https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

PSyclone 3: A source-to-source Fortran compiler for developing maintainable and performance-portable HPC applications

Rupert W. Ford[1,†], Aidan B. G. Chalk[1], Joerg Henrichs[2], Iva Kavčič[3], Christopher Maynard[3], Andrew R. Porter[1], and Sergi Siso[1]

[1]STFC Hartree Centre, Daresbury Laboratory, UK
[†]deceased, 14 February 2024
[2]Bureau of Meteorology, Melbourne, Australia
[3]Met Office, Exeter, UK

Correspondence: Andrew R. Porter (andrew.porter@stfc.ac.uk) and Sergi Siso (sergi.siso@stfc.ac.uk)

Abstract. PSyclone is a source-to-source Fortran compiler designed to programmatically optimize, parallelize, and instrument HPC applications via user-provided transformation scripts. These scripts allow for a clear separation of concerns between the scientific model, described in Fortran, and the optimization choices. This separation improves the maintainability of complex scientific applications by enabling independent exploration and development of each aspect. In addition, it provides a solution to achieve better performance portability for Fortran applications by encoding the transformations that are beneficial to each platform and compiler in different transformation scripts.

PSyclone supports two modes of operation. The first mode optimizes existing source code, including MPI-based applications, by making the necessary code transformations to effectively use the capabilities and programming models supported by each CPU and GPU vendor. This approach is demonstrated using a benchmark extracted from the NEMO ocean model. The second mode defines a kernel-based parallelism model with domain-specific Fortran-embedded metadata. This approach enables a stricter separation of concerns, thereby allowing PSyclone to take full control of the data dependencies and iteration spaces in order to improve its capabilities and generate distributed and shared-memory parallelism. This approach has been co-designed with the Met Office and is used in the Finite-Element based dynamical core of the LFRic atmospheric model.

1 Introduction

PSyclone is an open-source code transformation and generation system designed to programmatically optimize, parallelize, and instrument Fortran HPC applications via user-provided transformation scripts. It has been designed to facilitate the development of Fortran weather and climate applications, but is also applicable to other domains.

The computational demands of weather and climate modelling require that simulations use some of the world's most advanced supercomputers, on which achieving optimal performance and parallel scalability are crucial. Furthermore, the widespread sharing of these applications among different institutions, illustrated, for example, by the wide adoption of the NEMO ocean model (Madec et al., 2026) in various weather and climate centers, requires that the applications also run optimally on a variety of platforms. At the same time, these are inherently complex codes that remain operational for decades and

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

are constantly being updated with new science by multiple developers. Therefore, they must be maintainable and extensible for future supercomputing architectures. The idea that a code can run well on different, including upcoming, architectures is often called *performance portability*.

Writing HPC applications that are maintainable, have good scalability, and are performance portable is challenging (Pennycook et al., 2021), especially for weather and climate applications (Lawrence et al., 2018). This challenge is exacerbated by the end of Dennard scaling (Dennard et al., 1974) and the increasing use of specialized accelerators such as GPUs in supercomputer facilities. The Fortran and MPI software toolchain that many HPC applications still use today is no longer sufficient (Lawrence et al., 2018), but adding support for architecture and vendor-specific optimizations with divergent code paths conflicts with the goals of maintainability and portability.

Several solutions have been developed for writing maintainable and performance portable HPC applications. These include systems such as Kokkos, RAJA and SYCL, which take advantage of C++ template metaprogramming to provide parallel abstractions that adapt to the needs of a specific platform at compile time. However, these are impractical for many pre-existing Fortran applications where significant rewrites would be too costly and domain scientists lack familiarity with complex C++ syntax. Other solutions such as OpenMP and OpenACC are applicable to Fortran, but these lack the full flexibility offered by metaprogramming, and their capabilities are limited by the functionality provided by the standard, with different compiler vendors providing varying levels of support for them. In addition, directives are tightly coupled to the underlying code, and for some applications, these are not considered viable by its developers. An example of this is the aforementioned NEMO ocean modelling framework (Madec et al., 2026), where the focus is on scientific functionality, maintainability, and enabling community contributions. Therefore, the code base is kept free of directives, relying solely on MPI to exploit data parallelism. Whilst this strategy has worked well for the past few decades, it is not possible to run the model on the GPUs, which are increasingly used to provide the bulk of the compute performance of modern HPCs.

In response to these challenges, we introduce PSyclone, a novel source-to-source Fortran compiler designed to programmatically modify Fortran code via user-provided transformation scripts. This approach decouples the scientific logic, expressed in the Fortran code, from the parallelization and optimization details encoded in separate Python scripts. This separation of concerns leads to maintainable code where each aspect can be investigated and developed largely independently. Additionally, it allows developers to tackle the performance portability challenge by writing different scripts for different platforms, each adhering to the specific programming model and preferred language features of the specific hardware vendor and compiler being targeted.

PSyclone offers two modes of operation: direct transformation of existing Fortran codes and support for developing Fortran-embedded domain-specific languages (DSLs). The first mode is relatively general and has proven successful for applications that use directly-addressed methods that express their data parallelism via loops over multidimensional arrays. It is being used to offload and add shared-memory parallelism in a number of models including NEMO, the UKCA atmospheric chemistry scheme, and physical parameterization schemes within the Met Office's atmosphere modelling system. In this paper, we demonstrate this approach with a simplified benchmark extracted from the NEMO code. There are, however, limitations to this approach; the model developers remain responsible for managing any MPI parallelization (such as halo exchanges),

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

the code must not use patterns that cannot be parallelized (such as linked lists), and PSyclone's ability to decide the validity of transformations is limited to a single file; therefore, some global transformations, like changing the data layouts, are not feasible.

In the second mode, PSyclone supports the development of Fortran-embedded kernel-based DSLs that conform to the PSyKAl (Ford et al., 2013; Adams et al., 2019; Porter et al., 2018) separation of concerns. PSyKAl specifies that code is separated into three layers, the algorithm code (Al), the kernel code (K), and the parallel system code (PSy). In this approach, the Algorithm operates on conceptually global fields while the Kernels are individual units of computation that operate on single elements. PSyclone uses this information to generate a middle layer (PSy-layer) that decides the execution order and parallelization policies to use to leverage the target platform. In this mode, PSyclone can guarantee that no domain data is touched without its knowledge outside a kernel and, therefore, can generate the full data-dependency graph to enable global transformations such as generating a distributed memory parallelization for the model.

To date, two DSLs conforming to the PSyKAl separation of concerns have been implemented in PSyclone: "LFRic", a mixed finite-element DSL which was co-designed with the dynamical core (known as Gung Ho) of the Met Office atmospheric model, LFRic (Adams et al., 2019) and "GOcean", a 2D finite-difference or finite-volume DSL (Henrichs et al., 2025).

The choice of which approach to use depends on the code, the community that is using it, and the willingness of code owners to accept the cost of restructuring or rewriting their code. Different components of an application can use different approaches, providing a path to iteratively migrate codebases to higher-level abstractions over time.

The remainder of this paper is structured as follows. Section 2 provides a brief overview of performance-portability solutions available in the literature and explains how PSyclone differs from them. Section 3 describes the details of the implementation of PSyclone. Section 4 presents the use of PSyclone for code transformation and evaluates this approach with a tracer-advection benchmark originating from NEMO. Section 5 expands this by introducing the PSyKAL DSL approach and evaluating it using the LFRic atmospheric model. The paper closes with some conclusions in Section 6.

2 Related Work

Metaprogramming abstractions are widely used in the development of high-performance portable applications because they can be used to optimize low-level aspects, such as data access patterns or loop iteration sequences, without introducing additional layers of indirection, which typically carry a performance penalty. Additionally, because metaprogramming operations are executed at or before compile time, they do not hinder the compiler's ability to perform additional optimizations. This ensures that the final code is both efficient and adaptable to varying architecture requirements.

Due to these benefits, C++ template metaprogramming frameworks have emerged as one of the most popular ways to develop maintainable, heterogeneous computing HPC applications. Frameworks like Kokkos (Trott et al., 2022), RAJA (Beckingsale et al., 2019) and SYCL (The Khronos SYCL Working Group, 2025) exemplify this trend, offering powerful tools for designing performance-portable HPC applications. However, a significant challenge is presented by the prevalence of pre-existing HPC applications written in Fortran, which lacks template metaprogramming. Moving large codebases to C++ is often prohibitively

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

expensive. Furthermore, domain scientists, who are integral to the development of these applications, may not possess expertise in C++ or template metaprogramming, and often prefer simpler languages that are more restricted to the domain logic. Thus, the prospect of migrating to a C++-based codebase introduces substantial risks to the application ecosystem.

Another popular option, available for both C++ and Fortran, is the use of language directives such as OpenMP (OpenMP Architecture Review Board, 2021) and OpenACC (OpenACC Standard, 2022), or higher-level standard language constructs such as the Fortran *do concurrent* construct (Reid, 2008). These provide additional semantic information to the compiler so that it can make the appropriate optimization decisions. However, they lack the flexibility of template metaprogramming as their capabilities are restricted to the constructs available in the standard. Standards bodies take time to formalize new capabilities, which then take time to be implemented by each compiler. So in practice, the performance-portability benefits are limited to the common denominator of features implemented by each vendor compiler on each platform of interest. Additionally, when the optimal set of directives differs between vendors and architectures, the portability benefits are diminished, sometimes requiring multiple implementations.

