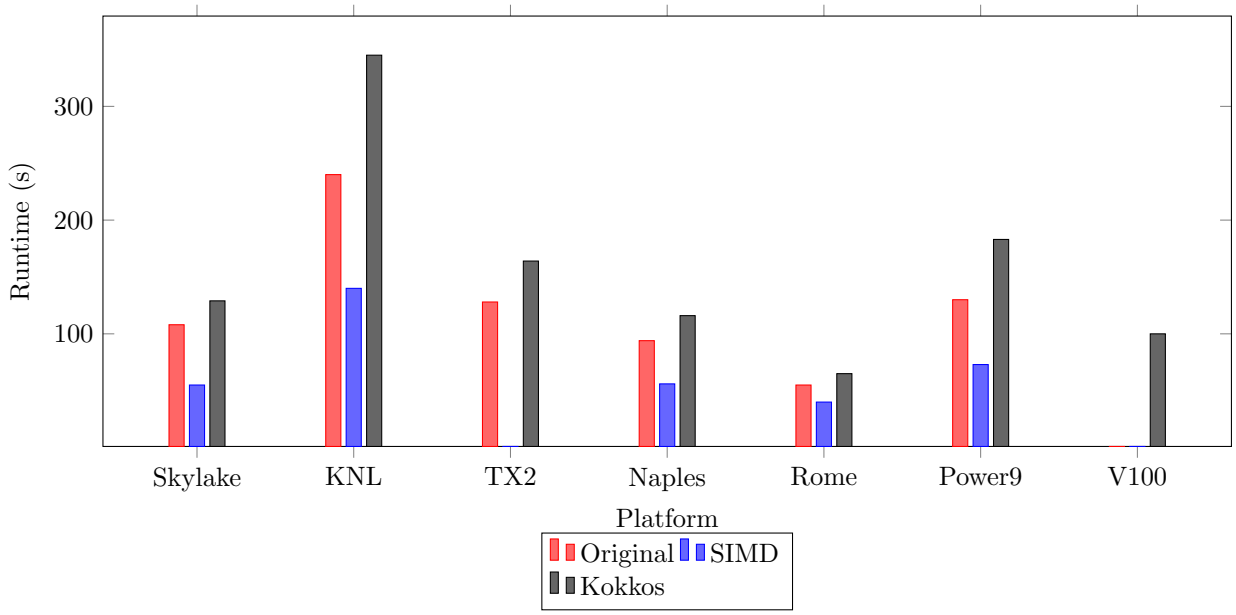


Figure 19: VPIC runtime data from Bird et al. [71]

In Figure 19 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, harming not only the performance portability, but also 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.

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

5.6.2 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. Figures 20 and 21 visualise 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 platforms, its performance portability is 0.

Figure 21 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 [75] has shown that the Kokkos runtime can be improved by up to 55% using Kokkos SIMD[24].

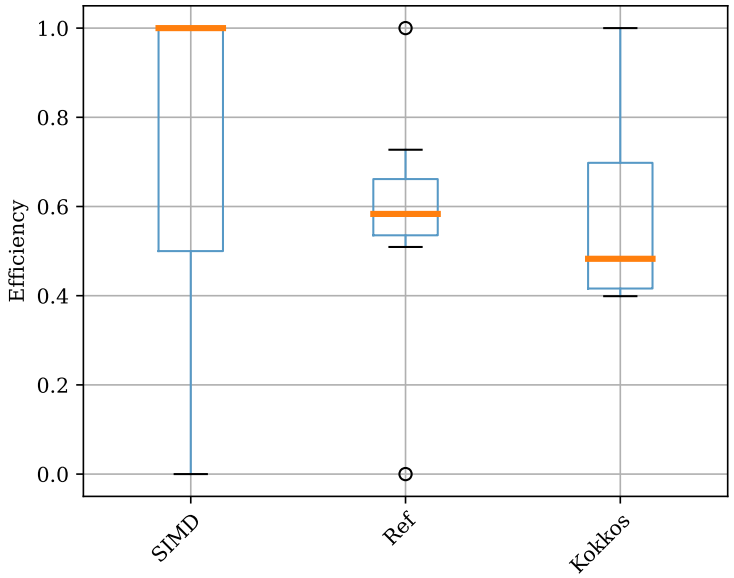


Figure 20: Box plot visualisation of performance portability of VPIC

[24]The data in this report will be updated to reflect this in future iterations.

