T/NA086/20
Code structure and coordination

Report 2047358-TN-03

Evaluation of Approaches to Performance Portability

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

University of York

Gihan Mudalige

University of Warwick

December 8, 2021

Contents

1 Introduction

The focus of the *code structure and coordination* work package is to establish a series of “best practices” on how to develop simulation applications for Exascale systems that are able to obtain high performance on each architecture (i.e. are performance portable) without significant manual porting efforts.

In the past decade, a large number of approaches to developing performance portable code have been developed. In this report we will begin to report on our evaluation of some of these approaches through the execution of a small number of mini-applications that implement methods similar to those likely to be required in NEPTUNE.

These applications are detailed in report 2047358-TN-02, but are summarised below for convenience:

TeaLeaf

A finite difference mini-app that solves the linear heat conduction equation on a regular grid using a 5-point stencil[1].

miniFE

A finite element mini-app, and part of the Mantevo benchmark suite[2].

Laghos

A high-order curvilinear finite element scheme on an unstructured mesh[3].

CabanaPIC

A structured PIC code built using the CoPA Cabana library for particle-based simulations[4].

VPIC/VPIC 2.0

A general purpose PIC code for modelling kinetic plasmas in one, two or three dimensions, developed at Los Alamos National Laboratory[5].

EMPIRE-PIC

An unstructured PIC code that uses the finite-element method.

[1] `http://uk-mac.github.io/TeaLeaf/`
[2] `https://github.com/Mantevo/miniFE`
[3] `https://github.com/CEED/Laghos`
[4] `https://github.com/ECP-copa/CabanaPIC`
[5] `https://github.com/lanl/vpic`

The selected applications broadly represent the algorithms of interest for the NEPTUNE project and fall in to two categories – fluid-methods and particle-methods. Within the fluid-method tranche, the applications are available implemented in a wide range of programming models, allowing us a good opportunity to evaluate the effect of programming model on the performance, and importantly the *performance portability* of that particular approach to application development. There are a relatively small number of particle-in-cell mini-applications available, and thus the selected particle-methods applications are only available implemented using Kokkos. However, this still allows us an opportunity to evaluate the appropriateness of Kokkos as a programming model for performance portable application development.

1.1 Method of Evaluation

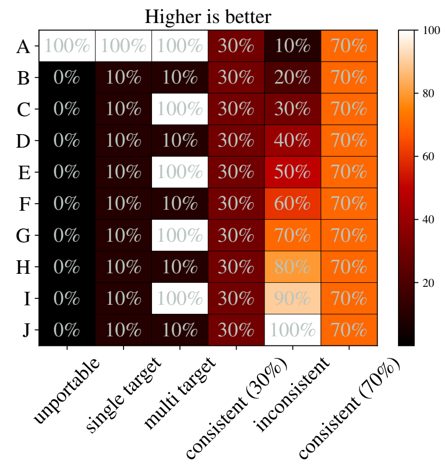
As stated previously, we will evaluate the performance portability of these applications using the metric introduced by Pennycook et al. [1], and use the visualisation techniques outlined by Sewall et al. [2]. The Pennycook metric allows us to calculate the *performance portability* of an application according to Equation (1).

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

In the equation, the performance portability (Φ) 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. [2]. These visualisations are best explained with an example.

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.

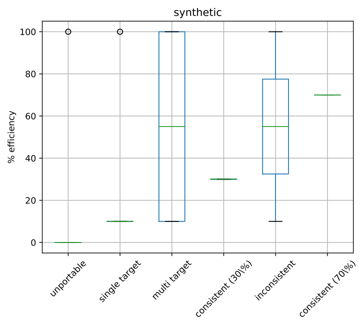


	unportable	single target	multi target	consistent (30%)	inconsistent	consistent (70%)
A	100%	100%	100%	30%	10%	70%
B	0%	10%	10%	30%	20%	70%
C	0%	10%	100%	30%	30%	70%
D	0%	10%	10%	30%	40%	70%
E	0%	10%	100%	30%	50%	70%
F	0%	10%	10%	30%	60%	70%
G	0%	10%	100%	30%	70%	70%
H	0%	10%	10%	30%	80%	70%
I	0%	10%	100%	30%	90%	70%
J	0%	10%	10%	30%	100%	70%

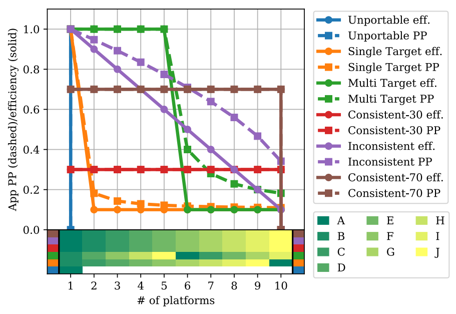
Figure 1: Synthetic data set for six implementations running across 10 platforms taken from Sewall et al. [2]

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. [2].



(a) Box plot



(b) Cascade plot

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

Where possible, performance data has been taken from previously published works. Where no data exists, the data has been collected from the UK's Tier-2 platforms, in particular Isambard's Multi-Architecture Comparison System (MACS), ThunderX2 system and A64FX system.

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 all the time and analysed, 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.

2 Application Evaluations

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++ [3]) to high-order finite element or spectral element (like Nektar++ [4]). Similarly, the particle-methods mini-apps all implement the particle-in-cell method (like EPOCH [5]).

2.1 TeaLeaf

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

It has been used extensively in studying performance portability already [6, 7, 8, 9], and is available implemented using CUDA, OpenACC, OPS, RAJA, and Kokkos, among others[6]. The results in this section are extracted from two of these studies, namely one by Kirk et al. [7] and one by Deakin et al. [6].

In both studies, the largest test problem size (`tea_bm_5.in`) is used, a 4000×4000 grid.

2.1.1 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 3[7]. Note that in the study, some results are missing due to incompatibility (e.g. CUDA on Broadwell/KNL).

