Towards Performance Portability and Modernization of FLASH via Transpilation with High-Level Intermediate Representation

(Unrefereed Workshop Manuscript)

Mateusz Bysiek[1,2,a] Saurabh Chawdhary[3,4] Mohamed Wahib[2]
Anshu Dubey[3,4] Satoshi Matsuoka[5,1,2]

Abstract:
With concurrent increase in application complexity and hardware heterogeneity, large multiphysics code FLASH faces huge challenges to its continued usability on high performance computing platforms. We are building a novel transpilation framework that relies on high-level intermediate representation to confront this challenge and enable FLASH to adapt to accelerator-based architecture via performance-oriented refactoring. Additionally, we use the framework to modernize code by enabling higher level of abstraction in expressing computations. We evaluate the effectiveness of the tool with respect to speedup obtained relative to original code performance, and also quantify productivity gains.

Keywords: high performance computing, performance portability

1. Introduction

Scientific computing has been presented with unprecedented opportunities and challenges in the recent years. With combined growth in model fidelity and numerical methods, significant growth in scientific insights seems within reach. But concurrent with that growth is the increasing complexity and heterogeneity in high performance computing (HPC) platforms. Large multiphysics codes have taken years to reach the stage of robustness and maturity that was all designed for a much simpler machine model; that of more or less homogeneous nodes with a few levels of memory hierarchy and interconnects between nodes for parallelism. Performance portability demanded maximizing locality and minimizing communication. Conflicting requirements from diverse solvers required for higher fidelity modelling strain even this simple machine model.

Current and future platform will achieve higher performance through higher degree of parallelism and accelerators in many different flavours. And each one will have its own preferred data layout and operation scheduling. When multicomponent codes meet platform heterogeneity the combinatorics of target specific optimization become intractable. We could list many examples of prominent scientific software that fit this profile. In this work, we focus primarily on FLASH [11], a highly capable multiphysics code with a wide user base in several science communities.

The challenge of exploding combinatorics of heterogeneous solvers and heterogeneous platforms has been known for a while and many projects have come into existence to mitigate it. These projects include domain-specific-languages (DSL) [1, 8, 9], tools exploiting template metaprogramming features of C++ to introduce abstractions [4, 14, 16], and new parallel languages, e.g. Julia. Of these the C++ tools have had the most success. However, there are many codes that have been written in Fortran which cannot make use of these tools; FLASH is one of them. Developing replacements for these codes in either C++ or new parallel languages would take several years of effort. In the meantime their continued viability on available HPC platforms is critical for many science domains. We have developed a transpiler framework specifically targeted at codes such as FLASH.

2. Background and Motivation

Our transpilation framework is relying heavily on Python. In this section we bring into some background as well as explain our motivation, and in later sections we rely on them to propose our solution.

[1] Tokyo Institute of Technology
[2] National Institute of Advanced Industrial Science and Technology
[3] Argonne National Laboratory
[4] University of Chicago
[5] RIKEN Center for Computational Science
[a] bysiek.m.aa@m.titech.ac.jp

© 2019 Information Processing Society of Japan

2.1 Python in HPC

Numerous solutions have been introduced into the Python ecosystem that aim at mitigating Python's performance issues. Below, we briefly introduce 3 widely employed approaches.

NumPy [23] is a BLAS-compliant numerical library for Python, which, when used correctly, can achieve very good performance. NumPy's approach to higher performance in Python is very straightforward. NumPy is partially implemented in Python, but it mostly consists of a big collection of low-level language implementations which are interfaced with Python through the CPython API. The implementations often contain manually unrolled loops to support SIMD compiler optimizations, or contain code that is preprocessed by the C compiler at installation time to generate the code. Those low-level implementation are very efficient, but are not useful in cases when they would need to be tailored for some specific use. NumPy cannot be used directly to accelerate any algorithm.

Numba [17] is a JIT compiler for Python. It does not compile Python directly, but instead translates it into LLVM Intermediate Representation, and delegates the compilation to the LLVM toolchain. Through an easy-to-use API, Numba enables JIT compilation of selected parts of Python code. The compiled code sections have to contain only a restricted subset of Python syntax. Additionally, Numba cannot shed all layers of indirection present in Python, because it is not capable of complete type analysis. There are inherent unavoidable problems with unavailability of reliable type inference in Python, and therefore if any elements of Python's dynamic indirection remain after transpilation, they significantly slow down the execution.

Cython [3] is a language derived from Python, and also a software solution that translates Cython language to C with certain extensions. Reason is performance, especially the case of numerical loops [25]. Cython can usually outperform NumPy in cases of construction of sparse matrices, data transformation, repacking, equation solving, among others [2]. Cython language is very similar to Python in a sense that a subset of Python is also valid Cython code. However, Cython extends Python syntax by adding few C-related constructs. This makes Cython code backwards-incompatible with Python – once the code is converted to Cython so as to benefit from its performance boost, it is no longer valid Python. Cython provides a so-called 'pure mode' via which the original Python code can be left untouched, and a separate file with static type information for that code needs to be created instead [2]. This additional file is ignored by Python interpreter, but used by Cython framework, which provides some level of compatibility, however this lowers maintainability of the code because two files have to be kept in sync manually. Cython framework, apart from providing application performance boost, incurs a significant compilation overhead, because Cython framework delegates the compilation of C code to an external compiler (as available in the system) [2], exactly as it is in the case of f2py.

From the above, 3 facts become evident. Python can efficiently invoke low-level implementations as it is in case of NumPy. Python is capable of introspection and self-alteration at the source code level during runtime, as it is with Numba. And thirdly, as seen in Cython, even if at runtime level Python is very far from optimal, at a language level it is not so far from its high-performing competitors like C.