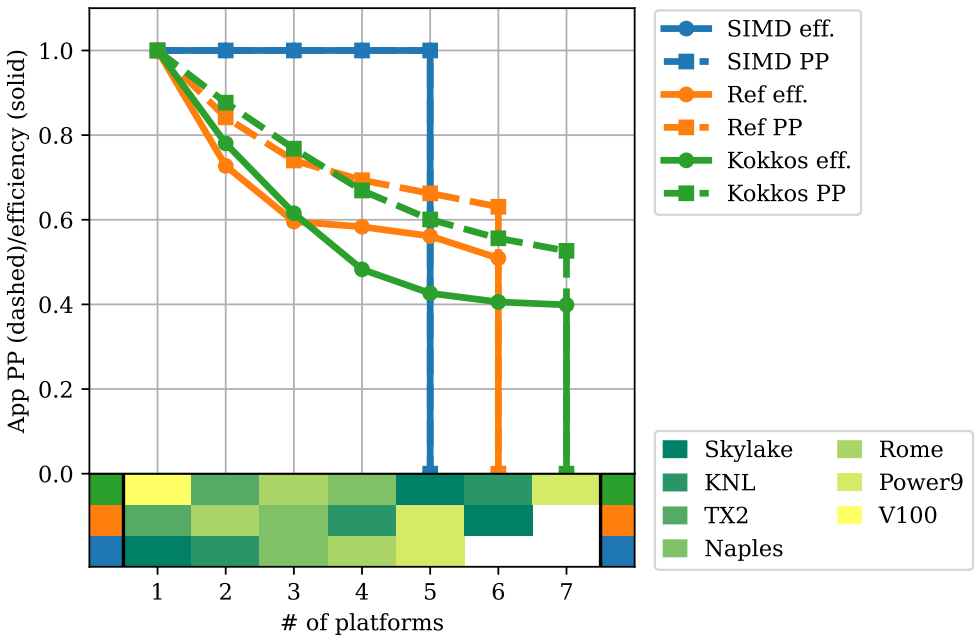


Figure 21: Cascade visualisation of performance portability of VPIC

5.7 EMPIRE-PIC

EMPIRE-PIC is the particle-in-cell solver central the the ElectroMagnetic Plasma In Realistic Environments (EMPIRE) project [72]. 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 [73, 74].

5.7.1 Performance

The EMPIRE-PIC application is export controlled, and thus the results in this section come from the study by Bettencourt et al. [73], looking specifically at the particle kernels within EMPIRE-PIC.

Figure 22 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. [73].

5.7.2 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. [73] to calculate the FLOP/s achieved and compare this to each machines maximum floating-point performance, thus calculating the *architectural efficiency*. The equations

