WARWICK
THE UNIVERSITY OF WARWICK

Manuscript version: Author's Accepted Manuscript
The version presented in WRAP is the author's accepted manuscript and may differ from the published version or Version of Record.

Persistent WRAP URL:
http://wrap.warwick.ac.uk/160916

How to cite:
Please refer to published version for the most recent bibliographic citation information. If a published version is known of, the repository item page linked to above, will contain details on accessing it.

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions.

Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher's statement:
Please refer to the repository item page, publisher's statement section, for further information.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk.

warwick.ac.uk/lib-publications

Warwick Data Store: A Data Structure Abstraction Library

Richard O. Kirk*, Martin Nolten†, Robert Kevis†, Timothy R. Law†, Satheesh Maheswaran†,
Steven A. Wright‡, Seimon Powell†, Gihan R. Mudalige*, Stephen A. Jarvis§
*Department of Computer Science, University of Warwick, Coventry, United Kingdom
Email: R.Kirk@warwick.ac.uk
†AWE, Aldermaston, United Kingdom
‡Department of Computer Science, University of York, York, United Kingdom
§University of Birmingham, Birmingham, United Kingdom

***Abstract*—With the increasing complexity of memory architectures and scientific applications, developing data structures that are performant, portable, scalable, and support developer productivity, is a challenging task. In this paper, we present Warwick Data Store (WDS), a lightweight and extensible C++ template library designed to manage these complexities and allow rapid prototyping. WDS is designed to abstract details of the underlying data structures away from the user, thus easing application development and optimisation. We show that using WDS does not significantly impact achieved performance across a variety of different scientific benchmarks and proxy-applications, compilers, and different architectures. The overheads are largely below 30% for smaller problems, with the overhead deceasing to below 10% when using larger problems. This shows that the library does not significantly impact the performance, while providing additional functionality to data structures, and the ability to optimise data structures without changing the application code.**

***Index Terms*—High Performance Computing, Data Structures, Mini-Applications**

I. Introduction

Over the last few years, there has been a noticeable shift in the development of new, high performance computing architectures. The disparity between processor speeds and memory speeds has resulted in an increased focus on the performance of the memory subsystem, as demonstrated by the increasing use of high-bandwidth memory in newer CPUs such as Fujitsu's A64FX [1] and GPUs such as NVIDIA's A100 [2]. This is a necessary development in order to close the gap between the speed of performing data read and writes compared to the speed of floating point operation in the same period of time. As such, applications that relied on large amounts of data movement between the processor and its memory would not see as much of a performance increase when executing on newer architectures. However, even with an increase in memory performance, the memory efficiency is often lower than its compute counterpart [3]. Combine this fact with the increasing complex data structures used within applications, and the need for code to be performance portable [4], the structure of the data becomes incredibly important for ensuring high memory and application performance.

In this paper we present the design and implementation of a library that allows data structures to be abstracted away from applications and algorithms. By so doing, we gain two major benefits. Firstly, we are able to perform large-scale changes to the data structure, without the need for a significant proportion of the program to be re-written. These can range from restructuring the data for the code to placing the data onto a different level in the memory hierarchy or altering the existing data structure. The changes specified here are set by the user. Secondly, we are able to tweak the data structures for different applications and hardware, thus allowing for better utilisation across multiple architectures. Unlike the first benefit, these changes are made within the library itself. To demonstrate the need for a library such as the one described, we are focusing on the challenges faced by different physics applications.

The Warwick Data Store (WDS) is a template C++ library, designed to replace hard-coded data structures in applications [5]. The resultant library provides a means whereby data structures can be altered and optimised, without the risk of large-scale changes to the code. Alongside this, further functionality can be provided that would otherwise be difficult, such as the ability to easily switch between different data structures, and change the data adjacencies of given variables (for example, changing from row-major to column-major indexing, and vice versa). All of this would need to be achieved with as little cost to an application's performance as possible.

We aim to demonstrate the key criteria of WDS as extensibility, minimal size, ease of implementation and minimal performance impact on an application, along with additional functionality provided in the library. In order to do this, we will show the following contributions:

- Design and implementation of a data structure abstraction library.
- Development of a multi-material extension to the data structure abstraction library and evaluation the performance utilising representative multi-material kernels from previous literature.
- Investigation of the on-node performance of the data

© British Crown Owned Copyright 2020/AWE. Published with permission of the Controller of Her Britannic Majesty's Stationery Office. This document is of United Kingdom origin and contains proprietary information which is the property of the Secretary of State for Defence. It is furnished in confidence and may not be copied, used or disclosed in whole or in part without prior written consent of Defence Intellectual Property Rights DGDCDIPR-PL—Ministry of Defence, Abbey Wood, Bristol, BS34 8JH, England.

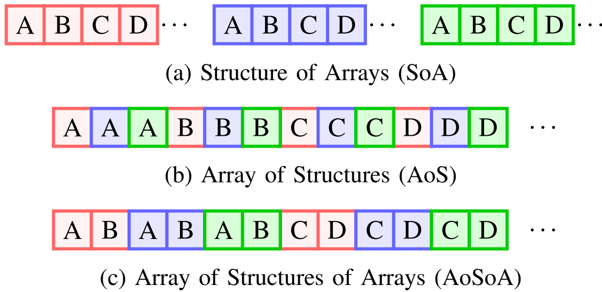


Fig. 1: Basic data structures

structure abstraction library across different architectures, compilers and mini-applications

The remainder of this paper is structured as follows: Section II explores other data structure and abstractions in different contexts. Section III, outlines the structure and implementation of Warwick Data Store, alongside additional functionality provided by the library; Section IV explains how the results were collected and analysed; Section IV-A details how Warwick Data Store can be expanded for customised, domain-specific data structures and the performance impact this has on representative kernels; Section IV-B explores the performance impact of Warwick Data Store on three physics mini-applications, BookLeaf, MiniMD and TeaLeaf; Finally, Section V concludes this paper.

II. Background

The way in which data is structured within an application is incredibly important to its performance, and is often hard to change once development has begun [6]. As such, it is key to look at the three basic data structures that are used as the building blocks for more complex data structures. These are Structure of Arrays (SoA), Array of Structures (AoS) and Array of Structures of Arrays (AoSoA).

Figure 1 shows each of these data structures for three variables, `x`, `y` and `z`. For ease of explanation, each variable contains the same number of elements. In examples where the variables are not the same size, a combination of these data structures is required.

The first is Structure of Arrays (SoA, Figure 1a), which consist of an array of each variable. The data within each is contiguous, allowing for programs to easily iterate through the variable, and allows for more efficient use of memory. SoA performs best when a few arrays are being used within a loop.

The converse of this is Array of Structures (AoS, Figure 1b), which sees each variables elements interleaved with one another. The data is placed in contiguous memory, but depending on implementation, may include padding at the end of each loop. Where required, the padding is used to ensure an element is not separated over cache lines, meaning each cache line can be pulled from memory independently of each other. AoS works well when the algorithm is required to process a large number of variables at the same time.

Finally, a hybrid of both of these approaches can be used, called Array of Structures of Arrays (AoSoA, Figure 1c). This data structure interleaves multiple elements of the same variable, between the variables. As such, it can make better use of the cache, if done correctly. AoSoA is also the most flexible data structure out of the three, as the number of elements for each variable loop can differ between variables. For example, rather than each loop having 2 elements from each variable, the loop could contain one element from `x`, four from `y` and two from `z`. However, because of the extra flexibility and the need to fit well into cache to ensure good performance, the data structure is more complex to design and manage.

A. *Data Structure Abstractions*

Two contemporary projects that provide the capability to work with abstract data layouts are RAJA [7] and Kokkos [8]. The goal of both of these tools is to facilitate development of performance-portable applications that can execute on a wide range of hardware and achieve good performance. In this paper, we instead make data storage and manipulation our primary focus. We aim to inject domain-specific knowledge into the library, and make converting between different layouts a first-class feature. The functionality provided by Warwick Data Store is therefore orthogonal to that offered by Kokkos and RAJA.

