Parallel Computing 108 (2021) 102830

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Towards performance portability in the Spark astrophysical magnetohydrodynamics solver in the Flash-X simulation framework

Sean M. Couch [a], Jared Carlson [a], Michael Pajkos [a], Brian W. O'Shea [a], Anshu Dubey [b,*], Tom Klosterman [b]

[a] Michigan State University, United States of America
[b] Argonne National Laboratory, United States of America

ARTICLE INFO

MSC:
00-01
99-00

Keywords:
Exascale
MHD
Hybrid node

ABSTRACT

Simulations of core-collapse supernovae, and other astrophysical phenomena, are quintessential extreme-scale computing challenges. For core-collapse supernova simulations to be carried out by the ExaStar project under the Exascale Computing Project umbrella, a robust, efficient, and state-of-the-art magnetohydrodynamics solver is a critical requirement. In Flash-X, the primary software instrument for ExaStar, a new magnetohydrodynamics solver has been designed and implemented from the ground up to achieve accuracy and efficiency for simulations of complex astrophysical flows. This new solver, dubbed Spark, uses high-order spatial reconstruction, Runge–Kutta time integration, and an efficient cell-centered approach to satisfying the divergence-free condition for the magnetic fields. Spark was written to be optimized for data locality in cache hierarchy of CPUs. Since data locality optimizations for cache hierarchy are not directly compatible with those of accelerators, we have taken the approach of using program synthesis to avoid massive amounts of code replication that would be necessary if we were to maintain two different versions of the solver. Our program synthesis relies on a simple key-dictionary approach, implemented in python, that enables us to assemble the version of the solver suitable for the target hardware from code fragments identified by specific keys. In this paper, we describe the data locality optimizations of the solver for CPUs and accelerators and the program synthesis tools that enable this portability. We also detail the parallel performance of Spark for both CPUs and accelerators.

1. Introduction

Our understanding of complex astrophysical phenomena has been dramatically enhanced through the application of high-performance computing. For perhaps no problem is this more true than for the simulation of the collapse and explosion of massive stars as core-collapse supernovae (CCSNe). Stellar core collapse is quintessentially multi-physics, involving 3D magnetohydrodynamics (MHD), general relativity, the equation of state of matter at supranuclear densities, nuclear kinetics, and multi-flavor neutrino transport. The ExaStar project, under the umbrella of the Exascale Computing Project is developing simulation capability for exascale computation of CCSNe and similar relativistic systems dominated by neutrino transport such as binary neutron star merger. With these simulations, the ExaStar project [1] seeks to answer the question: what is the source of the heaviest elements in the universe? It is known that roughly half of the isotopes heavier than iron were formed via a slow neutron capture process (the s-process) with most of the remaining half due to a rapid neutron capture process (the r-process) [2,3], and with a scattering of rarer isotopes ascribed to what was originally thought to be a proton capture process (the p-process). Of these the ExaStar project addresses itself to the r-process.

Several astrophysical environments are suspected to produce the conditions needed for r-process nucleosynthesis. An essential component of such capability, and critical for almost all extreme-scale astrophysics applications, is an efficient and solution-accurate MHD solver. To this end, we have developed a new MHD solver targeted at emerging extreme-scale platforms, named Spark [4]. Spark is implemented in the new Flash-X framework, the primary software instrument for ExaStar. Flash-X builds on the long legacy of FLASH [5,6] and aims to deliver a flexible, high-performance platform for computational science, leveraging adaptive mesh refinement (AMR) via both Paramesh [7] and AMReX [8].

* Corresponding author.
E-mail address: adubey@anl.gov (A. Dubey).

https://doi.org/10.1016/j.parco.2021.102830
Received 31 October 2020; Received in revised form 21 June 2021; Accepted 24 August 2021
Available online 14 September 2021

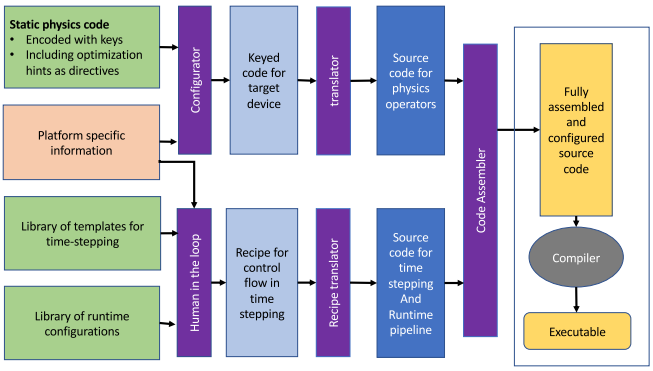

Fig. 1. An overview of the tool-chain that produces optimized code for target platform ready to be executed by the runtime.

Here we describe our efforts to make Spark performance portable across different heterogeneous platforms. The fundamental computational motif of Spark is stencil operations with finite-volume discretization. While stencils are relatively easy to scale to larger sizes, they have relatively low arithmetic intensity that makes them challenging for obtaining data locality suitable for deep memory hierarchy found on the CPUs. At the same time the presence of AMR can limit the extent of vectorization and other aspects of fine-grained parallelism suitable for accelerators. Heterogeneity is to be expected in large computing platforms currently and in the foreseeable future makes it imperative to address both these challenges without duplicating large amounts of code for different platforms. Several efforts such as Raja [9], Kokkos [10], and GridTools [11] provide abstractions that address some of these challenges, however, all of these solutions are C++ language specific and not easily applicable to Flash-X because it is largely written in Fortran.

We take a novel approach of division of labor among relatively simple collection of tools that together deliver a powerful methodology for performance portability for Flash-X. A key design principle of our tool-chain, *A Composer and Orchestrator for Multiphysics Software* (ACORMS), is to enable both domain-specific knowledge and platform knowledge to be utilized in application configuration without placing undue burden on either the tool developers or the domain experts. The system relies on a pipeline built out of steps consisting of assembly, configuration, transformation, and orchestration. The first three of these are static, and are used to configure an instance of an application suited to a target platform are shown in Fig. 1 in the form of a workflow that will generate the executable. The generated executable contains all the information necessary for the orchestration runtime to be able to dispatch data as needed to various devices and launch execution on those devices [12]. Two types of templates are used in the system. One set implements the logic of time-stepping algorithms with hooks for various solvers. The second set provides a library of resource configurations tailored to specific platforms that can be used by the time-steppers to distribute the work among available resources efficiently Code transformation tools statically map the timestepping algorithm to the most suitable platform configurations available [13] that can be used by the runtime to orchestrate work.