Metaprogramming can also be achieved by using preprocessing tools for code-generation and transformation in the application build system. This can range from generally applicable source-to-source compilers to more narrowly applicable domain-specific language constructs. It is to this category that PSyclone belongs.

The ROSE compiler framework (Quinlan and Liao, 2011) provides a general-purpose environment that can be used to develop source-to-source translation tools (such as OP2 and CHiLL, mentioned below) and to perform compiler research. ROSE provides a rich suite of functionality, but this comes with a comparatively complex installation process with many dependencies. It also requires developers to be familiar with both both C++ and compiler architecture concepts. In contrast, PSyclone is much narrower in concept and aims to lower the barrier to entry through the use of Python (widely used in scientific computing) and an intermediate representation much closer to Fortran semantics.

The kernel-based DSL approach of PSyclone is similar to that first developed in OP2 (Bertolli et al., 2012). OP2 supports the idea of a top-level algorithm code and separately-written kernel function calls. The algorithm code contains `OP_PAR` function calls. An `OP_PAR` function call takes as arguments a single kernel function call and a description of that kernel function. The implementation of the `OP_PAR` function call is then responsible for running its kernel function in parallel and adding inter-processor communication calls, if required. PSyclone expands on these concepts by adding inter-kernel optimizations and more flexible transformations through programmatic mutations of its intermediate representation. Similarly to CHiLL (Chen et al., 2008), PSyclone also enables transformations to be applied by the HPC expert via a script rather than being fixed by the system.

There are several DSL-based approaches that focus on allowing the developer to express their problem at a higher level, either in terms of the mathematical formulation or the stencil operations that it performs. For example, the Python library GT4Py (Ubbiali et al., 2025) enables scientists to describe the computations to be performed at a relatively high-level (in Python). An Intermediate Representation is created for these computations, which is then optimized and passed to one of a choice of backends, enabling execution on both GPU and CPU hardware. Although stencil-specific optimizations are performed by the toolchain, they are not under the control of the model developer. GT4Py is itself an evolution of the C++ templating approach taken in STELLA (Gysi et al., 2015), with the aim of improving the ease of use by providing a Python rather than

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

C++ frontend. This highlights a difficulty that is common to all DSL approaches: the need for the model to be rewritten in a new language and the need for that language to be able to express all of the forms of computation performed by a model.

Loki (Ubbiali et al., 2025; Lange and Reuter), and the now dormant Claw project (Clement et al., 2018), provide(d) support for parallelizing the single-column abstraction pattern typically encountered in the physical parameterizations of the IFS model. While Claw is a DSL with its own compiler, Loki is a source-to-source transformation tool conceptually similar to PSyclone in that it allows the modification of source code through programmable transformations.

In the DaCe programming environment (Ben-Nun et al., 2019), domain scientists develop their model in either Python, MATLAB or TensorFlow, these are translated into a Stateful Dataflow Multigraphs (SDFGs) describing the computation and associated dataflow. As in PSyclone, HPC specialists can choose which transformations to apply to the IR before the final, code-generation step emits a flavor of C++ (e.g. using CUDA for NVIDIA GPUs). Since the dataflow is explicitly captured in the IR, the generated code can include the necessary instructions to transfer data to the local memory space of an accelerator device. More recently, DaCe has been extended with a custom Fortran parser and has demonstrated good performance with a kernel of the ICON model (Klocke et al., 2025).

Firedrake provides an even higher level of abstraction by enabling the domain scientist to express the mathematics of a Finite-Element model in Unified Form Language (Alnæs et al., 2014), embedded in Python. At runtime, this is lowered to C++, compiled and linked against PETSc (Balay et al., 2025). As such, it is a powerful tool for enabling domain scientists to rapidly experiment with different Finite-Element formulations and be able to do so at scale.

The Multi-Level IR (MLIR) compiler framework (Lattner et al., 2021) provides interesting opportunities for future DSL development within the LLVM ecosystem, offering robust portability to any system supported by an LLVM-based compiler. However, bringing the extensive institutional knowledge on the performance aspects of weather and climate models into a fast evolving, C++-based compiler framework will be a difficult challenge.

PSyclone does not try to reinvent the full compiler stack, instead, it provides transformations at the language level. That is, transformations applied directly to the source code and expressed using concepts that correspond closely to the constructs of the original programming language. This allows HPC developers to abstract performance decisions while still working in a familiar environment and leveraging the directives and optimizations provided by the vendor compilers. It also provides a unique range of options from flexibly accepting any input Fortran to restricting the input to specific DSL expectations, given the developers freedom to choose the best model for their particular application and community.

3 The Architecture of PSyclone

This section describes the choices in the design of the PSyclone tool. At its core, PSyclone uses a custom-made intermediate representation called PSyIR (PSyclone Intermediate Representation). PSyIR can be converted to and from other languages or intermediate representations by using its frontend and backend infrastructure. The tool also provides transformations to mutate the PSyIR and some utilities to help analyze the code. Each of these components is described in this section in turn.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

3.1 PSyIR: the PSyclone Intermediate Representation

The PSyIR is an Abstract Syntax Tree (AST) with nested scopes that PSyclone uses to represent both existing code and DSL abstractions. It is designed to represent high-performance, parallel applications in an expressive, mutable, and extensible way:

- It is *expressive* because it is intended to be human-readable and manipulated by HPC software experts when optimizing or porting applications. This means that the node semantics are intentionally kept close to the Fortran language concepts and each node provides semantically-relevant operations.
- It is *mutable* because it is intended to be manipulated programmatically.
- It is *extensible* to allow representing different levels of abstraction, from language-level semantics to higher-level DSL concepts such as HaloExchanges, GlobalSums, loops over a domain characteristic, etc.

  Importantly, in PSyclone, nodes from different abstraction levels are interoperable and can be found mixed in the same PSyIR tree. The frontends and backend operate on language-level nodes which through different uplifting and lowering methods can be converted to and from DSL nodes. Figure 1 shows the typical lifetime of a node in PSyclone.

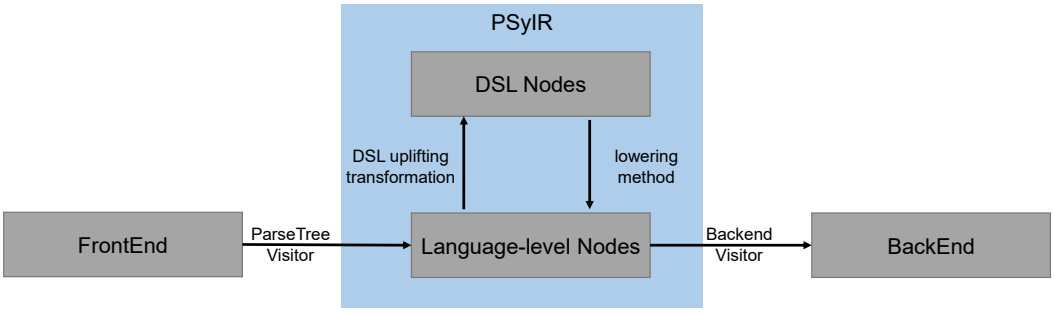

Figure 1. PSyIR node abstraction-levels flowchart

Another feature of the PSyIR is that, since it is designed solely for source-to-source transformations, it can get away supporting only the subset of the input code that is relevant for performance. Within the PSyIR any unsupported constructs are placed in nodes called *CodeBlocks*, and the unsupported data types are marked as *UnsupportedTypes*. This allows the PSyIR transformations to gracefully skip the parts of the tree that are not understood and helps to keep the PSyIR simple. Concepts such as file-system operations, uncommon Fortran features, and labeled statements are currently placed on the CodeBlock nodes. Although CodeBlocks make it possible for PSyclone to accept any valid Fortran code, it limits the dependency analysis and application of transformations, as it must assume the worst-case scenario for the symbols inside the CodeBlock. The presence of a CodeBlock node also prevents PSyclone from generating code in a language other than that of the input source code.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

3.2 PSyIR frontends and backends

To convert a code to or from PSyIR PSyclone provides multiple readers and writers, implemented using a traditional visitor pattern architecture. The main supported language is Fortran. For this, PSyclone creates a parse tree using *fparser* (Peterson et al., 2026) that is then converted to PSyIR using the relevant frontend. If PSyclone is parsing one of the supported DSL applications, a domain-specific uplifting transformation is applied. The uplifting step identifies patterns that define the domain model and introduces higher-level abstraction PSyIR nodes (e.g. kernels, halo exchanges, etc.) that provide additional semantic domain information to the tree.

Similarly, the backends translate the PSyIR into other languages or IRs. To do this, PSyclone follows again a two-step process. First, it lowers any DSL nodes to language-level PSyIR nodes: each domain-specific node is responsible for providing a lowering method to create the language-level representation of themselves. Then, a language backend visitor can be used to traverse the tree and create a textual source code representation of each node. Figure 2 provides a flow chart with all the described steps.

