
Peer reviewed version

Link to published version (if available): 10.1109/P3HPC54578.2021.00007

Link to publication record in Explore Bristol Research

PDF-document

This is the accepted author manuscript (AAM). The final published version (version of record) is available online via IEEE at https://doi.org/10.1109/P3HPC54578.2021.00007. Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/red/research-policy/pure/user-guides/ebr-terms/
Analyzing Reduction Abstraction Capabilities

Tom Deakin and Simon McIntosh-Smith  
Department of Computer Science  
University of Bristol, Bristol, UK  
Email: {tom.deakin, S.McIntosh-Smith}@bristol.ac.uk

S. John Pennycook and Jason Sewall  
Intel Corporation  
Santa Clara, CA, USA  
Email: {john.pennycook, jason.sewall}@intel.com

Abstract—Reductions are a common pattern in parallel programming, and every parallel programming language or framework has its own reduction abstraction with its own idiosyncrasies. These abstractions differ not only in their syntax, but also in their semantics and their ability to express certain types of reduction. Such differences may prevent specific combinations of abstraction and hardware platform from reaching high levels of performance, with consequences for portability and programmer productivity.

In this paper, we present a set of representative reduction benchmarks to explore the capabilities of five contemporary programming languages and frameworks – OpenMP, Kokkos, RAJA, SYCL, and the oneAPI DPC++ Library (oneDPL) – across a variety of hardware platforms, including CPUs and GPUs from multiple vendors. We discuss the advantages and disadvantages of each reduction abstraction, and conclude with recommendations to improve their design and implementation.

I. INTRODUCTION

The parallel pattern of a reduction is a crucial part of many high-performance computing applications. It is a generalized operation that extends into a higher level of abstraction than most concepts found in parallel programming environments, such as threads and synchronization primitives. For reductions, the best choice of algorithm to use depends on the types, size and structure of the data as well hardware configuration and support for concurrent data updates (such as atomics). It is increasingly common for parallel programming models to provide their own abstraction for reduction operations. By providing said abstraction, it gains the freedom to choose among many suitable implementations for their target platforms.

However, these abstractions vary considerably across programming models. In this study, we explore reduction abstractions with a particular focus on how they affect the performance, portability and productivity of an application. We explore reductions in a number of common parallel programming models and frameworks and see how they affect the implementation of a number of representative benchmark reduction kernels. The models we explore are OpenMP, Kokkos, RAJA, SYCL and Intel’s oneDPL. oneDPL is indicative of the direction of parallelism being introduced into standard programming languages (“stdpar” [1]).

In particular, we make the following contributions:

• We present a new open-source suite of representative reduction kernels. These kernels are derived from typical reduction operations as well as examples modelled on real applications. The code is available on GitHub¹. We describe the kernels in Section III.

• We explore the reduction abstractions used by different parallel programming models and frameworks, and how this impacts the implementation of applications, including custom reduction operations.

• We explore how those abstractions impact the performance portability of applications, in particular as a result of memory abstraction.

II. RELATED WORK

Where parallel programming models provide a reduction abstraction, it is usual that they supply a well-optimized implementation for the appropriate target platforms. There are many reduction case studies (such as [2], [3] to name just two) that compare implementations of reductions on specific platforms. A recent two-part study by Peri looks at seven reductions for DPC++ on Intel GPUs [4], [5].

In contrast, the focus of our study is not on the best way to implement these reductions in the “backend”, but rather on the abstractions themselves, their ability to allow an implementation to write an optimized algorithm, and their impact on performance portability and productivity.

Where the OpenMP programming model does not provide suitable implementations, in particular for sparse reduction operations, the Spray project provides reducer objects encapsulating different reduction schemes [6]. For those sparse operations, the optimal scheme may be dependent not only on the architecture but also the sparsity pattern of the data, and so Spray provides a much greater level of control than the abstractions discussed in this paper.

The study by Gayatri et al. looks at some reduction kernels from the material science code, BerkeleyGW [7]. They compare the performance of using atomic operations and built-in reduction implementations in OpenMP and OpenACC. Of particular relevance is a complex number summation, however the study avoided using custom reductions citing performance impacts; we will address some of the reasons for them in Section IV-F1.

III. TYPES OF REDUCTIONS

To evaluate these reduction mechanisms, we have implemented a variety of reduction benchmark kernels in each of the

¹https://github.com/UoB-HPC/everythingsreduced
frameworks. These kernels are designed to be representative of reductions found in scientific applications. They each represent a particular characteristic of a reduction operation and so can naturally be extended or adapted to more general situations.

Many implementations of reductions distinguish between those that combine results to produce a single variable (i.e., scalar reductions) and reductions that produce a multi-dimensional quantity (i.e., array reductions). We do not consider array reductions in depth in this study; reductions of small arrays are similar to reductions of structures containing uniform types (which we discuss in Section III-C), while reductions of large arrays invite a wide variety of implementation strategies and complexities that we have chosen to avoid to maintain a focused study.

Each of the kernels is predominantly main memory bandwidth-bound when large problem sizes are used. Our benchmark suite includes a simple cache-oblivious performance model for memory bandwidth: bandwidth is calculated using the total number of unique bytes read during the reduction operation, and so represents the sustained memory bandwidth. We do not attempt to count any memory accesses associated with the accumulation of the reduction result – a hypothetical optimal processor could read each data item once and update the reduction variable atomically with no cost. This gives us a reasonable bound on the performance of each kernel.

