

T/NA086/20
Code structure and coordination

Report 2047358-TN-04

*Approaches to Scientific Software
Development at Exascale*

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

University of York

Gihan Mudalige

University of Warwick

November 16, 2021

Contents

1	Executive Summary	3
2	Hardware Targets	5
2.1	Exascale Efforts	5
2.1.1	The United Kingdom	5
2.1.2	Europe	6
2.1.3	United States	7
2.2	Hardware Overview	8
2.3	Summary	10
3	Developing Performance Portable Software	11
3.1	OpenMP	11
3.2	Kokkos and RAJA	12
3.3	SYCL and DPC++	13
3.4	Other Approaches	15
3.5	Portable Data Structures	16
3.6	Summary	18
4	Analysis of Approaches	20
4.1	Pragma-based Approaches	20
4.2	Programming Model Approaches	22
4.3	High-level DSL Approaches	28
4.4	Summary	29
5	Key Findings and Recommendations	31

1 Executive Summary

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 perhaps the first *modern* heterogeneous system. 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 different optimisation strategies, developing software that is both *performant* and *portable* across systems is becoming increasingly difficult.

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 our first report (2047358-TN-01), we provided a survey of the hardware and software that is likely to be prevalent on next-generation HPC systems. The hardware landscape is diversifying and because of this, new approaches to developing performance portable software are emerging.

Our second report (2047358-TN-02) set out to identify a number of test applications that could be used to evaluate some of the options available. Further, it established a series of ground rules, setting out the process through which we would evaluate the performance portability of approaches to software development.

The third report (2047358-TN-03) analysed a number of these approaches, through existing performance studies of these mini-applications, or from gathering execution data from the UKs HPC offerings, in order to evaluate approaches to developing performance portable scientific simulation applications.

In this final report, we summarise the key findings from our investigations and provide recommendations for developing a new plasma fusion simulation application.

The remainder of this report is structured as follows:

Section 2 summarises the hardware that is likely to be available at the Exascale in the UK, Europe and the US;

Section 3 outlines the possible approaches to developing software that is portable across these architectures;

Section 4 provides a brief analysis of these approaches, using the data collected in 2047358-TN-03;

Section 5 provides recommendations for future development in the NEPTUNE project.

As the HPC landscape shifts over the coming years, it is likely that some of the data or recommendations in these reports will change; therefore, these reports should be considered *living documents* that will be updated as new evidence emerges.

2 Hardware Targets

As we approach the era of Exascale computing, it is clear that heterogeneity is likely to be prevalent in the first generation of systems. This shift towards *accelerated* computing has been coupled with increasing diversity in the architectures available in HPC. Developing applications for these post-Exascale system therefore requires careful consideration of and preparation for the systems they are likely to be executed on.

2.1 Exascale Efforts

There are on going efforts towards Exascale happening around the world, and it is possible the first Exascale system will appear at the 2021 Supercomputing Conference (SC'21). Here we summarise the ongoing efforts in the UK, Europe and the United States, that are perhaps most relevant to NEPTUNE and UKAEA.

2.1.1 The United Kingdom

In the UK, Supercomputing is focused around Universities, often funded by UKRI, and a small number of commercial sites. Currently, the biggest systems in the UK are those found at research laboratories such as the Met Office (#58, #152 and #153 on Top500.org at the time of writing), ECMWF (#103 and #104) and AWE (#125). Each of these systems are homogeneous clusters using Intel CPUs, typically supporting applications that have been developed over a long period of time, in Fortran or C/C++, using MPI to distribute work across the cluster.

In 2021, the Met Office announced that its next system will also be a homogeneous cluster, but will be based on AMD Milan CPUs and delivered by Microsoft. It will deliver approximately 60 PetaFLOP/s of performance (i.e. 8× more powerful than their current XC40 system).

The fastest machine in the UK currently is the NVIDIA Cambridge-1 system, dedicated to use by Pharmaceutical companies. It is comprised of 80 NVIDIA

DGX nodes, each containing 8 GPUs, for a total of 640 A100 GPUs. Its 16 PetaFLOP/s performance puts it at #41 in the June 2021 Top500 list.

The HPC provision provided by UK Universities is structured in the form of a tiered system. The UK's national (Tier-1) supercomputer, ARCHER2, is currently in the process of being commissioned at the Edinburgh Parallel Computing Centre (EPCC). Like many of the HPC systems in the commercial sector, ARCHER2 is an homogeneous system with AMD Rome CPUs and will deliver approximately 28 PetaFLOP/s of performance.

It is at the regional (Tier-2) centres where there is a wealth of architectural diversity. The Isambard system, installed at the University of Bristol, is predominantly an ARM-based system, with one cabinet of Marvell ThunderX2 compute nodes. Besides this, it also contains a cabinet of Fujitsu A64FX CPUs, and the Multi-Architecture Comparison System (MACS), which consists of a range of alternative platforms for evaluation, including NVIDIA P100 and V100 GPUs, and CPUs from IBM, AMD and Intel.

The N8's Bede system is an IBM system that is similar in construction to the US Department of Energy Summit (#2) and Sierra (#3) systems. It consists of 32 nodes, each with two IBM Power9 CPUs and four NVIDIA V100 GPUs.

Besides these systems, York and Warwick also each have compute clusters for their own researchers. Each of these are predominantly homogeneous clusters with Intel CPUs, but both containing small GPU accelerated partitions.