[6]`http://uk-mac.github.io/TeaLeaf/`

[7]*Hybrid* represents the best performing configuration of a MPI/OpenMP hybrid execution

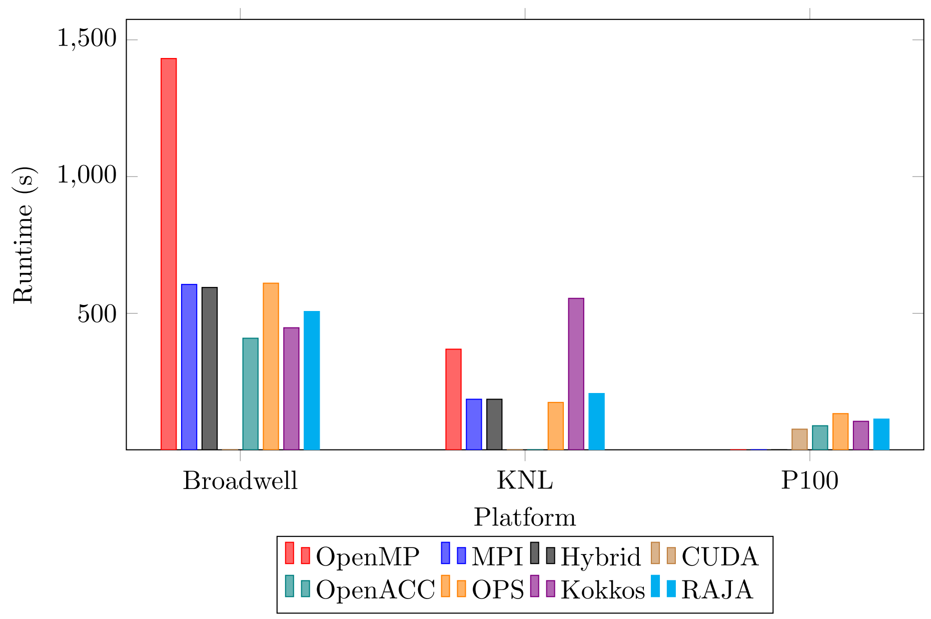


Figure 3: TeaLeaf runtime data from Kirk et al. [7]

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 4.

2.1.2 Portability

Both studies evaluate some portable and 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. [7], Figures 5 and 6 allow us to visualise the performance portability of each approach to application development. Figure 5 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

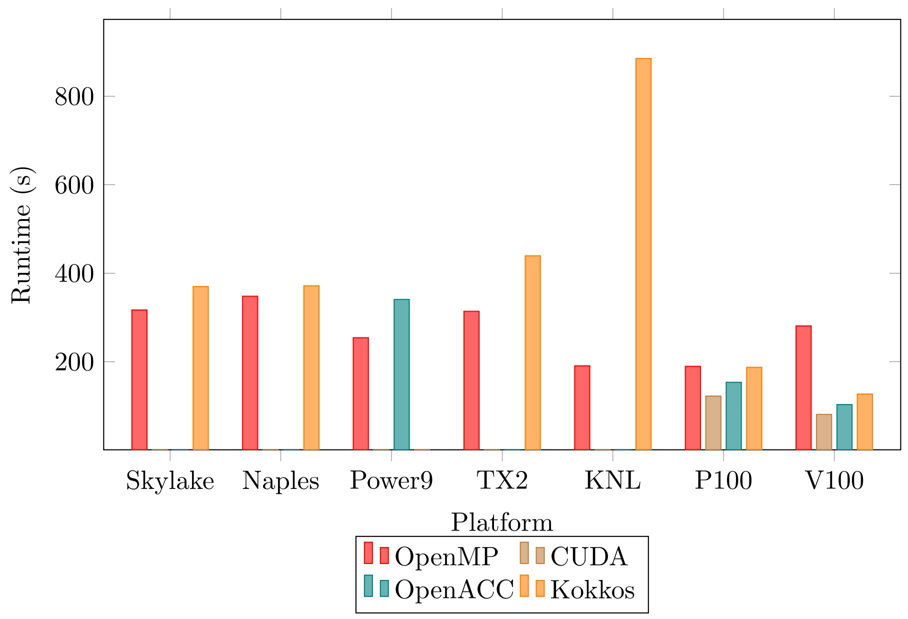


Figure 4: TeaLeaf runtime data from Deakin et al. [6]

a much smaller range of efficiencies.

The cascade plot in Figure 6 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 3, 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 7 and 8 show the same visualisations for the data from Deakin et al. [6]. 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

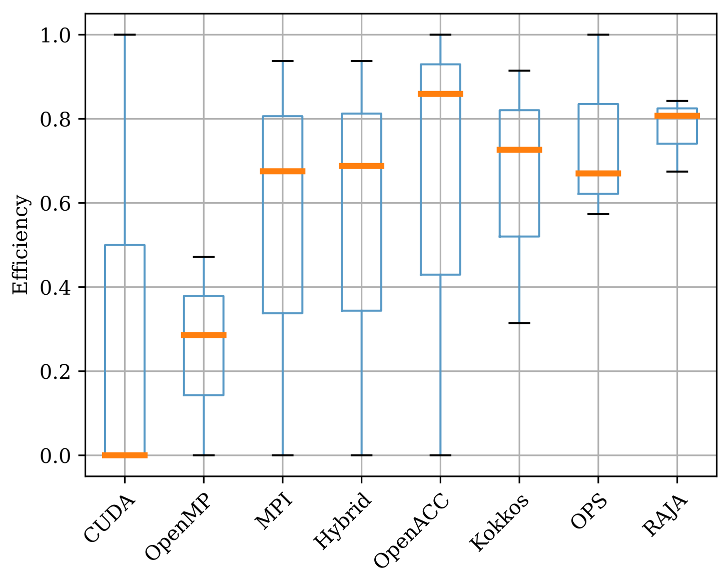


Figure 5: Box plot visualisation of performance portability from Kirk et al. [7]