Leveraging the above, together with Type Hints [32] mechanism available in modern Python, we came up with a concept of using Python language without modifying its syntax in any way to enable Fortran-level runtime performance on the in Python, and implemented a prototype [6]. At first, we experimented with simple algorithms and Fortran benchmark applications and achieved Python code execution performance equivalent to pure Fortran execution performance. Our prototype outperformed Numba in selected benchmarks. Additionally, our approach enabled bi-directional translation between small confined subsets of Python and Fortran, thus enabling migration of selected small Fortran-based benchmark applications into Python while preserving their performance. Later, we modified our approach and published our framework [5] (called Transpyle) on GitHub. We also reasoned about how the approach could be applied to full applications, however until now we haven't published any results related to larger codes. Here comes FLASH. In remaining subsections we introduce it, as well as explain why it is a good fit for experiments with our framework.

2.2 FLASH

FLASH is a multiphysics, multicomponent code that has been in development and use since 1998 [10], and serves several science communities [13]. The software architecture of FLASH [11] relies on inter-operating components that have multiple alternative implementations. An extremely limited configuration domain-specific-language (DSL) encodes meta-information about various components and their alternative implementations, and ways in which they can be combined with other components. This language is interpreted by the "setup" tool written in Python. Applications are configured by selecting specific implementations of a subset of components following the rules encoded by the DSL. Alternative implementations may exist either to provide different numerical methods for different physical regimes, or may be targeted to be performant on specific platform architectures. Note that the configuration DSL does not play any role in execution, it is an intelligent configuration tool that can provide the high degree of composability demands placed on FLASH by its diverse users.

Being a large code whose lifespan has covered several generation of largest HPC platforms, performance portability has always been critical for FLASH. Hand tuning

© 2019 Information Processing Society of Japan

the entire code base for every platforms has never been an option, however, targeting distributed memory machine model and good data locality gave sufficiently good performance as long as platforms remained largely homogeneous. Arrival of multicore platforms was managed with use of OpenMP directives with modest development efforts. However, these options are not sufficient for the upcoming platforms with increasing heterogeneity. There is no longer a general machine model that works across platforms. Devices on systems differ in terms of optimal data layout, data movement, and memory management.

For example, the new (e.g. Summit) and future systems (Aurora, Frontier) are not only expected to have accelerators, but also, the accelerators are expected to be different on different machines. In certain situations, where specific physics dominates the runtime, and lends itself to easy re-implementation, the obvious solution of writing optimized codes for each type of machine may be viable [24]. However, in vast majority of simulations using FLASH there are no such hotspots, and therefore, this approach is neither productive nor feasible for even a fraction of the code. We need a paradigm shift in the way scientific codes are designed and developed.

One possible option is to build upon the idea of separation of concerns, which has been the basis for FLASH design, wherein, the core structure of the code remains agnostic of the system architecture. And now, instead of relying upon locality to deliver good performance as in the past, we use a tool to generate target platform specific code. In the next section we describe how FLASH's architecture makes it amenable to this approach.

2.3 A Deeper Look Into FLASH

The key to using Transpyle (and transpilation in general) with FLASH lies in its separation between concerns of micro-parallelism and macro-parallelism. The macro-parallelism is managed at the level of adaptive mesh refinement (AMR), which tackles the load distribution and distributed memory parallelism. The physics solvers in the code are largely oblivious of this parallelism. The micro-parallelism, that pertains to shared memory, threading and vectorization etc. comes into play at the solver level. The physics solver units operate with the "block" abstraction; a section of the domain with surrounding halo is operationally indistinguishable from the whole domain. Vast majority of floating point operations involve a physics operator being applied separately on each such block. Thus it is best to employ transpilation in the code implementing the physics operators.

The operators themselves vary greatly in complexity, and there are two aspects to their complexity; logical and numerical. We focus on operators that have numerical complexity and intensity, because that is where performance bottlenecks are likely to occur. Since an important idea behind the use of Transpyle is to retain a readable code to maintain, we simplify the structure of the operators by breaking up the loop nests into function calls. The expectation is that the overhead of very fine grained function calls can be eliminated by inlining them, which is one of the simplest features in Transpyle. With this transformation, the implementation begins to resemble a sequence of function calls interspersed with some logic. This exercise serves two purposes; the logical flow of the calculation is clarified, and the semantics of calculation are encoded in smaller chunks more easily digested by any code transformation engine.

FLASH, having a vibrant and performance-conscious user community, have seen several attempts at adapting it to GPU-based machines [15, 19, 24], and now one of the members of the Exascale Computing Project, it is committed to adapt to emerging architectures.

2.4 Challenges

To address the challenge of adaptation, several approaches are available. First, one may chose to migrate legacy software such as FLASH to a new programming language, that attempts to solve the issues at the language level. However, it is hard to predict if a given programming language will enjoy a community large enough to sustain the project currently implemented in Fortran for the years to come. Additionally, a new programming language may be an option for a new project, but with large enough codebase full migration of an ongoing one is not an option.

Second, even in existing languages, there is sometimes an option to abstract out a lot of functionality by employing metaprogramming. Among widely-adopted HPC programming languages, it is possible only in C++. Also, by using templates, user loses control over, and potentially understanding of, what is going on.

Finally, directives such as OpenMP and OpenACC promise to enable adaptation of existing software to new architectures with little effort. However, critical features such as abstracting out the data layout, and inter-loop optimizations is outside of scope of directives. In addition, achieving high performance on different targets often requires non-trivial directive customization for each target in a programming style that introduces technical debt.