A. Anatomy of a Reduction

A reduction operation combines multiple input values using a specific reduction operator to produce a single result into a reduction variable. The operator is commonly associative and commutative, although this is not necessarily a requirement (such as subtraction). To simplify discussion of different reduction implementations in this paper, it is useful to consider a reduction as consisting of four steps:

**INITIALIZATION** — Zero or more copies of a reduction variable are initialized. Usually, the initial value is chosen to be the identity value associated with the reduction operator.

**COMPUTATION / ACCUMULATION** — Input values are read, potentially transformed, and accumulated into a copy of the reduction variable using the reduction operator.

**COMBINATION** — Any copies of the reduction variable that exist are combined using the reduction operator to produce a single value. The original reduction variable itself may or may not be included in the combination.

**FINALIZATION** — The result of the combination step is potentially transformed, and the result of the reduction is written to some final output location.

Describing reductions this way may seem unnecessarily verbose, but breaking things apart in this manner will allow us to more precisely compare the reduction abstractions of different frameworks. As we will see, different frameworks provide different levels of control for each of the four steps.

B. Dot product

The dot product of 64-bit floating-point scalars is a very common pattern in numeric codes; two arrays of scalar values are multiplied, element-wise, and accumulated.

\[ r = a \cdot b = \sum_{i \in [0, N]} a_i b_i \]  

The Initialization step is just to set identities under addition; the Computation/Accumulation step is just to set the multiplication and local additions. The Combination step simply further accumulates these, and the Finalization step is just to write the final sum.

While a simple benchmark, its ubiquity makes it a good initial test of any reduction mechanism.

C. Sum of complex numbers

This kernel accumulates an array of complex numbers into a single complex number. Complex numbers are represented by a real and imaginary part, each a floating-point value; adding two complex numbers sums the real and imaginary parts independently. Conceptually, this kernel performs two independent scalar reductions with summation.

\[ z = \sum_{i \in [0, N]} c_i \]  

There are two common representations for an array of complex numbers: an array consisting of structures of two numbers (array-of-structures, or AoS); or two separate arrays of scalars, one for the real and and one for the imaginary parts (structure-of-arrays, or SoA). We have implemented both data structures, as a `std::complex<T>` and a struct of two `T`s, respectively. Some frameworks provide their own complex type, and we use those where they are provided.

We test 64-bit floating-point numbers (`std::complex<double>`); the AoS implementation therefore is 128-bit in size. This may therefore limit the choice of underlying reduction implementation based on the hardware support. For instance, if the hardware supports only 64-bit atomics, a parallel accumulation of double-precision complex numbers must use an alternative scheme to update the scalar reduction result. In contrast, implementing this using a SoA precludes the use of any 128-bit atomics, as the two independent reductions are already exposed in the code.

The steps from Section III-A are the same here as they are for the dot product, independent of SoA or AoS techniques.

D. Minimum value of complex numbers

It is common to consider the magnitude of a complex number; this kernel finds a complex number among those with the smallest magnitude in the given array.

\[ z, \text{ such that } |z| = \min_{i \in [0, N]} |c_i| \]  

The result of this operation is not deterministic for a general input array c; multiple elements may have the minimal
magnitude and therefore correctly satisfy the reduction. This becomes particularly evident when this operation is executed in parallel. This highlights an interesting aspect of reduction abstractions, namely that they provide no explicit restrictions or guarantees on having a deterministic result. A few different implementations have exposed non-standard controls that allow for determinism in specific scenarios where it might be required (at the expense of performance).

This operation is significant because it cannot be separated into multiple independent reductions (as is it for complex sum) and because the reduction operator uses a transformation of the underlying value. We must implement a custom reduction operator; this kernel serves as a proxy for user-defined reductions. We discounted using constructs like Kokkos::minloc to find the minimum value and its location in an array because it had such first-class support in the Kokkos programming model, although such support is rare. Our kernel allows us to explore the challenges associated with defining custom reduction operations, and in particular where the operation occurs in the Combination step.

In terms of the steps from Section III-A, in Initialization, we must set the identity for the custom min operator. The Computation/Accumulation step involves the computation of the magnitude and then computing the minimum of the magnitudes of the accumulator and the candidate value; the Combination step does the same. The Finalization step simply stores the result.

E. Field summary

The Field Summary kernel from the CloverLeaf hydrodynamics mini-app [8] takes a number of input arrays and produces several scalar outputs; this proxy produces five 64-bit floating-point values. The input data is manipulated and combined in various ways to produce the output reduction variables. In each case, each reduction variable is a summation of data derived from one or more of the input arrays. This kernel therefore allows us to test whether a reduction abstraction allows data to be transformed generally to produce computed temporary variables which are the object of the reduction. That is, the data itself is not the object of reduction, but variables derived from various combinations of input data are reduced. It also tests simultaneously reducing multiple such values.

As this is a common pattern arising in many other codes, programming models often fuse transformation and reduction operations to be more convenient for the user, and also remove the need to store the temporary results prior to the reduction.

Field summary has a straightforward interpretation in the decomposition given in Section III-A; in Initialization, each result field is initialized according to the additive identity, then the Computation/Accumulation step has a (non-trivial) amount of computation performed to obtain the values to be reduced, which are then reduced. The Combination step is much simpler, performing only additions for the independent reduction variables, and Finalization is merely to write the output. Note that there are multiple independent (final) reduction variables.