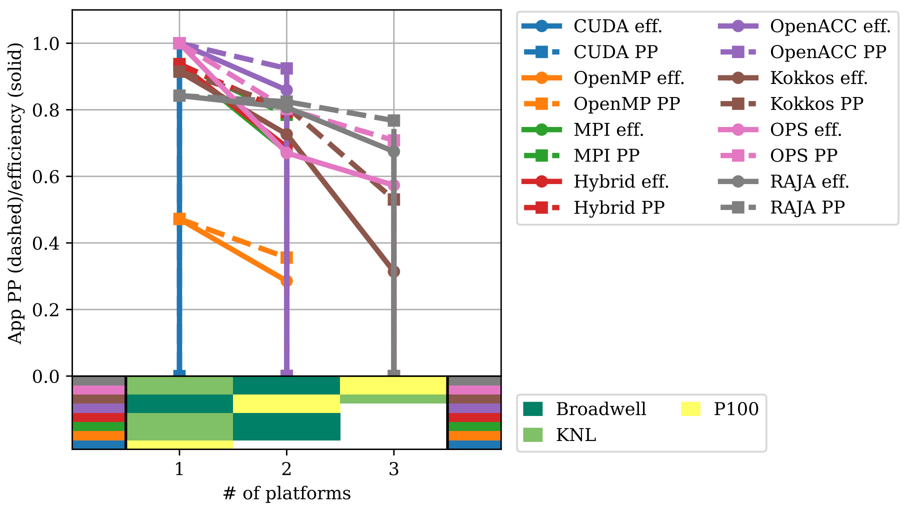


Figure 6: Cascade visualisation of performance portability from Kirk et al. [7]

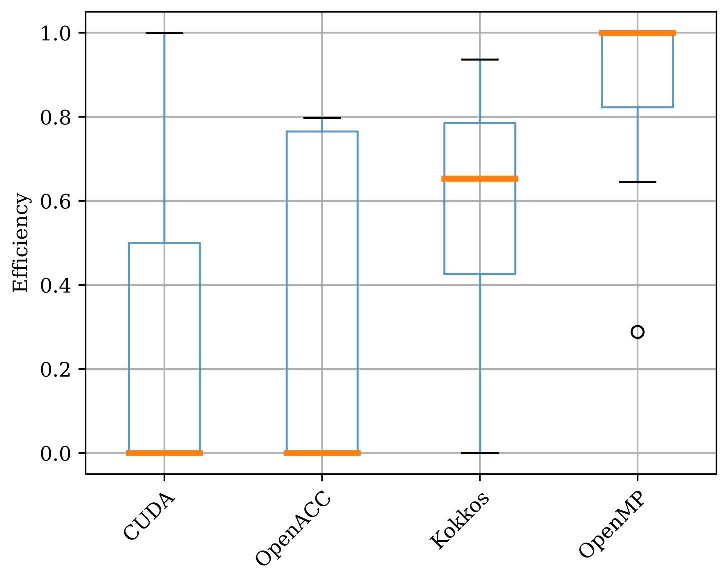


Figure 7: Box plot visualisation of performance portability from Deakin et al. [6]

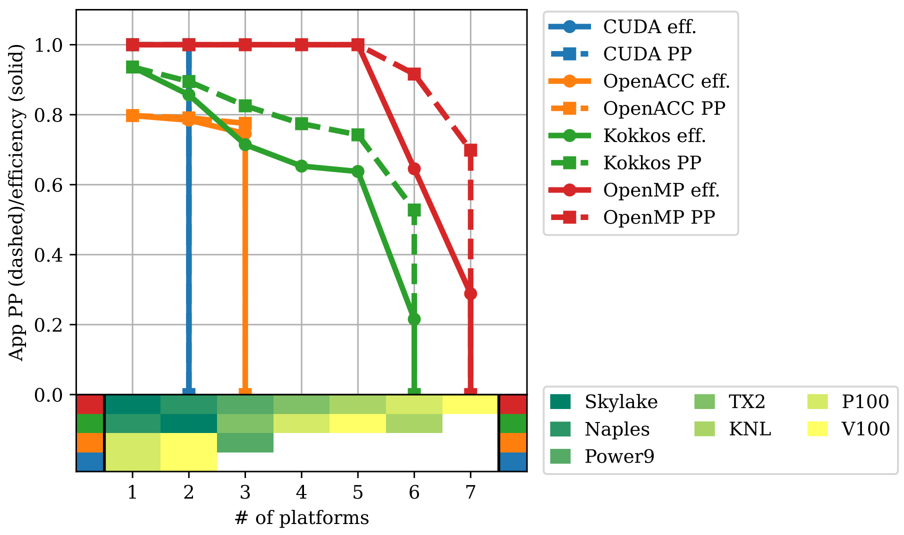


Figure 8: Cascade visualisation of performance portability from Deakin et al. [6]

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.

2.2 miniFE

miniFE is a finite element mini-app, and part of the Mantevo benchmark suite [10, 11, 12, 13]. It implements an unstructured implicit finite element method and has versions available in CUDA, Kokkos, OpenMP (3.0+ and 4.5+) and SYCL[8].

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.

2.2.1 Performance

The raw runtime results for these runs can be seen in Figure 9. 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[9]; 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 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 9 shows that SYCL performance on the KNL and Rome platforms is far

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

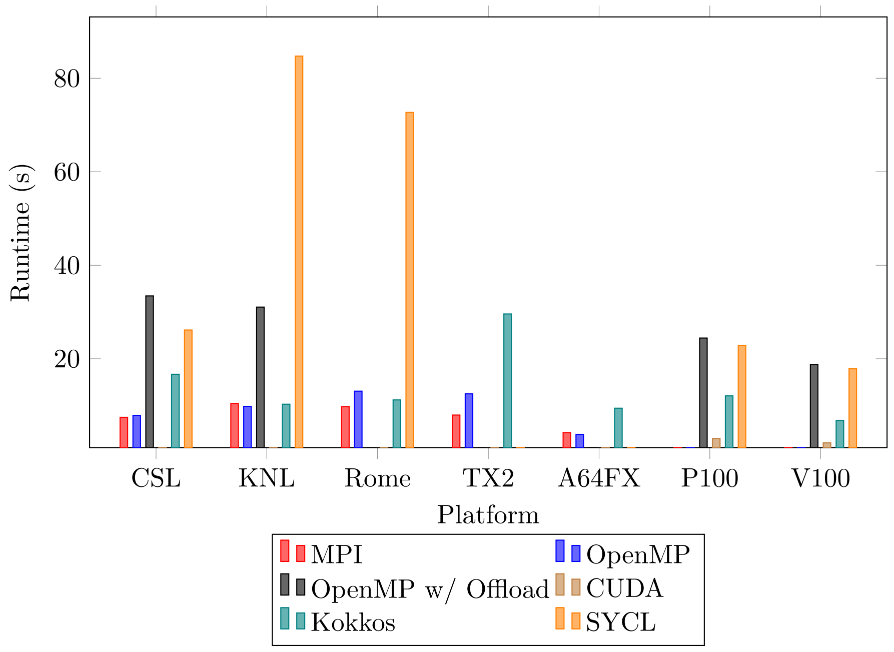


Figure 9: miniFE runtime data

in excess of any other execution (with the exception of OpenMP w/ Offload on Rome which is not shown), and on the GPU platforms the SYCL runtime is on par with OpenMP w/ offload. This is likely due to the hipSYCL compiler generating OpenMP w/ 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.