The UK Government has stated that it is UKRI's intention that an Exascale-class supercomputer is available for UK researchers by 2025¹. Given the current status of HPC in the UK, it is likely that if this were to be achieved, it would require widespread adoption of heterogeneous compute.

2.1.2 Europe

In Europe, PRACE (Partnership for Advanced Computing in Europe) provide access to a number of PetaFLOP-class HPC systems (Tier-0). The current Tier-0 systems are:

¹<https://www.theyworkforyou.com/wrans/?id=2021-02-22.156386.h&s=exascale#g156388.q2>

Marconi, a 30 PetaFLOP/s IBM Power9 and NVIDIA V100 system installed at CINECA;

Hawk, HLRS's 25 PetaFLOP/s homogeneous system using AMD Rome CPUs;

JUWELS, a 70 PetaFLOP/s system with AMD Rome CPUs and NVIDIA A100 GPUs installed at FZJ;

SuperMUC, a 26 PetaFLOP/s system installed at LRZ, using Intel Xeon Skylake CPUs.

Joliot-Curie, a 12 PetaFLOP/s homogeneous AMD Rome system at CEA;

Piz Daint, a Intel Xeon and NVIDIA P100 system, delivering 27 PetaFLOP/s at ETH Zurich;

and, **MareNostrum 4**, installed at BSC consisting of 4 separate systems: an Intel Xeon cluster, an IBM Power9 and NVIDIA V100 system, an AMD Rome and Radeon Instinct MI50 system, and an ARMv8 cluster.

In July 2019, the EuroHPC Joint Undertaking governing body selected 8 sites across the EU to host new HPC systems. Of these 8 sites, 3 will host pre-Exascale machines capable of at least 150 PetaFLOP/s.

LUMI will be installed in Kajaani, Finland, and will be a Cray Shasta system comprising of AMD EPYC CPUs and AMD Radeon Instinct GPUs. It is expected to be capable of approximately 550 PetaFLOP/s.

LEONARDO will be installed at Cineca, Italy, and will be a Atos BullSequana system. It will be constructed of Intel Sapphire Rapids CPUs, coupled with 14,000 NVIDIA A100 GPUs, connected with Infiniband.

MareNostrum 5 will be installed within the Barcelona Supercomputing Centre. Like its predecessor, MareNostrum 5 will be two distinct (and currently unknown) systems, but may feature some use of the ARM and RISC-V architectures currently being explored by the EU's Mont Blanc project [4].

2.1.3 United States

In the United States, there is a long history of supercomputing within the Department of Energy. Currently the largest systems are both IBM Power9 and NVIDIA V100 systems installed at Lawrence Livermore National Laboratory and Oak Ridge National Laboratory.

The first phase of the new Perlmutter system was recently installed at the Lawrence Berkeley National Laboratory, with 1,500 nodes, each with dual AMD Milan CPUs, coupled with four NVIDIA A100 GPUs. Its achieved performance of 65 PetaFLOP/s placed it at #5 in the most recent Top500 list, with the second phase to be delivered at a later date, adding 3,000 more CPU-only nodes.

The DoE are currently in the process of building and installing their first three Exascale systems, namely Aurora, Frontier and El Capitan. Each of them are heterogeneous systems, consisting of mixture of CPUs and GPUs.

Frontier is a planned 1.5 ExaFLOP/s system being installed at Oak Ridge National Laboratory in 2021 and will consist of AMD EPYC Milan CPUs with AMD Radeon Instinct MI200 GPUs. Argonne’s 1 ExaFLOP/s **Aurora** system will follow in 2022 and will be constructed with Intel CPUs and GPUs – with each node being two Sapphire Rapids CPUs, with six Ponte Vecchio GPUs. These systems will be followed in 2023 by El Capitan at LLNL, which is expected to exceed two ExaFLOP/s. Like Frontier, El Capital will consist of AMD hardware, with EPYC Genoa CPUs and a next generation Radeon Instinct architecture.

2.2 Hardware Overview

It is clear from the ongoing efforts towards Exascale that the first generation of machines to break the ExaFLOP/s barrier will do so with heterogeneous architectures. It is also clear from the machines in development, that these heterogeneous architectures will feature a range of different CPU and GPU combinations, highlighting the issue of portability between systems.

Examining the systems listed in Section 2.1, we can see that all of the currently planned machines will consist of CPUs from Intel or AMD, and GPUs from Intel, AMD or NVIDIA. There may also be some machines with Arm-based CPUs, similar to those found in Fugaku, Isambard and EPCC’s Fulham.

There have been a number of recent announcements on these architectures that may be relevant to the development of a future NEPTUNE code.

The two major CPU vendors, Intel and AMD, are currently in the process of

developing their Exascale-era processors. Intel's recent *Architecture Day* in August 2021 provided details about their new CPUs and GPUs, while information from AMD has not yet been publicly released, but has been the subject of recent data leaks.

The upcoming **Xeon Sapphire Rapids** CPU includes a number of important improvements over current generation Xeon CPUs. In particular, there will be new instruction sets specifically aimed at AI inference and training called Advanced Matrix eXtensions (AMX), and there will be new acceleration engines aimed at offloading common tasks such as data movement, to free up CPU cycles. Additionally, there will be an increased last level cache (LLC) capacity, and an option for on-board high bandwidth memory (HBM2). There are a number of connectivity improvements that will also be included in Sapphire Rapids. The CPUs will include Intel's new Compute eXpress Link (CXL), designed for connecting CPUs and Accelerators, and will see an uplift to 8 DDR5 memory channels².