Specialised data storage libraries have also been designed, but these have been for specific use cases. One such example of this is Atlas, designed by European Centre for Medium-Range Weather Forecasts (ECMWF) [9]. This library is designed to store unstructured mesh data within climate and weather simulations, and provides a variety of layout options depending on the type of discretisation used. Warwick Data Store aims to support a wider range of applications.

Another example is Axom, developed by the Lawrence Livermore National Laboratory [10]. This project aims to provide tools for multi-physics applications, with one such tool being a data management tool called Sidre. Sidre's aim is to allow for transparent data accesses for physics applications across a large range of hardware options [11]. Sidre provides similar functionalities and capabilities as WDS and was developed at the same time, but independently to Warwick Data Store. Sidre's development shows the demand in a library that performs these data structure abstractions.

Libraries have also been created to abstract the data layout, allowing for auto-vectorisation, more utilisation of bandwidth and thus higher performance. The most commonly used library is Intel's SIMD Data Layout Templates (SDLT) [12]. While the main aim for this library is to manipulate the data in order to increase performance, Warwick Data Store aims to extend the available features, such as the ability to convert between data structures, and to allow more flexibility in how the data is defined. One such example of WDS' flexibility is that domain-specific data structures can be created within the library, such as those required for multi-material physics applications [13].

B. *Multi-material data structures*

In real-world problems of interest, it is common to find highly specialised data structures that have been carefully

designed to address a particular issue. Examples include lock-free hash tables for k-mer counting in Bioinformatics [14], Morton-ordered texture caches in computer graphics [15], and *Compressed Sparse Row* (CSR) data structures for sparse linear algebra [16].

Another example, which we consider in some detail here, are *interface tracking algorithms* in solid-fluid mechanics applications. Numerical methods designed for such applications often run into difficulties treating the sharp discontinuities in state variables that occur at boundaries between two distinct physical materials. Interface tracking methods are a broad family of approaches designed to ameliorate these issues by keeping a record of exactly where such boundaries are located, and applying correction terms to the solution variables in these areas. The methods used to store this boundary information are sometimes termed *multi-material data structures* [13].

A key design goal for WDS is to provide sufficient flexibility that specialised domain-specific data structures such as those required for multi-material problems can be efficiently described and stored using its mechanisms.

III. Warwick Data Store (WDS)

The Warwick Data Store (WDS) library is a template C++ library designed to allow for the abstraction of data structures within applications [5]. By doing this, we can manipulate and optimise the data structure without needing to change the programs code. We can also implement additional functionality that would be time-consuming to implement, such as converting between different data structures. When developing WDS, we focused on the core functionality alongside four key criteria: extensibility, minimal size of library, ease of implementation into applications, and the performance impact of the library. By ensuring these four criteria are met alongside the core functionality, we can ensure that the library maximises its effectiveness and usability to the user.

The library has been designed to be as extendable as possible, allowing for a wide range of applications. In order to achieve this, the library consists of three collection of classes; the high-level controller, a hierarchy of classes designed to allocate and manage the memory (the variable classes), and a hierarchy of classes to allow for quick and easy access to the memory (the view classes). By splitting the library in this way, we have the ability to extend one without impacting the functionality of the others. This allows for new data structures to be done with relative ease. Both the variable and view classes consist of interface classes, that should be extended when developing a new data structure. This provides basic functionality, but will not be performant due to inheritance and VTable lookups. Thus, the functionality in `WDS`, `Controller`, `ViewSpec` and potentially `View` classes should be extended to include the new data structure variable and view classes. Each of these three collection of classes can be seen in Figure 2.

The high level functionality collection of classes manages large requests from the users, such as allocation and management of the underlying variables. This includes the setup

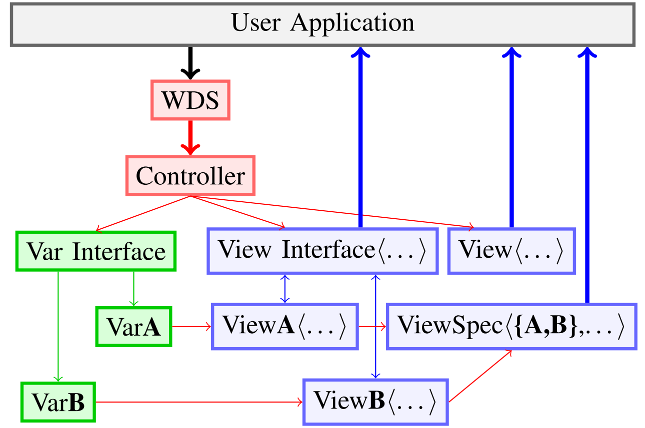


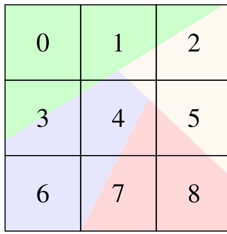
Fig. 2: Control flow diagram of WDS. **Bold** arrows shows the flow of data to and from the user. The red classes provides the high level functionality to the user. The green classes are variable classes. The blue classes are view classes.

of variables, the changing of dimensions and converting a collection of variables from one data structure to another. By not allowing the user direct access to the variable management classes, we can extend the available data structures the user can use, without affecting applications which do not require this. Because this abstraction can see all variables and possible data structures, adding and extending high-level functionality becomes simpler compared to if this was done within, or between specific programs.

The variable collection of classes controls the allocation and management of the data itself. These classes are referred to as `Var` within the library. Each of the data structures implemented within WDS is created within a separate class in this collection of classes, and inherits from the parent interface. This ensures the data structures are sandboxed from each other, and that the high level classes can access all of the necessary functionality. These variable classes include everything that would be needed for the data structure, except for how the user accesses the data. This is managed by the final collection of classes, the views.

The view collection of classes controls how the user accesses the data stored within the library. These classes intentionally uses a very similar design to the variable collection of classes. When a variable class is created for a new data structure, at least one new view class should also be created. This allows for a data structure with multiple parts to be represented easily, as each part would be accessed through its own view class. Each of these view classes inherit from a common interface, to allow for the same flexibility as the variable collection, and to allow this interface to be passed to the user. This allows for the appropriate view class to be called, while being data structure agnostic.

While the view interface is a good way to access the variable without worrying about its data structure, it comes at the cost of the code using VTable lookups. This means that, for every data access, the application has to lookup



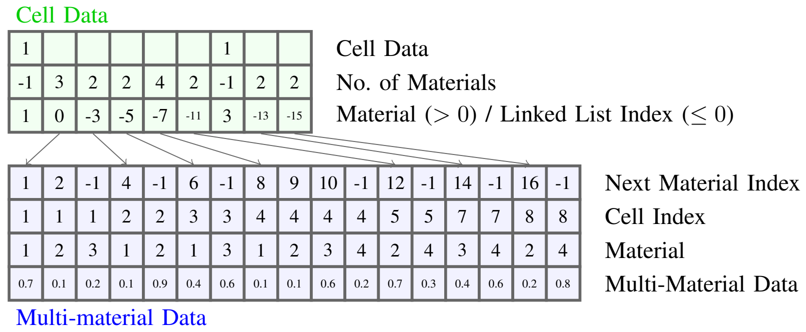
Material 1
(Cells 0, 1, 2, 3, 4)

Material 2
(Cells 1, 2, 4, 5, 8)

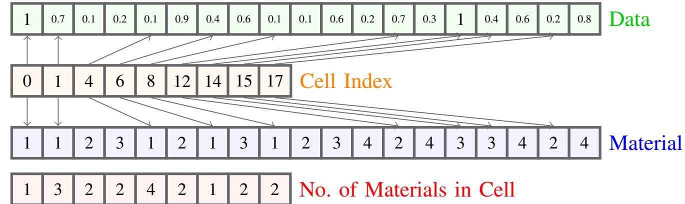
Material 3
(Cells 1, 3, 4, 6, 7)

Material 4
(Cells 4, 5, 7, 8)

(a) Graphic representation of multi-material mesh 3×3 mesh with four materials



(b) Fogerty et al. Compact Cell [13] data structure implementation example using the example mesh



(c) Warwick Data Store Compact Cell Flat data structure implementation using example mesh

Fig. 3: Multi-material example 3×3 mesh with four materials. Each sub-figure shows the same mesh in different representations.

how to access the data. To get around this, two separate view classes were created. The first, simply named `View` is designed to provide data access for any data structure with a uniform striding pattern, and as such, is data structure agnostic. This is very common within data structures, as it allows for memory optimisations such as cache prefetching. The second class is `ViewSpec`. This class allows for direct access to the original view, without having to use VTables. To do this, each of the functions are inlined within specialised versions of `ViewSpec`. As such, the data structure is required to be passed to the library by the user.