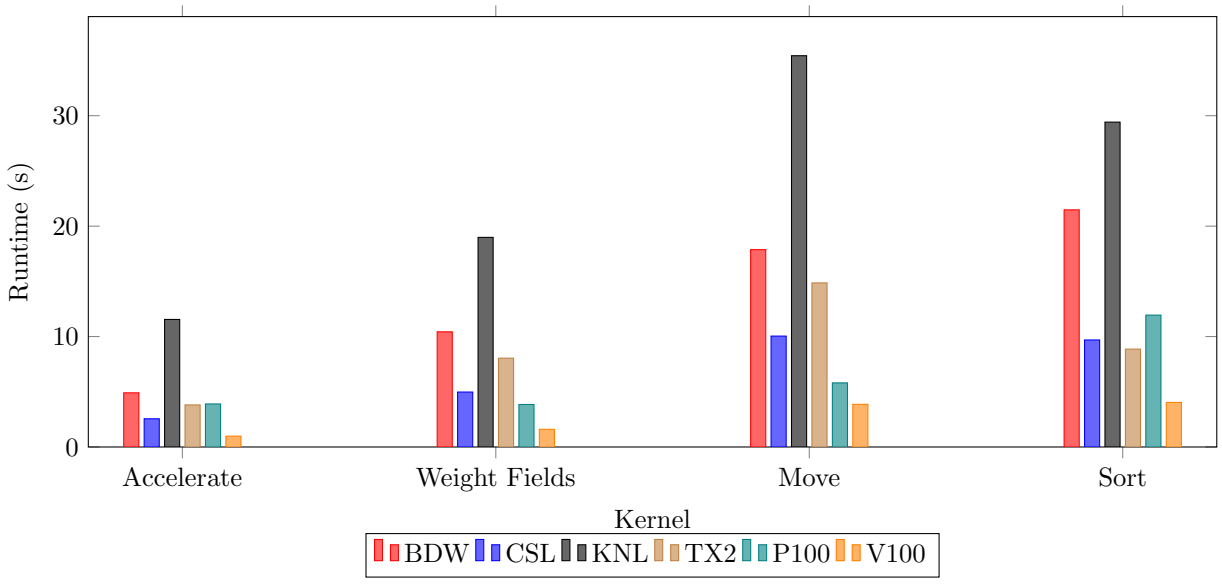


Figure 22: EMPIRE-PIC runtime data

presented assume the best case performance, whereby particles are evenly distributed across the domain, there is no particle migration throughout the simulation, and they are sorted at the start of the simulation. Nevertheless, they provide a useful opportunity to analyse the performance portability of Kokkos for particle-based kernels.

Figures 23 and 24 provide visualisations of EMPIRE-PIC's performance portability across six platforms[25].

It is important to note that although Figure 23 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 Figures 23 and 24 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.

The data suggests that EMPIRE-PIC is not able to fully exploit the on-core parallelism available through vectorisation. Figure 25 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. [73], 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

[25]Please note that the y-axis in each of these Figures has been scaled, since the architectural efficiency is very low.

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 25, 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. [74], 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.

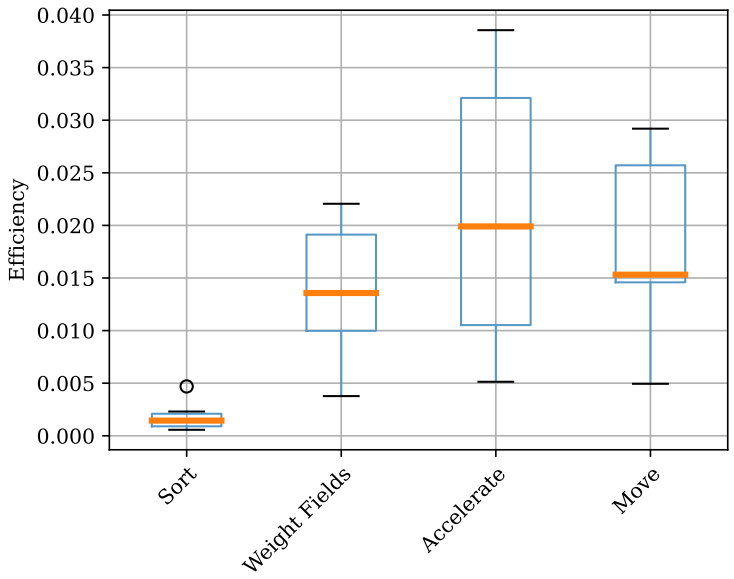


Figure 23: Box plot visualisation of performance portability for four particle kernels in EMPIRE-PIC