The next generation of AMDs EPYC CPUs will be codenamed **Genoa** (Zen 4). Details of Genoa have not yet been officially announced, but a recent data leak has provided some details. The Genoa CPU may include support for the AVX-512 instruction set, along with 12 DDR5 memory channels. It may also come in configurations up to 96 cores, with 2-way simultaneous multithreading (SMT)³.

In the GPU space, there are three architectures likely to be present in Exascale systems. Again, details were provided for the upcoming Intel Xe GPU at the recent Architecture Day, whereas the other two vendors are still subject to leaked information.

Intel's first generation Xe-HPC GPUs will be **Xe Ponte Vecchio**. Much like recent NVIDIA and AMD GPUs, it will include second-generation High Bandwidth Memory (HBM2e), and will allow GPU-GPU and CPU-GPU communication via Intel's CXL. Recent prerelease A0 silicon has been able to achieve over 45 TFLOP/s of performance for 32-bit floating point computation.

²<https://www.servethehome.com/intel-details-sapphire-rapids-xeon-at-architecture-day-2021/>

³<https://www.techradar.com/uk/news/gigabyte-hacker-spills-details-on-next-generation-amd-epyc-genoa-series>

The upcoming AMD **Radeon Instinct MI200** will be present in the Frontier system. As with the next generation EPYC CPU, information about the MI200 is only available from recent data leaks. The GPU is expected to double the performance of the MI100 (11.5 FP64 TFLOP/s) and will be constructed from two dies connected using Infinity Fabric. It is also expected to contain 128 GB of High Bandwidth Memory (HBM2e)⁴.

Details of the next NVIDIA architecture targeted at HPC workloads are still scarce. However, it is predicted that **Hopper** will launch in 2022, and will likely be the first multi-chip-module design from NVIDIA. It is also likely to triple the performance of the current Ampere generation GPUs.

2.3 Summary

The shift towards accelerated computing has made the task of efficiently programming these systems much more difficult. For homogeneous platforms, standard programming models (i.e. Fortran, C/C++, etc) along with well maintained compilers is sufficient for developing complex physics simulations. For accelerated platforms, hierarchical parallelism is usually exposed through a custom API and compiler developed specifically for the accelerator in use. For NVIDIA, this is the CUDA programming model; for AMD, this is HIP; and, for Intel, this will be SYCL/DPC++.

Although both AMD and Intel provide source-to-source translators that can take already developed CUDA code, and generate equivalent code for their accelerators, there are a number of efforts aimed at developing platform-agnostic applications from the outset. Whether applications developed using these platform-agnostic frameworks can be both *performant* and *portable* remains an open question.

⁴<https://www.tomshardware.com/uk/news/amd-begins-initial-shipments-of-aldebaran-cdna-2-gpu>

3 Developing Performance Portable Software

Our previous report (2047358-TN-01) provides a summary of a number of the available approaches to developing performance portable software for heterogeneous architectures. For brevity, the main approaches considered in this project are repeated here along with short listings demonstrating how parallelism is achieved in each of these programming models (using a simple vector-add example).

It is likely that any chosen programming model will be coupled with a message passing library such as MPI (i.e. the so called MPI+X model), and so in this work programme we have only considered single nodes, assuming node-to-node communication will still be handled through MPI-like libraries.

3.1 OpenMP

OpenMP is a standardised implementation of a shared-memory fork-join model, whereby annotations are used in code to signify work that can be multi-threaded. In scientific workloads, it is typically the case that loop structures are annotated with an OpenMP `#pragma`, such that each iteration can be executed in parallel.

Listing 1 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.

```
1 #pragma omp parallel for
2 for (int i = 0; i < 100; i++) {
3     c[i] = a[i] + b[i];
4 }
```

Figure 1: OpenMP code listing

In 2015, the OpenMP 4.5 standard introduced *offload* annotations, that can enable the compiler to generate hybrid executables for hosts with accelerator devices. Compiler support for the latest features of the OpenMP standard is often lagged, but the majority of compilers used in HPC now support a good

subset of `omp target` directives⁵.

An example of the same vector addition seen previously is provided in Figure 2 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.

```
1 #pragma omp target map (to:a[:size]) map (to:b[:size]) map (tofrom:
   c[:size])
2 #pragma omp teams distribute parallel for default(none)
3 for (int i = 0; i < 100; i++) {
4     c[i] = a[i] + b[i];
5 }
```

Figure 2: OpenMP 4.5 using target directives

An advantage of these directives is that code written using target directives can be executed on host platforms with no code changes, if no accelerator is available. This means that a single code base can target both homogeneous and heterogeneous platforms.

In our last report we evaluated the performance of TeaLeaf, miniFE and Laghos implemented in OpenMP, with TeaLeaf and miniFE also available implemented with OpenMP 4.5 target regions.

3.2 Kokkos and RAJA

An alternative approach that has emerged from the US Department of Energy’s Exascale Computing Project (ECP) is the use of C++ template libraries to enable compile-time code generation for different architectures. Kokkos, from Sandia National Laboratories, and RAJA, from Lawrence Livermore National Laboratory, are two such libraries.

Both Kokkos and RAJA provide a number of APIs that are primarily aimed at loop-level parallelism. Through using C++ templates, appropriate code is generated at compile-time to enable efficient execution on various target architectures. Figures 3 and 4 provide equivalent implementations of a vector add.