The remainder of the paper is organized as follows. Section 2 describes basic features of Spark. In Section 3 we describe how the code was refactored to support tailoring of code for accelerators through leveraging program synthesis. Section 4 discusses code transformation necessary to use the program synthesis and its impact on developer productivity. Section 5 describes the performance of Spark in various settings, and Section 6 presents our conclusions and future work.

2. MHD implementation and on-node parallelism

2.1. The trouble with stencils

Systems of coupled PDEs are common in many physics application. Solving such systems often involves stencil-based operations. Optimizing stencil-based solvers for both CPUs and accelerators is challenging since the performance bottlenecks in the code may be different between the two architectures, necessitating different optimization strategies.

Complete details about the solution approach to the MHD equations employed in Spark may be found elsewhere [4]. Here, we focus on the details of the numerical implementation that are relevant to describing the on-node performance for hybrid nodes that have multiple cores and accelerators. Spark is implemented within the Flash-X adaptive mesh refinement (AMR) simulation framework. Flash-X employs parallelism based on domain decomposition in which the fundamental unit is an AMR block of cells. In Flash-X, each block is of the same size, and blocks are distributed amongst separate MPI ranks for parallel processing. The AMR packages in Flash-X, both Paramesh [7] and AMReX [8], handle communication between ranks for the sharing of data, e.g., "guardcell" (or "halo" or "ghost" cell).

Spark solves the equations of ideal MHD using finite-volume discretization and explicit time integration. These equations form a coupled system of hyperbolic PDEs of the form,

$$\frac{\partial \mathbf{U}}{\partial t} = -\frac{\partial \mathbf{F}_i}{\partial x_i} + \mathbf{S}, \tag{1}$$

where $\mathbf{U}$ is a vector of conserved fields, $\mathbf{F}_i$ the respective spatial fluxes of those fields, and $\mathbf{S}$ is a vector of sources and sinks. The general outline of the algorithm for updating the conserved quantities is the following. First, we perform high-order spatial reconstruction on the cell-averaged primitive variables to estimate their values at the interface between cells. Running offset stencils for this reconstruction results in two approximations for the vector of primitives at each interface. Second, these left and right states collocated at each interface are used as input to a Riemann solver to compute the average flux across each interface between cells. Spark performs these first two steps dimension-by-dimension to calculate the spatial fluxes in all directions for each conserved field. Third, the interface fluxes are used to compute conservative flux divergences for the update of the field variables according to Eq. (1). Any necessary source terms are also computed at this point and applied during the update. The update is "unsplit" in fashion, accounting for the flux divergences in each spatial direction simultaneously.

Spark uses second- or third-order strong stability preserving (SSP) Runge–Kutta (RK) time integration [14,15]. This approach requires multiple sub-stages in which the vector of conserved quantities is updated via Eq. (1) in what amounts to a Forward Euler step, requiring that the general procedure outlined above be repeated two or three times depending on whether second- or third-order time integration is used. By default, Spark uses fifth-order weighted essentially non-oscillatory (WENO) spatial reconstruction [16] with the smoothness indicators of [17], and the HLLD approximate Riemann solver [18]. Other options for both the spatial reconstruction and Riemann solver also exist.

The most computational intensive part of the solve is the calculation of the interface fluxes, including reconstruction of the left and right states and solving the resulting Riemann problem. Given the hyperbolic nature of the MHD equations solved by Spark, calculation of the fluxes requires a spatial stencil of some variable size, depending on the spatial and temporal orders of accuracy, along with guardcell layers and inter-block communication. In Spark, we have focussed heavily on optimizing the flux computation steps of the MHD algorithm. In doing so, our chief concerns were two-fold: first, to maximize cache reuse, and second, to exploit vector instructions to the fullest extent possible. Additionally, we have aimed to achieve efficient on-node parallelism using

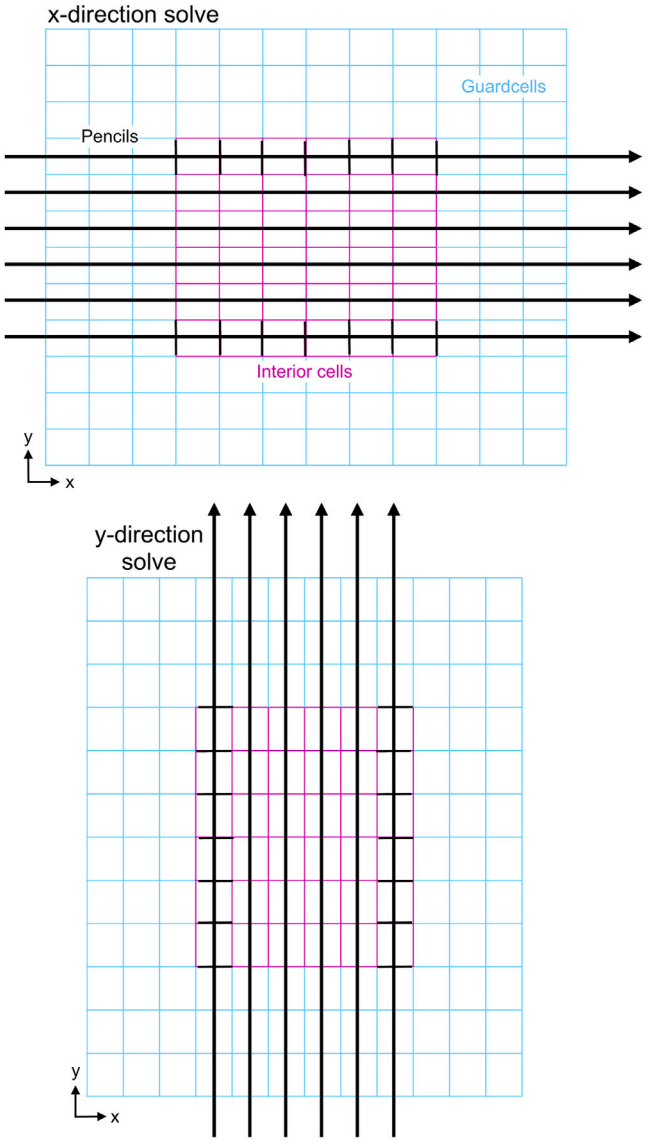

Fig. 2. Illustration of Spark's approach to computing numerical fluxes using auxiliary 1D contiguous data structures called *pencils*. For each solution direction, spatially contiguous data needed for the reconstruction and flux calculation are mapped into the pencils so that the data are also contiguous and spatially local in memory. The top panel illustrates the flux calculation in the x-direction and the bottom panel the same for the y-direction.