After extended experimentation on our early prototype [6] with larger codes and after facing significant challenges related to engineering effort required to translate large production-grade Fortran codes to Python, we changed our (in hindsight, unrealistic) goal of bi-directional transpilation using (a1) custom high-level intermediate representation to translate (b1) whole applications, to (a2) using extended Python AST as intermediate representation [7] to translate (b2) only a subset of application code.

3. Contributions

We propose a compiler-based method of adapting mutliphysics framework FLASH to modern architectures while also enhancing its maintainability.

© 2019 Information Processing Society of Japan

Specifically, we present a method that by employing our transpilation framework Transpyle, coupled with targeted code refactoring, enables obtaining both (1) more maintainable modularized FLASH Fortran codebase and (2) generated platform-tuned Fortran codebase that is still readable and can be reasoned about further.

4. Dive Into Transpyle

Relying on our experiences with the early prototypes we subsequently designed and implemented an open-source transpilation framework Transpyle [5] with applications that go beyond the scope of this paper. Here we will focus on its application to FLASH, however we will maintain more general perspective wherever it is useful.

4.1 Mapping Between Python and Target Languages

To translate between Python and any other language, we need to first define the mapping between Python and that language. In case of languages tackled so far in our work, Python, Fortran and C++, we focus on 3 areas in particular: (1) data types, (2) basic syntactic structures, (3) selected common idioms, (4) internal API calls, and (5) external APIs. In all of those cases, we take a pragmatic approach of looking at the types of codes that we would like to handle, selecting example benchmarks for translation correctness, and implementing just the functionality necessary for them to work. Similar approach is taken by developers of mypyc [30] where the authors aim at successful translation of Python to C for only a single Python package called mypy [18, 29].

In case of data types, the most important are fundamental types, and arrays in case of numerics. For basic syntactic structures, of course the various unary and binary operations, loops, and other elements encountered in even the simplest code are essential if the transpiler is not to give up on the most basic examples. Among those, many individual instances can be trivially passed through, and that what makes them basic. Idioms are often an answer to a question like "how to print to standard output stream" or "how to initialize an empty array", therefore transpiler needs to understand what calling a built-in print function in Python actually means.

Mapping internal and external APIs in case of Python requires similar effort because of enormous size of Python standard library. NumPy is a prominent example of external API, the translation of which is also essential for our work. Coincidentally, the basic NumPy API bears high resemblance to the interface of built-in Fortran arrays, therefore implementing the translation or array operations was

4.2 Transpiler with Human-readable Output

On top of the above requirement we impose on ourselves the requirement of preserving high-level structure of the code, as well as most of the human-required properties of it, such as the naming of variables and comments. With regards to code formatting, we decided to take the approach of generating code that is as readable as possible. However, because what is considered a reasonable indentation and recommended code style changes between programming languages, we did not make an effort to preserve original indentation style. Instead, our approach is to generate code which is objectively readable in a given programming language.

Most notably, Python has very strict rules about indentation. The indentation itself affects control flow, therefore if we would generate code while preserving the original formatting, we might end up with a completely different software.

Conversely, although we assume that the syntax of the input code is correct, we do not assume that its formatting is ideal. Therefore, we are of the opinion that by not preserving the original formatting we avoid some problems while not introducing any new ones, including any possible negative impact on readability.

Generating readable code is also worth the effort because of control over the code that end user gets. I.e. if the code generated by our tool would be further hand-tuned by a developer with deep knowledge of the original algorithm as well as the language of the generated code, such tuning should not come with additional strain, confusion or surprises about the shape of the code.

5. Dive Into Transpiling FLASH

FLASH design aligns surprisingly well with the Transpyle framework. FLASH itself being a framework, puts restrictions on the input code for physical operators, in effect forcing it to be relatively clean and well-organized. We pragmatically utilize this property of FLASH, because successful application of our transpilation framework is also partially based on input restrictions.

Also, compiler-based approach is ideal for code with clear separation of concerns. Abstractions that require the user to go all the way and specify everything from data layout through execution policies to operators, cannot be applied in part – user has to commit fully to using the framework, moreover if an abstraction would be used in one part of FLASH, it would have to be used in all of them, and in all user code as well. Selective transpilation has no such limitations - just because we rely on certain transformations in one part of the code, does not require us to do the same in any other part. This presents us with a unique opportunity to gradually apply Transpyle to more parts of FLASH, without need of substantial code refactoring.

Additionally, in case of FLASH, it not only separates concerns into two clear-cut kernel and framework spaces but the way it does the separation is very favourable for our approach. The framework space consists of unrestricted Fortran, with many latest Fortran features being employed, and therefore the syntactic complexity of the code being relatively high. On the other hand the code in kernel

© 2019 Information Processing Society of Japan

space is very strictly regulated, with even variable names up to their casing being pre-decided. As the transpilation framework is applied only to the kernel space, we can make many useful assumptions about the input and operations on it that would be very hard or impossible to make for Fortran code in general.

Finally, producing human-readable code gives the user control over the code, especially in contrast to using C++ templates for example, with which the user loses control of what is happening with code and cannot easily debug it. Ability to be able to see what the framework has done, or what it hasn't done, is also useful for FLASH developers because they can experiment with various code changes and be immediately able to judge in what way such changes affect the generated architecture-specific code.