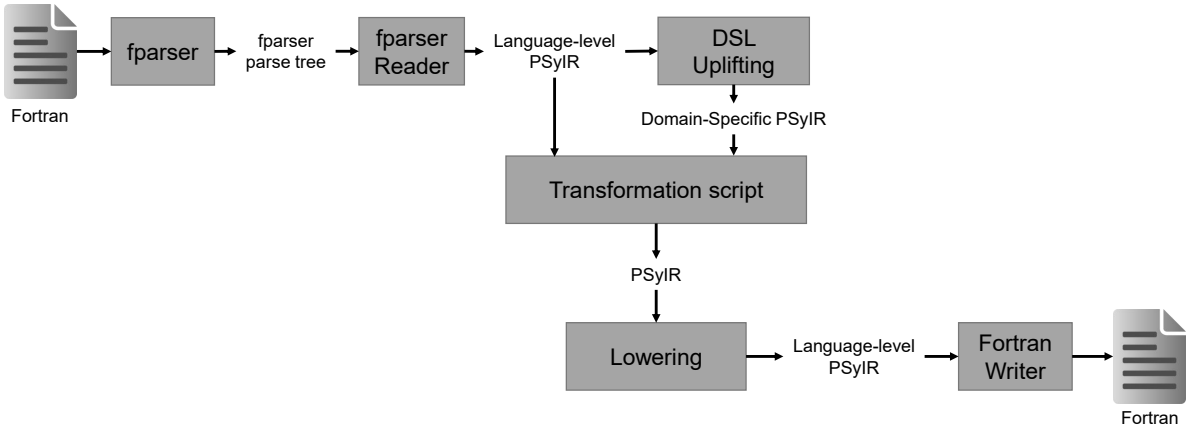

Figure 2. Typical PSyIR workflow within the PSyclone tool.

This architecture provides a modular approach that allows the reuse of the uplifting and lowering logic of the supported DSLs for different frontends and backends. It also makes the PSyIR easy to extend, as new nodes can be added to the PSyIR without modifying the language backends, as long as they provide their lowering method.

In addition to the Fortran backend, PSyclone has an experimental OpenCL backend for the PSyIR subset that is specifically involved in data and arithmetic computations. This has been used in the NemoLite2D application to run on multi-core CPU, GPU and FPGA (Siso et al., 2023) architectures. Internally, PSyclone also uses front- and/or backends for the Sympy (Meurer et al., 2017), Dawn (Osuna et al., 2020), and MLIR (Bisbas et al., 2024) intermediate representations.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

3.3 Transformations

Directly mutating the IR is generally a dangerous operation due to the need to maintain a structure that still represents a semantically valid program. To improve the safety of programmatic modifications, PSyclone has the concept of PSyIR Transformations. These are coherent sets of operations that can be applied to the IR and provide a validation mechanism to ensure that the result is still a semantically equivalent (or valid) program. Additionally, each node provides a global validation check that is executed during the backend visitor traversal.

PSyclone comes with transformations that will typically be used to optimize and parallelize computational code. These range from generic loop and array transformations such as *LoopTilingTrans* or *ArrayAssignment2LoopTrans*; to transformations that insert parallelism using OpenMP (e.g. *OMPParallelTrans*, *OMPTargetTrans*) or OpenACC (e.g. *ACCKernelsTrans*, *ACCEnterDataTrans*). There are also transformations to assist with debugging or performance monitoring such as *ProfileTrans*, *ValueRangeCheckTrans* or *ExtractTrans*, which creates standalone applications for a selected code region.

Each transformation provides a *validate* and an *apply* method. The *validate* method checks, without modifying the tree, whether the tree complies with all requirements to guaranty that the transformation will be applied successfully; otherwise, a *TransformationError* will be raised with information about the missing requirements, so that the user script can still attempt to recover from it or optimize the code with an alternative approach. The *apply* method always calls *validate* first, and, if successful, proceeds to perform the tree modifications in-place. See Section 4 for an example of the use of transformations.

3.4 Dependence Analysis

Many transformations use dependency analysis for their validation. The dependence analysis implemented in PSyclone consists of two phases. First, the PSyIR is traversed and each variable access in the region of interest is categorized as a read, write, or read-write access. If the sub-tree contains a (DSL) Kernel, the access information from the Kernel metadata (see 5.1) is used to determine how the parameters are used. Once the variable information is captured, these are used to generate DefinitionUseChains and loop independence analysis as described in Kennedy and Allen (2001). The PSyclone implementation uses SymPy (Meurer et al., 2017) to symbolically compare expressions and improve the analysis.

Users have the option to selectively ignore symbols in the dependency analysis. This is useful when constraints not expressed in the source code can help in applying optimizations to the code. For example, knowing that an indexing map does not contain repeated indices, or that a particular dimension is extruded and does not contain data dependencies on that direction can assist in determining that a loop is safe to parallelize.

4 Using PSyclone to transform existing Fortran applications

As discussed in the Introduction, there is a large ecosystem of existing weather and climate models written in Fortran. These represent a huge investment of effort and scientific knowledge, therefore, it is desirable to build on this investment by having the ability to adapt their source code to make use of the modern developments in the HPC landscape. This is difficult to do

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

manually because these models are large and typically have a flat performance profile, meaning that any optimization must be applied to the vast majority of the code base.

PSyclone enables HPC experts to apply optimizations that they would normally do manually across a whole code base in an automated fashion. To date, this approach has been used or trialed with various models including NEMO (Madec et al., 2026), CROCO (Auclair et al., 2022), MEDUSA (Yool et al., 2013), NEMOVAR (Waters et al., 2015) and UKCA (Waters et al., 2015).

For these applications, one prepares a set of PSyclone transformations in a python script, which we call a *recipe*. All PSyclone recipes have the same basic structure: at minimum they must contain a function called `trans` which takes a single object containing the PSyIR of the file being processed and modifies it in-place. A simple example of such a recipe script is shown in Listing 1.

```python
def trans(psyir: FileContainer):
    ''' Add OpenMP Target and Loop directives to each outermost loop.

    :param psyir: the PSyIR of the provided file.
    '''
    # Use `stop_type=Loop` to ensure we traverse outermost loops only
    for loop in psyir.walk(Loop, stop_type=Loop):
        try:
            OMPLoopTrans(omp_directive="teamsloop").apply(loop, collapse=True)
            OMPTargetTrans().apply(loop.ancestor(Directive))
        except TransformationError as err:
            # Cannot be transformed, add a comment and proceed to next loop
            loop.preceding_comment = f"Loop not parallelized because: {err}"
```

Listing 1. The `trans` function of an example PSyclone recipe that attempts to add OpenMP offloading for every loop (nest) in the supplied PSyIR.

The recipe can be applied to each Fortran file in an application by inserting the `psyclone` command, as shown in Listing 2, into the application build system as a preprocessing step to the actual compilation.

```
$ psyclone -s transformation_recipe.py -o output.f90 input.f90
```

Listing 2. Example application of PSyclone to an existing Fortran source file.

For example, given the Fortran code in Listing 3, and applying the transformation in Listing 1 results in the creation of the Fortran shown in Listing 4.

```fortran
DO ji = 1, n
   DO jj = 1, m
      factor = 0.5*log(ji)
      var(ji,jj) = factor * (fld(ji-1, jj) + fld(ji+1, jj))
   END DO
END DO
DO ji = 1, n
```

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026


```
      var(map(ji), 1) = var(ji, 1)
    END DO
```

Listing 3. Some example Fortran input.

```
    !$omp target
    !$omp teams loop collapse(2) default(shared) private(factor,ji,jj) schedule(auto)
    do ji = 1, n, 1
      do jj = 1, m, 1
        factor = 0.5 * LOG(ji)
        var(ji,jj) = factor * (fld(ji - 1,jj) + fld(ji + 1,jj))
      enddo
    enddo
    !$omp end teams loop
    !$omp end target

    ! Loop not parallelized because:
    ! Error: The write access to 'var(map(ji),1)' [...] and the read access to 'var(ji,1)' [...] are
    dependent and cannot be parallelized. Variable: 'var'.
    ! Consider using the "ignore_dependencies_for" transformation option if this is a false
    dependency
    ! Consider using the "array_privatisation" transformation option if this is a write-write
    dependency
    do ji = 1, n, 1
      var(map(ji),1) = var(ji,1)
    enddo
```

Listing 4. The result of using PSyclone to apply the recipe from Listing 1 to the Fortran in Listing 3. The first loop has been successfully offloaded while the second cannot be transformed due to a possible dependency. Suggestions to parallelize the second loop are added as comments.

One can write a different recipe for each platform, or even for each specific compiler, to take advantage of the programming model and features available for that particular tool chain. This includes potentially working around known compiler limitations or inserting compiler-supported directives. Since every established model that makes use of HPC will almost certainly already have an optimized distributed memory implementation, there is no immediate need for PSyclone to handle distributed memory explicitly when processing existing code. Therefore, PSyclone simply treats any calls to data-exchange routines as any other subroutine call. This leaves the onus on the model developer to understand when halo exchanges are required and when they can be safely omitted. However, PSyclone can be used to optimize the per-node performance by reorganizing loops, inlining calls, applying shared-memory parallelization or offloading computations to accelerator devices.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

4.1 Evaluation with the NEMO Tracer-Advection Benchmark

The original version of the tracer-advection benchmark was constructed by Silvia Mocavero of the Euro-Mediterranean Center on Climate Change (CMCC) as part of the IS-ENES2 Project. It is a stand-alone program (single source file) based on the MUSCL tracer-advection scheme as implemented in NEMO version 4. The only significant differences to the original NEMO version are that all halo exchanges have been removed and synthetic data is used to populate all input fields. Once the computation is complete, the updated field is written to disk (to enable validation). Note that this benchmark code also forms the core of one of the ESCAPE-2 Weather and Climate Dwarfs (Behrens et al., 2021) where it is embedded within a more generic routine to emulate time-stepping behavior.

This is a good benchmark for PSyclone for transforming existing code because, as mentioned above, in this mode it does not currently attempt to modify any distributed-memory implementation and the tracer advection (the propagation of quantities such as salinity) within NEMO is a relatively expensive part of the model. The benchmark itself is contained within the *PSycloneBench* repository (Porter et al., 2026).