OpenMP. Initial implementation was targeting multithreaded execution on many-core nodes. For such nodes, the primary tool used to achieve these goals is the creation and re-use of auxiliary data structures we call "pencils" that contain all the field variables for a contiguous line of cells. Reconstruction and flux calculation is performed on each pencil followed by an update to the field variables. Since each pencil updates only its own cells, there is no write conflict with other pencils and they can each be given to separate threads to be computed in parallel using OpenMP. Only variables that are reconstructed are copied to the pencil arrays, increasing efficiency and data locality. This is an especially important optimization for multiphysics simulations, such as those targeted for ExaStar, wherein other physics modules may require large sets of variables that are not involved in the MHD update. For instance, for ExaStar, explicit two-moment neutrino transport requires over *one hundred* additional variables, none of which are required for the flux calculation during the MHD solution step.

Fig. 2 illustrates the flux calculation procedure in Spark for a single representative AMR block in a two-dimensional configuration. Interior cells are shown in magenta while guardcells needed for the flux calculation are shown in cyan. Black arrows represent the auxiliary pencil arrays, and the interfaces at which fluxes are computed are highlighted in black. In Spark, all the fluxes in a given direction are computed before moving on to compute the fluxes in the transverse directions. The pencil arrays are re-used and new data in the transverse direction are copied to the pencils. This copying incurs some overhead, about 10% of the total flux calculation cost, and may involve strided access of memory for the copy. But once the data are copied to the pencils, subsequent accesses to the pencil data for the flux computation in each direction along each pencil is highly local and the desired data locality is obtained.

In addition to the above optimizations for efficient CPU performance, in Spark we make extensive use of function inlining to avoid overhead costs and require that options such as the method of reconstruction and Riemann solver are selected at compile time, so that the compiler is better able to optimize these aspects of the calculation. We avoid unnecessary code duplication as much as possible by utilizing Flash-X's program synthesis tools described later in Section 3.2.

2.2. Adaptation for AMReX

With portability in mind, Spark has been written to work with the multiple grid treatments, such as a uniform grid or with adaptive mesh refinement (AMR). The native AMR in FLASH has been Paramesh [7] an octree-based implementation that evolves all leaf blocks in the octree in a single iteration. Another AMR library that recently achieved compatibility with Flash-X [19], and therefore, Spark is AMReX: a software framework for massively parallel, block-structured AMR applications [8]. Unlike Paramesh, AMReX evolves its blocks on a level by level basis. The two principal changes made within Spark to be compatible with AMReX are related to storing primitive variables on the computational domain and correcting fluxes at refinement boundaries.

When solving hyperbolic PDEs such as the MHD equations described above, the cell averaged data must be stored on the computational grid. In block structured AMR, the grid is divided into logically cartesian blocks of data at different levels of refinement. In the case of AMReX, this collection of blocks storing data is called 'multifabs'.[1] In between timesteps of the MHD evolution, the primitive variables of interest (eg. pressure, density, magnetic field components) are stored in the multifabs. During each RK stage, these data are then accessed and moved into the pencil structures for the subsequent reconstruction and update steps. Once the update is complete, the newly calculated variables of interest are stored once again in the multifabs.

For a grid utilizing AMR, the fluxes calculated on fine-coarse boundaries must be treated with proper care to retain the desired order of accuracy for flux calculations. To maintain this accuracy, fluxes calculated from the coarse zones and fine zones of a fine-coarse interface must be reconciled, and the mechanism differs between AMReX and Paramesh. The AMReX data object responsible for storing these weighted fluxes is the 'flux register', which exists as an object between two levels of refinement. Thus there is a separate instance of flux register between each pair of level of refinements. As is the case with multistage methods – like RK in Spark– a linear combination of flux calculations can also be stored in the flux register for a correction to be applied to the fluxes at the end of the update step.

[1] For more details on the abstractions of AMReX data structures, see the AMReX documentation: https://amrex-codes.github.io/amrex/docs_html/index.html.

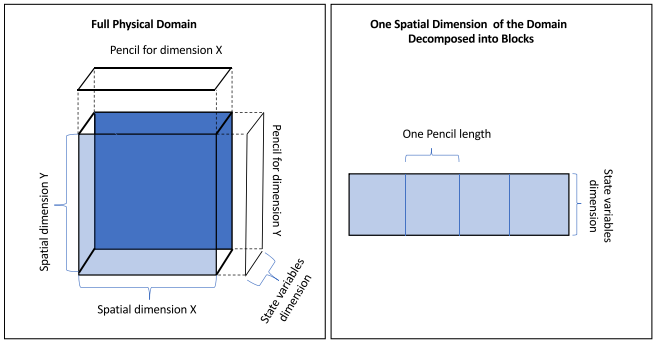

Fig. 3. A pencil extracted from a two-dimensional domain with a few state variables. The right panel shows how domain decomposition along any spatial dimension impacts the length of the pencil.

3. Adapting for accelerators

Spark was originally written for CPU with deep cache hierarchy, and therefore, with the data locality for lowest level caches baked into its structure. As mentioned earlier in Section 2, this data locality is achieved by configuring the computation into a pencil, which is a 2-dimensional array consisting of one spatial dimension and all the relevant state variables, as shown in the left panel of Fig. 3. Normally, this structuring would also be perfect for offloading the computation to accelerators by adjusting the size of the pencil. However, because AMR decomposes domain into small blocks (typically 16^3 in most Flash-X computations), and adjacent blocks may not have the same resolution, pencils end up being too small for the GPU, as shown in the right panel of Fig. 3. Additionally, as mentioned earlier, AMR itself can be exercised in two modes; one where all the leaf blocks advance all at once, the default behavior in Paramesh, or on a level by level basis where all active blocks on a level evolve in one step, the default in AMReX. Therefore, these two approaches need different ways of handling flux correction at fine-coarse boundaries

3.1. GPU offloading with penMP

To better utilize the increasingly heterogeneous computational architectures, it was imperative to provide Spark with an option to run on GPUs. Our implementation relies upon OpenMP 4.5 target offloading capabilities for this purpose. However, to be effective on GPUs, Spark needed changes in how memory is accessed and the granularity of the kernels.

On CPUs, utilizing the cache structure requires data reuse, which is accomplished by loading data in a local stencil, on which kernels operate (see Section 2). In contrast, GPUs have relatively smaller caches and larger bandwidth and perform best when each kernel operates on many elements at once. Therefore instead of creating and operating on 1D pencils of cells, the entire 3D block is operated on by each kernel. The data required for the solve are still mapped into a contiguous memory space by essentially stacking the many 1D spatially contiguous pencils end-to-end. The ordering of the data in this array is remapped depending on the direction of the flux solve being performed to take advantage of locality and expose better computational performance. To increase memory coalescing, the flux variables for the whole block are copied to a temporary array when updating fluxes in the y or z directions. These changes require an increase in the size and number of temporary variables but significantly increases the code's ability to utilize the GPU's computational resources.