⁵<https://www.openmp.org/resources/openmp-compilers-tools/>

```

1 Kokkos::parallel_for(100, KOKKOS_LAMBDA (const int& i) {
2     c[i] = a[i] + b[i];
3 });

```

Figure 3: Kokkos

```

1 RAJA::RangeSegment seg (0, 100);
2 RAJA::forall<loop_exec> (seg, [=] (int i) {
3     c[i] = a[i] + b[i];
4 });

```

Figure 4: RAJA

In both cases, a simple for-loop is replaced with a templated lambda function that is expanded at compile-time to the appropriate target. Kokkos is able to target CUDA, OpenMP, pthreads, HIP or SYCL, while RAJA can target OpenMP, Intel Thread Building Blocks (TBB) or CUDA. Because the target platform is usually specified at compile-time, architecture-specific code can be generated, potentially enabling greater performance (unlike with a pure OpenMP approach, where architecture-specific pragmas would require potentially complex C-preprocessor

In our previous report, we evaluated a number of applications in both Kokkos and RAJA; in particular, all of the particle-in-cell codes we have evaluated are parallelised through Kokkos.

3.3 SYCL and DPC++

Another approach to developing portable applications is the Open Computing Language (OpenCL) framework, maintained by the Khronos Group. More recently, the Khronos Group ratified SYCL, a higher-level programming model that builds on the underlying concepts of OpenCL, but with a focus on improving programmer productivity. SYCL is a single-source embedded domain specific language based on C++17.

In SYCL, there is typically a queue that work items can be submitted to. Work items are typically written in the code as anonymous functions, much like in Kokkos and RAJA. Parallelisation is achieved using constructs such as

the `parallel_for` function.

Figure 5 provides an example of a 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.

```
1  sycl::queue myqueue;
2  std::vector h_a(100), h_b(100), h_c(100);
3  sycl::buffer d_a(h_a), d_b(h_b), d_c(h_c);
4
5  auto ev = myqueue.submit([&](handler &h){
6      auto a = d_a.get_access<access::read>();
7      auto b = d_b.get_access<access::read>();
8      auto c = d_c.get_access<access::write>();
9      h.parallel_for(count, kernel_functor( [=](id<> item) {
10         int i = item.get_global(0);
11         c[i] = a[i] + b[i];
12     }));
13 });
```

Figure 5: SYCL

Support for SYCL exists in a number of compilers, with a variety of target architectures⁶. The ComputeCpp compiler, from Codeplay, has multiple backends, allowing it to target a range of CPUs and GPUs from Intel, AMD and Arm; the triSYCL compiler, developed by Xilinx, can generate OpenMP-compliant applications, and can additionally target Xilinx FPGAs; Heidelberg University’s LLVM-based hipSYCL compiler can generate OpenMP, CUDA, ROCm or oneAPI Level Zero code, allowing it to target CPUs and GPUs from the three major hardware vendors expected to be present in post-Exascale systems.

SYCL has additionally been adopted and extended by Intel (as Data Parallel C++) for its oneAPI programming model. While initially appearing in Intel’s (now branded “Classic”) C++ Compiler in 2020, aimed primarily at Intel hardware, the adoption of an LLVM-backend in 2021 has meant that Intel’s compiler can now natively support NVIDIA and AMD targets also, through CUDA and HIP, respectively.

The maturity of SYCL toolchains has been the subject of recent work, with performance still typically lagging native alternatives [5, 6]. Whether this performance gap can be reduced remains an open question.

⁶<https://www.khronos.org/sycl/>

3.4 Other Approaches

A number of other approaches are also available, some of which have been evaluated in this project but only in a limited capacity.

The OpenACC (Open Accelerators) programming model is an annotation-based approach similar to OpenMP but specifically aimed at accelerators. Adoption of the OpenACC standard into common compilers is often lacking, with the exception of the Cray and NVIDIA (PGI) compilers. The introduction of `target` pragmas in the OpenMP standard, may ultimately render the OpenACC standard unnecessary. Nonetheless, in our previous report, an evaluation of TeaLeaf with OpenACC is presented, providing a realistic target for future OpenMP target offload implementations.

Our previous report also provides an evaluation of TeaLeaf developed using the Oxford Parallel library for Structured mesh solvers (OPS). OPS is a domain specific language (DSL) that exists at a higher level than the previously mentioned programming models, targeting multi-block structured mesh computations. OPS provides an API for storing blocks of data, and abstractions to iterate over these blocks in parallel. The OPS compiler can then generate code for CUDA, HIP, OpenMP and SYCL, allowing it to target any of the post-Exascale hardware platforms currently announced.

There are a number of other DSLs aimed at solving computational fluid problems. Two notable examples are PSyclone, from the Met Office, and UFL (Unified Form Language), from the FEniCS project. PSyclone is a code generator specifically aimed at Finite Element, Finite Difference and Finite Volume methods, that can generate OpenMP, OpenCL and OpenACC code. UFL is a DSL for expressing partial differential equations (PDEs) that can be solved by the FEniCS or Firedrake platforms. PDEs are expressed in Python using the mathematical operators provided by UFL; the compiler then generates C code (using PyOP2) to solve the PDEs in parallel.

For particle methods, there is a distinct lack of DSLs. The only notable example we are currently aware of is PPMD (Performance Portable Molecular Dynamics), another Python-based DSL. As the name suggests, PPMD is specifically aimed at molecular dynamics simulations. We are unaware of any DSLs aimed at the Particle-in-Cell (PIC) methods that will likely be required in NEPTUNE.