Benchmarking Setup

To demonstrate the performance portability capabilities of writing multiple psyclone transformation scripts, a range of diverse architectures and compilers from different vendors have been targeted:

- A dual-socket Intel Xeon Gold 6448Y CPU, from the Mary Coombs HPC system hosted at the Hartree Centre. Compiled with Intel ifx 2025.2.1 with the following flags: `-O3 -fopenmp`. Thread-to-core binding was controlled by setting `OMP_PLACES` to `core` and `OMP_PROC_BIND` to `close`. The result shown fully uses a single socket.
- A dual-socket AMD EPYC 9175F, from a Hartree Centre internal workstation. Compiled with gcc 16.1.0 with the following flags: `-Ofast -mtune=native -ffast-math -fopenmp`. Thread-to-core binding was controlled by setting `OMP_PLACES` to `core` and `OMP_PROC_BIND` to `close`. The result shown fully uses a single socket.
- A discreet H100 SMX GPU, from the Mary Coombs HPC system hosted at the Hartree Centre (only 1 of the 4 GPUs in the node). Compiled with nvfortran 25.5 with the following flags: `-O3 -gpu=cc70,managed` and either `-mp=gpu` or `-acc=gpu`. For the "explicit memory" results the `managed` flag was omitted.
- A MI300A APU, from AMD's internal AAC7 cluster [1]. Compiled with the AMD ROCM Afar 22.2.0[2] with the flags: `-lomp -lomptarget -fopenmp=libomp -fno-lto -offload-arch=gfx942 -mp-fopenmp-offload-mandatory -fopenmp-force-usm -lflang_rt.hostdevice -fopenmp=libomp`.

Benchmark Configuration

Beginning with a source code describing only the serial version of the benchmark, PSyclone has been used to create several different parallel versions:

[1] Development cluster not intended for large-scale performance assessment.
[2] Pre-release version of the amdflang compiler, which is still in active development. The current performance may not be indicative of the final release.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

1. OpenMP threading version;
2. OpenMP offloading version on GPU;
3. OpenACC kernels version on GPU;
4. OpenACC loops version on GPU with and without explicit memory movement directives;

Each recipe logic can differ according to the capabilities of each programming model and, crucially, the capabilities of the target compilers. For instance, NEMO encourages the use of Fortran Array Assignment Statements, which allow array operations to be expressed in a readable syntax without an explicit loop construct. In the OpenACC recipes, we chose to insert an *OpenACC kernels* directive around these statements. However, the support for this functionality in OpenMP is limited, so in the OpenMP recipe we chose to first convert these statements to explicit loops with the `ArrayAssignment2LoopTrans` transformation, and then apply as OpenMP offloading logic as demonstrated in Listing 1.

Each of the GPU versions of the benchmark, along with the Intel CPU version, have been run for a square horizontal domain size of 512. The number of vertical levels has been held constant at 75 since this is the value used in the ORCA25 and ORCA12 configurations of NEMO. The AMD CPU benchmark uses a square horizontal domain size of 1024, as the 512 squared benchmark could fit into the cache of the CPU.

To normalize the results these are presented as the percentage of measured memory bandwidth achieved in each platform. The bandwidth of an application is calculated by diving the solver elapsed time by the memory footprint on that region, which is based on the number of reads and writes to different arrays summed over the size of iteration length of each loop. Finally this is divided by the platform memory bandwidth, measured using the BabelStream Triad benchmark Deakin et al. (2018).

The results of the 7 variants are shown in Figure 3. The performance of all examples is at least 45% of the measured Stream Triad bandwidth available, with the highest GPU run (NVIDIA OpenACC kernels) reaching over 85%.

4.2 Limitations of the code transformation approach

There are limits to what can be achieved when working with existing code, and in our work to date we have encountered several limitations. Some of them are related to the performance achievable with code transformations alone:

- When an application contains significant computation that follows a pattern that is not dominated by array-based computation. For instance, the iceberg modelling component of NEMO v.5.0 is currently implemented using a linked-list data structure which cannot be parallelized and makes poor use of a machine's memory hierarchy. In this case a solution needs to be co-designed with the model owners.
- When an application contains significant regions of data-parallel work that are exposed at different levels in a call hierarchy, there is no single level that expresses enough parallelism. Achieving good GPU performance for such a model can require significant manual refactoring (Zhang et al., 2021).

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

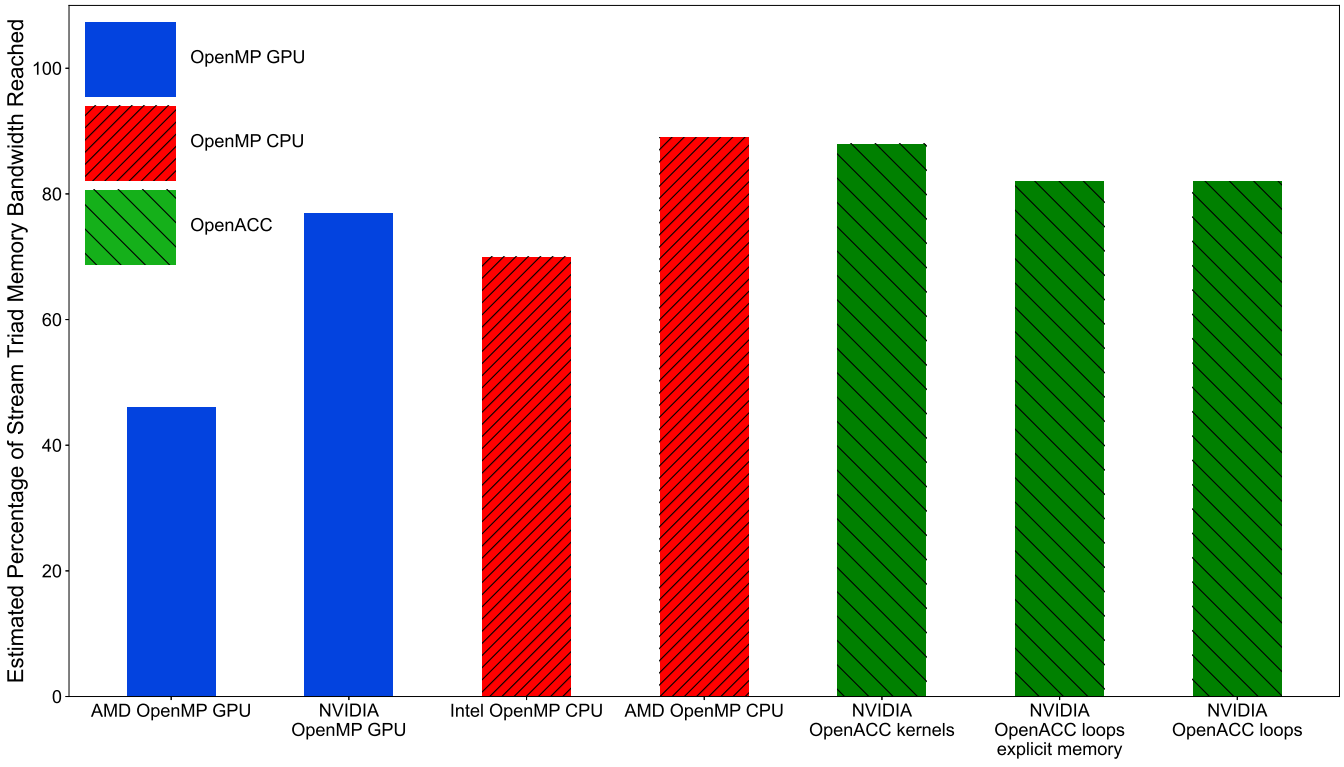

Figure 3. Estimated (see text) percentage of Stream triad memory bandwidth achieved by the tracer advection benchmark on different platforms with different code transformation recipes applied with PSyclone. The AMD CPU result shown here uses a larger problem size than the other tests shown to avoid excessively high results due to its large cache size.

– When a code has previously been optimized for a particular architecture. For example when a conditional computation must be done, some codes have previously introduced optimizations for caching architectures that gather the data into a packed list, perform the computation and then scattering the results back to their original locations. On a highly parallel architecture such as a GPU, the cost of the gather and scatter operations outweighs that of the actual computation, and simply computing everywhere and masking the results is more performant. We suggest that the best strategy is to undo manual optimizations that are hard for transformation scripts to undo, and let different optimization scripts prepare the code for each architecture (e.g. with a packing and a masking transformation). This can both improve the readability of the science code and also enable better performance portability.

In addition to the performance limitations, this approach can also suffer from what might be termed *performance-portability fragility*. This can occur when a transformation script looks for a specific pattern in the input code in order to apply its transformation(s). If a change is made to the input code that breaks this pattern then one or more transformations may not be applied. Even though the resulting code is still correct, there may be a large performance penalty. It is therefore important that transformation scripts be as general as possible. If some specificity is unavoidable and performance is known to be critical, the failure

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

to apply a transformation could be treated as an error. On top of this, we strongly advocate for the use or extension of existing test suites to monitor application performance as well as correctness.

The limitations arising from working with existing, unconstrained code can also be tackled by re-writing that code to make use of a DSL. In the next section we describe PSyclone's support for this type of approach.