2.2.2 Portability

Figures 10 and 11 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 10 shows that CUDA has the worst lowest median performance, because

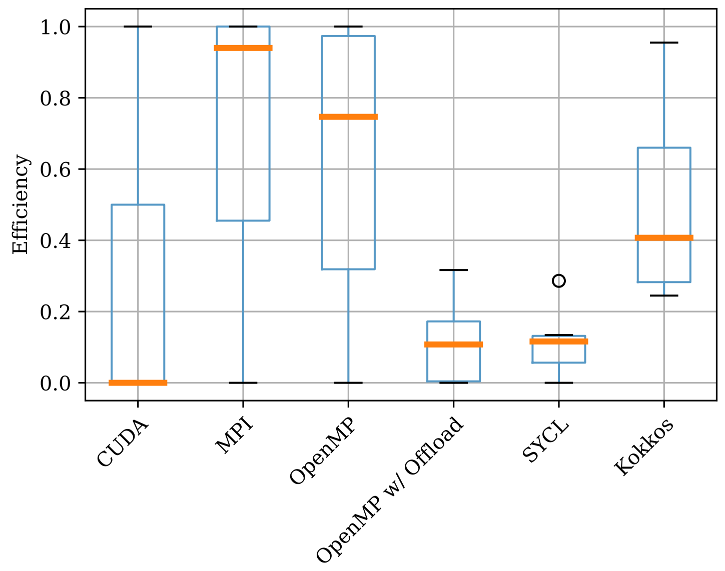


Figure 10: Box plot visualisation of performance portability of miniFE

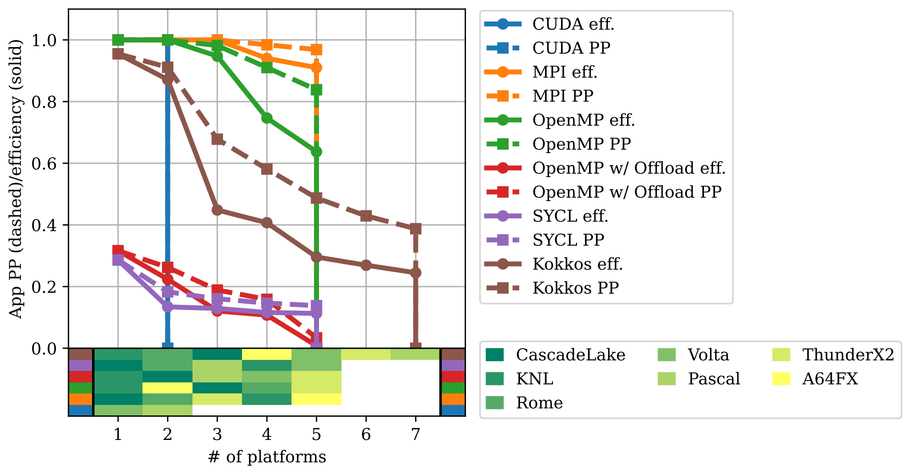


Figure 11: Cascade visualisation of performance portability of miniFE

it only runs on the two GPU systems, but is the best performing on each. The boxplots for both OpenMP w/ Offload and SYCL are very similar, and this is likely an artefact of SYCL being translated to OpenMP w/ 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 sacraficing a significant proportion of performance.

Figure 11 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 w/ 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. 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.

2.3 Laghos

Laghos is a mini-app that is part of the ECP Proxy Applications suite [14, 15, 13]. 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[10].

The results presented below have all been collected from the Isambard platform.

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

2.3.1 Performance

Figure 12 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.

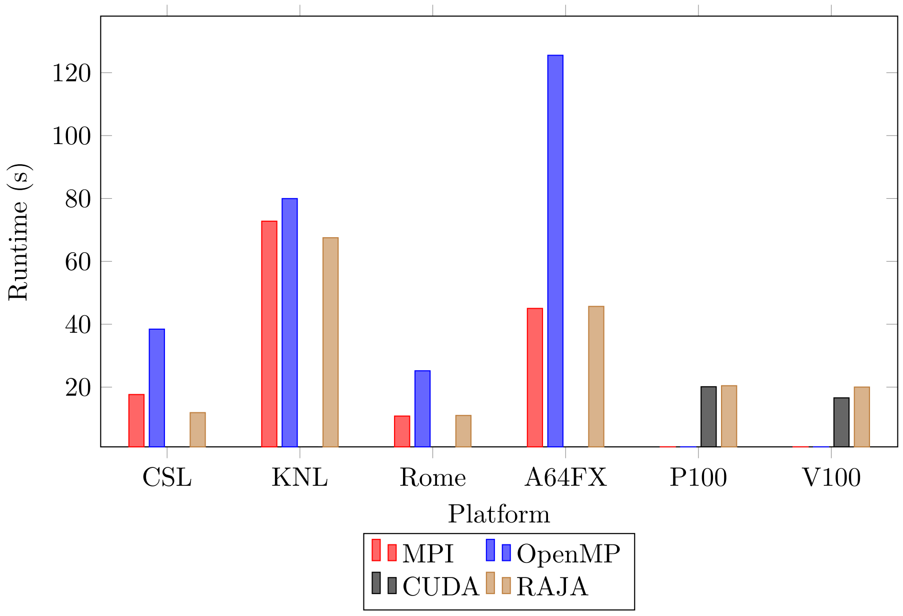


Figure 12: Laghos runtime data

2.3.2 Portability

Portability visualisations of each implementation of Laghos are provided in Figures 13 and 14.