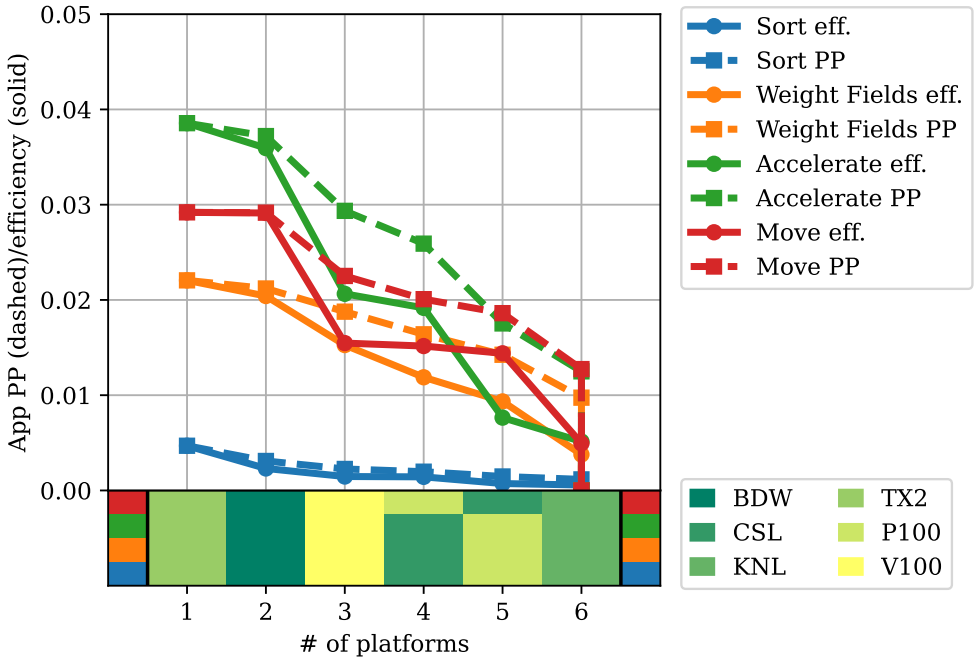
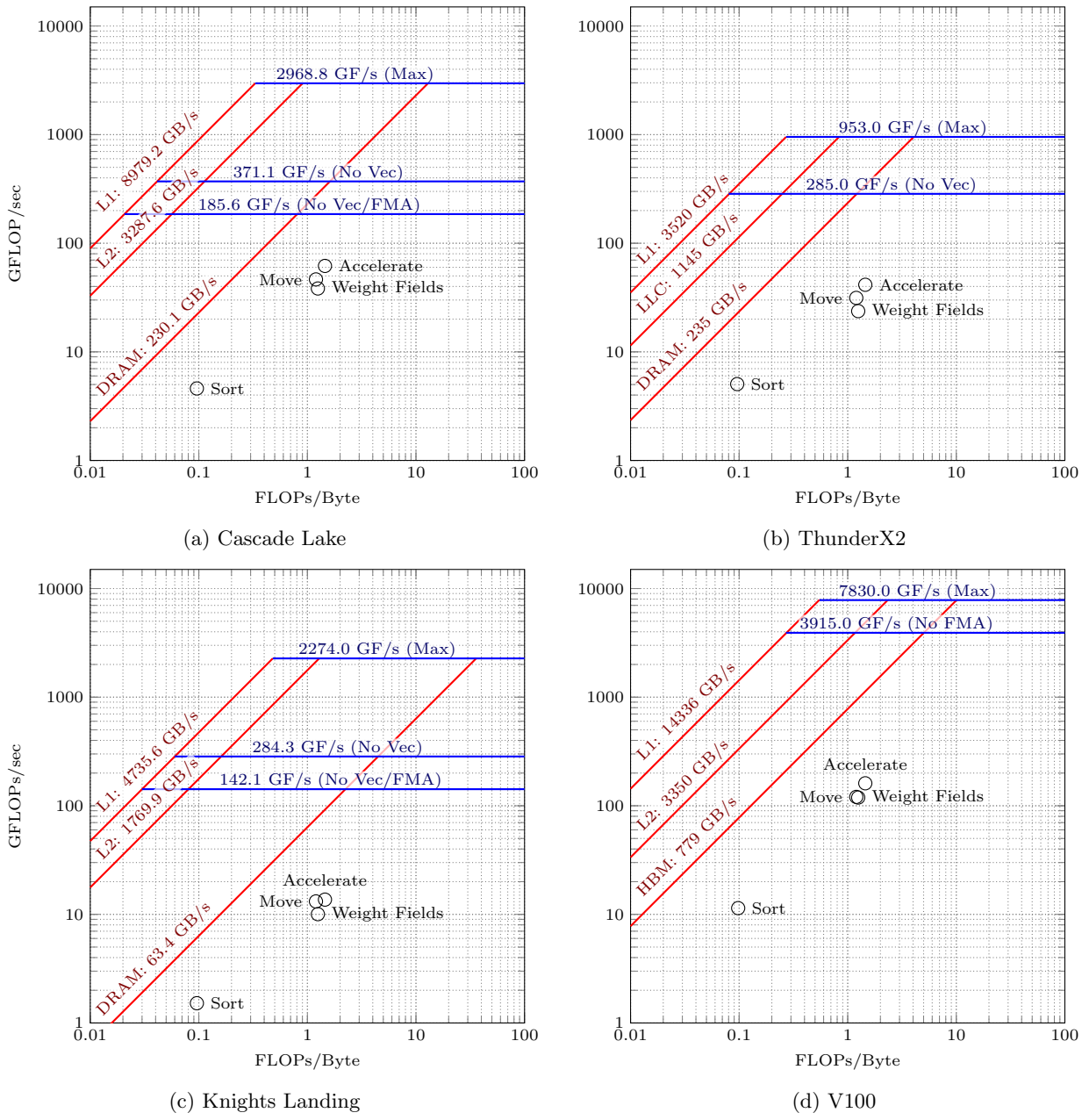