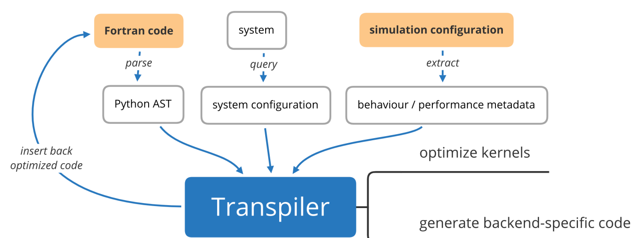

Figure 1 Transpiling FLASH: overview.

The workflow of running a FLASH simulation consists of 3 phases: (1) simulation setup (2) compilation and (3) execution. For our work, especially the first phase is interesting. The setup phase works akin to configure script present in many source code releases of open-source software. Because of modularity of FLASH, user can choose which physical operators are to be involved in their simulation, dimensionality, resolution, domain size and many other parameters can also be adjusted. The setup phase is responsible for setting up compilation according to all those settings, and it is through the setup script that the most insight into the requested settings (and in turn required transformations) can be achieved. Therefore, also the way FLASH is implemented also aligns well with compiler-based approach to optimization, as shown in Figure 1.

5.1 A Future Outlook: Path to Migrate FLASH to High-performance Python

There are more reasons that can justify the use of transpilation with Python as IR in this case.

First of all, the scientific community is converging around Python and Julia for productivity. Especially NumPy, pandas, and other Python packages in their respective domains see many new users, and scientists from many new domains are learning those languages and subsequently develop software packages to further enhance productivity in their field.

Secondly, especially with Python community embracing Type Hints and therefore accepting that Python should under certain circumstances provide static type information, Python can continue on the path towards being recognized as a fully-blown HPC scientific language. More and more packages add support for type hints, and it is easy to imagine that computing packages for Python could one day too benefit from almost free speedups that static type information combined with JIT compilation can offer.

Even now, FLASH developers often prototype new code in Python before later rewriting it to Fortran. We could imagine a future in which Python is the language in which FLASH is written and maintained, and Fortran a language in which it is executed on various HPC systems.

5.2 FLASH on GPUs

In current experiments, we focus on adapting FLASH to one new platform: GPU-based accelerated systems. In recent years, there has been a boom of accelerator-based systems. Most notable GPU-based HPC systems are ABCI, Piz Daint, Sierra, and Summit [33] – as of June 2019 these 4 systems are in global top 10 according to both Top 500 [28] and HPCG [27] lists.

Wanting to avoid porting parts of FLASH to another language, we had 2 options to choose from. Rely on CUDA Fortran API, or use directives (OpenMP or OpenACC).

Because transpiling numerical Fortran kernels reliably to CUDA would be an engineering challenge beyond our current capabilities, we opted for relying on compiler directives. Additional benefit of that approach is that Fortran code with directives is far more readable to Fortran developers than CUDA code would be, therefore verification or debugging of the code would be much easier.

There are two realistic directive languages to choose from: OpenMP and OpenACC. In our assessment, although latest OpenMP standards provide a mechanism to execute code on GPUs [21, 22], the implementations are lagging behind the standard and performance of OpenMP-annotated code on GPUs would not be satisfactory. OpenMP for GPUs is simply not ready yet, therefore we opted for OpenACC.

6. Transpilation Process

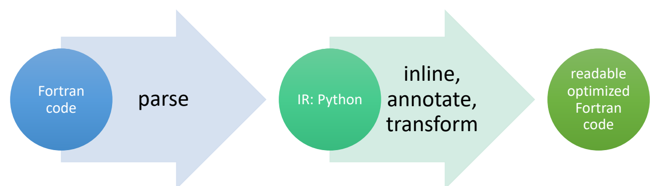

Figure 2 Overview of the transpilation process.

The end-to-end transpilation process might at first glance look very similar to the process taken by any transpiler, or compiler for that matter. Framework takes source code in source language, converts it into an intermediate representation, applies transformations on the intermediate representation, and finally outputs code in the target language, as seen in Figure 2.

© 2019 Information Processing Society of Japan

In case of our approach, the important distinction is that all languages involved in the process, including the intermediate representation language, are high-level languages.

The framework takes source code in source language, and feeds it to the parser that generalizes the syntax into an abstract syntax tree which supports direct expression of universal high-level constructs. In our approach, we take Python syntax together with basic NumPy API as the set of those universal high-level constructs. In principle, however, any language powerful enough to express the space of high-level syntax, while also including data types, coupled with well-defined APIs for numerical array operations could be used as the intermediate representation. We chose Python because of many reasons: its ability to parse/unparse itself, the high level AST format used internally by CPython, availability of numerical primitives, and finally ease of use and adoption of Python in science, as well as many other reasons.

After transformation of source syntax into the generalized one, the framework is able to apply various high-level syntactic transformation in a predictable manner. By definition, such transformations are not defined at low level, and are in fact closer to the concept of refactoring.

As an upside of operating at higher level, the framework facilitates the ease of definition of custom transformations as well as selection of targets for transformations. The conditions for transforming as well as the transformations themselves are also written in the high-level language Python, and therefore can also be dynamic so that transformation can depend on additional contextual data that would be unavailable to the static compiler/transpiler.

After transformations on the generalized syntax are complete, syntax is unparsed into source code in the target language. It is a key requirement for the process of transpilation to be realized in such a way that it does not take too long, and that the readability of the code is preserved as much as possible. Output of the framework should be still very well readable for a person.

7. Implementation Choices

The framework does not aim to support any specific kind of Fortran code or Fortran standard, rather, we aim at supporting certain carefully selected parts of Fortran syntax and very carefully selected subset of intrinsic functions. Notably, we neither support syntax obsoleted by new language standards, which many compilers support because of compatibility requirements, nor vendor language extensions, which again many compilers support because they are implemented by those vendors.