WDS has been designed to be easily implemented into an application. Take Listing 1 as an example application. In this block of code, two arrays are created `a` and `b`, which are then passed into a kernel, performing a series of calculations on `a` and `b`. The conversion of this application can be seen in Listing 2.

Listing 1: Example of a kernel to be implemented with WDS

```
int main(int, char **)
{
  // Create data arrays
  // Specify the variables "a" and "b"
  const int len = 250, depth = 4;
  double* a = new double[len];
  int* b = new int[len, depth];
  {
    // Kernels operate on arrays, and has
    // to have knowledge of the underlying
    // layout or the kernel to operate
    kernel(a, b, len, depth);
  }
 return 0;
}
```

As can be seen, there are three main changes between Listings 1 and 2. First is the inclusion of the WDS header file. This ensures the inclusion of all relevant classes are available to the application.

The second and biggest change is the generation of the variables `a` and `b`. WDS builds a queue of variables required by the application through the use of the function `addMeta`. This function takes the variable name, an array of dimensions for the variable, and an example of the type required. Whilst primitive types have been used in this example, user-defined classes would also work. The variables are not created until `buildVar` is called, where a data structure is passed to the library. WDS then takes the queue, sanitises the meta data for the required data structure, then allocates blocks of memory as required.

The third and final difference is the parameters for the kernel. Rather than passing pointers to the kernel, `View` objects are passed. These are generated through the use of the `getView` functionality in WDS, and uses the variable names defined earlier to identify the correct variable objects to generate the required `View` objects. Other than the alteration to the parameters, the only other change required in the kernel function is swapping `[]` for `()`.

Listing 2: Implementation of the code in Listing 1 with WDS

```
#include <wds.hpp>
int main(int, char **)
{
  // Create data store
  wds::WDS datastore;
  // Specify the variables "a" and "b"
  const int len = 250, depth = 4;
  datastore.addMeta("a", {len}, double());
```

```
  datastore.addMeta("b", {len, depth}, int()));
  // Build the variables given the metadata
  // provided, specifying the layout desired
  datastore.buildVar(WDS_DT::SOA);
  {
    // Views used to access and modify data
    auto a = datastore.getView<double>("a");
    auto b = datastore.getView<int>("b");
    // Kernels operate on views without
    // knowledge of the underlying layout
    kernel(a, b, len, depth);
  }
 return 0;
}
```

A. Data Structure Specialisation

One of the key requirements of the data structure abstraction library is that it should be extensible. To demonstrate this, a specialised data structure for multi-material applications has been implemented, in particular, the Compact Cell data structure outlined by Fogerty et al. [13]. The data structure consists of two parts, one for storing all cells containing only a single material and associated metadata, and another for storing multi-material cells in the form of a packed linked list. We have also developed a variant on this entitled Compact Cell Flat, where all the cell data, both single and multi-material, is laid out contiguously in cell order, and then in material order. Some computational kernels do not care whether a given material state is a whole cell or a component of a multi-material cell, so this latter arrangement avoids the need to perform indirect accesses in these cases.

Figure 3 shows the same multi-material example in three different formats. In the first format (Figure 3a), a graphical representation of the mesh can be seen, showing that we are representing a 3×3 mesh with four different materials. The second format (Figure 3b) shows the Compact Cell equivalent of the mesh, consisting of the cell data and multi-material data, and shows how the two blocks are linked. The third section (Figure 3c) depicts the Compact Cell Flat representation of the mesh, and shows how the main components alongside how they are linked.

We commenced by implementing Compact Cell Flat as a separate variable class, with a corresponding view class. The variable class create the arrays outlined in the overview shown in Figure 3c, and outlines how elements from the array can be added and removed. In its API, WDS treats the data as a two-dimensional array, the first dimension being the cell index and the second being the material. The library passes this information to the variable and view classes, which interprets it appropriately, depending on the required functionality. As an example, when adding a new material to the cell, the variable class will use this data to expand the arrays rather than altering the dimensions of the variables. Another example which demonstrates the flexibility of Warwick Data Store is how the dimensions are treated in the Compact Cell Flat view class. WDS allows for that data to be accessed in different ways within the same view, depending on the data structure and the need of the user. For Compact Cell Flat, `[int]` allows for direct access to the data array, `(int)` allows for access to the first element of the given cell index thus allowing the user to iterate through the cell without needing to know the materials in the cell, and `(int, int)` allows for access to the given material for a given cell. If an incorrect element is given, the WDS specification specifies that this should be classed as undefined behaviour.

We additionally implemented Compact Cell as originally outlined. The variable class contains the required arrays for storing the single material and multi-material data, in addition to methods for extending the appropriate arrays for adding new materials. Due to the arrangement of this particular data structure, it would not be possible to easily implement Compact Cell in a single view class. Instead, two views are required, one for accessing data in the single material data, and one for the multi-material data. WDS has the flexibility to achieve this, by allowing a variable class to generate multiple views, and for the controller to allow for multiple views to be created from a single variable class, depending on the user requirements.

One of the key benefits of abstracting the data structure is that we are able to allow the user to carry out large changes to the data with minimal effort. For example, we were able to extend the functionality of the high-level controller to allow for the user to convert between the different Compact Cell implementations. This meant that the user can benefit from iterating through all the data where appropriate, and has the ability to iterate through just the multi-material data without having to write large amounts of conversion code within the application.

IV. Experimental Results

Although the primary goal of Warwick Data Store is to abstract data layout from an application and improve programmer productivity, minimising the impact on application performance was also a significant target. As such, the goal of this section is to show how the overhead incurred by WDS is acceptably low for use in production applications. To achieve this, we will present the overhead of a variety of mini-applications. We will also show the impact of specialisation of data structures within WDS by using these data structures and extra functionality with key multi-material kernels outlined by Fogerty et al. [13]. Finally, we will show that the performance impact of implementing WDS into a full mini-application is minimal, through the presentation of the overheads of multiple applications using different parallelisation techniques and problem sizes.

To ensure that the performance impact of Warwick Data Store is small across a variety of HPC systems, we therefore ran all problems across three different systems, each of which use a different processor, and across two different compilers on each system. Specifically, we use an ARM ThunderX2 system, an Intel Xeon Cascade Lake AP system, and a AMD Rome Epyc system. Details for each of these systems can be seen in Table I. Each of these have varying amounts of cache, bandwidth and processing power, allowing for a wide range of

TABLE I: Systems used to measure the performance impact of WDS

System	Processor	Per Socket			Compiler	MPI
		Cores	STREAM BW	Cache		
Isambard	ARM Marvell Thunder X2	32	116.5 GB/s	32 MB L3	CCE 9.1.3 GNU 9.2.0	Cray MPICH 7.7.12
CLX-AP	Intel Xeon 9242	48	187.3 GB/s	71.5 MB Smart Cache	Intel 19.1.1 GNU 8.2.0	IMPI 7.217
Rome	AMD EPYC 7742	64	176.4 GB/s	256 MB L3	AOCC 2.2.0 GNU 9.2.0	OpenMPI 4.0.3 OpenMPI 4.0.2

architectural differences to be inspected. To calculate the bandwidths, we used the STREAM benchmark [17] with optimisation flags. We used both the `Ofast` and appropriate OpenMP flags across all systems. Where a compiler with streaming stores was available, this was utilised along with the appropriate flag. For the Cascade Lake system, this meant using the Intel 19.1.1 compiler with `qopt-streaming-stores` flag. For Rome, we used AOCC 2.2.0 with `fnt-store` and transparent huge pages tuned off.