Figure 24: Cascade visualisation of performance portability for four particle kernels in EMPIRE-PIC



(a) Cascade Lake

(b) ThunderX2

(c) Knights Landing

(d) V100

Figure 25: Roofline plots on four platforms, gathered using the Empirical Roofline Toolkit [76]

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 [77, 58, 78, 59]. 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 12 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. Conversely the OpenMP 4.5 implementation does run on all platforms, but provides poor performance on the CPU-only platforms, and significantly lags behind the best performing (CUDA) miniFE variants on the GPU platforms. It should be noted that the OpenMP 4.5 implementation does run on the Rome and TX2 platforms, but the runtime is several orders of magnitude higher than all other variants and are therefore omitted[26].

Figure 14 shows a cascade plot for all miniFE variants, showing that OpenMP offers good portability across the CPU platforms but no portability to accelerator devices. The OpenMP 4.5 variant is portable to all architectures, but is significantly less performant on all platforms. From this data it seems that different parallelisation strategies may be required for high performance between different platforms, and therefore it is likely 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

[26] A runtime for the A64FX platform has not been collected due the lack of support for OpenMP 4.5 in the Cray compiler, but will be collected in due course.

et al. [58]. In Figure 7 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[27]. 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 [77, 58, 78, 59]) have shown that both are typically able to deliver good and portable performance from a single source code base.

The results in Figure 9 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 15 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 19 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[28]. 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

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

[28] 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 [73]. The new SIMD feature in Kokkos should reduce this performance gap significantly [75]

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 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[29].

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 diffusiong code and a miniFE port generated using Intel's DPC++ Compatibility Tool (which converts from CUDA), compiled with the hipSYCL compiler. Figure 5 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 14), the performance portability of SYCL is similar to the performance portability of OpenMP 4.5. At the time these results were gathered, hipSYCL used OpenMP to target CPUs, and so this result is perhaps not surprising. It is quite possible (perhaps even probable) that the newer versions of the hipSYC compiler, and Intel's DPC++ compiler, will yield better performance; collecting this data, and expanding our evaluation platforms to include Intel and AMD GPUs 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 [16]. Figure 26 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 [17]. 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 UHD P630 GPU. Although their study is primarily tracking absolute performance changes with compiler version, rather than comparing to "best