The same arguments apply to other supported languages. We chose to support translation only between a selected subset of syntactic features of each programming language. We did this for a pragmatic reason of being able to test our ideas without need to implement all possible features. The subset we have chosen fits the requirements of typical computational kernels we have been working with. Since our framework is open-source, addition of missing useful constructs is possible in the future.

Secondly, in current implementation, types of all variables have to be explicitly annotated when coding in Python. Otherwise the framework will not work. We made this pragmatic choice to avoid the need to implement type inference, a feature which can be added later. Moreover, since the focus of this work is translation from Fortran, types of all variables are already present in the source code.

Thirdly, there are many libraries on which scientific software depends. We chose to support only few libraries in our initial implementation, and even for those, vast majority of library API is left unhandled. Later, to achieve more generality, we could implement a feature to translate while wrapping calls to the dependent libraries. This will enable us to effectively dragging libraries between languages. However, since in our current work we focus on source-to-source translation between the same source and target language, all of the library calls can be taken as-is and as long as we can make sure that they do not collide with each other or with idioms defined for Python.

8. Evaluation

For the evaluation of our approach, we chose several applications available through FLASH framework. All of the chosen applications are scientifically meaningful and are used by scientists to solve real domain problems. Through choosing several real-world applications we demonstrate both the utility and generality of our approach.

8.1 Simulations

To study the effect of transpilation on the performance of FLASH code, we chose two different simulations – (1) The Sod problem [26] and (2) the carbon detonation in Type Ia Supernovae [31].

The Sod problem simulates the propagation of a shock in a fluid medium where fluid is initially at rest. The initial conditions also specify presence of a vertical shock (parallel to Y axis) in the middle of the rectangular domain. The density and pressure jumps across the shock result in all three types of non-linear, hydrodynamic waves (shock, contact, and rarefaction). This simulation demonstrates the FLASH solver's ability to capture sharp gradient and discontinuities across the shock.

The second simulation, referred to here as *Cellular*, simulates the carbon detonations highlighting the cellular structure of the detonation front which are characteristic in Type IA supernovae. More details of this simulation can be found in [31]. The most relevant feature of this simulation from our perspective is a parameter that defines nuclear reactions simulation complexity, onwards referred to as *species*. The higher the species count, the more complex the reactions simulated.

We experiment with up to several different base implementations for each simulation, and for each of these, we perform computations in more than one concrete setup.

© 2019 Information Processing Society of Japan

8.2 Environment Setup

We run the simulations on Summit supercomputer [33]. It consists of 4608 nodes, however in our current experiments we are utilizing up to around 800 nodes at any given time. Each node is equipped with 2 IBM POWER9 CPUs and 6 Nvidia Tesla V100 GPUs, over 600 GB of memory, 800 GB of NVRAM (allocatable as burst buffer or extended memory) and Nvidia NVLink connection between CPUs and GPUs. The nodes are connected by a dual-rail Mellanox EDR InfiniBand interconnect in a non-blocking fat-tree topology.

In our experiments we use a development version of FLASH [20]. Other software we rely on includes: GNU compiler suite version 6.4.0, PGI compiler version 18.10, CUDA version 10.0.130, MPI implementation MPICH version 3.3, HDF5 library version 1.10.5, openblas version 0.3.5, hypre library 2.15.1 and superlu version 5.2.1. We also rely on Python 2.7.16 and Python 3.5.6.

8.3 Sod Simulation

For the Sod problem, because of the way the shock wave propagates, we can set up the scaling experiment by increasing the resolution of the domain across the shock wave front. After aligning one of the mesh axes with the shock wave front, we can expect a very regular mesh refinement pattern repeating itself across the whole domain. Therefore, even with unpredictable nature of AMR algorithms, we are able to scale the problem in a very predictable manner which enables us to run a weak scaling experiment.

We perform two series of runs of the Sod simulation - in 2 dimensions (Figure 3) as well as 3 dimensions (Figure 4). In this experiment we a baseline implementation that uses only CPUs.

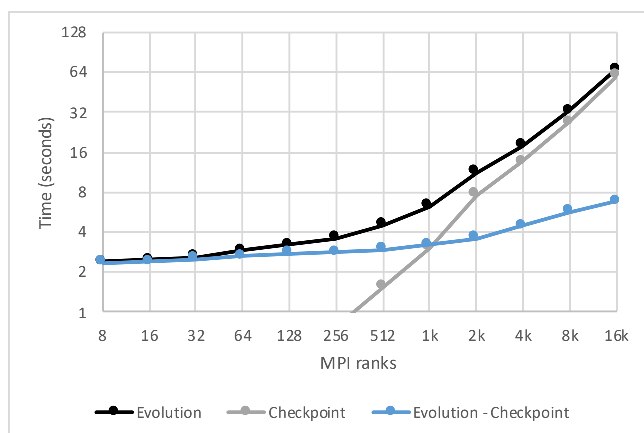

Figure 3 Weak scaling of Sod simulation in 2 dimensions on Summit supercomputer.

42 ranks fit on a single node, therefore the 2D series is executed on from 1 to around 400 nodes. We observe that the simulation does not scale very well. The runtime at larger scale is dominated by checkpointing, however even the physics reach about 50% parallel efficiency at 4000 ranks when compared to 8 ranks.