The kernels in Spark also need to change based on the architecture being used. For GPUs, the kernels need to be combined or split to have as much data reuse as possible without register spilling. Where possible, each kernel is written in such a way that all the cells in a block can be updated in parallel, so that the entire iteration space can be collapsed into a single loop nest. Thus the granularity of the kernel can be chosen to be such that there is optimal data reuse, which increases the effective arithmetic intensity of the kernel, without overloading the registers. This can require additional storage space for temporary variables but increases performance by reducing the number of kernel launches on the GPU.

In effect, we were confronted with two alternative implementations of the solver, and faced with the prospect having to write one each time a different device showed up. Additionally, Spark is not the only solver in Flash-X. Therefore, in addition to rethinking some of the logic and bookkeeping of the implementation to accommodate accelerator constraints we would face a possible combinatorial explosion if we were to have different implementations to support all possible configurations. While Flash-X supports multiple alternative implementations during configuration, having a great deal of code duplication and several copies of similar functionality with slight variations would have been detrimental for maintainability of the code. To tackle this challenge we capitalize on program synthesis through component assembly that has always existed in FLASH.

3.2. Program synthesis

From its conception FLASH was designed to be a component based software system where permutations and combinations of different components would create different applications. Coarsest components are *Units*, representing a major capability, for example, the *Grid* unit implements all features of the discretized mesh, whereas the *Hydro* unit implements numerics of hydrodynamics solvers. Units can have a hierarchy of subunits, and also multiple alternative implementations of features at any level of granularity as long as it is at least a full function or subroutine. The mechanism by which synthesis occurs in FLASH is through a configuration domain-specific language (DSL) that encodes metadata about code components. The metadata lives in *config* files that live side by side with the source code of the corresponding component. The setup tool parses these config files recursively and assembles a complete application by selecting which code components, versions, and specific implementation to include. Simultaneously, it also synthesizes the build system and the necessary runtime environment for the simulation.

For the same arithmetic to work with different data layouts with code duplication we need to further lower the granularity of assembly to within functions and subroutines. We achieve this through using macros with key–value dictionary approach. The fundamental idea is simple; for any code snippet in the source, it is possible to define a macro, which becomes a key. Anytime a key is encountered in the codebase it is replaced by the code snippet that is its definition or "value". Figs. 4 and 5 illustrate how key–value pairs can be used to allow alternative implementations of code segments within a function. Fig. 4 shows a pseudo code written with keys (in red) interspersed with computations (in black). Some keys have common definitions while others have alternative definitions for different target devices. Fig. 5 shows how the process of translation emits different codes based on the definitions given to it. Note that some of the definitions are *null* – this is necessary to allow optimization options such as keeping the loops separate, or fusing them, depending upon the platform. In the example shown in the figure it is assumed that the GPU prefers to keep the two loops separate while the CPU prefers them fused.

However, these features do not suffice for our purpose. For example, different data layouts make the corresponding array shapes differ, which would need the computations described in Fig. 5 to be expressed differently too. To get around such difficulties we add inlining, recursion, and arguments in the definitions. An example of code translation using a key that includes all three of the above-mentioned features is shown in Fig. 6. The code snippet is a part of a larger calculation being done on a 4-dimensional array where the last three are spatial

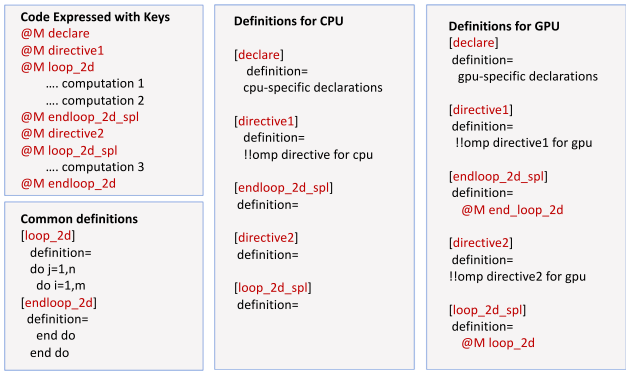

Fig. 4. An example of the key–value dictionary pairs where the key is replaced by the code snippet in the source.

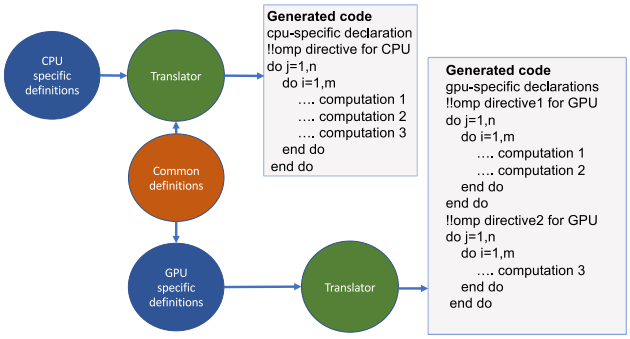

Fig. 5. Code generated for CPU and GPU through the use of appropriate definitions of the used keys.

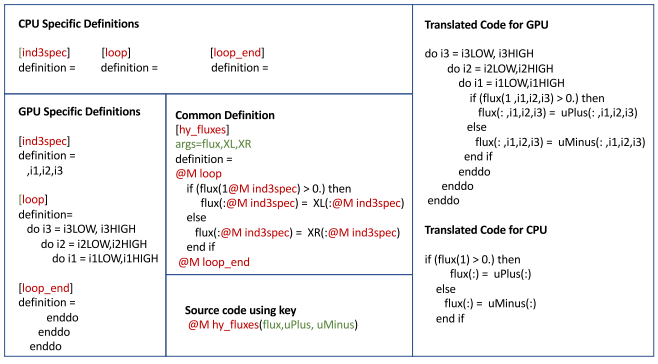

Fig. 6. Example of keys with inlining, recursion and arguments. Arguments and inlining are explicitly specified in the definition of the corresponding keys.

dimensions. It is known that for the CPU the best option is to use pencils, while for the GPU the entire array should be operated on for each kernel launch. With that in mind we define keys for GPU that iterate over spatial dimensions, while the same keys have null implementation for the CPU. The definition of *hy_fluxes* is common to both devices through recursion by using already defined keys that have different definitions for different devices. The key *hy_fluxes* also has three arguments, flux, XL, and XR. Arguments work in the same ways that keys themselves do. Thus whenever the text *XL* is encountered in *hy_fluxes* in this example, it is replaced by uPlus, and similarly, *XR* is replaced by uMinus. Of the keys being used recursively *ind3spec* is inlined, while the other two are regular keys with no arguments. The rightmost box displays the generated code for both CPU and GPU versions.