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

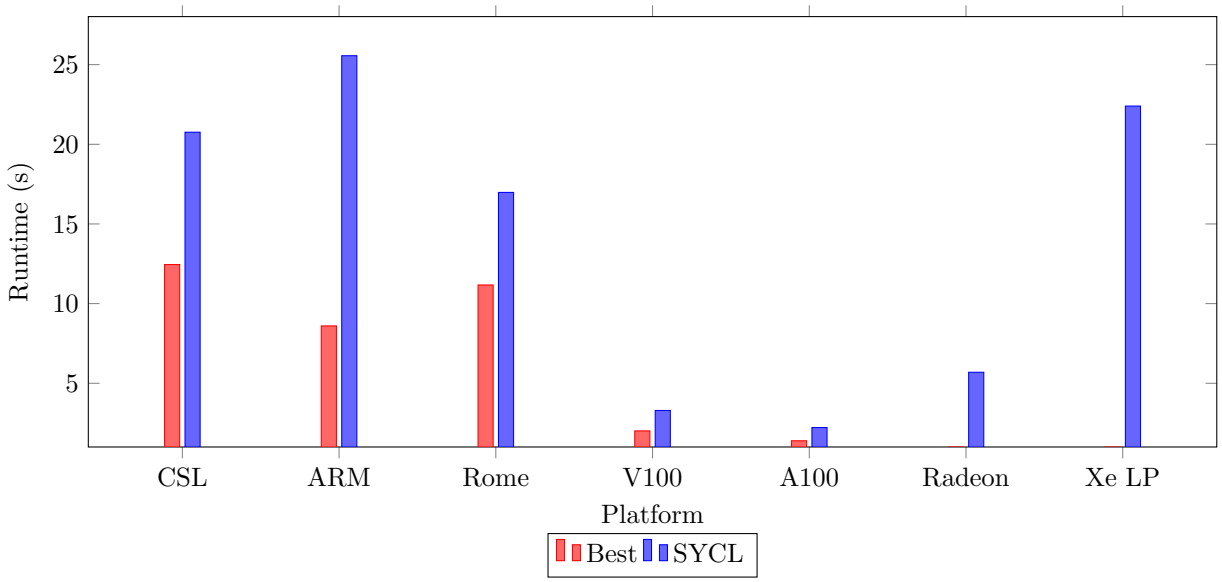


Figure 26: MG-CFD runtime data from Reguly et al. [16]

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.

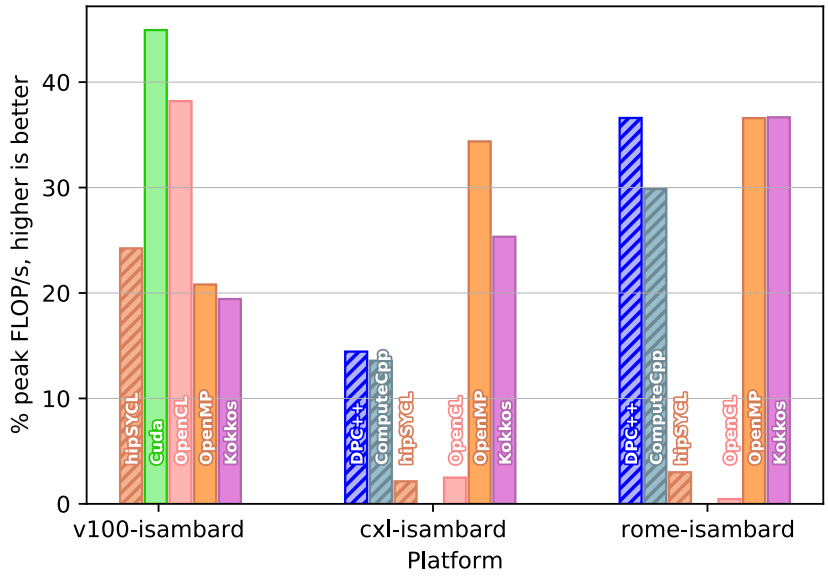


Figure 27: CloverLeaf: SYCL vs. alternative frameworks from Lin et al. [17]

For the two mini-applications the results are more varied; in some cases there are large differences between

the compilers (see Figure 27). 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 9 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*[30].