Figure 13 demonstrates the remarkable efficiency of the RAJA MFEM and HYPRE implementations, showing consistently above 80% performance effi-

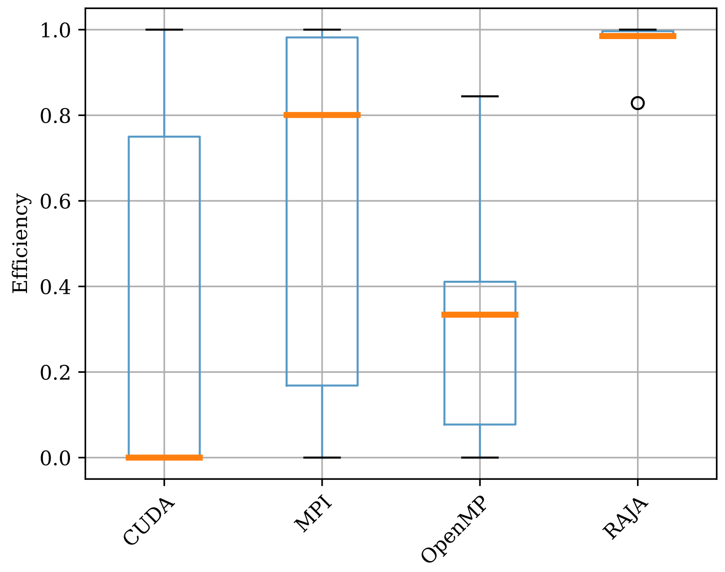


Figure 13: Box plot visualisation of performance portability of Laghos

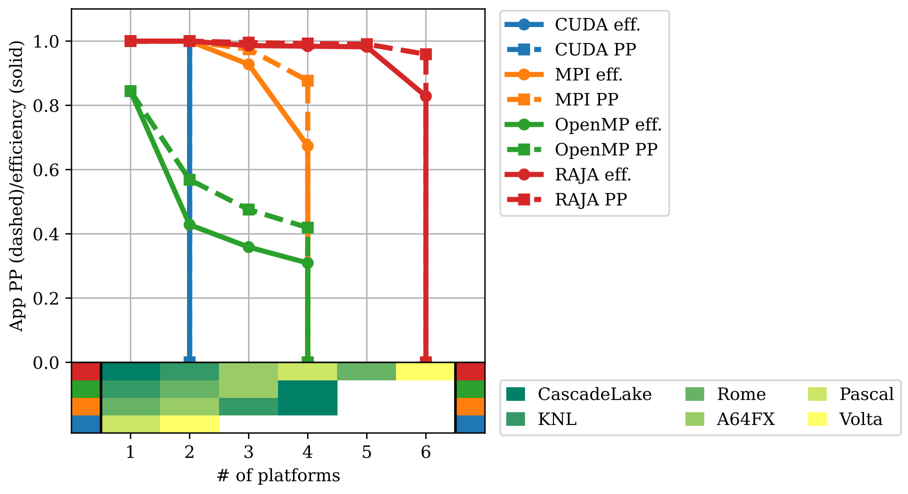


Figure 14: Cascade visualisation of performance portability of Laghos

ciency. 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.

2.4 CabanaPIC

CabanaPIC is a structured PIC demonstrator application built using the CoPA/Cabana library for particle-based simulations [13]. 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[11].

2.4.1 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 15 shows the achieved runtime for CabanaPIC across four of Isambard's platforms, running a simple 1D 2-stream problem with 6.4 million particles.

Approximately equivalent performance can be seen on the CascadeLake, 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.

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

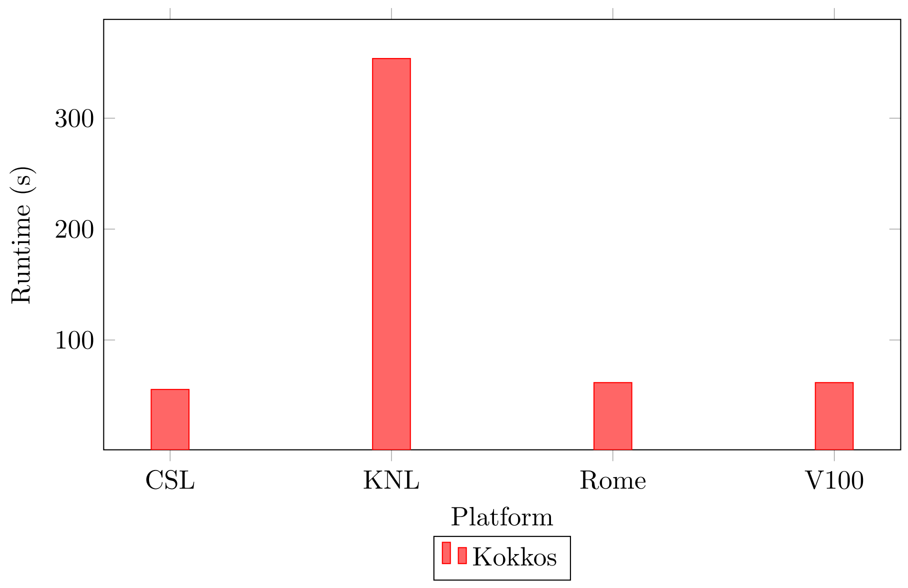


Figure 15: CabanaPIC data

2.5 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 [16]. 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[12].

Recently, VPIC 2.0 [17] has been developed that adds support for heterogeneity by using Kokkos to optimise the data layout and allow execution on accelerator devices.

2.5.1 Performance

Figure 16 shows the runtime for the three variants of the VPIC code running on seven platforms[13]. 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,

[12]`https://github.com/lanl/vpic`
[13]`https://globalcomputing.group/assets/pdf/sc19/SC19_flier_VPIC.pptx.pdf`

the runtime is the time taken for 500 time steps, with 66 millions particles.