We do not impose any explicit constraints on the number of keys that any code component can define. A few restrictions apply to recursion, for instance a key cannot self reference, and level of recursion deeper than 2 has neither been needed nor tested. The readability and maintainability of the code depends on how well the developers define and use key–value pairs. These pairs are typically local with their scope being limited to the corresponding code component. However, often repeated code patterns such as start and end of iterators, loop bounds for loop nests, and repeated array bounds etc. have been made available as globally defined keys. This has proven to be an invaluable by-product of this approach in that it has reduced code duplication and increased the overall readability and maintainability of the code.

3.3. Translator

As mentioned earlier, the tool that is used to do program synthesis in `Flash-X` is called the *Setup Tool*, which parses *Config* files written in the `FLASH` specific configuration DSL. The new capability added to the Setup Tool is through another tool, *the macroProcessor* that implements the key–value dictionary feature of the program synthesis. This tool, like the Setup Tool, is written in Python and is the last feature invoked during the configuration of an application instance. Definitions are stored in ".ini" files, according to the Python ConfigParser format. Style guidelines are documented for ease of use by the developers. When a new definition file is created, the corresponding Config file is informed about its existence so that the Setup Tool includes it during configuration. The source files needed by an ".ini" file are given the extension ".F90-mc" to differentiate them from the source files that do not need to have the macroProcessor applied on them. The macroProcessor tool can also be invoked independently if a developer wishes to inspect how keys are replaced by their definitions. The output comes out in a ".F90" file which is a normal Fortran file and can be compiled by any Fortran compiler. Various command-line arguments can specify which files to process and which definitions files to read. By default, the tool will read all '.ini' files in the directory, then process all '.F90.mc' files and write the modified files to '.F90' format.

4. Code transformation

The approach of using macro-like functionality in key–value pairs, while powerful, is not backed by any compiler or auto-tuning technology. It does not obviate the need to understand the data locality of the solver on different hardware and exploring the available design space. What it does provide is a way to express that design in a cleaner way with one high level source code that encodes all the numerics of the solver. We had started out with two different versions of `Spark`, one in pencil form and one in full block form, but with identical numerics and logic. Here, we outline the steps in refactoring `Spark` to obtain a single source, and its impact on developer productivity and being prepared for the future.

4.1. Refactoring

The first step was to appropriately tailor data declarations to provide good locality on each device and extract all scratch memory allocations interspersed within the code. This is so that the device's memory can be allocated and managed consistently. We do this by registering all needed instances of scratch memory and the size associated with each instance with the configuration system. A new interface in the infrastructure can be invoked to allocate all scratch space needed by a code component at the host or the destination device. To access the scratch space another interface gives access through a pointer to the corresponding portion of the memory where it is also possible to specify the shape of the array to be associated with the pointer. For example, if a three dimension scratch space of size $nx \times ny \times nz$ is needed, the allocation step will ask for a 1D chunk of that size. However, when

asking for access, the calling code can specify that the pointer be shaped as $dimension(nx, ny, nz)$.

The second step was elimination of the knowledge about the shape of array from the arithmetic code blocks by using inlined keys for array shapes similar to the example in Fig. 6. Similar inlined keys for argument lists and declaration blocks complete the abstraction of arrays shapes from the numerics. We used this option Instead of overloading the interfaces because this method allows any new device to be accommodated with alternative definitions of each of the keys in question, instead of changing the interfaces and all instances of use of the corresponding interfaces in the code. The next step is separating out arithmetic expressions from logical ones. Since we place no restriction on the number or size of the dictionary elements, the logic of this step is easy, especially because many of the needed keys (start the iterator) etc are global. Because we have made provision for arguments in the keyword definitions, the same keyword can be applied in multiple situations and eliminates error-prone code repetitions. After this step, the logical structure of functions doing numerical computations in the code resembles kernels ideally suited for accelerators.

Next, we generate keywords for invoking the numerical kernels described above. Here, we make use of the feature that allows multiple alternative implementations of any arbitrary functionality within the code to implement different methods of invoking the kernels. Thus by placing them in appropriate places in the source tree we can have several alternative dictionary definitions for the same key, each one tailored for the specific compute environment. Provision for null implementations has been among the most powerful features of FLASH's architecture that has allowed configurability and extensibility. We exploit null definitions in `Flash-X` in similar ways to maintain the flexibility in configuration and, more critically, invariant arithmetic blocks. The final step in refactoring is to take a union of all the OPM directives in the code and define keys for each, making full use of null implementations where one or more devices do not have a corresponding directives, similar to the example in Fig. 4. An added advantage is that very likely another set of alternative keys might suffice to use a different off-loading option, though this has not been explored so far.

4.2. Productivity

In principle the approach of key–value dictionary is similar at the level of code interface to the abstraction tools such as Kokkos and Raja. Those tools use C++ templates for providing single expression of the source code with language supported specialization. At this level our approach has two strong advantages over the others. Our approach is completely language agnostic, and is obtained with a really simple tool. It took less than a week to develop, test and deploy. Also, the intermediate code generated is in the target language (Fortran/C/C++ or any other), therefore human readable, unlike templates that can be extremely difficult to unravel. The second advantage is that level of complexity to be deployed is under the control of the developer since they define the keys and determine where and how to use them.

By itself the key–value pairs do not replace the capabilities offered by the C++ based tools. They have full featured backends for all available flavors of accelerators, and are able to infer most optimization opportunities without any intervention from the developers. Other components of the tool-chain shown in Fig. 1 provide some of those capabilities on a much smaller scale. The tool-chain is designed to make it possible to expose the optimization opportunities to a generic compiler. In that sense optimizations through our tool chain are still driven by the developers themselves, and considerable refactoring of the code is needed.

The biggest value of our tool-chain, however, lies in being prepared for further out in the future. Once the code has been refactored such that parts of the code that are platform invariant (the arithmetic and logic of the calculation) have been separated from those that are platform dependent, it is potentially much easier to generate alternative definitions. It would be much faster than building a new back-end and will, therefore, make deploying on a new machine much faster and less expensive.