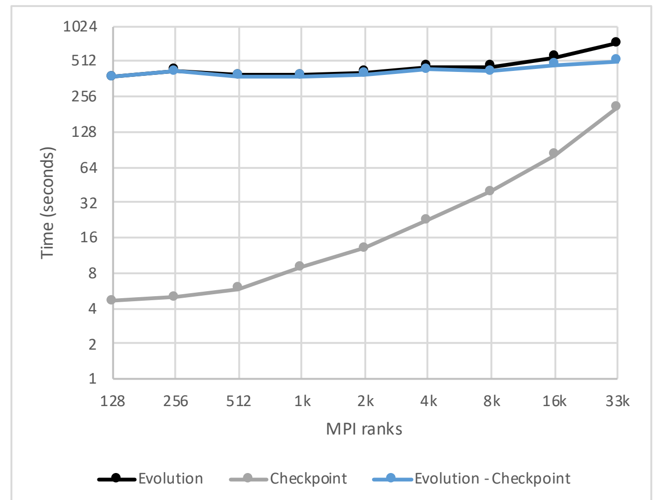

Figure 4 Weak scaling of Sod simulation in 3 dimensions.

From computational perspective, Sod is not a heavy simulation as only one kernel (hydrodynamics) is involved and it is not a particularly complex kernel. However, in 3 dimensions the simulation is dominated by the physics computations, and we observe that it scales much better in such scenario. Still at about 32000 ranks there is a observable slowdown which cannot be attributed only to the checkpointing overhead.

8.4 Cellular Simulation

The Cellular simulation is computationally much more complex than Sod. Because it aims at aiding the explanation of where and how are (were) the isotopes heavier than iron made, and merger of supernovae and neutron stars is a candidate for origin of those elements, there are several physics involved. Kernels involved in the simulation are (1) hydro, (2) gravity, (3) nuclear burn and (4) EOS (equation of state). For this simulation, we have several baseline implementations available.

8.4.1 Core Implementation

First, we establish the baseline performance of 2D Cellular simulation on CPU (Figure 5) using implementation available in FLASH.

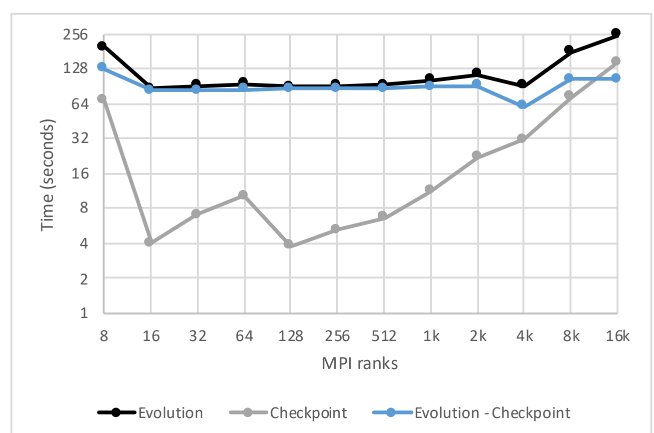

Figure 5 Weak scaling of Cellular simulation in 2 dimensions on Summit supercomputer.

© 2019 Information Processing Society of Japan

The 3D Cellular simulation, although also available, remains to be profiled. We expect that impact of checkpointing to be less than in case of 2D simulation, because of higher compute load.

8.4.2 Manual GPU-based Implementation

There is an ongoing effort by the an independent team [15, 24] to port the Type IA supernovae double detonation simulation to GPUs. Work has been done on analysis of performance, including scaling, for that implementation as well [12]. Here we summarize and discuss those results. There were 4 series of runs, two of them on CPU-only implementation (Figure 6 and Figure 7) and 2 on GPU-enabled version of the code (Figure 8 and Figure 9).

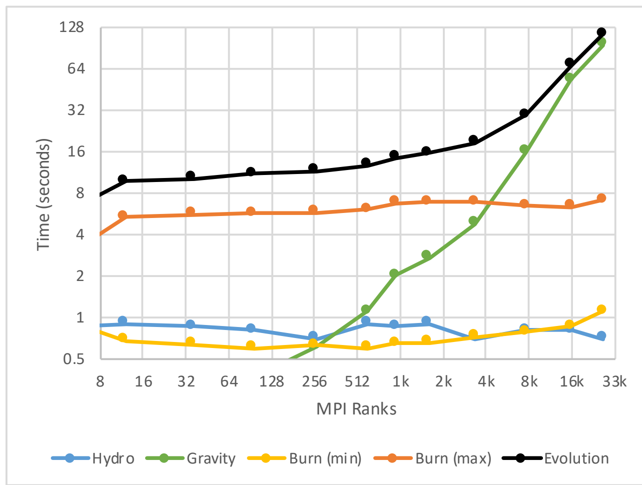

Figure 6 Scaling of the double-detonation simulation with 13 species on CPU [12].

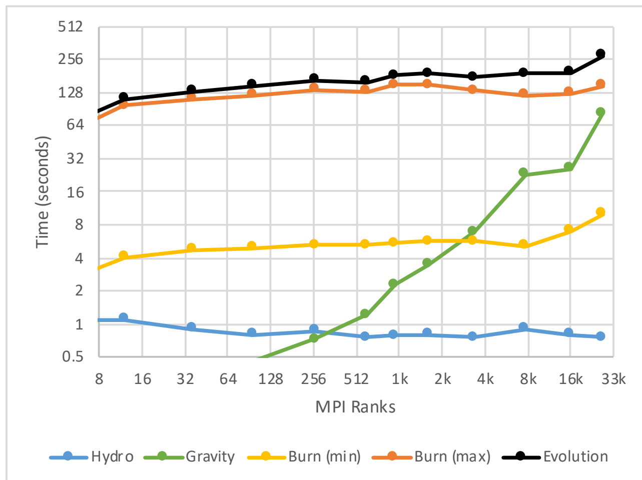

Figure 7 Scaling of the double-detonation simulation with 150 species on CPU [12].