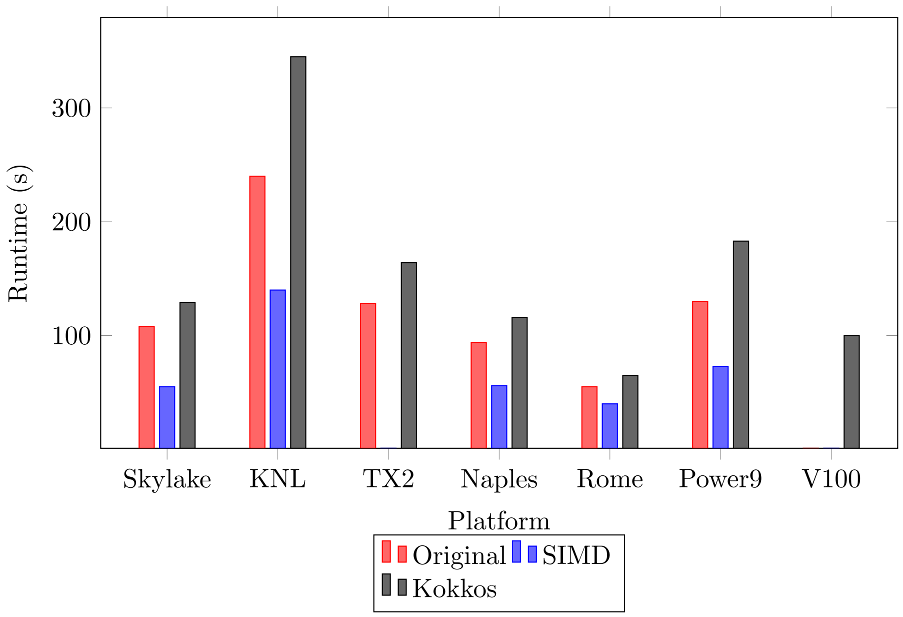


Figure 16: VPIC runtime data from Bird et al. [17]

In Figure 16 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.

2.5.2 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 17 and 18 visualise how the performance portability varies as more platforms are evaluated.

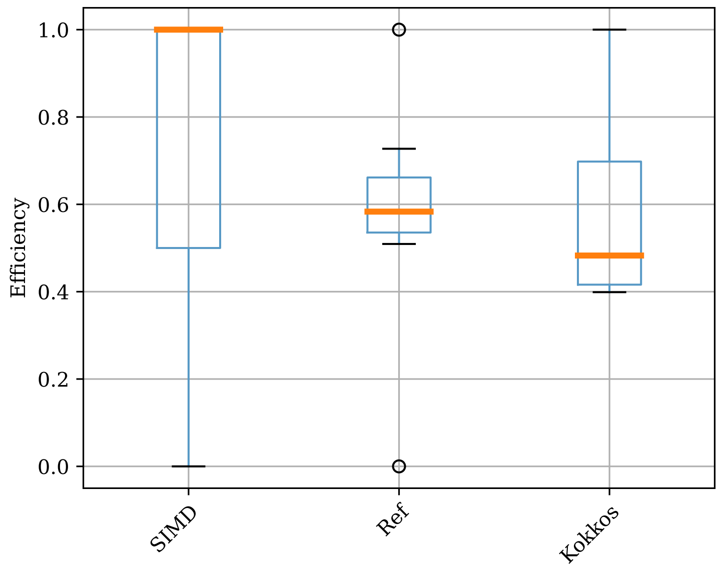


Figure 17: Box plot visualisation of performance portability of VPIC

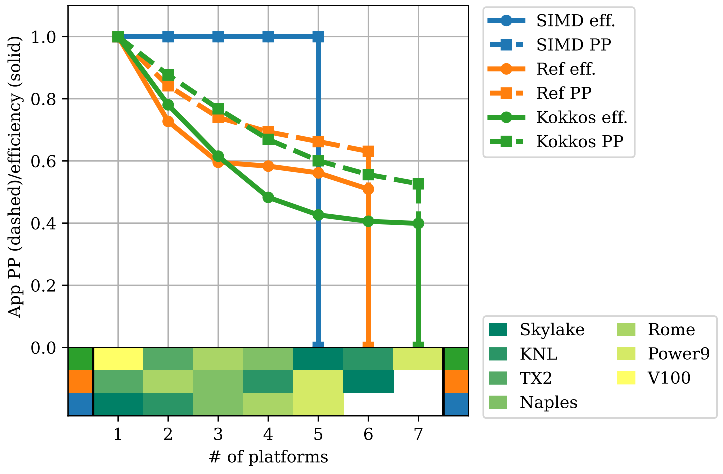


Figure 18: Cascade visualisation of performance portability of VPIC

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, Figure 17, when evaluating the entire set of platforms, its performance portability would be 0, due to non-execution on the V100 platform.

Figure 18 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.

2.6 EMPIRE-PIC

EMPIRE-PIC is the particle-in-cell solver central the the ElectroMagnetic Plasma In Realistic Environments (EMPIRE) project [18]. 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 [19, 20].

2.6.1 Performance

The EMPIRE-PIC application is export controlled, and thus the results in this section come from the study by Bettencourt et al. [19], looking specifically at the particle kernels within EMPIRE-PIC.

Figure 19 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. [19].

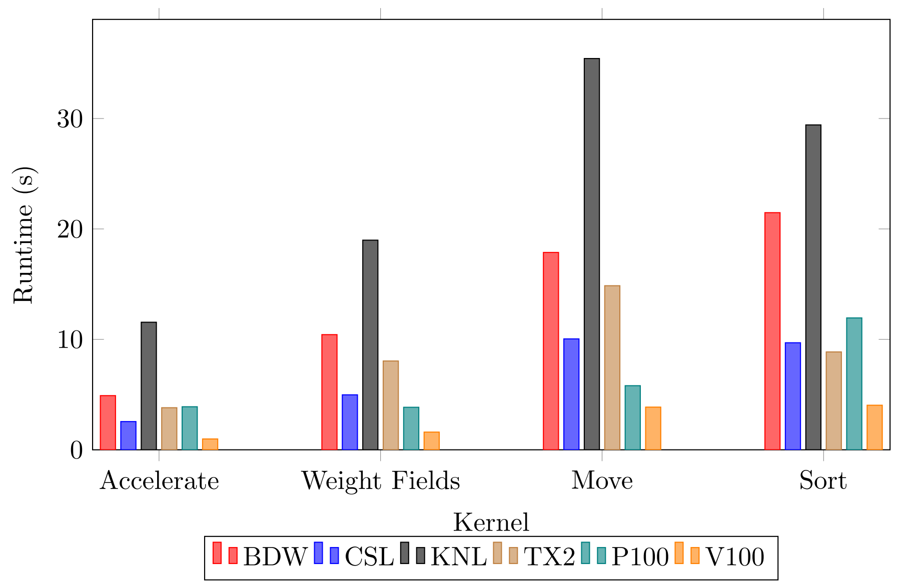


Figure 19: EMPIRE-PIC runtime data

2.6.2 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. [19] to calculate the FLOP/s achieved and compare this to each machines maximum floating-point performance, thus calculating the *architectural efficiency*. The equations presented assume the best case performance, 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 20 and 21 provide visualisations of EMPIRE-PIC's performance portability across six platforms[14].