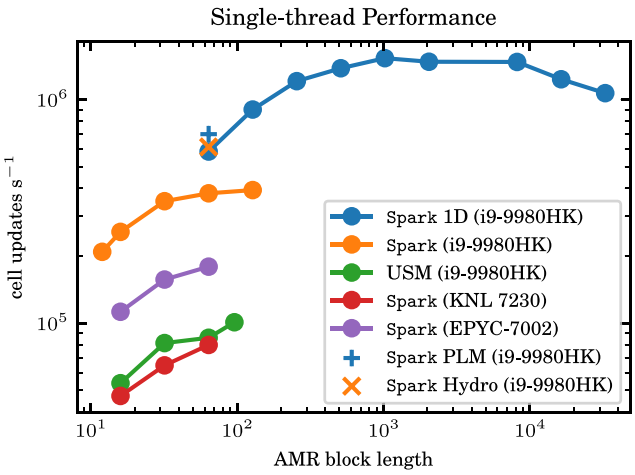

Fig. 7. Single-thread performance of Spark for multiple different hardware architectures as a function of the AMR block length in one dimension. Performance is measured in cell updates per core-second. Here, "i9-9980HK" refers to an Intel Core i9-9980HK laptop processor, "KNL 7230" to an Intel Xeon Phi Knights Landing processor as deployed in *Theta* at ALCF, and "EPYC-7002" to an AMD EPYC-7002 processor. All tests are for a single-block 3D run except for the blue curve which is for 1D runs. For comparison, identical 3D runs were performed with FLASH's unsplit staggered mesh (USM) MHD solver, shown in green. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5. Parallel performance

We now turn to the single-node performance of Spark on CPUs and GPUs. We begin with an analysis of Spark's single-thread performance on CPUs and how it varies with the size of the AMR block used. We then consider Spark's multi-threaded performance on commodity CPU nodes as well as many-core Intel Phi nodes on the Argonne Leadership Computing Facility's *Theta* machine. We do not consider multi-node performance since in `Flash-X`, this is handled entirely by the AMR grid package via MPI and, hence, does not reflect any particular performance characteristics of Spark.

5.1. Single-thread performance

Fig. 7 shows the single-thread performance of Spark as a function of the AMR block length. For these tests, we use a standard 3D magnetized shock tube test problem carried out with a single AMR block of equal length on each side. All tests were performed with a single execution thread on the respective hardware. Performance is measured in cell updates per second for a full update step, including setting boundary condition values in the guard cells and allocation of scratch memory storage. We test Spark's performance on three different CPU architectures: Intel i9-9980HK, AMD EPYC-7002, and Intel Knights Landing KNL 7230. The Intel i9-9980HK is an 8-core CPU with a 16 MB L3 cache and each core has a 256 KB L2 cache. The AMD EPYC-7002 is a 64-core CPU with a 256 MB L3 cache. The Intel KNL 7230 is a 64-core CPU with a total of 32 MB of L2 cache and utilizes 16 GB of high-bandwidth memory in addition to traditional DRAM. For Spark on these different architectures, the performance improves steadily as the block length is increased. This is true up to block sizes of 128^3 at which point total memory available for the calculation became limiting. This implies that the CPU caches are not yet fully utilized. To illustrate this, in Fig. 7 we also show the same test carried out with

Spark in 1D, for which we can extend the linear AMR block size to much larger values. For 1D, Spark's performance continues to increase until a block length of 1024, at which point performance drops slightly. Beginning at a block length of 8096, performance begins to steadily decrease as the block length is increased, indicating cache is being saturated and performance is limited by memory bandwidth. None of the 3D cases are able to reach this point since the linear length of the pencil arrays are not large enough before the overall memory footprint becomes limiting. This implies room for optimization of our cache reuse and memory layout scheme in Spark for 3D applications.

For the architectures considered, the single-thread performance is as-expected. The i9-9980HK processor performs best, the EPYC-7002 about a factor of two slower, and the KNL 7230 more than a factor of four slower. Both of the slower architectures, the KNL in particular, rely heavily on multi-threaded execution to achieve peak performance so this result is no surprise. Multi-threaded performance on these two architectures is considered in the following subsection.

As a benchmark comparison, we also ran the same 3D test problem using the unsplit staggered mesh (USM) constrained-transport MHD solver available in FLASH [20] using the i9-9980HK hardware. We find a similar increase in performance with larger block size as in Spark for the USM solver but in nearly every case Spark is nearly four times faster. This is true even though the USM scheme is a "single-step" time integration method, meaning that reconstruction and flux calculations are performed only once per time step, while Spark's second-order RK scheme that is used in these tests requires two such sub-steps per full time-step. Caution should be taken, however, in drawing conclusions about the relative expense between Spark and USM for full applications since RK time integration schemes have a more stringent time step size restriction, meaning that more overall time steps are required by Spark to complete the same calculation than for USM [4].

We also evaluate the performance of Spark in two special cases: second-order piecewise linear method (PLM) spatial reconstruction and non-MHD, pure-hydrodynamics applications. WENO reconstruction requires a far greater number of operations per cell than second-order linear reconstruction with suitable slope limiting. The blue cross in Fig. 7 shows that for the same 3D MHD shock tube problem, replacing WENO reconstruction with PLM increases performance by about a factor of two. Switching to pure-hydrodynamics results in a relative speedup of nearly two, as well. This is also to be expected since including MHD requires nearly double the number of field variables to be solved for in Spark.

5.2. Multi-threaded performance

We now consider the multi-threaded performance and efficiency of Spark. For these tests, the same 3D magnetized shock tube problem as above is used and we restrict the calculation to a single AMR block. As described in Section 2, threading is achieved with OpenMP by distributing separate pencil arrays amongst the thread team for the parallel calculation of the spatial fluxes, the most expensive part of the MHD update step. The solution update step using the divergence of the fluxes is also OpenMP threaded but is typically a small fraction of the overall runtime so here we focus on the threading performance of the flux calculation alone. Other parts of the MHD update, such as the exchange of guard cell information with neighboring AMR blocks and the allocation of scratch memory storage, are not currently threaded and can become a significant expense at high thread counts (i.e., Amdahl's Law [21]).