F. Pandas describe

We have implemented a kernel inspired by the Pandas.describe API. This routine takes a data series (an array), and produces common summary statistics: the minimum and maximum value, the mean and the standard deviation. This differs from the other kernels in our suite as there is a single input array which is used for a number of different reduction operations. In particular, order statistics (min/max) and summation (for mean and standard deviation).

The primary motivation for including this kernel is to explore the ability of each programming model to express multiple independent reduction types. Some programming models may provide dedicated support for this case, while others may require the variables to be grouped as a structure or array.

We use the Kahan-Neumaier [9], [10] algorithm for calculating the mean for the OpenMP, Kokkos and RAJA implementations. This is in order to ensure the benchmark validates on CPUs with large input arrays; we believe this is due to the order of operations. Kokkos and RAJA use OpenMP as a backend of CPUs and therefore use this algorithm too. Note that this algorithm relies on using the reduction variable inside the function, i.e. as an OpenMP threadprivate variable containing its local contribution to (copy of) the reduction variable. SYCL and oneDPL, and in some cases RAJA, do not follow this model and this algorithm is unimplementable there. We therefore use the standard single-variable approach for SYCL and oneDPL which did not suffer from the same correctness issues. Accumulating many floating-point numbers sufficiently carefully to avoid round off errors is a known phenomenon and no abstraction to date assists the programmer to mitigate this without relying on implementation-specific hints. For instance, an abstraction might internally choose to use the Kahan-Neumaier algorithm when there is a high risk for floating-point round off errors to occur.

Describe is perhaps the most interesting kernel to consider in terms of the breakdown given in Section III-A. There are two reduction phases as the standard deviation uses the result of the mean, and there is computation besides storing the result in the Finalization step. For the first phase, in Initialization, the addition, min, and max fields must be initialized according to the identities of these respective operators. The Computation/Accumulation step will merely apply min and max for those fields, but the Kahan-Neumaier algorithm does additional work. The Combination step will follow the reduction operators themselves. For Finalization, min and max are trivial, but for computing the mean, the division by the count of entries is sometimes performed as an additional last step rather than as part of the computation.

For the second phase of reduction to compute the standard deviation, the Initialization is the additive identity, the Computation/Accumulation step involves computing the squared difference and adding, the Combination step is simple addition, and the Finalization step is to compute the square root of the result.
IV. LANGUAGES & ABSTRACTIONS

This paper evaluates the reduction abstractions of five different programming languages and frameworks. In this section, we cover some basic background of each language and share observations about how they approach reductions.

The reduction mechanisms we consider in this study each have their own unique abstractions, reflecting the goals of the languages or frameworks to which they belong. Notwithstanding, these abstractions were not developed independently, but were informed by one another.

In our efforts to evaluate and contrast these abstractions, we have consciously avoided opportunities to form further abstractions of our own to improve code re-use or encapsulate multiple would-be “backends”; while this is possible in some cases and may provide productivity benefits for real applications, our focus here is to understand the strengths and weaknesses of each of these languages or frameworks on their own.

A. OpenMP

OpenMP 1.0 [11] was introduced in 1997 as a set of compiler directives for shared memory parallelism, allowing developers to decorate sequential loops to direct compiler transformations. Notably, the reduction directive was already present in OpenMP 1.0. Over time, as OpenMP has grown to support task parallelism, SIMD vectorization and offloading to accelerators, the reduction syntax has evolved too: user-defined reductions in OpenMP 4.0 [12]; array reductions in OpenMP 4.5 [13]; and task reductions in OpenMP 5.0 [14]. Of these changes, only task reductions introduce new syntax and semantics – unlike “traditional” OpenMP reductions, which guarantee a private copy of the reduction variable for each thread or SIMD lane, the task_reduction and in_reduction clauses allow an implementation to create an unspecified number of private reduction variables.

OpenMP is unique among the languages in consideration in many ways: it is by far the oldest language we have considered in this study, and the only meta-language. It is also unique in that—as a product of its age—its heterogeneous/offload capabilities were added well after the initial development of the language and its reduction abstraction; each of the other languages under discussion were developed specifically to deal with heterogeneous computing environments.

We consider two separate implementations for each kernel in OpenMP; one in traditional, “native” form that runs on the CPU only, and another using the OpenMP target capabilities for use on offload devices. While updates in OpenMP 5.0 have required that a CPU “device” be supported in this model, we feel it is important to include both as a reflection of current usage models of OpenMP. In nearly all cases, the target code is a superset of the CPU code; the additional code is concerned with data mapping and movement, as well as additional target directives.

User-defined reductions are necessary for the complex minimum kernel, and these are simple to define in the modern language both for native and target.

OpenMP attempts to allow users to write a minimal amount of annotation if they so desire; it has aggressive firstprivate behavior in many environments that can allow a user to forget about the details of how some data is mapped to an offload read, for example. This is powerful, but can lead to unexpected errors or performance problems.

OpenMP historically has not constrained how reduction variables—the variable marked in the reduction clause with specific reduction semantics—are to be used in the body of code. A user could read the intermediate variable, or perform modifications on that variable other than those specified as the reduction operation. This may limit the implementation’s freedom of choice when generating efficient code, and could cause potential unexpected behavior. On the other hand, this allows for more flexibility on the user side, enabling the Kahan-Neumaier summation strategy described earlier.