It is important to note that although Figure 20 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

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

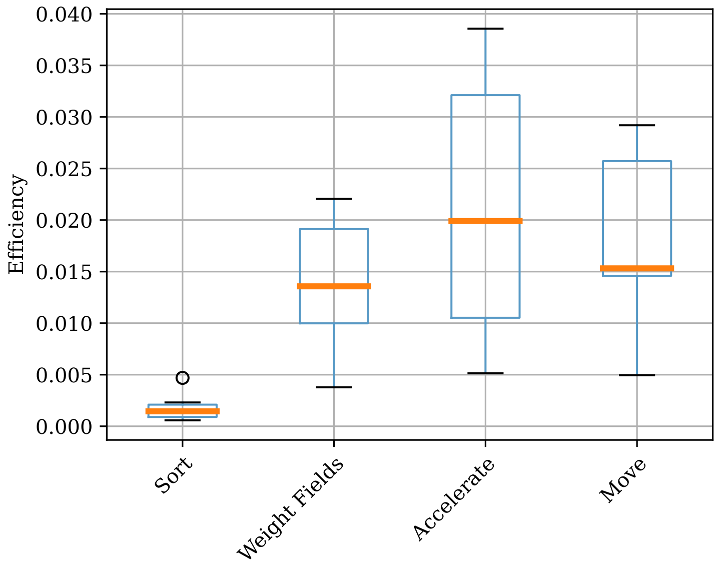


Figure 20: Box plot visualisation of performance portability for four particle kernels in EMPIRE-PIC

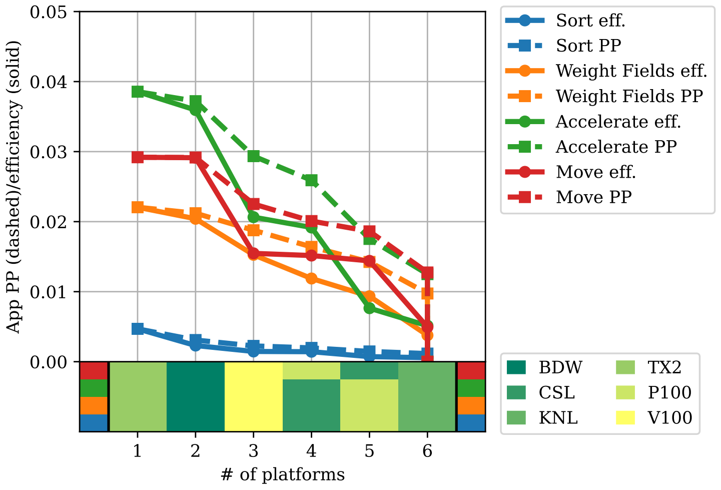


Figure 21: Cascade visualisation of performance portability for four particle kernels in EMPIRE-PIC

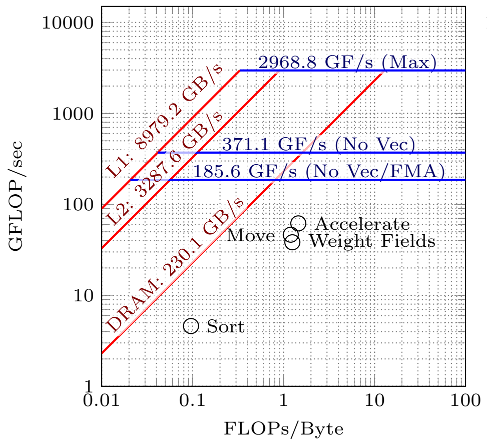
10% of this peak performance is not unusual for a real application. In the case of the Sort kernel, the efficiency is lower still, as this is not a kernel that is bound by floating point performance.

What is clear from Figures 20 and 21 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.

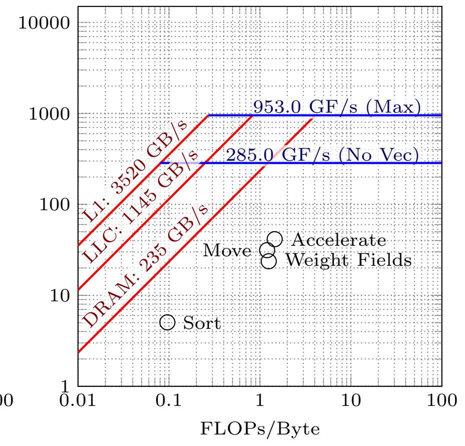
The data suggests that EMPIRE-PIC is not able to fully exploit the on-core parallelism available through vectorisation. Figure 22 shows roofline models for four of these platforms, with the four particle kernels plotted according to their arithmetic intensity and achieved FLOP/s.

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. [19], the control flow required to handle particles crossing element boundaries leads to warp divergence on GPUs and makes achieving vectorisation difficult on CPUs. Nonetheless, on the Cascade Lake and ThunderX2 platforms, we are within an order of magnitude of the non-vectorised peak performance for the three main kernels, and the sort kernel (with low arithmetic intensity) is heavily affected by main memory bandwidth. For the two many-core architectures (KNL and V100), floating-point performance is further from the peak, and the performance of each kernel is further hindered by the DRAM/HBM bandwidth.

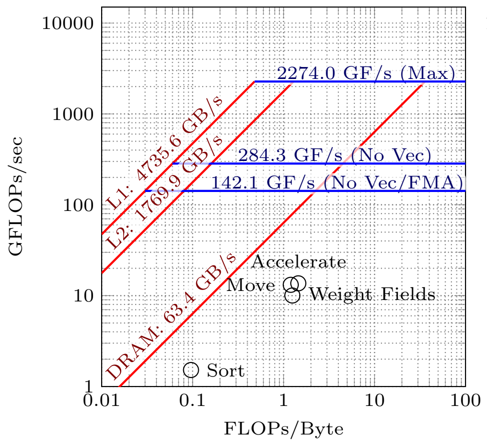
Roofline analyses, like Figure 22, are effective at demonstrating how vital to performance it is to balance efficient memory accesses with arithmetic intensity. This is especially important in PIC codes, where some of the kernels are relatively low in arithmetic intensity when compared to the amount 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. [20], 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.