Fig. 8 shows the strong scaling efficiency of Spark's flux calculation as a function of thread count and AMR block length. We focus here only on one of the CPU architectures to demonstrate the typical behavior of Spark's performance on CPUs. These tests were performed on an AMD EPYC-7002 node at Michigan State University's Institute for Cyber-Enabled Research and on an ALCF *Theta* KNL 7230 node. For 16^3 blocks

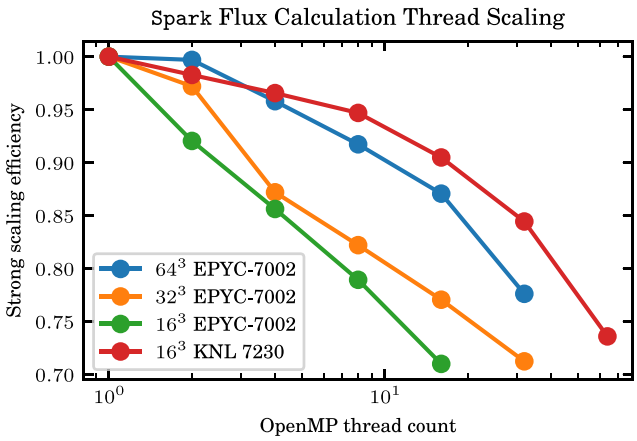

Fig. 8. Multi-threaded strong scaling efficiency of Spark's flux calculation step using OpenMP. Results are shown for an AMD EPYC-7002 node for different AMR block sizes. As the block size is increased, scaling efficiency improves. For reference, we also show results for ALCF *Theta* (KNL 7230) for 16^3 blocks. For all thread counts greater than one we find that the KNL 7230 architecture shows better strong scaling efficiency than the comparable setup on the EPYC-7002.

on the EPYC hardware, the strong scaling efficiency drops steadily to about 70% at a thread count of 16. Strong scaling efficiency is improved as the block size is increased. For 64^3 blocks, efficiency is 87% at 16 threads and 77% at 32 threads. This is a product of reduced overheads associated with thread spawning and memory rearrangement for the pencil arrays as the block length increases. Thread scaling efficiency is generally better for the KNL 7230 than for the EPYC architecture, as also shown in Fig. 8. For a *Theta* node and only 16^3 blocks, efficiency is 91% at 16 threads per block and 74% at 64 threads. This indicates that architecture details (cache layout, memory available, etc.) can impact the scaling behavior of stencil-based codes like Spark even between different CPUs. As we describe below, the significant differences between CPU and GPU architectures, however, requires a completely different strategy to realize acceptable performance.

5.3. Single-node GPU performance

To test the performance increase of the GPU code, we run on a single IBM Power System AC922 node on OLCF *Summit*.[2] Each node has two IBM POWER9 processors (42 usable cores) and six NVIDIA Volta V100 GPUs. The processors are connected to the GPUs via dual NVLink connections. The two processors have an aggregate 512 GB of DDR4 memory and the GPUs 96 GB of High Bandwidth Memory (HBM2) per node. We assign one MPI rank to each core and have all cores offload work to the GPUs. This means each GPU has seven cores submitting work to it, with the mapping of cores to GPUs such that the memory transfer latency between the cores and GPUs is minimized. Each CPU core executes four OpenMP threads.[3]

We profile the performance of Spark's flux calculation step on a single *Summit* node, comparing the optimal CPU implementation with our GPU implementation. A sophisticated on-node memory management and orchestration system is under development for Flash-X whose prototype has proven to be effective in latency hiding through asynchronous communications [12]. At this writing it has not yet been integrated into the production. As such, in our current GPU implementation of Spark, the runtime is dominated by the movement of memory back and forth between the CPUs and the GPUs. As Flash-X's orchestration systems becomes integrated with the production code,

[2] https://www.olcf.ornl.gov/summit/.

[3] The exact execution call on Summit is `>jsrun -n6 -a7 -c7 -g1 -l GPU-CPU`.

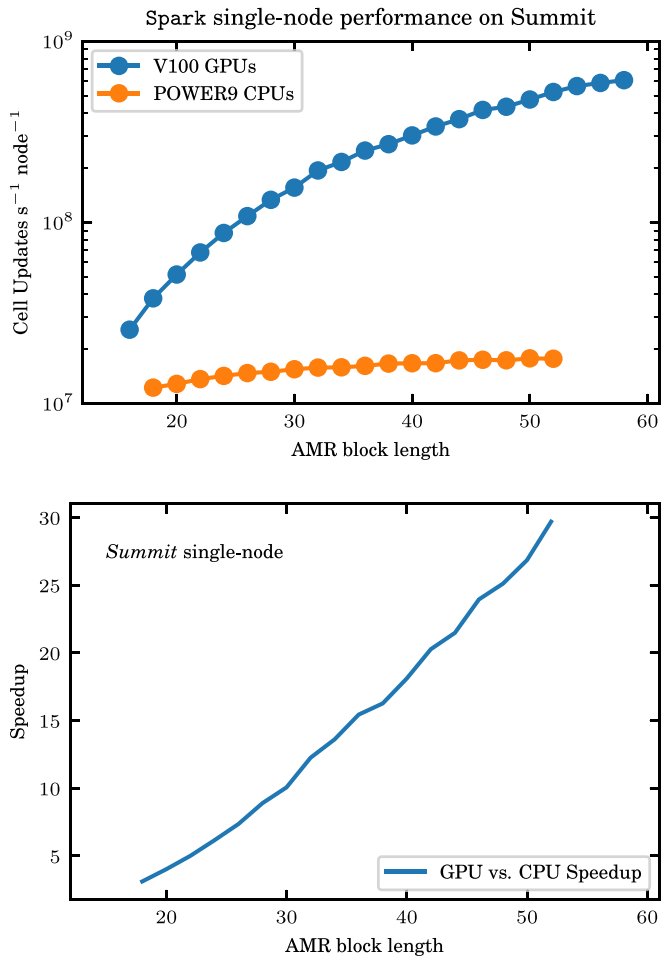

Fig. 9. Single-node performance on OLCF *Summit* comparing CPU-only runs ("POWER9") to CPU+GPU runs ("V100") considering only the Spark flux calculation and update steps (i.e., neglecting initial memory movement between the CPU and GPU). The top panel shows the performance in cell updates per second per node for both cases. In each case, a full node's worth of resources are used. Details of the run configuration are given in the text. The bottom panel shows the GPU-to-CPU speedup as a function of the linear AMR block length. The GPU continues to speedup substantially as the size of the block is increased.

we anticipate this becoming a negligible cost in our target production simulations. Therefore, here, we only consider the on-device MHD solution step involving the calculation of spatial fluxes and the resulting updates to the conservative and primitive variables.

For the performance study presented here, we use a standard 3D Sedov–Taylor blast wave problem on a uniform grid consisting of 120 AMR blocks. We repeat this test for varying AMR block lengths. Fig. 9 shows the resulting performance in cell updates per second per node for both the CPU-only and GPU versions of Spark as functions of AMR block length. As seen above in Section 5.1, CPU performance steadily improves as the AMR block size increases. This is also true, and to an even greater extent, for the GPU version of Spark. Between a block size of 16^3 and 58^3 the GPU performance increases by a factor of 24. Over roughly the same range in block size, the CPU-only performance increases by a factor of only 1.5. The bottom panel of Fig. 9 shows the relative speedup of the GPU version over the CPU version as a function of the block length. The speedup is a factor of three for 16^3 blocks but increases rapidly to a factor of 30 at block sizes larger than 50^3.