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 uses 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

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

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 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. In this project, our evaluation has been limited to the hipSYCL compiler, which uses OpenMP to target CPU platforms, and so this is not entirely surprising. With the recent introduction of a new Intel DPC++ compiler, based on LLVM SPIR-V, it is likely performance will improve across many of the platforms benchmarked. 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[31];
- 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.

[31] Although Fortran introduced object orientation in the 2003 standard, it lacks many of the advanced features present in C++ [79].

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 [80]. 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 Fluid and Particle applications to the evaluation set (e.g. HipBone, SheathPIC, etc).
2. Develop an FEM-PIC mini-application to evaluate the coupling of an unstructured mesh and particles in a selection of parallel programming models.
3. Enhance our evaluation of SYCL-based codes.
4. Add new hardware targets, specifically AMD and Intel GPUs.
5. Evaluate the similarity of mini-applications to host codes such as Bout++.

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] Andrew Turner. Parallel Software usage on UK National HPC Facilities 2009-2015: How well have applications kept up with increasingly parallel hardware? Technical report, Edinburgh Parallel Computing Centre, April 2015.

[10] Christopher Rackauckas and Qing Nie. Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software*, 5(1), 2017.

[11] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. Jarvis. Exploring simd for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors. In *Parallel and Distributed Processing Symposium, International*, pages 1085–1097, Los Alamitos, CA, USA, may 2013. IEEE Computer Society.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January–March 1998.

[13] Message Passing Interface Forum. MPI: A Message Passing Interface Standard Version 2.2. *High Performance Computing Applications*, 12(1–2):1–647, 2009.

[14] David Truby, Carlo Bertolli, Steven A. Wright, Gheorghe-Teodor Bercea, Kevin O'Brien, and Stephen A. Jarvis. Pointers inside lambda closure objects in openmp target offload regions. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 10–17, 2018.

[15] Tom Deakin and Simon McIntosh-Smith. Evaluating the Performance of HPC-Style SYCL Applications. In *Proceedings of the International Workshop on OpenCL*, IWOCL'20, New York, NY, USA, 2020. Association for Computing Machinery.

[16] 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.

[17] 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.

[18] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[19] Junchao Zhang et al. The petscsf scalable communication layer. *arXiv*, page 2102.13018, 2021.

[20] Los Alamos National Laboratory. CoPA Cabana - The Exascale Co-Design Center for Particle Applications Toolkit. `https://github.com/ECP-copa/Cabana` (accessed April 20, 2021), 2021.

[21] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[22] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[23] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[24] Hartmut Kaiser, Bryce Adelstein Lelbach, Thomas Heller, Agustín Bergé, Mikael Simberg, John Biddiscombe, Anton Bikineev, Grant Mercer, Andreas Schäfer, Adrian Serio, Taeguk Kwon, Kevin Huck, Jeroen Habraken, Matthew Anderson, Marcin Copik, Steven R. Brandt, Martin Stumpf, Daniel Bourgeois, Denis Blank, Shoshana Jakobovits, Vinay Amatya, Lars Viklund, Zahra Khatami, Devang Bacharwar, Shuangyang Yang, Erik Schnetter, Patrick Diehl, Nikunj Gupta, Bibek Wagle, and Christopher. STEllAR-GROUP/hpx: HPX V1.2.1: The C++ Standards Library for Parallelism and Concurrency, February 2019.

[25] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[26] Rich Hornung, Holger Jones, Jeff Keasler, Rob Neely, Olga Pearce, Si Hammond, Christian Trott, Paul Lin, Courtenay Vaughan, Jeanine Cook, et al. ASC Tri-lab Co-design Level 2 Milestone Report 2015. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.

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

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[29] B. Mostafazadeh, F. Marti, F. Liu, and A. Chandramowlishwaran. Roofline Guided Design and Analysis of a Multi-stencil CFD Solver for Multicore Performance. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 753–762, May 2018.

[30] C. Yount, J. Tobin, A. Breuer, and A. Duran. Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 30–39, Nov 2016.