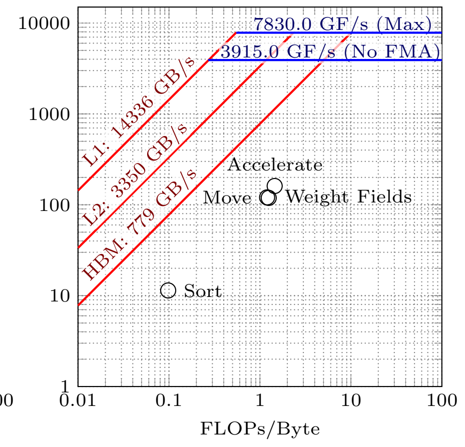
(a) Cascade Lake



(b) ThunderX2



(c) Knights Landing



(d) V100

Figure 22: Roofline plots on four platforms, gathered using the Empirical Roofline Toolkit [21]

3 Conclusions

This report serves as a living document of the performance of applications that implement algorithms of interest to the NEPTUNE project. For each of the applications in this report, there are typically a number of alternative implementations, solving the same algorithm but using a different parallel programming model. This allows us an opportunity to assess these programming models and their appropriateness for the NEPTUNE project, with the goal of creating a set of best practices to developing plasma physics applications that are both *performant* and *portable*.

The results presented in the previous section show that in many cases, OpenMP and/or MPI provide the best performance on CPU platforms, while CUDA typically provides the best performance on NVIDIA GPUs. However, these programming models significantly affect the portability of these applications, with the former unable to use accelerators, and the latter unable to use host platforms. Developing an application that can exploit all available parallelism that is likely to be present on post-Exascale systems would therefore require developers to maintain multiple implementations of a code – potentially one for each class/generation of host or accelerator platforms.

For fluid codes, there are a number of domain specific languages (DSLs) that provide abstractions for grid-based algorithms. OPS is one such DSL targeted at structured mesh applications, and capable of code generation targeting MPI, OpenMP, OpenACC, CUDA and HIP. Our study with TeaLeaf shows that it is able to provide performance that in many cases is on par with native OpenMP and MPI, and within 2× native CUDA performance on a P100. However, such DSLs often reduce the flexibility afforded to a developer.

Besides code generation from a higher-level abstraction, GPUs can be targeted using pragma-based language extensions such as OpenMP 4.5 and OpenACC. Both offer similar functionality, but only OpenMP 4.5 allows portability between accelerator and non-accelerator platforms. However, our evaluation has shown that although OpenMP 4.5 allows us to target GPUs, different pragmas are often required to achieve sufficient performance on accelerators when compared to host systems, meaning that multiple implementations would likely need to be maintained. This is well demonstrated by our miniFE results, where the

OpenMP with offload code does successfully execute on the CPU architectures but offers significantly worse performance than OpenMP itself.

The template libraries, Kokkos and RAJA are both capable of providing full portability across all architectures, and in most cases offer good performance. The significant exception from our results is for the Intel Knights Landing platform, where Kokkos performance is typically poor. This performance gap is likely the result of a bug or memory configuration issue, but will not be investigated further due to the discontinuation of the KNL architecture. Regardless, where we are able to compare Kokkos or RAJA to a native programming model, they are typically able to achieve a runtime that is no more than 20% greater than the native programming model on CPUs and no more than 50% greater than the native programming model on GPUs, but from a single code base.

Another approach that is gaining traction is that of SYCL/DPC++. In our current benchmark set, only a single application is available implemented in SYCL (miniFE), and that implementation has been generated using Intel's DPC++ Compatibility Toolkit. The resulting application is portable across platforms but in most cases has performance that is only slightly better than the available OpenMP 4.5 implementation. This warrants additional exploration to account for this performance difference; for such an immature programming model, it is likely that choice of compiler, and some very simple optimisations will bring performance more in line with other approaches to portability. As this project progresses, hopefully more applications will be available for evaluation, and compiler support will evolve.

For the particle methods tranche of applications, they are predominantly available using Kokkos as a parallel programming model. This does allow portable execution across all available platforms, but makes it difficult to compare performance against native implementations. In the case of VPIC, we can see that Kokkos provides performance that is in line with the original, unvectorised implementation on all platforms, and allows us to extend our platform set to include GPU devices. However, the greatest performance comes from using non-portable vector intrinsics, which in this case means maintaining an implementation for each set of vector instructions (i.e. SSE, AVX, AVX-2, Altivec, etc.).

3.1 Limitations

The work presented in this report represents our initial evaluation of approaches to performance portability. We intend that this document is continually updated as new data becomes available, and as applications and implementations are developed. Currently, the data in this report contains a few limitations that we aim to rectify in future.

Firstly, due to its immaturity relative to other approaches, there are a lack of relevant fluid and particle-in-cell applications available that use the SYCL/DPC++ programming model. This means that with the exception of miniFE, it is difficult to assess its appropriateness as an approach to performance portable application development. A recent study by Reguly et al. has shown that for a computational fluid dynamic application SYCL may be able to achieve comparable performance, though this may require different code paths for different hardware [22].

Secondly, the PIC codes assessed in this report all use the Kokkos programming model. Again, this limits our ability to reason about the appropriateness of this approach for PIC codes, but we can use the VPIC data to show that while we cannot match native, hand-vectorised performance, it can provide performance that is similar to the original implementation, and can be extended to heterogeneous architectures.

Finally, we have not currently evaluated performance on any AMD Radeon Instinct or Intel Xe hardware, due to availability of test platforms. We aim to add these platforms in the near future, when available, either through the COSMA8 system at Durham University, or through Amazon EC2 instances.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] Istvan Z. Reguly, Andrew M. B. Owenson, Archie Powell, Stephen A. Jarvis, and Gihan 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, *High Performance Computing*, pages 391–410. Springer International Publishing, 2021.