5 Using PSyclone to support Fortran kernel-based embedded DSLs

A solution to the fragility of applying user-transformations to any arbitrary input code is to enforce the science code to always follow a defined pattern, thus, creating Domain Specific Languages (DSLs) catering to the needs of the application. One pattern for structuring parallel code in a robust way is to describe each individual unit of compute that can be run concurrently, together with the dependencies between them. A global description of the algorithm then implements the control flow and the iteration space over which the units of computation must be executed. Such an architecture is often called a kernel-based parallelism model, and it has proven successful in many well-established parallel computation frameworks (e.g. CUDA, OpenCL, SyCL, Kokkos).

PSyclone supports the implementation of applications using a Fortran-embedded kernel-based programming model, called PSyKAl (Adams et al., 2019). This model distinguishes between three layers: the Algorithm layer, the Kernel layer, and the Parallelization System (PSy) layer; which together give the model its name.

The Algorithm layer is responsible for providing a logically-global (i.e. without reference to whether or not the domain has been decomposed) description of the algorithm while the Kernel layer specifies individual units of compute operating on a single item within the domain. Depending on the DSL, these items can be a single element, a vertical column, or a set of vertical columns. The kernels also contain metadata that allow the data dependencies between kernels to be determined.

PSyclone reads the Fortran files containing the Algorithm and its associated Kernel-layers and generates an initial version of the code for the PSy layer. This layer acts as a bridge between the algorithm and kernels and contains the code that is responsible for performing concurrent execution of kernels, including the way in which the iteration space is traversed while respecting kernel dependencies.

In this approach, a transformation recipe is applied to the PSy layer. Listing 5 shows an example PSyclone command for a PSyKAl application:

```
$ psyclone --psykal-dsl lfric -s recipe.py -opsy psy_layer.f90 -oalg alg_layer.f90 algorithm.f90
```

Using the transformation recipe, the scheduling order of kernels can be modified and parallelization directives introduced to enable execution for specific platforms such as multi-node, multi-core and GPU architectures. Additionally, it is possible for scripts to also access the algorithm layer and the kernel implementation for additional code analyses or optimization.

In this model each layer has well defined responsibilities, providing a more robust separation of concerns between the domain science code and the parallelization and optimization logic. This is shown in Figure 4.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

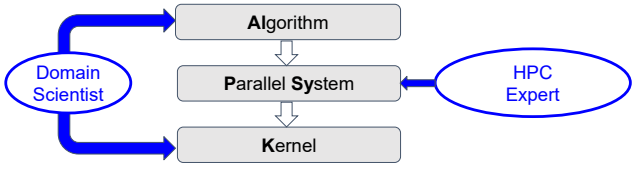

Figure 4. The Parallel System, Kernel, Algorithm (PSyKAl) Separation of Concerns.

5.1 The LFRic DSL

PSyclone's LFRic DSL has been developed for the Met Office's next-generation Weather and Climate model of the same name (Adams et al., 2019). It supports models that use a mixed finite-element scheme with a grid that is directly-addressed in the vertical and indirectly-addressed in the horizontal.

In LFRic, kernels are written to operate on either a single, vertical column of cells or on individual degrees-of-freedom (DoFs) of a field. Such kernels may be passed fields, operators or scalars. Fields and operators are associated with function-spaces, labeled "W1", "W2", "W3", "Wchi" etc. Adams et al. (2019); UK Met Office (2026b). The access pattern for each argument is also specified. Listing 5 provides an example of LFRic kernel metadata showing how some of these elements are specified. Note that the metadata is itself a valid Fortran derived type and contains a reference to the subroutine containing the implementation of the kernel. For the full specification of the LFRic Kernel metadata, please see the PSyclone User Guide Ford et al. (2025).

```
module apply_variable_hx_kernel_mod
  ! boilerplate code omitted ...
  type, public, extends(kernel_type) :: apply_variable_hx_kernel_type
    private
    type(arg_type) :: meta_args(9) = (/                              &
         arg_type(GH_FIELD,    GH_REAL, GH_WRITE, W3),               &
         arg_type(GH_FIELD,    GH_REAL, GH_READ,  W2),               &
         arg_type(GH_FIELD,    GH_REAL, GH_READ,  Wtheta),           &
         arg_type(GH_FIELD,    GH_REAL, GH_READ,  W3),               &
         arg_type(GH_OPERATOR, GH_REAL, GH_READ,  W3,     W2),       &
         arg_type(GH_OPERATOR, GH_REAL, GH_READ,  W3,     Wtheta),   &
         arg_type(GH_OPERATOR, GH_REAL, GH_READ,  Wtheta, W2),       &
         arg_type(GH_OPERATOR, GH_REAL, GH_READ,  W3,     W3),       &
         arg_type(GH_SCALAR,   GH_REAL, GH_READ) /)
      integer :: operates_on = CELL_COLUMN
    end type
  contains
    procedure, nopass :: apply_variable_hx_code
  end type

```

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026


```
  subroutine apply_variable_hx_code( cell, nlayers,            &
                                     lhs, x, mt_inv, pressure, &
                                     ncell_3d_1, ...)
  ! Implementation omitted ...
  end subroutine apply_variable_hx_code
end module apply_variable_hx_kernel_mod
```

Listing 5. An example of Kernel metadata for the LFRic API. The type, access and function space of each argument to the kernel is specified using an `arg_type` constructor.

The Algorithm layer contains a description of which kernels to call and which fields, operators or scalars they act upon through the specification of calls to a special `invoke` routine. This routine is purely a DSL construct and is not implemented anywhere. Listing 6 gives an example of one of the most computationally-costly `invoke` calls in the GungHo dynamics part of LFRic 2.1. The kernels are supplied to this `invoke` call in the form of calls to the structure constructors corresponding to their metadata.

```
call invoke( setval_c( grad_p, 0.0_r_solver ),                          &
             setval_c( y_vec, 0.0_r_solver ),                           &
             scaled_matrix_vector_kernel_type(grad_p, x_vec,            &
                                              div_star, hb_lumped_inv,  &
                                              u_normalisation),         &
             enforce_bc_kernel_type( grad_p ),                          &
             apply_variable_hx_kernel_type(y_vec, grad_p,               &
                                           mt_lumped_inv, x_vec,        &
                                           compound_div, p3theta, ptheta2, &
                                           m3_exner_star, one) )
```

Listing 6. LFRic Algorithm-layer invoke call involving the Kernel described in listing 5.

5.2 LFRic Transformations

Initially, the PSy-layer that PSyclone generates uses kernel dependency analysis (using the kernel metadata information) to determine the halo exchanges that are definitely required and adds run-time setting of `dirty` flags (contained within an LFRic field object) to keep track (between separate `invokes`) of what other field halos will need to be communicated for safe execution of the code. In the DSL approach, the PSyclone transformation script is applied to the PSy-Layer. PSyclone comes with a set of transformations that target PSy-layer code. A typical production build of LFRic uses the following:

1. The `LFRicRedundantComputationTrans` increases the number of halo layers that are redundantly computed by each partition in exchange for decreasing the number of halo exchange communications needed during the simulation. In addition, the `annexed dofs` parameter in the PSyclone configuration file can also be set to redundantly compute any shared dofs at the edge of the partition, further reducing the number of halo exchanges required.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

2. The `LFRicColourTrans` splits the iteration space into multiple "colors" for those loops whose iterations are not independent from each other (due to updates to shared dofs). After the transformation, each "color" contains a set of independent iterations that can be computed in parallel with each other, but different colors must be run in sequence. The colors can currently be individual cells (columns), or tiles of cells of the same color.

3. The `LFRicOMPParallelLoopTrans` (typically applied in combination with the `LFRicColourTrans`) adds OpenMP directives to parallelize any loop that has independent iterations. The transformation has parameters to select different OpenMP schedules and a parameter to enable run-to-run reproducible results (by implementing an alternative to the native OpenMP reduction support), which is a capability sometimes needed in certain simulations that the MetOffice runs. Note that the locality (shared or private) of variables do not need to be specified here, as PSyclone handles this internally.

The next section will evaluate the performance impact of transformation (1) and the combination of (2 and 3). However, PSyclone's LFRic DSL also has support for the following transformations that will not be evaluated in this paper:

- `LFRicAsyncHaloExchangeTrans` to split a halo exchange into a "start" call and a "waiting-for-completion" call. Subsequently, the transformation script can move these apart in order to overlap the time spent in communication with other computation available in the same `invoke` routine.
- `LFRicLoopFuseTrans` to fuse loops for adjacent kernels that do not have inter-dependencies.
- `LFRicKernelConstTrans` to substitute runtime-invariant variables inside kernels with their literal values. Examples include the number of dofs of a particular function space, the number of vertical layers in the mesh or the number of quadrature points. This trades generality of the resulting executable for enabling the compiler to make better decisions about how to optimize certain inner loops.
- Inlining transformations for Fortran intrinsics (such as `Matmul2CodeTrans`) to replace intrinsic operations with native code. This enables limitations in compiler implementations (whether performance or correctness) to be overcome and also permits further optimization of the source by subsequent transformations.

5.3 Evaluation of LFRic Performance and Scalability

The time-step of the dynamical core of LFRic (known as Gung Ho) is the performance-critical part of most LFRic applications. It has also been co-designed with PSyclone using the full DSL approach. The scaling behaviour of distributed-memory parallelism expressed over MPI and shared-memory parallelism expressed over OpenMP is reported in Bull et al. (2025). The time-step shows good scaling with both shared and distributed memory. Those results are not reproduced below. Instead, this section examines the impact of some of the PSyclone transformations described in the previous section. In particular, those of Redundant Computation (1), with annexed dofs enabled; the insertion of Coloring (2) and OpenMP parallelism (3); and a combination of all these transformations together.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

Table 1. The time taken for 384 time-steps on a C256 cubed-sphere mesh, in seconds, for different resource configurations and optimization choices and the percentage speed-up, RC_S, of the optimized case over the non-optimized case. Also shown are the sizes of the interior region and boundary per MPI rank and OpenMP thread in terms of the number of columns.