Non-monotonic time to solution change with increasing number of ranks until order of few thousands of ranks comes from the fact that burn kernel runtime varies between ranks. In favourable conditions the load is distributed more equally while in other cases long runtime on some ranks dominates the simulation. Load balancing might be necessary.

Scaling issues at scale beyond few thousand ranks come from reduction-based implementation of the Gravity kernel, which simply fails to scale at this point.

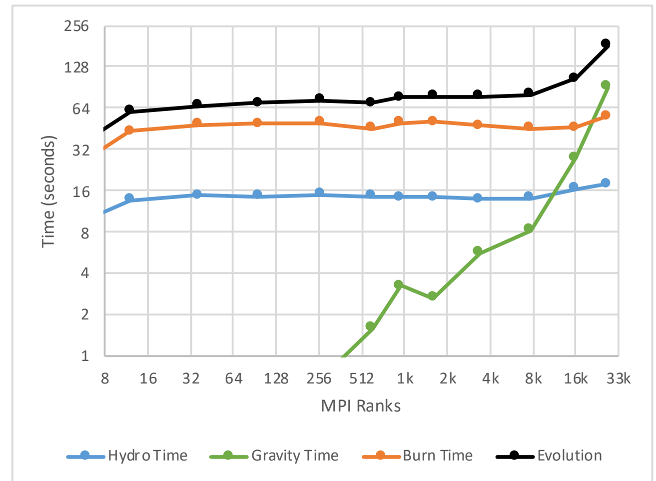

Figure 8 Scaling of the double-detonation simulation with 150 species on GPU [12].

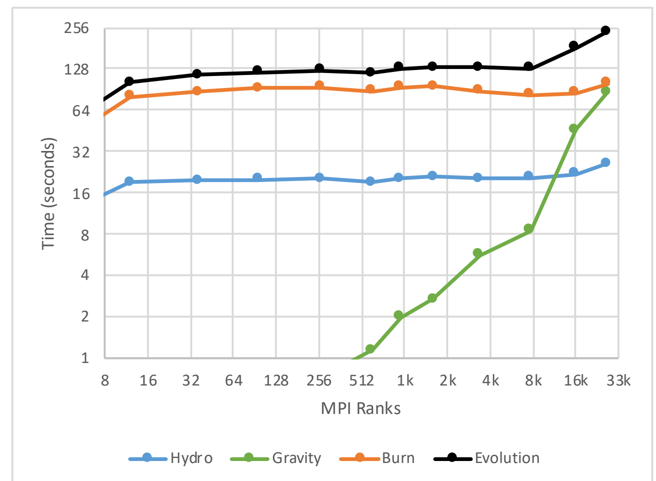

Figure 9 Scaling of the double-detonation simulation with 231 species on GPU [12].

The Burn kernel was ported to CUDA, whereas EOS module relies on OpenACC. The runtime of Burn is improved, however the load balancing issue is still present. Moreover, since the Gravity kernel is not improved, its scaling issues again dominate the runs at largest scales.

8.5 Transpiled Simulations

The transpyle framework works during the setup phase, where it can access configuration parameters. Depending on the involved kernels and their configuration, it automatically alters the code towards most efficient execution. Parameters under consideration include: internal structure of the kernels (operations, data dependencies, locality); relationships between the kernels (as above); system architecture (is it multi-core, many-core, GPU, or heterogeneous?); as well as simulation configuration (blocks per rank, grid size, other parameters).

© 2019 Information Processing Society of Japan

The experiments on transpiled code will also be ran on Summit supercomputer, however currently performance of the experimental implementation is unsatisfactory, therefore we do not have any good results to report.

9. Conclusion and Future Work

We presented intermediate results of our ongoing effort towards performance portability and modernization of multipysics Fortran framework FLASH via transpilation with Python as high-level intermediate representation.

As shown in the evaluation section, we are currently analysing baseline performance of various configurations of FLASH and working on the performance of automatically generated GPU-enabled FLASH kernels.

The next steps are as follows: hot-spots in respective kernels will be transpiled, with nuclear burn kernel being of particular interest as it will be offloaded to GPU via automatically inserted OpenACC prgamas.

Most conservative goal is for GPU-enabled Cellular simulation to outperform the CPU-only version. The computational complexity of the Burn kernel, and at larger scales the Gravity kernel limit performance improvements. We expect most improvements for compute-intensive and point-wise Burn kernel. 3D simulations are more favourable than 2D, and increasing chemical complexity (number of species) is favourable as well.

With regard to possible extensions of the work, as the new EOS implementation is being prototyped in Python, translation to Fortran instead of reimplementation is being considered.

References

[1] Bauer, M., Treichler, S., Slaughter, E. and Aiken, A.: Legion: Expressing locality and independence with logical regions, *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE, pp. 1–11 (2012).

[2] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S. and Smith, K.: Cython: The Best of Both Worlds, *Computing in Science & Engineering*, Vol. 13, No. 2, pp. 31–39 (online), DOI: http://dx.doi.org/10.1109/MCSE.2010.118 (2011).

[3] Behnel, S., Bradshaw, R., Seljebotn, D., Ewing, G. et al.: Cython: C-Extensions for Python, 2008.

[4] Bianco, M., Benedicic, L. et al.: GridTools.