The overall per node performance of Spark on *Summit*'s GPUs is comparable to that obtained with K-Athena by [22] using Kokkos. We caution, however, that until an effective memory management system is implemented in Flash-X and put into use in the GPU implementation of Spark, real-world applications with Spark will not achieve these levels of performance since the runtime is currently dominated by memory movement between the host and device. Nevertheless, our results are encouraging and indicate that we are able to achieve speedups on GPUs using OpenMP offloading comparable to other GPU-optimized codes.

6. Conclusions

We have presented a new MHD solver in this paper that uses program synthesis to maintain a single code base for all the arithmetic involved in the computation, and obtains performance portability to different architectures through alternative definitions of keys in key–value pairs. In terms of functionality this technique resembles template meta-programming of C++ that is used in several abstraction layers being used by various scientific codes. Our approach is unique in that its fundamental design is language agnostic. Additionally, our program synthesis tools are uncomplicated, the macroProcessor tool took roughly 2–3 person-week to complete. The refactoring of Spark to enable alternative implementations with a single code base took 3–4 person weeks and generation of alternative definitions of keys for the two modes of operation took less than one person-week. An added advantage is that macroProcessor can be applied to any code component as a stand alone tool to generate the corresponding Fortran file that can be inspected by the developers and can be debugged like any other Fortran source. The MHD solver is already demonstrating consistently good performance on different architectures, and our expectation is that once the runtime being developed is integrated with Flash-X, we will have an easy to maintain performance portability solution, certainly for Flash-X, but quite possibly also for non C++ scientific codes.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Exascale Computing Project, United States (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, United States.

S.M.C. is supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics, Early Career Research Program, United States under Award Number DE-SC0015904. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and Office of Nuclear Physics, Scientific Discovery through Advanced Computing (SciDAC) program under Award Number DE-SC0017955.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, and Argonne Leadershop Computing Facility.

References

[1] A. Harris, et al., Exascale models of stellar explosions: quintessential multi-physics simulation, Int. J. High Perform. Comput. Appl. (2021) (in press).
[2] E.M. Burbidge, G.R. Burbidge, W.A. Fowler, F. Hoyle, Synthesis of the Elements in Stars 29 (1957) 547–650.
[3] A.G.W. Cameron, Nuclear Reactions in Stars and Nucleogenesis 69 (1957) 201–222, atomic Energy of Canada, Ltd. CRL-41.
[4] S.M. Couch, Spark: A New Solver for Self-gravitating Compressible Magnetohydrodynamics Implemented in the FLASH Simulation Framework, (in press).
[5] B. Fryxell, K. Olson, P. Ricker, F.X. Timmes, M. Zingale, D.Q. Lamb, P. MacNeice, R. Rosner, J.W. Truran, H. Tufo, FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes, ApJS 131 (2000) 273–334.

[6] A. Dubey, K. Antypas, M.K. Ganapathy, L.B. Reid, K. Riley, D. Sheeler, A. Siegel, K. Weide, Extensible component-based architecture for flash, a massively parallel, multiphysics simulation code, Parallel Comput. 35 (10–11) (2009) 512–522.
[7] P. MacNeice, K.M. Olson, C. Mobarry, R. de Fainchtein, C. Packer, PARAMESH: A parallel adaptive mesh refinement community toolkit, Comput. Phys. Comm. 126 (2000) 330–354.
[8] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, M. Zingale, AMReX: a framework for block-structured adaptive mesh refinement, J. Open Source Software 4 (37) (2019) 1370, http://dx.doi.org/10.21105/joss.01370.
[9] R. Hornung, J. Keasler, et al., Raja: managing application portability for next-generation platforms, 2020, URL: https://computation.llnl.gov/projects/raja-managing-application-portability-next-generation-platforms.
[10] H.C. Edwards, D. Sunderland, Kokkos array performance-portable manycore programming model, in: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores, ACM, 2012, pp. 1–10.
[11] M. Bianco, L. Benedicic, et al., Gridtools, 2020, URL: https://github.com/GridTools/gridtools.
[12] J. O'Neal, M. Wahib, A. Dubey, K. Weide, T. Klosterman, Domaine-specific Runtime to Orchestrate Computation on Heterogeneous Platforms, Technical Report ANL-20/77, Argonne National Laboratory, 2020.
[13] J. Rudi, J. O'Neal, M. Wahib, A. Dubey, K. Weide, CodeFlow for FLASH: Code Generation System for FLASH-X Orchestration Runtime, Technical Report ANL-21/17, Argonne National Laboratory, Lemont, IL, 2021.
[14] C.-W. Shu, S. Osher, Efficient implementation of essentially non-oscillatory shock-capturing schemes, J. Comput. Phys. 77 (1988) 439–471.
[15] S. Gottlieb, D.I. Ketcheson, C.-W. Shu, High order strong stability preserving time discretizations, J. Sci. Comput. 38 (3) (2009) 251–289.
[16] C.-W. Shu, High order weighted essentially nonoscillatory schemes for convection dominated problems, SIAM Rev. 51 (2009) 82–126.
[17] R. Borges, M. Carmona, B. Costa, W.S. Don, An improved weighted essentially non-oscillatory scheme for hyperbolic conservation laws, J. Comput. Phys. 227 (2008) 3191–3211.
[18] T. Miyoshi, K. Kusano, A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics, J. Comput. Phys. 208 (2005) 315–344.
[19] A. Dubey, J. O'Neal, K. Weide, S. Chawdhary, Distillation of best practices from refactoring flash for exascale, SN Comput. Sci. 1 (4) (2020) 223, http://dx.doi.org/10.1007/s42979-020-0077-x.
[20] D. Lee, A solution accurate efficient and stable unsplit staggered mesh scheme for three dimensional magnetohydrodynamics, J. Comput. Phys. 243 (2013) 269–292.
[21] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April (1967) 18-20, Spring Joint Computer Conference, AFIPS '67 (Spring), ACM, New York, NY, USA, 1967, pp. 483–485, http://dx.doi.org/10.1145/1465482.1465560.
[22] P. Grete, F.W. Glines, B.W. O'Shea, K-Athena: a performance portable structured grid finite volume magnetohydrodynamics code, arXiv e-prints.