In this work package, we have not attempted to evaluate these high-level DSLs, as they each code-generate to lower level approaches that have been evaluated. Nevertheless, these DSLs do provide useful examples for how an interface might look for scientists wishing to express their computational problems.

Beyond specific programming languages and programming models for developing performance portable applications, some of the applications evaluated as part of this project are available written using scientific computing libraries. For example, the Laghos mini-application is a high-order finite element code implemented using the HYPRE and MFEM libraries. Computation in HYPRE is then parallelised with OpenMP, OpenMP 4.5, CUDA, HIP, RAJA or Kokkos. Similarly, the EMPIRE-PIC code makes extensive use of Kokkos, and the linear solvers in the Trilinos library that are also parallelised through Kokkos. Many scientific applications also rely heavily on the linear solvers in common math libraries such as LAPACK.

3.5 Portable Data Structures

Besides the selection of programming model, data access patterns are perhaps one of the most important considerations for achieving high performance. In scientific computing applications, matrices and tensors are typically stored in multi-dimensional arrays, while particle-like data is usually stored in structures/objects. When looping over these data in parallel, the order of traversal can be beneficial or adversarial to the host architectures data caches. Furthermore, expressing multi-dimensional data in C and C++ may be difficult if the dimensions are not fixed at runtime, requiring programmers to potentially manually de-reference data in one-dimension (i.e. using `array[col + (num_cols * row)]`).

Both Kokkos and RAJA provide a solution to this by offering views over data. In the case of RAJA, this is a simple View class that can be used on initialised 1-D arrays to provide simple de-referencing. Figure 6 shows the use of the `RAJA::View` class on a simple two-dimensional array.

Kokkos provides fully managed multi-dimensional arrays through its View class. Figure 7 provides the same simple example of a two dimensional array in Kokkos. Because Kokkos Views are fully managed, they are allocated and reference

```

1 const int DIM = 2;
2 double *array = new double[num_rows * num_cols];
3 RAJA::View<double, RAJA::Layout<DIM> > array_view(array, num_rows,
    num_cols);
4 Aview(0,0) = ...;
5 ...
6 free(array);

```

Figure 6: Use of RAJA::View for multi-dimensional arrays

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.

```

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

```

Figure 7: Use of Kokkos::View for multi-dimensional arrays

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 8 provides a simple example of particle storage using an array-of-structs (AoS) and a struct-of-arrays (SoA) approach.

<pre> 1 #define N 1024 2 typedef struct { 3 // position 4 float x, y, z; 5 // momentum 6 float ux, uy, uz; 7 // weight 8 float w; 9 } Particle; 10 Particle particles[N]; 11 // access x field from particle 12 particles[0].x; </pre>	<pre> 1 #define N 1024 2 typedef struct { 3 // position 4 float x[N], y[N], z[N]; 5 // momentum 6 float ux[N], uy[N], uz[N]; 7 // weight 8 float w[N]; 9 } Particles; 10 Particles particles; 11 // access x field from particle 12 particles.x[0]; </pre>
--	--

Figure 8: 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.

Intel’s SIMD Data Layout Templates (SDLT) offers a convenient way to abstract the in-memory data layout transparently to the developer. Figure 9 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.

```
1 #define N 1024
2
3 typedef struct particle_data {
4     float x, y, z;
5     float ux, uy, uz;
6     float w;
7 } Particle;
8
9 SDLT_PRIMITIVE(Particle, x, y, z, ux, uy, uz, w)
10 ...
11 sdlt::soa1d_container<Point2D> pContainer(N);
12 auto particles = pContainer.access();
13 #pragma omp simd
14 for (int i = 0; i < 1024; i++) {
15     particles[i].x() = ...;
16     ...
17 }
```

Figure 9: Intel SDLT

A similar approach, using Kokkos Views, can be found in the VPIC 2.0 application [7]. 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 10). 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.

3.6 Summary

Due to the nature of how most scientific problems are expressed in code, the majority of the programming models explored express parallelism through loop-structures. For this reason, the difference between many of the available programming models is primarily semantics.

```

1 Kokkos::View<float*[7]> particles(N); // particle data
2 namespace particle_var {
3     enum p_v { // particle member enum for clean access
4         x, y, z,
5         ux, uy, uz,
6         w,
7     };
8 };
9 View<int*> particle_indicies(N); // Particle indices
10 // Access x from particle 0
11 particles(0, particle_var::x) = ...;

```

Figure 10: Using Kokkos to convert AoS to SoA

Where the models diverge significantly is in their language, compiler and hardware support profiles. Pragma-based standards like OpenMP are available in all of the major HPC languages (C, C++, Fortran), and usually benefit from support in many of the main HPC compilers. Additionally, applications written with pragma-based approaches have the advantage of falling back to single thread, host-only code, where support is unavailable. Unfortunately, it is often difficult to express complex parallel schemes with these simple code annotations, and implementation of these standards often lags development of the standard itself.

Template libraries like Kokkos and RAJA partially avoid this issue, by using template metaprogramming to generate relevant code at compile time. This restricts the choice of programming language to modern C++, but does mean that support for new hardware targets (and architecture-specific optimisations) come “free” as the Kokkos and RAJA teams develop and optimise new backends. However, this does build a dependency into any applications developed using these third-party libraries. This dependency may extend beyond just inclusion of the library, to adopting custom build instructions.

In contrast to the DoE-developed programming models, SYCL is an open standard and Intel’s DPC++ is a C++ language extension that is being built into their own compiler. That the standard is maintained by a group of organisations, and the investment by Intel, bodes well for its use in HPC.