For all systems, we run all MPI problems across all physical cores in a node, all OpenMP problems across a single NUMA region within a node, and all hybrid (MPI + OpenMP) problems such that the MPI ranks are allocated to seperate NUMA regions, with OpenMP threads filling each NUMA region. For Isambard, each socket consists of a NUMA region, so when running hybrid problems, two MPI ranks are used, with each rank consisting of one thread per core. As such, each Isambard NUMA region has 32 threads. Each socket in the Cascade Lake node consists of two NUMA regions. This means that when running hybrid problems on the Cascade Lake system, four MPI ranks are used with each NUMA region containing 24 threads. For Rome, the processor was split into four NUMA regions. The hybrid runs were achieved by splitting the processor further, and using a single MPI rank per L3 region, consisting of four cores. Each NUMA region consists of four L3 regions, so a configuration of 32 MPI ranks, each with four OMP threads, was used. It should be noted that the AMD processor can be configured to consist of one NUMA region if required.

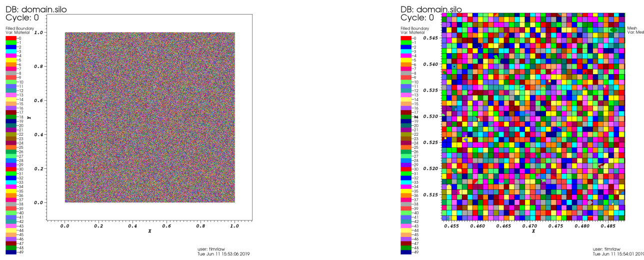
To ensure we measure just the performance impact of the Warwick Data Store, we convert the kernels and mini-applications from the reference version to a WDS version. For this, we aim to make as few changes to the applications logic as possible. This allows for a direct comparison between both of the versions. The only exception to this is the multi-material example, where the reference version uses the reference Compact Cell, while WDS uses both Compact Cell and Compact Cell Flat data structure as appropriate.

In order to compare the effectiveness of WDS, we measure the effect of the library by analysing the overhead. To calculate this, we take the library time and divide this by the reference time. This provides us with a time and scale independent metric to track how well the library performs.For the majority of tables, both the time taken in seconds for the reference version to complete, and the overheads, have been given. These have been labelled as *Ref* and *%* respectively. If no indication is given, the results presented is the overhead. For all results tables within this section, a colour scheme has been used for overheads to show the difference in results. All green cells are values below 10%, orange cells are values between 10% and 30% inclusive, and red cells are values above 30%. The aim for the library is to get the overhead as low as possible. It should be noted that we expect to see trends across compilers and architectures.

A. *Multi-Material Kernel Results*

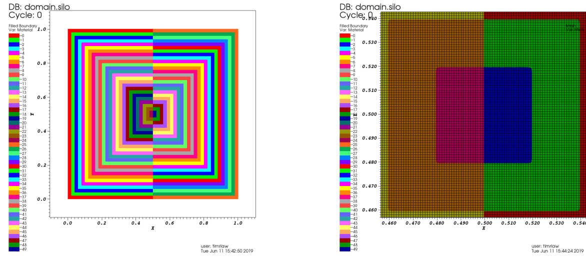
One of the key features of Warwick Data Store is that new data structures can be added with little cost to performance, and are able to perform transformations transparently to the user. To demonstrate this, we implement two multi-material data structures into WDS, as described in Section III-A. To ensure the data structure validates, and to test the performance of these data structures, we used two multi-material kernels. The first takes an average of all multi-material cells in a mesh, and stored this in a single-material array. The second performs an EOS material lookup. Two versions of the second kernel were made; one which iterated though the cells and found the EOS material it corresponded to, and the other iterated through the EOS materials and then searched for cells with that material. By using these kernels in this way, we demonstrate the library can handle the two most common ways in which multi-material data structures are used, either iterating through single/multi-material cells, and iterating through all data regardless of whether the cell is single or multi-material.

In order to exhaustively test the multi-material data structure and kernels, we use the two extreme meshes outlined by Fogerty et al. [13]. The first is a randomised mesh, with a given proportion of cells containing two, three and four materials within a cell. In particular, 20% of cells were randomly picked to be multi-material cells. Of these multi-material cells, 62.5% were allocated two materials, 25% were allocated three materials, and the remaining 12.5% were allocated four materials. A visual representation captured using the reference data structure can be seen in Figure 4. The second multi-material mesh is a geometric patterned mesh, as seen in Figure 5. This consists of a much lower portion of multi-material cells to single material cells, compared to the randomised mesh. To ensure the kernels validate when using both meshes, and in



(a) Whole Mesh (b) Zoomed Portion of Mesh

Fig. 4: Multi-material random mesh example



(a) Whole Mesh (b) Zoomed Portion of Mesh

Fig. 5: Multi-material geometric mesh example

order to guarantee a fair comparison, the mesh is generated once, and duplicated for both the reference and WDS versions.

Table II shows the kernels and which WDS data structure was used for each. Specifically, we use the Compact Cell data structure outlined by Fogerty et al. for the average kernel, and the specialised version for the EOS kernels. For all reference runs, the Compact Cell data structure is used.

We would expect to see a small overhead for the average kernels, due to the addition of the libraries View object. However, we would also expect to see a speedup when using the library, as all the required memory in the Compact Cell Flat data structure is concurrent in one block, rather than split over two blocks. From Table II, we can see that the average kernels have a small overhead. We can also see that the overhead for the EOS kernels have a negative overhead for the majority of systems and configurations, with the lowest being -32.1%. This negative overhead is most likely due to the fact that under the reference version, the EOS kernel has to be called twice (once for the single-material data, then again for the multi-material data), whereas the Compact Cell Flat version only needs to be called once. As well as this, the

TABLE II: Results of multi-material kernels within Benchmarking suite, across different architectures and compilers

Mesh	Kernel	WDS Data Structure	Isambard		CLX-AP		Rome	
			Cray	GNU	Intel	GNU	AOCC	GNU
Random	Average	Compact Cell	2.03	-2.74	-1.54	-7.46	5.30	11.7
	EOS (Cell)	Compact Cell Flat	-14.5	-8.74	-30.8	-25.3	-25.2	-21.4
	EOS (Mat)		4.74	-10.6	17.8	-0.55	8.81	3.42
Geometric	Average	Compact Cell	-1.79	-3.32	-2.44	-6.35	7.78	16.2
	EOS (Cell)	Compact Cell Flat	-3.57	-1.95	-5.83	-2.09	6.37	7.48
	EOS (Mat)		-6.61	-27.5	-23.7	2.33	-45.5	-32.1

TABLE III: Input sizes for small and large problems across all mini-aplications

Mini-app	Input Deck	Small (S)	Large (L)
BookLeaf	Noh Problem Size Sedov Problem Size	1200×60 179×179	2530×126 566×566
MiniMD	Problem Size Timesteps	$64 \times 64 \times 64$ 1000	$128 \times 128 \times 128$ 500
TeaLeaf	Problem Size Timestep	1000×1000 20	8000×8000 10

Compact Cell Flat data structure ensures all the valid data is contiguous allowing for better utilisation of vectorisation. This negative overhead is across all architectures, as expected. Whilst the data is accessed in contiguous order for Compact Cell, there are gaps where data does not need to be processed. Finally, when looking at the version of the EOS kernel which iterates through the cells first, the reference version is required to check that there is a valid material at all given positions before performing the calculations. This is not required in the Compact Cell Flat version, as the data structure ensures all pieces of data has a corresponding material.

There is also a difference between the two EOS kernels for each of the meshes. For the random mesh, the method of iterating through cells and then locating the materials performs better than the reverse. The opposite is true for the geometric mesh. However, on the kernels where the mesh does not match the best EOS kernel, the overhead is not large across all architectures.