[31] István Z. Reguly, Gihan R. Mudalige, Michael B. Giles, Dan Curran, and Simon McIntosh-Smith. The OPS Domain Specific Abstraction for Multi-block Structured Grid Computations. In *Proceedings of the 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 58–67, Washington, DC, USA, 2014. IEEE Computer Society.

[32] Sebastian Kuckuk, Gundolf Haase, Diego A. Vasco, and Harald Köstler. Towards generating efficient flow solvers with the ExaStencils approach. *Concurrency and Computation: Practice and Experience*, 29(17):e4062, 2017.

[33] T. Zhao, S. Williams, M. Hall, and H. Johansen. Delivering performance-portable stencil computations on cpus and gpus using bricks. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 59–70, Nov 2018.

[34] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *2012 Innovative Parallel Computing (InPar)*, pages 1–12, May 2012.

[35] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1116–1123. IEEE, 2012.

[36] Pietro Incardona, Antonio Leo, Yaroslav Zaluzhnyi, Rajesh Ramaswamy, and Ivo F. Sbalzarini. OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers. *Computer Physics Communications*, 241:155–177, 2019.

[37] Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas Schulthess. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomputing Frontiers and Innovations*, 1(1), 2014.

[38] PSyclone Project, 2018. `http://psyclone.readthedocs.io/`.

[39] Michael Baldauf, Axel Seifert, Jochen Förstner, Detlev Majewski, Matthias Raschendorfer, and Thorsten Reinhardt. Operational convective-scale numerical weather prediction with the COSMO model: description and sensitivities. *Monthly Weather Review*, 139(12):3887–3905, 2011.

[40] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E. Osuna, Robert Pincus, Jon Rood, and William Sawyer. The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '18, pages 2:1–2:10, New York, NY, USA, 2018. ACM.

[41] V. Clément, P. Marti, O. Fuhrer, and W. Sawyer. Performance portability on GPU and CPU with the ICON global climate model. In *EGU General Assembly Conference Abstracts*, volume 20 of *EGU General Assembly Conference Abstracts*, page 13435, April 2018.

[42] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[43] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, December 2016.

[44] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. ExaStencils: Advanced Stencil-Code Engineering. In Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, pages 553–564, Cham, 2014. Springer International Publishing.

[45] David J. Lusher, Satya P. Jammy, and Neil D. Sandham. Shock-wave/boundary-layer interactions in the automatic source-code generation framework OpenSBLI. *Computers & Fluids*, 173:17–21, 2018.

[46] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman. Devito: Towards a generic finite difference dsl using symbolic python. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 67–75, Nov 2016.

[47] 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.

[48] 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.

[49] Robert D Falgout, Jim E Jones, and Ulrike Meier Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*, pages 267–294. Springer, 2006.

[50] D. Beckingsale, M. Mcfadden, J. Dahm, R. Pankajakshan, and R. Hornung. Umpire: Application-Focused Management and Coordination of Complex Hierarchical Memory. *IBM Journal of Research and Development*, 2019.

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

[63] 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.

[64] 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.

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

[66] 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.

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

[68] 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.

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

[70] 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.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

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

[78] 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.

[79] 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.

[80] 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 28 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 28: OpenMP code listing

A.2 OpenMP Target Directives

An example of the same vector addition seen previously is provided in Figure 29 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 29: OpenMP 4.5 using target directives

A.3 SYCL and DPC++

Figure 30 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 30: SYCL

A.4 Kokkos

Figure 31 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 31: Kokkos

Kokkos also provides fully managed multi-dimensional arrays through its View class. Figure 32 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 32: 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 33.

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

Figure 33: RAJA

Like Kokkos, RAJA provides a view class for handling multi-dimensional arrays. Figure 34 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 34: 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 35.

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

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

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

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

```
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 35: BOUT++ MHD equations implementation

A second eDSL is provided in Bout++ input files. Figure 36 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 36: 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 [32], 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 37.

[32] 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 37: 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 38 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 38: 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 39 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 [71]. 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 40). 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 39: 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 40: Using Kokkos to convert AoS to SoA