The new reductions in OpenMP 5.0 (task-based and in_reduction) expose a more modern interface that explicitly labels this undefined behavior, but the distinct mechanisms co-exist and may be confusing to users.

1) Interaction with offload directives: The use of reductions on target regions is complicated by several factors. First, since a target region is an implicit task, a user must decide whether they are performing a reduction over tasks or a reduction within the task – here, we consider only the latter case. Second, and most importantly, the user must become familiar with OpenMP's rules for mapping variables and how these rules impact data transferred into and out of the reduction kernel.

A naïve OpenMP target reduction may decorate a loop with #pragma omp target teams distribute parallel for reduction(+:x) to reduce a variable x. By default, a target region executes synchronously (from the view of the host), and any variable appearing in a reduction clause is treated as though it appeared in a map(tofrom:) clause. The net effect of these two default behaviors is that x will be transferred to the device, the reduction kernel will execute, and the host will wait until the reduction result is transferred back from the device.

While this is understandable when considered from the legacy of the reduction clause and its semantics on the host, it is often the case that the finalized reduction is to be used on the device, and that control should not be handed back to the host but instead another kernel should be launched. Our describe kernel—where we use the finalized mean value to compute the standard deviation—fits this model.

In order to ensure that the reduction variables are not transferred more times than required, the user must create a data region containing the reduction variables and explicitly describe the desired data transfers. Note that such a kernel chain is still synchronous – enabling asynchronous execution requires yet another clause (nowait), as does describing the data dependencies between the kernels (depend).

OpenMP gives the developer a great deal of flexibility when indicating reductions; so much so, that it can lead to unexpected behavior. Consider a double loop nest performing a reduction. After the developer has specified
omp parallel for reduction(...) on a loop, a number of things can happen with the inner loop. With no further annotation, the compiler may or may not vectorize the inner loop. Regardless, it must still properly perform the reduction. If instead a user chooses to specify pragma omp simd on that inner loop, they must also specify a corresponding (likely identical) reduction clause, or the result is undefined behavior. It would be useful for toolchains to issue warnings when these detectable errors are made by the developer.

B. Kokkos

Kokkos is a performance portability framework introduced in 2011, implemented (primarily) as a header-only library written in C++ [15]. Kokkos classes and functions provide a portable interface hiding lower-level primitives provided by different “backends”; this interface provides a rich set of programming patterns, including parallel loops, scans and reductions. It is important to note that the design of Kokkos is effectively that of an embedded language: a correct Kokkos program should work on any backend, without any changes to user code.

The reduction abstraction in Kokkos is semantically very similar to that of OpenMP, creating a private copy of the reduction variable for each instance of a kernel function. The variable is available for use with its normal type inside the lambda function in arbitrary ways, and is made available through an argument to the kernel lambda function. The reduction combines the value from each iteration (instance) using a summation operation by default, unless the program specifies a different binary operation. Whilst this default behaviour is likely concise, compilers are not able through an argument to the kernel lambda function to identify appropriate patterns from the underlying programming model and apply them to their loops. RAJA does not provide or enforce a general data abstraction, although additional frameworks can be combined with RAJA to provide this.

The reduction abstraction in RAJA is not tied to a particular expression of parallelism, and a user is not required to declare that a given loop contains a reduction. Instead, RAJA introduces the concept of a reduction variable—a variable exposing a limited interface dependent on the type of reduction being performed, that multiple iterations of a RAJA loop can safely update concurrently. Note that user variables must be wrapped in a specific RAJA type to indicate it is a reduction variable—this is the only place where RAJA owns data. The user is responsible for wrapping the variable with a reduction execution policy that is compatible with the execution policy of the parallel loop. By wrapping variables, they may only be used inside the kernel as part of the reduction operation, and cannot be used in arbitrary operations; this is in contrast to both Kokkos and OpenMP which allow this. This is relaxed if OpenMP is used as the backend, since the execution policies simply provide a wrapper around the underlying abstraction. The lack of consistency in this policy creates a risk for writing unportable code. This prevents us from implementing the Kahan-Neumaier summation-base Describe benchmark of Section III-F on GPUs using RAJA.

At the time of writing, RAJA does not provide a mechanism for user-defined reductions. As such, we also cannot implement our complex number minimum kernel described in Section III-D.

RAJA handles the Combination and Finalization stages of the reduction operation with deliberation; the final result is only available on the host after it calls the .get() member function of the reduction variable type. This gives the imple-
mplementation the flexibility to collate the partial copies of the reduction variable only when they are required on the host.

D. SYCL

SYCL is a Khronos industry standard based on C++ [17]. Although designed as a language, SYCL allows for flexibility in implementation—some implementations are library-only [18], [19], while others implement a compiler [20], [21].

The latest SYCL specification introduces a reduction abstraction heavily influenced by Kokkos, RAJA, and other C++ libraries [22]. The result is a mix of the previously described approaches: the user must declare when launching a kernel that it contains a reduction, but the reduction variable must be a `sycl::reducer` object exposing a limited interface.