B. Mini-applications with Warwick Data Store

In order to demonstrate how WDS performs in a more realistic setting we have implemented the library into three different mini-applications [18]: small and self-contained codes that are designed to be representative of larger applications, and can be used to test and develop new ideas in an agile way. We aim to demonstrate that using WDS incurs a low overhead relative to the reference versions of these applications. We have chosen BookLeaf [19], MiniMD [20] and TeaLeaf [21] for this study, which we describe in more detail below.

We use a small and large input deck for each mini-application in order to explore how the size of the problem affects the performance overhead of WDS. The different problem configuration for each mini-application can be seen in Table III.

Each run is performed five times, and the average times are used in our results. For each mini-application, WDS is set to run using the same data structure as the mini-application originally used. This is to ensure a fair comparison between the original mini-application runtime, and the runtime of the mini-application with WDS integration.

1) BookLeaf: BookLeaf [19], [22] solves the compressible Euler equations on an unstructured grid using an Arbitrary Lagrangian-Eulerian (ALE) formulation. These equations describe the dynamics of inviscid fluids, and are used widely to solve many problems in science and engineering. We use two

TABLE IV: Results for BookLeaf input decks across architectures, compilers and input decks.

Input Deck		Result	Isambard		CLX-AP		Rome	
			Cray	GNU	Intel	GNU	AOCC	GNU
Noh	S	Ref	32.0	35.4	15.9	19.9	16.8	20.4
		%	16.3	6.21	26.3	18.2	31.9	17.4
	L	Ref	62.7	63.2	48.9	49.1	112	112
		%	6.69	1.51	4.16	6.08	1.64	0.99
Sedov	S	Ref	8.00	12.4	4.69	5.59	5.27	5.68
		%	19.7	4.91	37.7	19.0	31.0	20.9
	L	Ref	57.1	54.1	46.1	43.5	100	99.8
		%	4.31	3.12	-1.50	5.85	2.16	1.49

classic test problems in our experiments: Sod's shock tube [23] and Noh's cylindrical artificial viscosity problem [24]. In this paper, we utilise the C++ OpenMP version of the code as the base for the WDS version, and also the reference version used to compare the performance. Implementing WDS into BookLeaf was relatively easy, as the reference C++ version of BookLeaf contains a data store abstraction already built into it. Thus, to implement WDS into BookLeaf involved swapping the reference data store for WDS.

Most routines within BookLeaf have a low arithmetic intensity, meaning that the code is typically memory-bound. As such, we expect to see a larger overhead with smaller problem sizes, and a relatively small overhead for larger problems. This is due to the fact that on larger problems, the processor will be more memory constrained, allowing for computation to be done in the time the processor is waiting on data.

Table IV shows the overheads for all variations of problem decks, processors and compilers. As expected, for both Noh and Sedov problems, the small problem sets have a larger overhead, than the large problem sets. This is independent of both the system and the compiler, though there is some fluctuations in how the compilers performed on the small problem sets. For the large problem set, the compilers produced close to the same overhead on the same system and across architectures.

Across both BookLeaf problem sizes, the WDS times are very similar across the different compilers within a given architecture. However, with the smaller problem size, the runtimes are shorter than there large problem size counter part, making the overhead more sensitive to differences. This is why there is a larger range of overheads for the smaller problem sizes, and why the slower runtime within an architecture has the smaller overhead. Even with these factors, it is clear that the overheads for the smaller problem is greater than the larger problems across all architectures and compilers.

2) TeaLeaf: TeaLeaf [25] solves the linear heat conduction equation on a structured grid. Parabolic equations like this are often solved using implicit methods, requiring the use of a linear solver. TeaLeaf's primary purpose is to support experimentation with different types of linear solver in a simple setting [21]. A C/C++ version has been created by the University of Bristol [26], which we use here, specifically the MPI+OpenMP variant. We focus on the CG, PPCG and

TABLE V: Results for all TeaLeaf solvers, across all parallelisations methods, architectures and compilers.

Input Deck			Result	Isambard		CLX-AP		Rome	
				Cray	GNU	Intel	GNU	AOCC	GNU
CG	MPI	S	Ref	5.82	3.97	0.65	0.99	1.51	0.75
			%	12.1	12.2	39.5	30.5	3.14	53.1
		L	Ref	1877	1441	862	854	1181	1182
			%	0.45	-1.38	-1.27	1.81	-0.52	-0.36
	OMP	S	Ref	12.3	11.3	4.17	7.75	8.22	
			%	3.25	-3.68	1.78	2.81	0.54	
		L	Ref	3614	3106	3398	3369	9362	
			%	2.05	0.63	-2.01	-0.48	-0.05	
	Hybrid	S	Ref	6.96	7.90	1.27	4.63	1.56	0.82
			%	4.24	-8.44	12.1	1.92	3.84	6.47
		L	Ref	1885	1591	838	839	1178	1180
			%	0.16	0.61	0.99	1.25	-0.17	-0.33
Chebyshev	MPI	S	Ref	1.82	1.96	0.33	0.53		0.37
			%	33.8	8.64	65.5	43.0		39.1
		L	Ref	1583	1501	808	805	1111	1111
			%	0.25	0.19	-2.07	-0.54	-0.10	-0.32
	OMP	S	Ref	4.45	5.35	2.48	3.02		2.46
			%	4.46	-1.13	2.92	-7.40		0.33
		L	Ref	3238	3454	3179	3204	8855	8856
			%	0.83	0.95	-0.96	-0.40	0.00	0.00
	Hybrid	S	Ref	1.88	3.32	0.66			0.41
			%	10.5	-8.15	13.9			16.9
		L	Ref	1647	1763	813	814	1113	1113
			%	1.03	2.02	-2.64	0.90	-0.28	-0.15
PPCG	MPI	S	Ref	2.54	2.74	0.41		0.59	0.55
			%	23.1	8.03	66.5		39.8	42.2
		L	Ref	2007	1970	1055	1055	1474	1479
			%	1.46	4.41	0.83	2.34	-0.25	-0.24
	OMP	S	Ref	4.01	6.25	1.98		1.89	2.47
			%	7.09	-3.16	1.63		10.4	2.02
		L	Ref	3562	4407	3669	3675	10348	10291
			%	2.68	0.26	-0.38	-0.42	-0.12	0.04
	Hybrid	S	Ref	2.33	4.33	0.74		0.58	0.60
			%	6.15	-8.40	8.64		34.9	24.1
		L	Ref	1824	2240	911	920	1379	1373
			%	8.26	4.16	0.88	2.59	-0.22	-0.08

Chebyshev solvers.

Implementing WDS into TeaLeaf was more difficult than the implementation of WDS into BookLeaf. For TeaLeaf, WDS replaced the internal data structure `Chunk`. The kernels parameters were then altered to pass the relevant view objects by reference. Finally, the kernels were altered to use `()` when accessing data, rather than `[]`.

TeaLeaf, like BookLeaf, is typically memory bound. As such, we expect the smaller problem size to have a larger overhead than the larger problem size, as less computation can be shadowed by memory accesses. Table V shows the overhead of the WDS version of TeaLeaf against the reference version. We can see that, for the majority of cases, the smaller problem sizes incur a higher overhead than the larger problem size counterpart regardless of architecture.

Out of all three parallelisation methodologies, we see that the MPI implementation incurs the highest overheads, followed by the the hybrid implementation then the OpenMP 3. This is especially true in the smaller problem sizes. For the smaller problem sizes, this is due to the fact that the communications take up a large fraction of the runtime. However, in the larger problems, the computation kernels take up a much larger proportion of the runtime, compared to the

TABLE VI: Results for MiniMD input decks, across all architectures and compilers.