[5] Bysiek, M.: transpyle: Human-oriented and high-performing transpiler for Python.

[6] Bysiek, M., Drozd, A. and Matsuoka, S.: Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance Through Transpilation and Type Hints, *Proceedings of 6th Workshop on Python for High-Performance and Scientific Computing*, Piscataway, NJ, USA, IEEE Press, pp. 9–18 (online), DOI: 10.1109/PyHPC.2016.12 (2016).

[7] Bysiek, M., Wahib, M., Drozd, A. and Matsuoka, S.: Towards Portable High Performance in Python: Transpilation, High-Level IR, Code Transformations and Compiler Directives (Unrefereed Workshop Manuscript), Technical Report 38, (2018).

[8] Chamberlain, B. L., Callahan, D. and Zima, H. P.: Parallel programmability and the chapel language, *The International Journal of High Performance Computing Applications*, Vol. 21, No. 3, pp. 291–312 (2007).

[9] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C. and Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing, *Acm Sigplan Notices*, Vol. 40, No. 10, ACM, pp. 519–538 (2005).

[10] Dubey, A., Antypas, K., Calder, A., Daley, C., Fryxell, B., Gallagher, J., Lamb, D., Lee, D., Olson, K., Reid, L., Rich, P., Ricker, P., Riley, K., Rosner, R., Siegel, A., Taylor, N., Timmes, F., Vladimirova, N., Weide, K. and ZuHone, J.: Evolution of FLASH, a Multiphysics Scientific Simulation Code for High Performance Computing, *International Journal of High Performance Computing Applications*, Vol. 28, No. 2, pp. 225–237 (online), DOI: 10.1177/1094342013505656 (2013).

[11] Dubey, A., Antypas, K., Ganapathy, M., Reid, L., Riley, K., Sheeler, D., Siegel, A. and Weide, K.: Extensible Component Based Architecture for FLASH, A Massively Parallel, Multiphysics Simulation Code, *Parallel Computing*, Vol. 35, pp. 512–522 (online), DOI: DOI: 10.1016/j.parco.2009.08.001 (2009).

[12] Dubey, A., Chawdhary, S., ... and Messer, B.: Simulation Planning Using Component Based Cost Model, *20th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2019).*

[13] Dubey, A., Tzeferacos, P. and Lamb, D.: The Dividends of Investing in Computational Software Design – A Case Study, *International Journal of High Performance Computing Applications*, (online), DOI: 10.1177/1094342017747692 (2018).

[14] Edwards, H. C. and Sunderland, D.: Kokkos array performance-portable manycore programming model, *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ACM, pp. 1–10 (2012).

[15] Harris, J. A.: Towards Exascale Simulations of Stellar Explosions with FLASH.

[16] Hornung, R., Keasler, J. et al.: RAJA: Managing Application Portability for Next-Generation Platforms.

[17] Lam, S. K., Pitrou, A. and Seibert, S.: Numba: A LLVM-based Python JIT Compiler, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, ACM, pp. 7:1–7:6 (online), DOI: 10.1145/2833157.2833162 (2015).

[18] Lehtosalo, J., van Rossum, G., Levkivskyi, I., Sullivan, M. J. et al.: mypy - Optional Static Typing for Python.

[19] Lukat, G. and Banerjee, R.: A GPU accelerated Barnes–Hut tree code for FLASH4, *New Astronomy*, Vol. 45, pp. 14–28 (2016).

[20] O Neal, J. P., Weide, K., Chawdhary, S., Dubey, A. et al.: FLASH5.

[21] of Advanced Scientific Computing Research, O.: Portability Across DOE Office of Science HPC Facilities.

[22] of Advanced Scientific Computing Research, O.: Porting BoxLib to OpenMP 4.x.

[23] Oliphant, T. E.: *A guide to NumPy*, Vol. 1, Trelgol Publishing USA (2006).

[24] Papatheodore, T. L. and Messer, O. E. B.: Improving Performance of the FLASH code on Summit and Other Architectures: First Steps, *2015 Smoky Mountains Computational Sciences and Engineering Conference*.

[25] Seljebotn, D. S.: Fast numerical computations with Cython, *Proceedings of the 8th Python in Science Conference*, Vol. 37 (2009).

[26] Sod, G. A.: A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws, *Journal of computational physics*, Vol. 27, No. 1, pp. 1–31 (1978).

[27] Strohmaier, E., Dongarra, J., Simon, H. and Meuer, M.: HPCG list – June 2019.

[28] Strohmaier, E., Dongarra, J., Simon, H. and Meuer, M.: Top 500 list – June 2019.

[29] Sullivan, M. J., Lehtosalo, J. et al.: mypy - Optional Static Typing for Python (mypyc-compiled version).

[30] Sullivan, M. J., Lehtosalo, J. et al.: mypyc: Mypy to Python C Extension Compiler.

[31] Timmes, F., Zingale, M., Olson, K., Fryxell, B., Ricker, P., Calder, A., Dursi, L., Tufo, H., MacNeice, P., Truran, J. et al.: On the cellular structure of carbon detonations, *The Astrophysical Journal*, Vol. 543, No. 2, p. 938 (2000).

[32] van Rossum, G., Lehtosalo, J. and Langa, .: PEP 484 – Type Hints (2014).

[33] Wells, J., Bland, B., Nichols, J., Hack, J., Foertter, F., Hagen, G., Maier, T., Ashfaq, M., Messer, B. and Parete-Koon, S.: Announcing supercomputer summit, Technical report, ORNL (Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States)) (2016).

© 2019 Information Processing Society of Japan