Nodes	MPI Ranks per node	Threads per rank	No Redundant (s)	Redundant (s)	RC_S(%)	Interior Columns (N_{col}^{Int})	N_{col}^{Int} per thread	Boundary Columns (N_{col}^{Bnd})	N_{col}^{Bnd} per thread
3	128	1	340.1	332.4	2.26	32×32	1024	132	132
	64	2	330.7	323.0	2.34	64×32	1024	196	98
	32	4	323.0	315.9	2.20	64×64	1024	260	65
12	128	1	80.8	77.3	4.28	16×16	256	68	68
	64	2	76.5	72.3	5.53	32×16	256	100	50
	32	4	72.8	68.8	5.39	32×32	256	132	33
48	128	1	29.2	27.5	5.79	8×8	64	36	36
	64	2	26.9	24.8	7.89	16×8	64	52	26
	32	4	26.1	23.9	8.39	16×16	64	68	17

The `gungho_model` application, from LFRic Apps, version 2.1, is used to test the performance of the different optimizations. This runs a deep baroclinic wave test on a C256 cubed-sphere mesh. The mesh has 256×256 cells on each of the six panels of the cube. The model has 30 levels in the vertical. Thus the total number of cells is $256 \times 256 \times 6 \times 30 = 11796480$. This mesh corresponds to a horizontal resolution of approximately 35-40 km.

The elapsed time, in seconds, for running 384 time-steps of the model is shown in Table 1. The data was produced from the Met Office HPE EX systems with AMD EPYC Milan 7763 processors. The code was compiled with the Cray compiler, CCE version 15.0.0 with high levels of optimization (`-O3`). Each Milan node is dual socket and each socket has 64 cores. With 1 OpenMP thread, the model runs with 64 MPI ranks per socket. With 2 OpenMP threads, 32 MPI ranks per socket, and with 4 OpenMP threads, 16 MPI ranks per socket. Thus, the number of CPU cores is fixed, but the ratio of MPI ranks to OpenMP threads changes. The affinity of MPI ranks and OpenMP threads is set such that the OpenMP threads are bound to CPU cores within a single chiplet on the AMD socket, thus preventing NUMA effects between threads for the same MPI rank. The number of nodes (3, 12, 48) controls the strong scaling and corresponds to 384, 1536 and 6144 CPU cores, respectively.

In Table 1, "No Redundant" refers to the default distributed-memory implementation generated by PSyclone and "Redundant" refers to code generated with the Redundant Computation transformation (together with the "annexed dofs" configuration option). For the No Redundant case, the halo exchange routine was called a total of $206,571$ times in a single model run while in the Redundant case this fell to $142,539$: a 31% reduction. *N.B.* for a fixed number of MPI ranks, the number of halo exchanges, and thus MPI calls, is independent of the number of threads.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

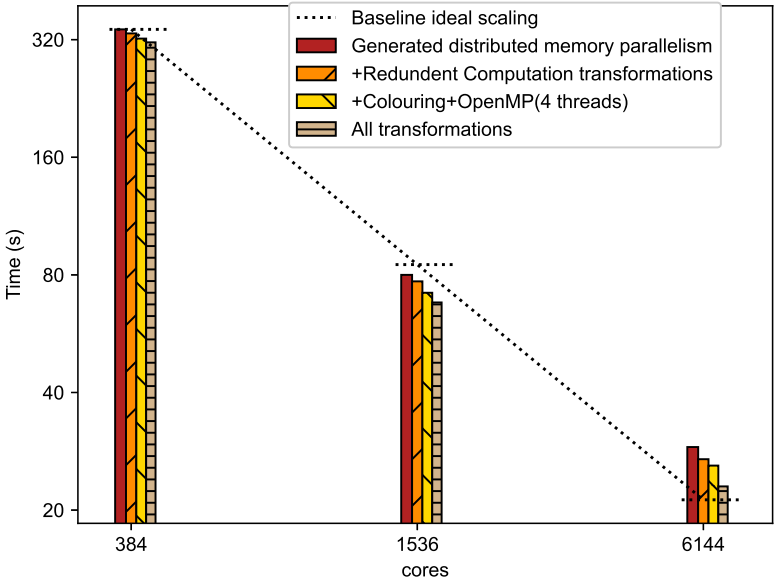

Figure 5. The effect of PSyclone optimizations on the run time of the deep baroclinic wave test with a C256 cubed-sphere mesh on the Met Office HPE EX system. The default, PSyclone-injected distributed memory parallelism achieves a Strong Parallel Scalability of $\times 11.6$ from 384 to 6144 ($\times 16$) cores (72.5% parallel efficiency). The super-linear scaling between 384 and 1536 cores is likely due to the smaller local volume fitting into cache. Applying all the transformations and running with four threads improves the per-core performance by $\times 1.22$ for the 6144 core case, achieving a Strong Parallel Scalability of $\times 13.2$ (82.5% parallel efficiency). The dashed line shows ideal scaling.

The percentage speed-up of applying the redundant-computation transformation, $\mathrm{RC_S}$, can be defined as $1 - \frac{T_{\mathrm{Redundant}}}{T_{\mathrm{NoRedundant}}}$ and this metric is also shown in Table 1. This data makes it clear that, for a given combination of ranks and threads, applying redundant computation is always beneficial. For all local volumes and cases (optimized and not optimized), the runs with four OpenMP threads perform best and those with one OpenMP thread the worst. The strong scaling data for the one- and four-thread cases is also plotted in Figure 5.

Although redundant computation significantly reduces the number of halo exchanges, this has to be traded against the increased computation that each thread must perform, as a percentage of its overall workload. This increased computation is governed by the size of the halo region. Figure 6 illustrates the conceptual horizontal layout of an example of a single MPI sub-domain. (Conceptual because LFRic uses an unstructured horizontal mesh and the actual data arrays are one dimensional.) This example has $\mathrm{N_x} = \mathrm{N_y} = 5$ and a halo of depth one. The diagram also shows how the cells (columns in 3D) are "colored" so as to avoid race conditions when updating shared DoFs using shared-memory parallelism. Only cells of the same color may be updated in parallel: the loop over colours is sequential. In this small example there are $\mathrm{N_{col}^{Int}} = \mathrm{N_x} \times \mathrm{N_y} = 25$ internal columns and $\mathrm{N_{col}^{Bnd}} = 2 \times (\mathrm{N_x} + 1) + 2 \times (\mathrm{N_y} + 1) = 24$ boundary or halo columns. Although halo exchanges are performed

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

by individual MPI processes, the redundant computation in the halo region is parallelized over OpenMP threads. Therefore, by design, the use of OpenMP threading improves the efficiency of the redundant-computation optimization.

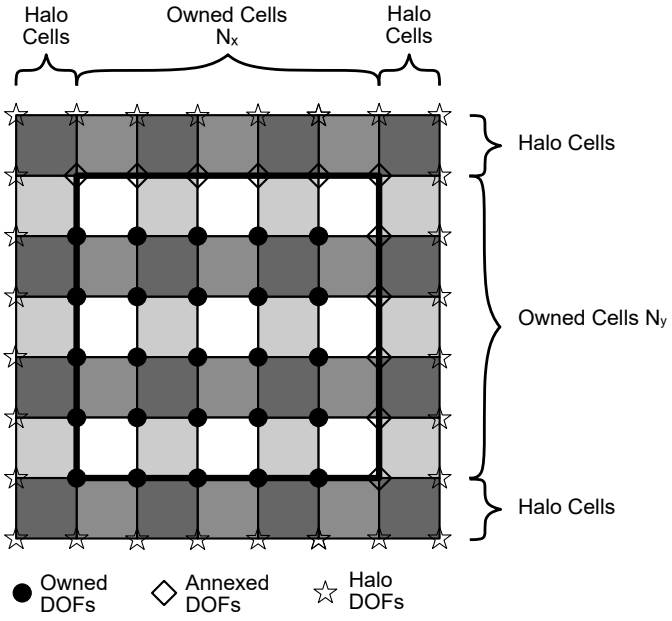

Figure 6. A single, 5×5 LFRic MPI sub-domain with a halo of depth one. Illustrative DoFs located on cell vertices are shown as small symbols. The "coloring" scheme used to prevent OpenMP race conditions while updating shared DoFs is indicated with shading (in LFRic four distinct "colors" are used).

The strong-scaling behavior shown in Figure 5 can therefore be explained as follows: as the number of nodes increases (and the MPI sub-domain size decreases), the relative cost of communication compared to computation increases. Thus the use of redundant computation to reduce the number of communication calls becomes relatively more effective. In the large number of nodes regime, the cost of redundant computation is relatively higher because of the small internal domain size (Table 1, but this is mitigated by parallelizing it over OpenMP threads.

6 Conclusions

This paper has introduced the open-source PSyclone tool for code generation and transformation and demonstrated its use in achieving performance portability for typical weather and climate applications. The code-generation and transformation capabilities enable machine- or tool-chain-specific code to be created as required at build time, thus avoiding pollution of the scientific code base. PSyclone supports two use cases:

1. code generation and optimization for models written in a DSL and following the PSyKAl separation of concerns;

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

2. code transformation for existing models that do not use a DSL.