SYCL is similar to RAJA in that its reduction abstraction provides a restricted interface, preventing users from modifying the reduction variable in ways that are incompatible with the chosen reduction operator. However, unlike RAJA, SYCL still allows user-defined reductions. Since such reductions do not have built-in accumulation operators (e.g. `+=`), users must call the `combine` function. This represents a trade-off in usability and portability: the type system enforces SYCL’s reduction semantics, preventing some mistakes and giving useful error messages, but the abstraction prevents the user expressing certain algorithms (such as the Kahan-Neumaier summation).

1) User-defined reductions: When defining a SYCL user-defined reduction, the user may provide an optional identity value that the implementation can use to initialize temporary accumulator variables. At time of writing, the SYCL specification provides no way to tie a user-defined reduction operator to an identity value, and the user must therefore remember to provide the same identity value to each reduction using their operator. If the SYCL specification were to allow users to provide their own specializations of the `sycl::known_identity` trait, this would simplify the reduction interface and bring it closer (in terms of usability) to OpenMP.

2) Asynchrony and memory movement: The `sycl::reduction` function accepts a buffer or Unified Shared Memory (USM) pointer, enabling the result of a reduction to be written directly into device memory; this allows reductions to be enqueued asynchronously without any dependency on host variables. Although this is very useful, it complicates expression in simple use-cases, and increases cognitive burden: the user must use a device allocation to store reduction results even when the result will be read immediately, and it becomes the user’s responsibility to decide how and when the reduction result should be transferred. This is a striking contrast to the aggressive auto-mapping behavior of the OpenMP library.

3) Basic vs ND-range parallelism: SYCL provides two flavors of its `parallel_for` kernel abstraction: a “basic” form where all instances are independent and can be scheduled in any order; and an ND-range form where instances are grouped together, with access to additional functionality (such as group-local memory). Both of these forms support reductions.

When using the “basic” form, the implementation has freedom to make scheduling decisions (e.g. work-group size) and reduction implementation decisions (e.g. how much group-local memory to use). When using the ND-range form, however, information that the user supplies and decisions made by the implementation become intertwined: the user’s selection of work-group size becomes dependent on how much group-local memory the implementation requires (if any) and vice versa. Although SYCL features query functions to help developers reason about the requirements of compiled kernels, they are arduous to use and do not adequately address this use-case; there is a clear need for SYCL to provide additional functionality to help users choose good work-group sizes.

E. oneDPL

oneDPL builds on SYCL to provide a high-productivity interface, inspired by the parallel algorithms in standard C++ [23]. Instead of writing SYCL kernels directly, developers express work in terms of high-level algorithms operating on C++ iterators. The library then maps these algorithms to SYCL kernels behind the scenes. Most relevant to this paper are the reduce and transform_reduce algorithms: the former performs a reduction over the input data, while the latter performs a reduction over a transformation of the input data. Recent work [1] examines the C++ standard parallel algorithms and its ilk in the performance, portability, and productivity light.

oneDPL adds support for wrapping SYCL buffers into C++ iterators so that they may be used as arguments to the various C++ algorithm functions. Because these buffers may only be accessed through these iterators when so used, code ends up adopting a very functional style. For kernels that involve simple operations that need only access any given input only once per iteration, this restriction is natural and unobtrusive.

However, as the complexity of the operation increases, the verbosity of buffer-based oneDPL code explodes very quickly in a fashion that is familiar to users of pure functional programming models. For stencil-like codes, each offset that will be read must be provided separately. This can quickly lead to dozens of input streams, which must be wrapped into aggregation constructs like `zip_iterator`.

A more imperative style can be used if USM is available, since the iterators can be used to simply control the extents of the computation and shared-memory pointers directly dereferenced in the device code. We exclude this as it requires further benchmarking to ensure buffers and USM attain equivalent performance in practice from all vendors.

Since the iterator model is fundamentally one-dimensional and precludes parallel scheduling into nested loops, two- and higher-dimensional computations can grow cumbersome. One solution is to nest oneDPL operations, one per dimension, but this requires devices with nested kernel launch support, and may carry additional overhead. Another solution is to recover indices from the linear space, much as OpenMP does for its `collapse` construct. Still another is to use custom
multi-dimensional iterators that properly describe the desired iteration space.

Like SYCL, oneDPL has stringent semantics about how reduction variables are used. The `reduce` primitive expects a binary operator with constant arguments to do the combination step and return a new object; this precludes the possibility of using the Kahan-Neumaier summation algorithm for describe and so we use the usual mean algorithm here.

oneDPL and its C++ parallel algorithm relative are very compact and expressive abstractions for simple operations, but they can quickly get cumbersome as the limits of the provided abstraction are tested.

**F. Common observations**

Having spent time developing in and discussing these language and frameworks, and the various reduction abstractions they provide, there are some general themes that deserve comment. To begin with, the lack of consistency among interfaces and their default behaviors and assumptions is a source of continual surprise for the developer working in many languages.

1) **Memory placement controls**: Performance of benchmarks with complex numbers in OpenMP on two-socket CPUs was initially surprisingly low. We determined that allocating arrays with `new std::complex<T>[N]` caused "value-initialization" of array members, as per the rules of C++.

Unfortunately, this is done by a single thread in the runtime library, which causes the first-touch NUMA policy in the operating system to place all of the data in the NUMA domain of one socket. This led to a cross-socket bandwidth bottleneck.

By using `malloc` to allocate data followed by placement `new` in a properly parallelized context to initialize the data, we were able to alleviate the issue. This approach would also be required to ensure good NUMA performance for other non-fundamental types, including user-defined types.