Input Deck		Result	Isambard		CLX-AP		Rome	
			Cray	GNU	Intel	GNU	AOCC	GNU
MPI	S	Ref	19.7	17.7	6.89	6.20	5.74	5.77
		%	6.33	1.37	0.41	6.20	0.72	0.81
	L	Ref	81.4	73.5	28.5	26.5	22.6	22.3
		%	5.54	-0.02	2.93	5.58	0.54	1.63
OMP	S	Ref	59.9	71.1	48.6	39.7	45.4	46.6
		%	3.61	2.22	11.5	10.1	6.50	5.21
	L	Ref			198	164	192	197
		%			12.0	9.31	6.26	5.74
Hybrid	S	Ref	33.9	40.1	14.1	11.5	6.49	6.59
		%	3.58	2.42	10.1	8.02	7.52	6.69
	L	Ref	124	147	51.3	42.9	26.0	26.6
		%	3.00	1.76	11.7	8.05	7.42	6.30

communications. Because this is not as much of an issue for the hybrid version, and not an issue at all for the OpenMP implementation, we see lower overheads.

Much like BookLeaf, the runtimes for TeaLeaf are consistent across compilers on the same architecture for the majority of cases. This is true for both problem sets, but is more prominent in the smaller problem size. When taking the looking at the variance for the larger problem size, we can see that the results overlap for the majority of cases. This means that these results will have a small, if not 0% overhead.

3) MiniMD: MiniMD is a proxy-application for the much larger LAMMPS molecular dynamics (MD) code developed and maintained by Sandia National Laboratories [27]. MD codes such as LAMMPS are widely used by scientists to study the microscopic properties of matter. MiniMD is designed to use the same algorithms as its parent code, but has been structured to be much simpler to support co-design. The mini-application supports two inter-atomic potentials: the Lennard-Jones potential and the embedded atom model (EAM). For the purposes of this paper, we test WDS with a simulation using the Lennard-Jones potential and MPI+OpenMP.

The results for MiniMD across all systems and compilers running on a single node can be seen in Table VI. As we can see, the overhead for the application does not go above 12% for any configuration of either problem size. Some of the overheads are slightly negative. This is most likely due to machine fluctuations. We also see that the OpenMP 3 implementation has the largest range of overheads, with the OpenMP + MPI approach coming in second for all systems.

V. Conclusion

In this paper, we have shown how a data structure abstraction library can be built in such a way that allows transparent data access to the user, incurs minimal cost to the performance of different applications, and provides additional functionality that would otherwise be time-consuming to develop and inflexible to use. We have shown Warwick Data Store, a data structure abstraction library that has been designed to allow for flexibility and transparent operations to applications and users. Through the use of multi-material data structures, we have demonstrated how additional data structures can be added without affecting the performance of other data structures or applications, and how additional functionalities can be included in to additional ease to the user. As well as this, we shown how the library can be expanded to include specialised data structures by using different multi-material data structures, and how some of the additional functionality, such as the ability to change between different data structures with minimal user interaction, can be achieved.

In order to show the performance affect of Warwick Data Store, we implemented the library into a variety of benchmark kernels and mini-applications, and tested WDS across multiple different architectures and compilers. We showed that the library impacted the multi-material kernels to a small degree, and that altering the data structure can give performance improvements. When used within a mini-application, we found that WDS generated a higher overhead when used on small problems, especially when the problem is more memory bound than compute bound. The overhead of the library is small when used on larger problems, not going above 5% in most cases.

A. Future Work

We aim to push the library into more memory-centric domains and optimisations. In particular, we aim to allow for better optimisation of different NUMA regions and High Bandwidth Memory, including those found on GPUs and Intel Xeon Phi Knights Landing systems. We also aim to develop the multi-material kernels into a full mini-application, to further test the specialisation of Warwick Data Store, and to further develop different multi-material data structures including a material centred data layout. We can then see the performance improvements of changing data layouts for different kernels, compared to the cost of data structure transformations. We also plan to further investigate the higher overheads presented. Finally, we aim to investigate whether Just-In-Time (JIT) compilation techniques would allow for better compiler optimisations when utilising WDS, that would not be possible otherwise.

Acknowledgements

Isambard is a UK National Tier-2 HPC Service based at the University of Bristol, operated by GW4 and the UK Met Office, and funded by EPSRC.

CLX-AP is an Intel development system managed and funded by Intel UK. We are grateful for the support of Dr. Andrew Mallinson at Intel for his comments and guidance.

This work was supported by the UK Atomic Weapons Establishment under grant CDK0724 (AWE Technical Outreach Programme). Prof. Stephen Jarvis is an AWE William Penney Fellow. This work would not have been possible without the assistance of a number of members of the Applied Computer Science team at AWE, to whom we would like to express our gratitude.

REFERENCES

[1] F. Limited, "Fujitsu Presents Post-K CPU Specifications," https://www.fujitsu.com/global/about/resources/news/press-releases/2018/0822-02.html (Accessed: 5th July 2020), 2018.

[2] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," NVIDIA, Santa Clara, CA, United States, Tech. Rep., 2020, V1.0.

[3] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal" for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, October 2013.

[4] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis, "Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. Honolulu, Hawaii, USA: IEEE Computer Society, Los Alamitos, CA, September 2017, pp. 834–841.

[5] R. O. Kirk, T. R. Law, S. Maheswaran, and S. A. Jarvis, "Warwick Data Store: A HPC Library for Flexible Data Storage in Multi-Physics Applications," in *SuperComputing 2019*, Denver, CO, USA, November 2019.

[6] R. F. Bird, P. Gillies, M. R. Bareford, J. A. Herdman, and S. A. Jarvis, "Mini-App Driven Optimisation of Inertial Confinement Fusion Codes," in *2015 IEEE International Conference on Cluster Computing*. Chicago, IL, USA: IEEE Computer Society, Los Alamitos, CA, October 2015, pp. 768–776.

[7] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. LLNL-TR–661403, 2014.

[8] H. C. Edwards and C. R. Trott, "Kokkos: Enabling Performance Portability Across Manycore Architectures," in *Extreme Scaling Workshop (XSW'13)*. Boulder, CO: IEEE Computer Society, Los Alamitos, CA, August 2013, pp. 18–24.

[9] W. Deconinck, P. Bauer, M. Diamantakis, M. Hamrud, C. Kühnlein, P. Maciel, G. Mengaldo, T. Quintino, B. Raoult, P. K. Smolarkiewicz, and N. P. Wedi, "Atlas: A library for numerical weather prediction and climate modelling," *Computer Physics Communications*, vol. 220, pp. 188–204, November 2017.

[10] LLNL, "Axom," https://github.com/LLNL/axom (Accessed: 4th July 2020), 2020.

[11] ——, "Sidre User Documentation," https://axom.readthedocs.io/en/develop/axom/sidre/docs/sphinx/index.html (Accessed: 4th July 2020), 2020.

[12] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier Science, 2016, pp. 251–267.

[13] S. Fogerty, M. Martineau, R. Garimella, and R. Robey, "A comparative study of multi-material data structures for computational physics applications," *Computers & Mathematics with Applications*, vol. 78, no. 2, pp. 565–581, July 2019.

[14] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, January 2011.

[15] D. S. Wise, "Ahnentafel Indexing into Morton-Ordered Arrays, or Matrix Locality for Free," in *Euro-Par 2000 Parallel Processing*. Munich, Germany: Springer, Berlin, Heidelberg, August 2000, pp. 774–783.

[16] J. B. White and P. Sadayappan, "On improving the performance of sparse matrix-vector multiplication," in *Proceedings Fourth International Conference on High-Performance Computing*. Bangalore, India: IEEE Computer Society, Los Alamitos, CA, December 1997, pp. 66–71.

[17] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[18] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[19] D. R. Truby, S. A. Wright, R. Kevis, S. Maheswaran, J. A. Herdman, and S. A. Jarvis, "BookLeaf : an unstructured hydrodynamics mini-application," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, November 2018, pp. 615–622.

[20] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2009–5574, 2009.

[21] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An Evaluation of Emerging Many-Core Parallel Programming Models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*. New York, NY, USA: ACM, 2016, pp. 1–10.