4 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. As shown in our previous reports 2047358-TN-01 and 2047358-TN-03, there are 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 [8, 9, 10, 11]. 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).

4.1 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 11 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,

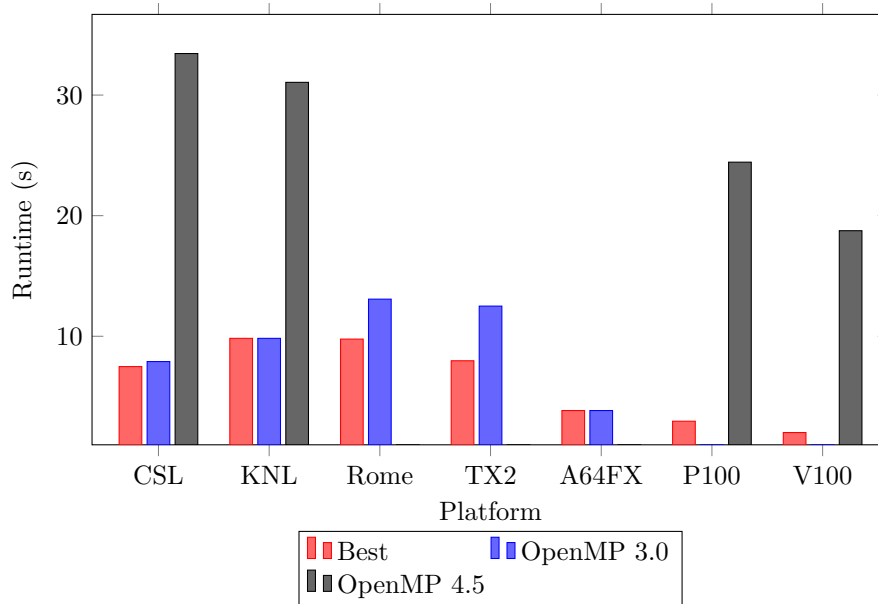


Figure 11: miniFE in OpenMP runtime data

but the runtime is several orders of magnitude higher than all other variants and are therefore omitted⁷.

Figure 12 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 et al. [9]. In Figure 13 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

⁷A 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.

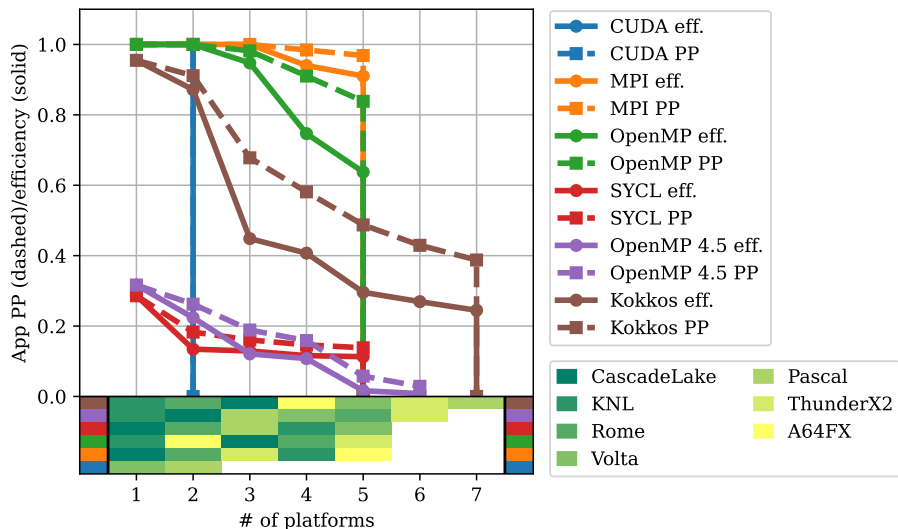


Figure 12: Cascade visualisation of performance portability of miniFE

different versions of the OpenMP specification, different hardware and even different compilers⁸. 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).

4.2 Programming Model Approaches

The next approach we have explored in this work package, 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 [8, 9, 10, 11]) have shown that both are typically able to deliver good and portable performance from a

⁸See: https://github.com/UoB-HPC/TeaLeaf/tree/master/2d/c_kernels

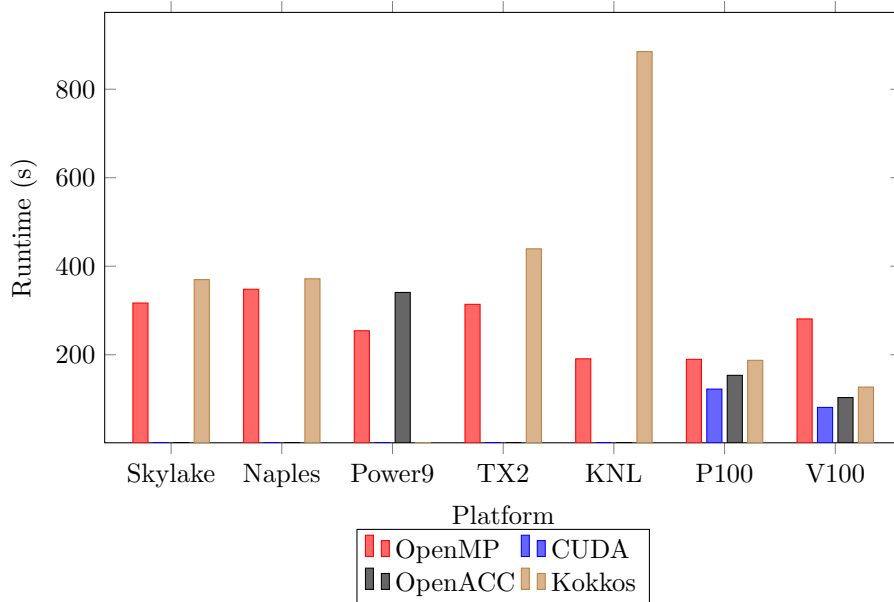


Figure 13: TeaLeaf runtime data from Deakin et al. [9]

single source code base.