Currently, PSyclone supports two flavours of DSL named "LFRic" and "GOcean". The LFRic DSL has been co-designed with the UK Met Office as part of the next-generation atmosphere model of the same name (Adams et al., 2019). The GOcean DSL has been used with the MOST Tsunami model (Henrichs et al., 2025). PSyclone's code-transformation capabilities are being used or trialled with existing models that have a large but homogeneous code base such as NEMO, CROCO, MEDUSA and UKCA. The application of PSyclone to transform existing models such as NEMO will be the subject of a future publication.

In all cases, PSyclone constructs the PSyIR of the code being processed. The PSyIR is designed to capture the compute aspects of a code while being language independent and mutable. It acts as the common target for all transformations. A variety of PSyIR transformations have been implemented, including, but not limited to: redundant computation to avoid halo exchanges, addition of OpenACC directives for expressing parallelism and data movement, addition of OpenMP directives (both for CPU and for offload to an accelerator), module-inlining of kernel code, loop fusion and code instrumentation. An HPC expert interacts with the PSyIR by constructing one or more transformation scripts in Python that use these transformations.

To date, development of the LFRic model and the associated PSyclone support has focused on scientific functionality and the ability to make use of the Met Office's existing, CPU-based machines. As such, PSyclone is used to add both distributed- and shared-memory parallelism to LFRic but support for OpenACC/OpenMP offload (so as to target GPUs) in this DSL is in its infancy. Currently PSyclone is able to generate functionally-correct OpenACC code for LFRic but more work is required to make this performant.

In contrast, PSyclone's support for OpenACC and OpenMP when transforming existing code is relatively mature and we have demonstrated that here with the NEMO tracer-advection benchmark. Distributed-memory parallelism currently relies entirely upon the functionality already present in the code being processed. Since this is an overhead on the scientific developer and a potential source of error, we plan to extend PSyclone so as to configurably recognize existing halo exchange calls and then build on PSyclone's existing dependence-analysis functionality to determine whether or not they are required (or indeed whether any are missing).

Author contributions. Rupert Ford was the original developer of PSyclone and wrote a lot of an early draft of this paper. Andrew Porter, Sergi Siso, Aidan Chalk and Joerg Henrichs are the core PSyclone developers and have all contributed to the text describing the implementation and use of PSyclone. Chris Maynard and Iva Kavčič lead the co-design process from the Met Office side and are heavily involved in the development of the LFRic model. They have contributed to the text describing the LFRic API as well as to the performance-analysis section. Iva has also contributed significantly to the PSyclone implementation for the LFRic API.

Competing interests. One of the GMD executive editors, David Ham, was an investigator on the original Gung-Ho project that saw the inception of the PSyclone software described in this work. However, he is not the handling editor for this submission.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

7 Code availability

PSyclone is available under a 3-clause BSD license and has a DOI of 10.5281/zenodo.11190457 (Ford et al., 2025). Note that while this DOI is for all versions, the DOI referenced by the citation refers specifically to version 3.1. All versions are also available from https://github.com/stfc/PSyclone/releases.

PSycloneBench is also available under a 3-clause BSD license and has a DOI of 10.5281/zenodo.20625745 (Porter et al., 2026). Again, the citation itself refers to the specific version used in this work. The achieved memory-bandwidths presented in figure 3 were computed from arithmetic intensity estimates calculated by the `memory_usage_estimate.py` script distributed with PSycloneBench (in `benchmarks/nemo/tracer_advection/scripts`).

LFRic is released under a 3-clause BSD license and version 2.1 is available from the GitHub repository here: https://github.com/MetOffice/lfric_apps/releases/tag/vn2.1. We used the "gungho_model" application in this work. To compile this model with the PSyclone transformation script that applies all transformations:

```
$ cd lfric_apps
$ ./build/local_build.py -j <num-procs> -p meto-ex1a -w working-directory gungho_model
```

For more details, see the "LFRic Apps Command Line Builds" section in the "Developer Guide" of (UK Met Office, 2026a). See also the "Mesh generation" section of (UK Met Office, 2026b) for instructions on using the mesh generator needed to construct the C256 mesh used here.

Acknowledgements. PSyclone was originally developed within the GungHo Project: a five-year programme (2011-2016) in the UK implemented by the Met Office, NERC and STFC, to research, design and develop a new dynamical core.

Subsequently, PSyclone development has been supported by the Met Office and by the UK's ExCALIBUR programme, which is funded by the UKRI Strategic Priorities Fund. The programme is co-delivered by the Met Office and EPSRC in partnership with the Public Sector Research Establishment, the UK Atomic Energy Authority (UKAEA), and UKRI research councils, including NERC, MRC, and STFC.

The GOcean API was begun with the NERC-funded one-year GOcean project (grant reference NE/L01209X/1).

Aspects of PSyclone development have also been supported by the ESiWACE2 project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements numbered 675191 and 823988.

AMD provided access to compute nodes with 4xMI300A devices.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

References

Adams, S., Ford, R., Hambley, M., Hobson, J., Kavčič, I., Maynard, C., Melvin, T., Mueller, E., Mullerworth, S., Porter, A., Rezny, M., Shipway, B., and Wong, R.: LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models, J. of Parallel and Distributed Computing, 132, 383–396, https://doi.org/10.1016/j.jpdc.2019.02.007, 2019.

Alnæs, M. S., Logg, A., Ølgaard, K. B., Rognes, M. E., and Wells, G. N.: Unified form language: A domain-specific language for weak formulations of partial differential equations, ACM Trans. Math. Softw., 40, https://doi.org/10.1145/2566630, 2014.

Auclair, F., Benshila, R., Bordois, L., Boutet, M., Brémond, M., Caillaud, M., Cambon, G., Capet, X., Debreu, L., Ducousso, N., Dufois, F., Dumas, F., Ethé, C., Gula, J., Hourdin, C., Illig, S., Jullien, S., Le Corre, M., Le Gac, S., and Theetten, S.: Coastal and Regional Ocean COmmunity model (1.3), https://doi.org/10.5281/zenodo.7415343, 2022.

Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., Kong, F., Kruger, S., May, D. A., McInnes, L. C., Mills, R. T., Mitchell, L., Munson, T., Roman, J. E., Rupp, K., Sanan, P., Sarich, J., Smith, B. F., Suh, H., Zampini, S., Zhang, H., Zhang, H., and Zhang, J.: PETSc/TAO Users Manual, Tech. Rep. ANL-21/39 - Revision 3.24, Argonne National Laboratory, https://doi.org/10.2172/2998643, 2025.

Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., Ryujin, B. S., and Scogland, T. R.: RAJA: Portable Performance for Large-Scale Scientific Applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 71–81, https://doi.org/10.1109/P3HPC49587.2019.00012, 2019.

Behrens, J., Osuna, C., Duras, J., Epicoco, I., Budich, R., and Korn, P.: Demonstration of Domain Specific Language Toolchain for Selected Weather and Climate Dwarfs, Report Deliverable D2.4, ESCAPE-2 Consortium, https://hpc-escape2.eu, european Union's Horizon 2020 Research and Innovation Programme, Grant Agreement No. 800897, 2021.

Ben-Nun, T., de Fine Licht, J., Ziogas, A. N., Schneider, T., and Hoefler, T.: Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, New York, NY, USA, ISBN 9781450362290, https://doi.org/10.1145/3295500.3356173, 2019.

Bertolli, C., Betts, A., Mudalige, G., Giles, M., and Kelly, P.: Design and Performance of the OP2 Library for Unstructured Mesh Applications, in: Euro-Par 2011: Parallel Processing Workshops, edited by Alexander, M., D'Ambra, P., Belloum, A., Bosilca, G., Cannataro, M., Danelutto, M., Di Martino, B., Gerndt, M., Jeannot, E., Namyst, R., Roman, J., Scott, S. L., Traff, J. L., Vallée, G., and Weidendorfer, J., pp. 191–200, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-642-29737-3, https://doi.org/10.1007/978-3-642-29737-3_22, 2012.

Bisbas, G., Lydike, A., Bauer, E., Brown, N., Fehr, M., Mitchell, L., Rodriguez-Canal, G., Jamieson, M., Kelly, P., Steuwer, M., and Grosser, T.: A shared compilation stack for distributed-memory parallelism in stencil DSLs, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 3, p. 38–56, https://doi.org/10.1145/3620666.3651344, 2024.

Bull, J. M., Coughtrie, A., Deeptimahanti, D., Hedley, M., Laoide-Kemp, C., Maynard, C., Shepherd, H., Van De Bund, S., Weiland, M., and Went, B.: Performance and scaling of the LFRic weather and climate model on different generations of HPE Cray EX supercomputers, in: Proceedings of the Cray User Group, CUG '24, p. 1–11, Association for Computing Machinery, New York, NY, USA, ISBN 9798400713286, https://doi.org/10.1145/3725789.3725790, 2025.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

Chen, C., Chame, J., and Hall, M.: CHiLL: A framework for composing high-level loop transformations, Tech. Rep. 08-897, University of Southern California, https://citeseerx.ist.psu.edu/document?doi=6a4620589f63f3385707d2d590f7b7dc8ee4d74f, 2008.

Clement, V., Ferrachat, S., Fuhrer, O., Lapillonne, X., Osuna, C. E., Pincus, R., Rood, J., and Sawyer, W.: The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models, in: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '18, Association for Computing Machinery, New York, NY, USA, ISBN 9781450358910, https://doi.org/10.1145/3218176.3218226, 2018.