[22] T. R. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. A. Herdman, and S. A. Jarvis, "Performance Portability of an Unstructured Hydrodynamics Mini-application," in *Proceedings of 2018 International Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*. Dallas, TX: Association for Computing Machinery, New York, NY, November 2018.

[23] G. A. Sod, "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, April 1978.

[24] W. F. Noh, "Errors for calculations of strong shocks using an artificial viscosity and an artificial heat flux," *Journal of Computational Physics*, vol. 72, no. 1, pp. 78–120, September 1987.

[25] UK-MAC, "TeaLeaf," https://github.com/UK-MAC/TeaLeaf (Accessed: 20th June 2020), 2020.

[26] University of Bristol, "TeaLeaf," https://github.com/UoB-HPC/TeaLeaf (Accessed: 18th June 2019), 2019.

[27] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

APPENDIX
ARTIFACT DESCRIPTION/ARTIFACT EVALUATION

A. Summary of the Experiments Reported

We ran two multi-material kernels and three different physics proxy applications. As well as this, counterparts with Warwick Data Store were also executed and presented in this paper. Each of these executions were done on three different systems, and two compilers. For each multi-material kernel, the problem was executed ten times, taking the minimum as the recorded value. For each proxy application, the program was executed using two different input decks, each being ran five times, removing any outliers. The average runtime reported by the program is presented within the paper.

B. Artifact Availability

1) Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

2) Hardware Artifact Availability: There are no author-created hardware artifacts.

3) Data Artifact Availability: There are no author-created data artifacts.

4) Proprietary Artifacts: No author-created artifacts are proprietary.

5) List of URLs and/or DOIs where artifacts are available:

- Reference BookLeaf C++: `https://github.com/UK-MAC/BookLeaf_Cpp`
- Reference TeaLeaf: `https://github.com/UoB-HPC/TeaLeaf`
- reference MiniMD: `https://github.com/Mantevo/miniMD`

All other code is available upon request.

C. Baseline Experimental Setup, and Modifications Made for the Paper

1) Relevant hardware details:

- Isambard:
  - Proccessor: ARM Marvell ThunderX2
  - Processor configuration: Dual socket system
  - Compilers:
    * CCE 9.1.3
    * GNU 9.2.0
  - MPI: Cray MPICH 7.7.12
- CLX-AP:
  - Proccessor: Intel Xeon 9242
  - Processor configuration: Dual socket system
  - Compilers:
    * Intel 19.1.1
    * GNU 8.2.0
  - MPI: IMPI 7.217
- Rome:
  - Proccessor: AMD EPYC 7742
  - Processor configuration: Dual socket system
  - Compilers:
    * AOCC 2.2.0
    * GNU 9.2.0
  - MPI:
    * When running AOCC 2.2.0 - OpenMPI 4.0.3
    * When running GNU 9.2.0 - OpenMPI 4.0.2

2) Input Decks: **BookLeaf (Noh, Small)**

```
TIME:
    time_start: 0.0
    time_end: 0.6
    dt_initial: 1.0e-4
    dt_max: 1.0e-2
    dt_min: 1.0e-8
    dt_g: 1.02

HYDRO:
    cfl_sf: 0.5
    cvisc1: 0.5
    cvisc2: 0.75

EOS:
    - { type: ideal gas,
        params: [1.6666667] }

CUTOFF:
    zcut: 1.0e-40
    accut: 1.0e-40
    pcut: 1.0e-40

MESH:
    type: LIN2
    dims: [1200, 60]
    material: 0
    sides:
        - - { type: LINE,  bc: SLIPY,
            pos: [0.005,0.0,1.0,0.0] }
        - - { type: ARC_A, bc: FREE,
            pos: [1.0,0.0,0.0,1.0,0.0,0.0] }
        - - { type: LINE,  bc: SLIPX,
            pos: [0.0,1.0,0.0,0.005] }
        - - { type: ARC_C, bc: WALL,
            pos: [0.0,0.005,0.005,0.0,0.0,0.0] }

INDICATORS:
    regions:
        - { type: mesh, name: air }
    materials:
        - { type: mesh, name: air }

INITIAL_CONDITIONS:
    thermodynamics:
        - { type: material, value: 0,
            density: 1.0,
            energy: 0.0 }
    kinematics:
        - { type: background, value: 0,
            geometry: radial,
            params: [-1.0, 0.0, 0.0] }
```

BookLeaf (Noh, Large)

```
TIME:
    time_start: 0.0
    time_end: 0.1
    dt_initial: 1.0e-4
```

```
    dt_max: 1.0e-2
    dt_min: 1.0e-8
    dt_g: 1.02

HYDRO:
    cfl_sf: 0.5
    cvisc1: 0.5
    cvisc2: 0.75

EOS:
    - { type: ideal gas,
        params: [1.6666667] }

CUTOFF:
    zcut: 1.0e-40
    accut: 1.0e-40
    pcut: 1.0e-40

MESH:
    type: LIN2
    dims: [2530, 126]
    material: 0
    sides:
        - - { type: LINE,  bc: SLIPY,
            pos: [0.005,0.0,1.0,0.0] }
        - - { type: ARC_A, bc: FREE,
            pos: [1.0,0.0,0.0,1.0,0.0,0.0] }
        - - { type: LINE,  bc: SLIPX,
            pos: [0.0,1.0,0.0,0.005] }
        - - { type: ARC_C, bc: WALL,
            pos: [0.0,0.005,0.005,0.0,0.0,0.0] }

INDICATORS:
    regions:
        - { type: mesh, name: air }
    materials:
        - { type: mesh, name: air }

INITIAL_CONDITIONS:
    thermodynamics:
        - { type: material, value: 0,
            density: 1.0,
            energy: 0.0 }
    kinematics:
        - { type: background, value: 0,
            geometry: radial,
            params: [-1.0, 0.0, 0.0] }
```

BookLeaf (Sedov, Small)

```
TIME:
    time_start: 0.0
    time_end: 1.0
    dt_initial: 1.0e-4
    dt_max: 1.0e-1
    dt_min: 1.0e-6
    dt_g: 1.02

HYDRO:
    cfl_sf: 0.5
    cvisc1: 0.5
    cvisc2: 0.75
    kappaall: 0.01

EOS:
    - { type: ideal gas,
        params: [1.6666666667] }

MESH:
    type: LIN1
    dims: [179, 179]
    sides:
        - - { type: LINE, bc: SLIPY,
            pos: [0.0, 0.0, 1.125, 0.0] }
        - - { type: LINE, bc: FREE,
            pos: [1.125, 0.0, 1.125, 1.125] }
        - - { type: LINE, bc: FREE,
            pos: [1.125, 1.125, 0.0, 1.125] }
        - - { type: LINE, bc: SLIPX,
            pos: [0.0, 1.125, 0.0, 0.0] }

ALE:
    zeul: true

INDICATORS:
    regions:
        - { type: background, name: air }
        - { type: cell, name: source, value: 0 }
    materials:
        - { type: background, name: air }

INITIAL_CONDITIONS:
    thermodynamics:
        - { type: region, value: 0,
            density: 1.0,
            energy: 0.0 }
        - { type: region, value: 1,
            energy_scale: volume, density: 1.0,
            energy: 0.2467966 }
```

BookLeaf (Sedov, Large)

```
TIME:
    time_start: 0.0
    time_end: 0.05
    dt_initial: 1.0e-4
    dt_max: 1.0e-1
    dt_min: 1.0e-6
    dt_g: 1.02

HYDRO:
    cfl_sf: 0.5
    cvisc1: 0.5
    cvisc2: 0.75
    kappaall: 0.01

EOS:
    - { type: ideal gas,
        params: [1.6666666667] }

MESH:
    type: LIN1
    dims: [566, 566]
    sides:
        - - { type: LINE, bc: SLIPY,
            pos: [0.0, 0.0, 1.125, 0.0] }
        - - { type: LINE, bc: FREE,
            pos: [1.125, 0.0, 1.125, 1.125] }
        - - { type: LINE, bc: FREE,
            pos: [1.125, 1.125, 0.0, 1.125] }
        - - { type: LINE, bc: SLIPX,
            pos: [0.0, 1.125, 0.0, 0.0] }

ALE:
    zeul: true
```

```
INDICATORS:
    regions:
        - { type: background, name: air }
        - { type: cell, name: source, value: 0 }
    materials:
        - { type: background, name: air }

INITIAL_CONDITIONS:
    thermodynamics:
        - { type: region, value: 0,
            density: 1.0,
            energy: 0.0 }
        - { type: region, value: 1,
            energy_scale: volume, density: 1.0,
            energy: 0.2467966 }
```