The results in Figure 14 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 16 demonstrates that

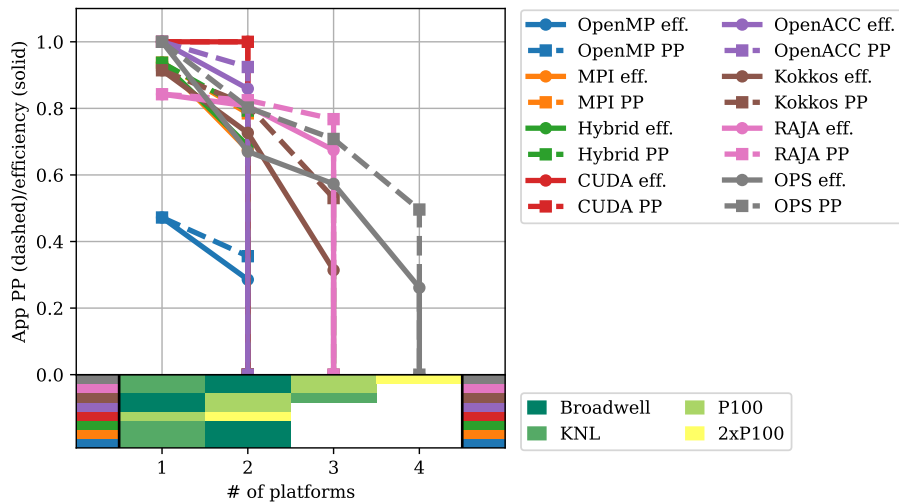


Figure 14: Cascade visualisation of performance portability of TeaLeaf from Kirk et al.

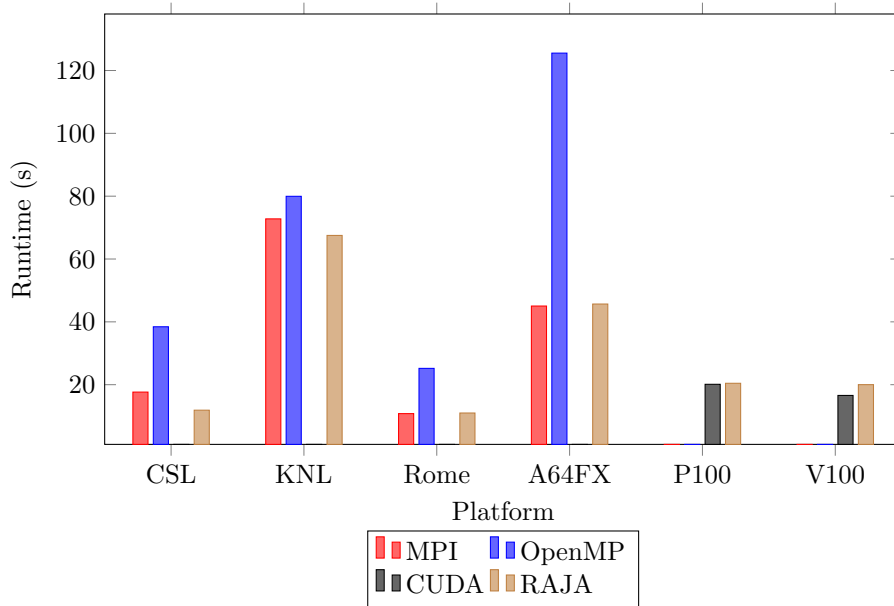


Figure 15: Laghos runtime data

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 future⁹. Importantly, the Kokkos variant can be executed across GPUs, where much of the available performance is likely to lie in post-Exascale systems.

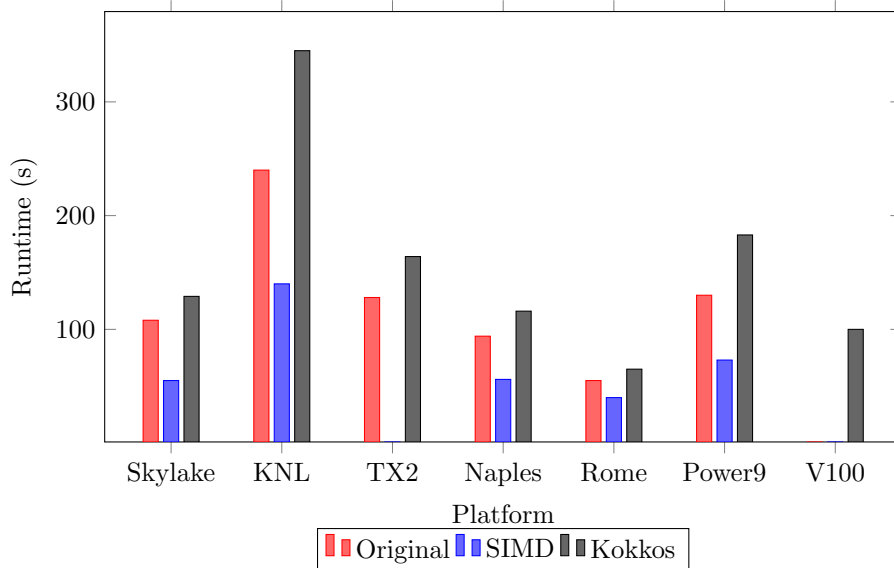


Figure 16: VPIC runtime data from Bird et al. [7]