Deakin, T., Price, J., Martineau, M., and McIntosh-Smith, S.: Evaluating attainable memory bandwidth of parallel programming models via BabelStream, International Journal of Computational Science and Engineering, 17, 247–262, https://doi.org/10.1504/IJCSE.2018.095847, 2018.

Dennard, R., Gaensslen, F., Yu, H.-N., Rideout, V., Bassous, E., and LeBlanc, A.: Design of ion-implanted MOSFET's with very small physical dimensions, IEEE Journal of Solid-State Circuits, 9, 256–268, https://doi.org/10.1109/JSSC.1974.1050511, 1974.

Ford, R., Glover, M., Ham, D., Maynard, C., Pickles, S., and Riley, G.: GungHo Phase 1: Computational Science Recommendations, Technical Report 587, Met Office, https://library.metoffice.gov.uk/Portal/Default/en-GB/RecordView/Index/197154, 2013.

Ford, R., Porter, A., Siso, S., Henrichs, J., and Kavcic, I.: PSyclone, https://doi.org/10.5281/zenodo.14925500, 2025.

Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., and Schulthess, T. C.: STELLA: a domain-specific tool for structured grid methods in weather and climate models, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, Association for Computing Machinery, New York, NY, USA, ISBN 9781450337236, https://doi.org/10.1145/2807591.2807627, 2015.

Henrichs, J., Junwei Lyu, J., Greenslade, D. J. M., and Allen, S. C. R.: Future-Proofing the MOST Tsunami Model Using the PSyclone Domain-Specific Language, Bureau Research Report 110, Australian Government Bureau of Meteorology, ISBN 978-1-923469-02-0, https://trove.nla.gov.au/work/260177317, includes bibliographical references (pages 25-26), 2025.

Kennedy, K. and Allen, J. R.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ISBN 1558602860, https://dl.acm.org/doi/book/10.5555/502981, 2001.

Klocke, D., Frauen, C., Engels, J. F., Alexeev, D., Redler, R., Schnur, R., Haak, H., Kornblueh, L., Brüggemann, N., Chegini, F., Römmer, M., Hoffmann, L., Griessbach, S., Bode, M., Coles, J., Gila, M., Sawyer, W., Calotoiu, A., Budanaz, Y., Mazumder, P., Copik, M., Weber, B., Herten, A., Bockelmann, H., Hoefler, T., Hohenegger, C., and Stevens, B.: Computing the Full Earth System at 1km Resolution, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '25, p. 125–136, Association for Computing Machinery, New York, NY, USA, ISBN 9798400714665, https://doi.org/10.1145/3712285.3771789, 2025.

Lange, M. and Reuter, B.: Loki: Freely programmable source-to-source translation, https://github.com/ecmwf-ifs/loki.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O.: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation, in: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 2–14, https://doi.org/10.1109/CGO51591.2021.9370308, 2021.

Lawrence, B. N., Rezny, M., Bauer, P., Behrens, J., Carter, M., Deconinck, W., Ford, R. W., Maynard, C., Mullerworth, S., Osuna, C., Porter, A. R., Serradell, K., Valcke, S., Wedi, N., and Wilson, S.: Crossing the chasm: how to develop weather and climate models for next generation computers?, Geoscientific Model Development, 11, 1799–1821, https://doi.org/10.5194/gmd-11-1799-2018, 2018.

Madec, G., Bell, M., Benshila, R., Blaker, A., Boudrallé-Badie, R., Bricaud, C., Bruciaferri, D., Carneiro, D., Castrillo, M., Calvert, D., Chanut, J., Clementi, E., Coward, A., de Lavergne, C., Dobricic, S., Epicoco, I., Éthé, C., Fiedler, E., Ford, D., Furner, R., Ganderton, J., Griffies, S. M., Graham, T., Harle, J., Hutchinson, K., Iovino, D., King, R., Lea, D., Levy, C., Lovato, T., Maisonnave, E., Mak, J.,

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

Sanchez, J. M. C., Martin, M., Martin, N., Martins, D., Masson, S., Mathiot, P., Mele, F., Mocavero, S., Moulin, A., Müller, S., Nurser, G., Oddo, P., Paronuzzi, S., Paul, J., Peltier, M., Person, R., Rousset, C., Rynders, S., Samson, G., Schroeder, D., Storkey, D., Storto, A., Téchené, S., Vancoppenolle, M., and Wilson, C.: NEMO Ocean Engine Reference Manual, https://doi.org/10.5281/zenodo.19206664, 2026.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A.: SymPy: symbolic computing in Python, PeerJ Computer Science, 3, e103, https://doi.org/10.7717/peerj-cs.103, 2017.

OpenACC Standard: The OpenACC Application Program Interface Version 3.3, https://web.archive.org/web/20260805102752/https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf, last access: 2026-08-05, 2022.

OpenMP Architecture Review Board: OpenMP Application Program Interface Version 5.2, https://web.archive.org/web/20260805102606/https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, last access: 2026-08-05, 2021.

Osuna, C., Wicky, T., Thuering, F., Hoefler, T., and Fuhrer, O.: Dawn: a High-level Domain-Specific Language Compiler Toolchain for Weather and Climate Applications, Supercomputing Frontiers and Innovations, 7, 79–97, https://doi.org/10.14529/jsfi200205, 2020.

Pennycook, S. J., Sewall, J. D., Jacobsen, D. W., Deakin, T., and McIntosh-Smith, S.: Navigating Performance, Portability, and Productivity, Computing in Science & Engineering, 23, 28–38, https://doi.org/10.1109/MCSE.2021.3097276, 2021.

Peterson, P., Ford, R., Porter, A., Siso, S., Chalk, A. B. G., and Henrichs, J.: fparser: a Fortran parser implemented in Python, https://doi.org/10.5281/zenodo.21804361, 2026.

Porter, A., Siso, S., Chalk, A., Ford, R., and Atkinson, V.: PSyclone Bench: benchmarks for the investigation of performance-portability in the earth-systems modelling domain., https://doi.org/10.5281/zenodo.21806332, 2026.

Porter, A. R., Appleyard, J., Ashworth, M., Ford, R., Holt, J., Liu, H., and Riley, G. D.: Portable multi-and many-core performance for finite-difference or finite-element codes–application to the free-surface component of NEMO (NEMOLite2D 1.0), Geoscientific Model Development, 11, 3447–3464, https://doi.org/10.5194/gmd-11-3447-2018, 2018.

Quinlan, D. and Liao, C.: The ROSE source-to-source compiler infrastructure, in: Cetus users and compiler infrastructure workshop, in conjunction with PACT, vol. 2011, p. 1, Citeseer, https://web.archive.org/web/20170830001555/https://engineering.purdue.edu/Cetus/cetusworkshop/papers/4-1.pdf, 2011.

Reid, J.: The new features of Fortran 2008, SIGPLAN Fortran Forum, 27, 8–21, https://doi.org/10.1145/1408643.1408645, 2008.

Siso, S., Porter, A. R., and Ford, R. W.: Transforming Fortran weather and climate applications to OpenCL using PSyclone., in: IWOCL '23: Proceedings of the 2023 International Workshop on OpenCL, 10, pp. 1–8, https://doi.org/10.1145/3585341.3585360, 2023.

The Khronos SYCL Working Group: SYCL 2020 Specification, Revision 11, https://web.archive.org/web/20260805102945/https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf, last access: 2026-08-05, 2025.

Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J.: Kokkos 3: Programming Model Extensions for the Exascale Era, IEEE Transactions on Parallel and Distributed Systems, 33, 805–817, https://doi.org/10.1109/TPDS.2021.3097283, 2022.

Ubbiali, S., Kühnlein, C., Schär, C., Schlemmer, L., Schulthess, T. C., Staneker, M., and Wernli, H.: Exploring a high-level programming model for the NWP domain using ECMWF microphysics schemes, Geoscientific Model Development, 18, 529–546, https://doi.org/10.5194/gmd-18-529-2025, 2025.

https://doi.org/10.5194/egusphere-2026-3844
Preprint. Discussion started: 19 August 2026

UK Met Office: LFRic Science Model Architecture, https://metoffice.github.io/lfric_apps/index.html, 2026a.

UK Met Office: LFRic Science Model Architecture, https://metoffice.github.io/lfric_core/how_it_works/core_data_model/science_model_architecture.html#lfric-function-spaces-and-element-orders, 2026b.

Waters, J., Lea, D. J., Martin, M. J., Mirouze, I., Weaver, A., and While, J.: Implementing a variational data assimilation system in an operational 1/4 degree global ocean model, Quarterly Journal of the Royal Meteorological Society, 141, 333–349, https://doi.org/10.1002/qj.2388, 2015.

Yool, A., Popova, E. E., and Anderson, T. R.: MEDUSA-2.0: an intermediate complexity biogeochemical model of the marine carbon cycle for climate change and ocean acidification studies, Geosci. Model Dev., 6, 1767–1811, https://doi.org/10.5194/gmd-6-1767-2013, 2013.

Zhang, W., Xu, M., Evans, K., Norman, M., Morales-Hernandez, M., Mahajan, S., Hill, A., Manners, J., Shipway, B., and Maynard, C.: Progress towards accelerating the unified model on hybrid multi-core systems, in: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '21, Association for Computing Machinery, New York, NY, USA, ISBN 9781450385633, https://doi.org/10.1145/3468267.3470612, 2021.