While this is fundamentally due to C++ semantics, it highlights the danger of a language or framework not having robust memory semantics. On the contrary, Kokkos's `view` policy allows it to control memory initialization on the device to suit its needs.

2) **Enforcing reduction semantics**: OpenMP and Kokkos expose reduction variables as their underlying types with no restrictions on semantics, while SYCL and oneDPL restrict the usage of the reduction variables, or hide it altogether. RAJA occupies a vague middle ground that lays out a policy of restricted use that it is not able to enforce consistently.

Unrestricted use of the reduction variable allows for a greater variety of user code in reduction kernels. While scenarios where partial reduction results are desired inside a kernel (as in the Kahan-Neumaier summation) are rare, this allows for that.

However, restricting semantics allows the language or framework more flexibility in choice of implementation, which can lead to better mappings to various hardware targets. Furthermore, restricted semantics allows for more rigorous error checking, and better cross-platform performance.

3) **Finalization support**: None of the abstractions discussed expose a Finalization interface. While far from ubiquitous, non-trivial Finalization steps do occur (e.g., in Describe's mean and standard deviation computations) and managing the data dependency and possibly involving a host device to finish the computation is cumbersome and potentially harmful to performance. Indeed, some of the abstractions provide the developer no way to control when the finalized result may be ready or where it will be transferred.

4) **Custom operators**: Most of the programming models allow custom reduction operations to be defined. The verbosity of the exact mechanism varies, impacting productivity: OpenMP and SYCL are fairly concise as they apply specification of the operator with the addition of a single compiler directive or as part of the interface, whereas Kokkos requires implementing two classes with a number of methods. No model however defines the extent to which an implementation may assume or otherwise determine that the operation is associative and/or commutative. This may preclude some optimizations, or even introduce incorrect code, depending on the properties of the operator.

**V. PERFORMANCE, PORTABILITY AND PRODUCTIVITY RESULTS AND ANALYSIS**

We evaluate the performance of our reduction kernels on five different platforms, as detailed in Table I. Where possible, we have run our benchmark kernels in each of the languages and abstractions outlined in Section IV.

Despite the great efforts each of these languages and frameworks have gone to to provide a performance portable programming environment, they seem to inevitably shift much of that complexity onto their backends and tooling. Each distinct platform may require a wholly unique toolchain including front-end compilers, runtimes, drivers, system configurations, etc. Building and running even these benchmarks across this diverse platform set remains challenging and time consuming. Although packages such as Spack [24] may offer promising solutions to these issues, a lack of recipes for experimental and developing compilers and runtimes tempers its effectiveness.

Performance results are presented in Figure 1 in the form of cascade plots [25], one for each reduction kernel. For each family of solid/dotted lines (which represent languages/frameworks), at each value $N$ on the horizontal axis,
the solid lines represent the minimum efficiency value of the $N$ platforms with the highest efficiencies; the dotted lines show the $P$ scores [26]. Better performance portability is visualised as higher values remaining high as the graph reads from left to right; when a language/framework is unable to run the kernel on further platforms, the lines drop to zero.

The efficiency depicted in these plots is the achieved main-memory bandwidth normalized by the peak achievable for each platform (i.e. architectural efficiency). To obtain the raw throughput numbers, the arithmetic mean value of five independent runs was taken, with each run reporting the average sustained bandwidth of 100 iterations of the kernel.

Portability is a requirement for performance portability [25]. No language or framework was able to run all kernels successfully on all platforms. Kokkos came closest to full support, only failing on Complex minimum on Intel® Iris® Pro. OpenMP also has good portability, although we have plotted “native” OpenMP and OpenMP target as separate languages to reflect the differences in the implementations used for each to support different classes of platforms.

A. Missing results

hipSYCL does not yet support the SYCL 2020 reduction interface and so SYCL did not run on the AMD CPUs and GPUs, nor NVIDIA GPUs.

oneDPL results are only available on Intel platforms.

As mentioned previously, it is not possible to implement the Complex Min kernel in RAJA, and the Describe kernel in RAJA on GPUs.

While RAJA’s SYCL backend in theory could target the Intel® Iris® Pro, we found insufficient documentation to perform the required specialisation for this platform.

On the AMD MI100 GPU, the OpenMP Describe kernel compiled with AOMP produced an incorrect answer.

The NVHPC compiler does not support std::complex with OpenMP, and so we could not compile the complex sum and complex min kernels; the Describe kernel crashed at runtime.

B. Observations

The dot product kernel is shown in the top-left of Figure 1. Most of the models attain good performance across all platforms here. The SYCL line (purple) shows that performance with the DPC++ compiler on CPUs is less efficient than the GPU implementation. On two sockets in particular we observe a drop in efficiency compared to single socket, indicating poor NUMA behaviour—this is somewhat expected given that oneTBB [27], which is not NUMA-aware, is the backend SYCL uses for Intel CPUs.

The oneDPL library is built on top of SYCL and so it is particularly interesting to see that it outperforms native SYCL code. We believe this is to do with pure SYCL using a GPU-optimised work scheduler when used on the CPU, resulting in access to memory with a large stride causing undesirable cache behavior.