While Kokkos and RAJA have both shown promise as approaches to performance portable application development, each also carry a small element of risk. For each API there is potentially a single point of failure – the API may be changed at short notice; support for the API or development of the library may be withdrawn at any time; and hardware backends may never be developed. Nonetheless, a high level of 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++ standard¹⁰.

In contrast to Kokkos and RAJA, the SYCL programming model is an open standard maintained by the Khronos Group. Interest in SYCL is growing

⁹Indeed, 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 [12].

¹⁰e.g. mdspan, <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0009r10.html>

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 miniFE port generated using Intel’s DPC++ Compatibility Tool (which converts from CUDA), compiled with the hipSYCL compiler. Figure 12 shows that the performance portability of SYCL in this configuration is similar to the OpenMP 4.5 variant of miniFE. Since hipSYCL typically uses OpenMP to target CPUs, this is perhaps not surprising. It is quite possible (perhaps even probable) that the Intel 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 [6]. Figure 17 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 [5]. 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

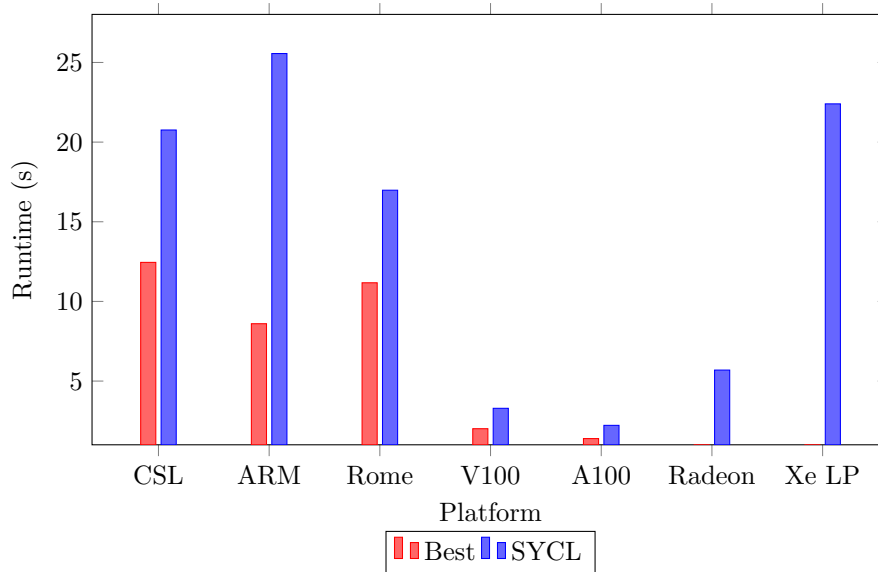


Figure 17: MG-CFD runtime data from Reguly et al. [6]

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 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.

For the two mini-applications the results are more varied; in some cases there are large differences between the compilers (see Figure 18). In this study, only hipSYCL was able to target the NVIDIA GPU, due to compatibility with NVIDIA’s 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 cur-

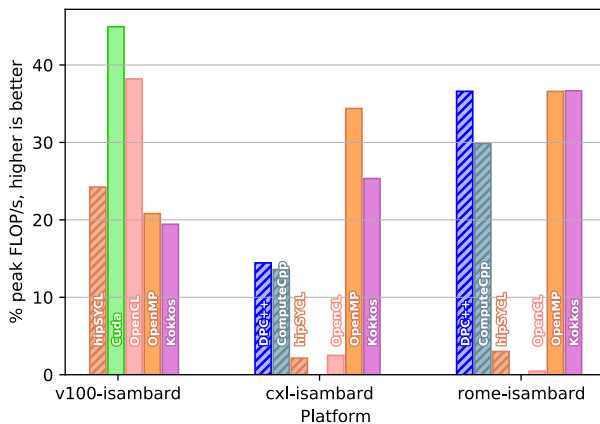


Figure 18: CloverLeaf: SYCL vs. alternative frameworks from Lin et al. [5]

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

4.3 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 14 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*¹¹.

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, embed-

¹¹See: <https://op-dsl.github.io/docs/OPS/tutorial.pdf>

ded 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.

4.4 Summary

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

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

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

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

5 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 reuse¹²;


¹²Although Fortran introduced object orientation in the 2003 standard, it lacks many of the advanced features present in C++ [13].

- 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.

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 [14]. 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.

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] Michael Feldman. Europe Will Enter Pre-Exascale Realm With MareNostrum 5, 2019.
- [5] 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.
- [6] 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.
- [7] Robert Bird, Nigel Tan, Scott V Luedtke, Stephen Harrell, Michela Tauffer, and Brian Albright. VPIC 2.0: Next Generation Particle-in-Cell Simulations. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2021.
- [8] Simon McIntosh-Smith. Performance Portability Across Diverse Computer Architectures. In *P3MA: 4th International Workshop on Performance Portable Programming models for Manycore or Accelerators*, 2019.

- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.