TeaLeaf (CG, Small)

```
*tea
state 1 density=100.0 energy=0.0001
state 2 density=0.1 energy=25.0
    geometry=rectangle xmin=0.0 xmax=1.0
    ymin=1.0 ymax=2.0
state 3 density=0.1 energy=0.1
    geometry=rectangle xmin=1.0 xmax=6.0
    ymin=1.0 ymax=2.0
state 4 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=6.0
    ymin=1.0 ymax=8.0
state 5 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=10.0
    ymin=7.0 ymax=8.0

xmin               = 0.0
ymin               = 0.0
xmax               = 10.0
ymax               = 10.0
x_cells            = 1000
y_cells            = 1000

use_cg
use_c_kernels
check_result

eps                = 1.0e-15
max_iters          = 5000

initial_timestep   = 0.004
end_step           = 20
end_time           = 100.0

halo_depth         = 2
num_chunks_per_rank = 1

ppcg_inner_steps   = 350
epslim             = 0.0001
presteps           = 20

*endtea
```

TeaLeaf (CG, Large)

```
*tea
state 1 density=100.0 energy=0.0001
state 2 density=0.1 energy=25.0
    geometry=rectangle xmin=0.0 xmax=1.0
    ymin=1.0 ymax=2.0
state 3 density=0.1 energy=0.1
    geometry=rectangle xmin=1.0 xmax=6.0
    ymin=1.0 ymax=2.0
state 4 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=6.0
    ymin=1.0 ymax=8.0
state 5 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=10.0
    ymin=7.0 ymax=8.0

xmin               = 0.0
ymin               = 0.0
xmax               = 10.0
ymax               = 10.0
x_cells            = 8000
y_cells            = 8000

use_cg
use_c_kernels
check_result

eps                = 1.0e-15
max_iters          = 5000

initial_timestep   = 0.004
end_step           = 10
end_time           = 100.0

halo_depth         = 2
num_chunks_per_rank = 1

ppcg_inner_steps   = 350
epslim             = 0.0001
presteps           = 20

*endtea
```

TeaLeaf (Chebyshev, Small)

```
*tea
state 1 density=100.0 energy=0.0001
state 2 density=0.1 energy=25.0
    geometry=rectangle xmin=0.0 xmax=1.0
    ymin=1.0 ymax=2.0
state 3 density=0.1 energy=0.1
    geometry=rectangle xmin=1.0 xmax=6.0
    ymin=1.0 ymax=2.0
state 4 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=6.0
    ymin=1.0 ymax=8.0
state 5 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=10.0
    ymin=7.0 ymax=8.0

xmin               = 0.0
ymin               = 0.0
xmax               = 10.0
ymax               = 10.0
x_cells            = 1000
y_cells            = 1000

use_chebyshev
use_c_kernels
check_result

eps                = 1.0e-15
max_iters          = 5000

initial_timestep   = 0.004
end_step           = 20
```

```
end_time            = 100.0

halo_depth          = 2
num_chunks_per_rank = 1

ppcg_inner_steps    = 350
epslim              = 0.0001
presteps            = 20

*endtea
```

TeaLeaf (Chebyshev, Large)

```
*tea
state 1 density=100.0 energy=0.0001
state 2 density=0.1 energy=25.0
    geometry=rectangle xmin=0.0 xmax=1.0
    ymin=1.0 ymax=2.0
state 3 density=0.1 energy=0.1
    geometry=rectangle xmin=1.0 xmax=6.0
    ymin=1.0 ymax=2.0
state 4 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=6.0
    ymin=1.0 ymax=8.0
state 5 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=10.0
    ymin=7.0 ymax=8.0

xmin                = 0.0
ymin                = 0.0
xmax                = 10.0
ymax                = 10.0
x_cells             = 8000
y_cells             = 8000

use_chebyshev
use_c_kernels
check_result

eps                 = 1.0e-15
max_iters           = 5000

initial_timestep    = 0.004
end_step            = 10
end_time            = 100.0

halo_depth          = 2
num_chunks_per_rank = 1

ppcg_inner_steps    = 350
epslim              = 0.0001
presteps            = 20

*endtea
```

TeaLeaf (PPCG, Small)

```
*tea
state 1 density=100.0 energy=0.0001
state 2 density=0.1 energy=25.0
    geometry=rectangle xmin=0.0 xmax=1.0
    ymin=1.0 ymax=2.0
state 3 density=0.1 energy=0.1
    geometry=rectangle xmin=1.0 xmax=6.0
    ymin=1.0 ymax=2.0
state 4 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=6.0
    ymin=1.0 ymax=8.0
state 5 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=10.0
    ymin=7.0 ymax=8.0

xmin                = 0.0
ymin                = 0.0
xmax                = 10.0
ymax                = 10.0
x_cells             = 1000
y_cells             = 1000

use_ppcg
use_c_kernels
check_result

eps                 = 1.0e-15
max_iters           = 5000

initial_timestep    = 0.004
end_step            = 20
end_time            = 100.0

halo_depth          = 2
num_chunks_per_rank = 1

ppcg_inner_steps    = 350
epslim              = 0.0001
presteps            = 20

*endtea
```

TeaLeaf (PPCG, Large)

```
*tea
state 1 density=100.0 energy=0.0001
state 2 density=0.1 energy=25.0
    geometry=rectangle xmin=0.0 xmax=1.0
    ymin=1.0 ymax=2.0
state 3 density=0.1 energy=0.1
    geometry=rectangle xmin=1.0 xmax=6.0
    ymin=1.0 ymax=2.0
state 4 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=6.0
    ymin=1.0 ymax=8.0
state 5 density=0.1 energy=0.1
    geometry=rectangle xmin=5.0 xmax=10.0
    ymin=7.0 ymax=8.0

xmin                = 0.0
ymin                = 0.0
xmax                = 10.0
ymax                = 10.0
x_cells             = 8000
y_cells             = 8000

use_ppcg
use_c_kernels
check_result

eps                 = 1.0e-15
max_iters           = 5000

initial_timestep    = 0.004
end_step            = 10
end_time            = 100.0

halo_depth          = 2
num_chunks_per_rank = 1
```

```
ppcg_inner_steps    = 350
epslim              = 0.0001
presteps            = 20

*endtea
```

MiniMD (Small)

```
Lennard-Jones input file for miniMD

lj       units (lj or metal)
none     data file (none or filename)
lj       force style (lj or eam)
1.0 1.0  LJ parameters (epsilon and sigma;
         COMD: 0.167 / 2.315)
64 64 64 size of problem
1000     timesteps
0.005    timestep size
1.44     initial temperature
0.8442   density
20       reneighboring every this many steps
2.5 0.30 force cutoff and neighbor skin
100      thermo calculation every this
         many steps (0 = start,end)
```

MiniMD (Large)

```
Lennard-Jones input file for miniMD

lj          units (lj or metal)
none        data file (none or filename)
lj          force style (lj or eam)
1.0 1.0     LJ parameters (epsilon and sigma;
            COMD: 0.167 / 2.315)
128 128 128 size of problem
500         timesteps
0.005       timestep size
1.44        initial temperature
0.8442      density
20          reneighboring every this many steps
2.5 0.30    force cutoff and neighbor skin
100         thermo calculation every this
            many steps (0 = start,end)
```

3) Paper Modifications: No modifications were made to the hardware or software. Each system was used as is. Further information is available upon request.