The complex summation is shown in the top-right of Figure 1. We show only the array-of-structures implementation. We observe the effects of serialised use of the std::complex constructor for RAJA here (recall Section IV-F1). Kokkos shows good performance portability for this, and required no user intervention to ensure good NUMA behaviour as it eschews pitfalls through the internals of its Kokkos::View class. Both SYCL and oneDPL again show reductions in efficiency on the dual-socket Cascade Lake. This kernel required providing a user-defined reduction in OpenMP, and we see that this reduces the performance on the GPU platforms more than on the CPU platforms.

The complex minimum kernel is shown in the centre-left of Figure 1. This requires a custom reduction in all the models. Note that support only extends to five of the six platforms, however Kokkos performance is similar to the other kernels. We see that this complex reduction reduces OpenMP performance: compared to the complex sum custom reduction operator for which OpenMP may deduce is associative and commutative, there is no way for this to be determined for our custom reduction operation and so the implementation may be making more conservative optimizations to ensure correctness. However, the gap is much smaller than reported by Gayatri et al. [7].

The Describe kernel is shown in the centre-right of Figure 1. The cascade plot makes it clear that the performance portability of this kernel is lower than the others for all models. This kernel requires the implementation to perform three types of reduction operation simultaneously (addition, minimum and maximum); note that the memory bandwidth model already accounts for reading the data twice. In comparison with the Field summary kernel in the lower-left of the figure, performing simultaneous reductions of the same operation is performant in most models.

Kokkos shows good levels of performance portability for all the benchmark kernels. The performance on the NVIDIA A100 was lower than expected, however this is a relatively new processor and so may need further backend optimization. The Field Summary and Describe kernel show low efficiency on all GPU platforms, indicating that the new combined reducers which bundle multiple reducer objects together is a relatively new addition to Kokkos, and so is likely to need optimization and evaluations in wider benchmarking such as this study.

For all these kernels, OpenMP also attains good performance portability, with efficiency values above 60% for all kernels on all CPUs. Performance on GPUs still needs improving on all platforms, most notably on the MI100 GPUs where we used the AOMP compiler. We are looking forward to all OpenMP implementations fully supporting the metadirective so that we can combine both implementations into a single code base.

RAJA performance is noticeably lower on the GPU platforms compared to the other models. Our chosen schedules followed those documented in the RAJA documentation.

If our paper is accepted, we will aim to build Intel’s DPCPP implementation for CUDA as an alternative.
Fig. 1. Reduction results, shown as architectural efficiency cascades based on sustained main memory bandwidth.
TABLE II
A BREAKDOWN OF COMMON SOURCE LINES OF CODE (SLOC) AS COMPUTED BY CODE BASE INVESTIGATOR [28]; SEE ALSO [29].

<table>
<thead>
<tr>
<th>Platform Set</th>
<th>Language/Framework</th>
<th>OpenMP</th>
<th>RAJA</th>
<th>SYCL</th>
<th>oneDPL</th>
<th>Kokkos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Xeon® Gold 6230</td>
<td></td>
<td>366</td>
<td>68</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>AMD EPYC 7742</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVIDIA A100</td>
<td></td>
<td>579</td>
<td>55</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AMD MI100</td>
<td></td>
<td></td>
<td></td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel® Iris® Pro Graphics 580</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td></td>
<td>579</td>
<td>915</td>
<td>1060</td>
<td>991</td>
<td>964</td>
</tr>
<tr>
<td>Total SLOC</td>
<td></td>
<td>1338</td>
<td>1108</td>
<td>1097</td>
<td>992</td>
<td>964</td>
</tr>
</tbody>
</table>

C. Productivity
Table II shows the source lines of code for the benchmark, as computed by Code Base Investigator, differentiated by platform specialization. The number of lines of code specific to a platform are shown in that cell, unless there is a span of multiple rows, in which case the code shown is common to all of those platforms. The “All” row shows the SLOC common to all supported platforms. Finally, the “Total SLOC” row shows the sum of SLOC used by all platforms, plus any unused code (e.g. lines that may have been guarded by inactive preprocessor directives).

We include these SLOC counts here to show that many of the models require no code changes across the platforms. OpenMP has larger divergence due to the use of target and non-target directives, however much code is shared. Note that RAJA’s requirement to specialize per backend requires code changes unique for each target platform.

VI. CONCLUSION
The programming models explored all provide a sophisticated, high-level mechanism with which users may express kernels with reductions. From a productivity standpoint this is key, as implementing performant reductions is a challenge made especially acute by the need for performance portability. In the main, they follow a similar design: provide a reduction object which is updated inside the kernel using the reduction operator, with the final result made available after finalization. They differ most significantly in whether that reduction variable is available for use in other operations inside the kernel: OpenMP and Kokkos allow this and do no check whether the reduction variable is modified, and SYCL, RAJA and oneDPL do not allow the reduction variable to take part in arbitrary operations.

We would like to encourage parallel programming models to define their reduction abstractions following the anatomy we outlined in Section III-A so as to create common semantics, in particular around initialization and finalization. This will help the productivity of writing custom reduction operations, and ensure that optimized implementations of them are possible. It is important that reduction customisation is included in frameworks so as to provide an alternative to application writers attempting their own implementations, impacting all aspects of performance, portability and productivity.

Our performance results presented in Section V highlight that there is still unexpected performance variation impacting the performance portability of these parallel programming frameworks. In particular, SYCL implementations should strive for conformance (implementing the latest reduction interface), and optimize on CPUs in particular.

VI. CONCLUSION
The broad interest in performance, portability, and productivity has not escaped the attention of language and tool architects; first-class reduction primitives are widely available in complex computing environments, and similarly high-level abstractions are sure to follow. It is our hope that application developers, language architects, and others can work together to learn from the mixed lessons of the past to design flexible, expressive, portable, performant interfaces.

Furthermore, we look to the application community to make earnest evaluations of the fruits of these efforts and to analyze and share insights in a rigorous, methodical fashion.

ACKNOWLEDGMENT
This work was in part funded by EPSRC through the Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems (ASiMoV) project, EP/S005072/1.

This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1).

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk).

This work used the DiRAC@Durham facility managed by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). The equipment was funded by BEIS capital funding via STFC capital grants ST/P002293/1, ST/R002371/1 and ST/S002502/1, Durham University and STFC operations grant ST/R000832/1. DiRAC is part of the National e-Infrastructure.

NOTICES & DISCLAIMERS
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. No product or component can be absolutely secure. Your costs and results may vary. Intel technologies may require enabled hardware, software or service activation.

The Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
REFERENCES


[12] *OpenMP Application Program Interface, Version 4.0."


[14] *OpenMP Application Program Interface, Version 5.0."


APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: ANALYZING THE CAPABILITIES OF REDUCTION ABSTRACTIONS

A. Abstract
This paper explores reduction abstractions in parallel programming models and frameworks. We developed the Everything’s Reduced suite of benchmark kernels, available as open-source software on GitHub with a permissive license. We included performance results on a number of processors hosted in various supercomputer systems. The repository also includes the scripts we used to run the results and produce the figures in this paper.

B. Description

1) Check-list (artifact meta information):

- **Program**: Everything’s Reduced
- **Compilation**: For our experiments, we produced and included `run_<arch>.sh` scripts to build the code on the platforms tested. They may be used as an example to configure on your systems.
- **Data set**: The benchmark auto-generates its own inputs based on a command line argument. We used a problem size of $2^{30}$ on all platforms, except the Intel® Iris® Pro where we used a problem size of $128 \times 2^{20}$. The Field Summary kernel has a fixed input size of $3840 \times 3840$.
- **Hardware**: The Intel® Xeon® CPU, AMD EPYC CPU were available in the GW4 Isambard system. The NVIDIA A100 GPU was available in the University of Cambridge’s CSD3 system. The AMD MI100 GPU was available in the University of Durham’s Cosma system. The Intel® Iris® Pro GPU was available in the University of Bristol’s HPC Zoo. Further details of the hardware was present in the main body of the paper in Table I.
- **Output**: All output from our runs is archived in the GitHub repository.
- **Publicly available?**: Yes, on GitHub.

2) How software can be obtained (if available): The Everything’s Reduced code is available on GitHub at https://github.com/UoB-HPC/everythingsreduced. All other compilers and software was either pre-installed, or obtained using the usual means.

3) Software dependencies: On the Intel® Xeon® Gold 6230 (“Cascade Lake”) CPU we used the Intel® icpc compiler version 19.1.3, oneAPI version 2021.3.0 and a nightly build of the DPCPP compiler from August 19, 2021.

The the AMD EPYC 7742 (“Rome”) CPU we used the Cray Compiler (CCE) version 11.0.4. The performance results were very similar with the AOCC compiler, version 2.3.

On the NVIDIA A100 GPU we used driver version 460.32.03 and CUDA Version 11.2. We used the NVHPC compiler version 21.7.

On the AMD MI100 GPU we used the HIP compiler version 4.2.21155. We used the AOMP compiler version 13.0-6 for OpenMP target.

On the Intel® Iris® Pro GPU we used the Intel NEO driver version 21.15.19533. We used some versions of oneAPI and DPCPP as for the Intel® Xeon® above.

We used Kokkos version 3.4.0 and RAJA version 0.14.0.

C. Installation

Everything’s Reduced using CMake to generate a suitable configuration for compiling for a particular combination of programming model and platform. The general procedure is as follows:

```cpp
    cmake -Bbuild -H. -DMODEL=<model>
    cmake --build build
```

Valid options for the `MODEL` parameter are: OpenMP, Kokkos, RAJA, OpenMP-target, SYCL and oneDPL.

Additional options are required based on the model and are detailed in the README.md file included in the repository.

D. Experiment workflow

The binary is named Reduced and appears in the build directory specified in the CMake step. It takes one or more arguments. The first argument is the name of the kernel to run. A list of these are is obtained by running the executable with no arguments.

All benchmarks (except Field Summary) take a second argument setting the problem size. The problem size is in array elements. Human-readable prefixes can be used to specify orders of magnitude in base 10 or base 2; e.g. `1gib` specifies a problem size of $2^{30}$ elements.

Relevant environment variables must be set depending on the model chosen. For example, with OpenMP on the Intel® Xeon® CPU the following:

```bash
    OMP_NUM_THREADS=40
    OMP_PLACES=cores
    OMP_PROC_BIND=true
```

E. Evaluation and expected result

Each benchmark self-validates with an expected input. If the results are not within a suitable tolerance, they are reported as incorrect.

The benchmark outputs the runtime and estimated sustained memory bandwidth of the application. In this study we presented application efficiency, calculated as the estimated sustained memory bandwidth as a fraction of the theoretical peak main memory bandwidth of the processor.