International Journal of Parallel Programming (2023) 51:209–230
https://doi.org/10.1007/s10766-022-00739-0

Portable C++ Code that can Look and Feel Like Fortran Code with Yet Another Kernel Launcher (YAKL)

Matthew Norman[1] · **Isaac Lyngaas**[1] · **Abhishek Bagusetty**[2] · **Mark Berrill**[1]

Received: 11 April 2022 / Accepted: 24 October 2022 / Published online: 8 December 2022

Abstract

This paper introduces the Yet Another Kernel Launcher (YAKL) C++ portability library, which strives to enable user-level code with the look and feel of Fortran code. The intended audience includes both C++ developers and Fortran developers unfamiliar with C++. The C++ portability approach is briefly explained, YAKL's main features are described, and code examples are given that demonstrate YAKL's usage. YAKL fills a niche capability important particularly to scientific applications seeking to port Fortran code quickly to a portable C++ library. YAKL places heavy emphasis on simplicity, readability, and productivity with performance mainly emphasizing Graphics Processing Units (GPUs). Central to YAKL's ability to allow Fortran-like user-level code are three features: (1) a multi-dimensional Array class that allows Fortran behavior; (2) a limited library of Fortran intrinsic functions; and (3) an efficient pool allocator that transparently enables cheap frequent allocations and deallocations of YAKL Arrays. While YAKL allows Fortran-style code, it also allows Arrays that exhibit C-like behavior as well, including row-major index ordering and lower bounds of "0". YAKL currently supports CPUs, CPU threading, and Nvidia, AMD, and Intel GPUs.

✉ Matthew Norman
normanmr@ornl.gov

Isaac Lyngaas
lyngaasir@ornl.gov

Abhishek Bagusetty
abagusetty@anl.gov

Mark Berrill
berrillma@ornl.gov

[1] National Center for Computational Sciences, Oak Ridge National Laboratory, 1 Bethel Valley Rd, Oak Ridge, TN 37830, USA

[2] Argonne Leadership Computing Facility, Argonne National Laboratory, 9700 South Cass Avenue, Building 240, Argonne, IL 60439, USA

Keywords C++ · Portability · GPU · HPC

1 Introduction

Code portability can be an overwhelming subject to consider because of how quickly configurations can tensor out to different coding languages / augmentations, compilers, programming interfaces, hardware targets, and system capabilities. This paper focuses on lower-level languages commonly used in scientific software programming such as Fortran, C, C++, CUDA, HIP, and SYCL – specifically Fortran and C++. This is not to say that other languages are not important or common in scientific software but rather to set the scope of what is considered here. Further, the primary focus of this paper is on accelerated Graphics Processing Unit (GPU) hardware targets, though attention is paid to Central Processing Unit (CPU) considerations and CPU-level threading that typically maps to POSIX (Portable Operating System Interface) threads, or "pthreads".

Already, five to six languages have been mentioned, not to mention that C++ itself has many expressions in terms of which features are used and whether modularity is achieved largely through inheritance or template expressions. There are quite a few compilers to consider, each capable of its own set of languages, APIs, and hardware targets, including but certainly not limited to: GNU,[1] Clang,[2] IBM,[3] Intel,[4] Cray / HPE,[5] and Nvidia (including what was formerly PGI).[6] There are many parallel programming interfaces / specifications to consider as well, including the Message Passing Interface (MPI)[7] [1], OpenACC,[8] [2] and OpenMP[9] [3] in its two somewhat distinct flavors of OpenMP $<= 4.0$ for CPU-level threading and OpenMP target offload for accelerators with disparate memory spaces. Hardware targets include but are not limited to CPUs, SIMD units, pthreads, and Nvidia, AMD, and Intel GPUs. There are systems that allow data allocated with `malloc` to be paged automatically to disparate accelerator memory spaces and device allocated memory to be paged automatically to host memory. There are systems that allow MPI to use pointers to data in separate device memory spaces. If one relies on these system features, then the code is likely no longer portable to machines that do not have these features.

[1] https://gcc.gnu.org/.

[2] https://clang.llvm.org/.

[3] https://www.ibm.com/products/xl-cpp-linux-compiler-power.

[4] https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html.

[5] https://www.hpe.com/us/en/compute/hpc/cray.html.

[6] https://developer.nvidia.com/hpc-sdk.

[7] https://www.mcs.anl.gov/research/projects/mpi/.

[8] https://www.openacc.org/.

[9] https://www.openmp.org/.

An approach to portability that is growing in popularity over time is the use of portable C++ libraries such as kokkos[10] [4], RAJA[11] [5], and SYCL / OneAPI / Data Parallel C++[12] [6]. These libraries rely on the ability of the C++ language to encapsulate code as an object, most often as a "functor" (a class that overloads the `operator()` operator). Often, this is achieved using a C++ "lambda" expression,[13] which conveniently creates an anonymous functor in-place for the programmer, automatically capturing the required data. Over the last three years, a C++ portability library called Yet Another Kernel Launcher (YAKL)[14] has been developed (and is still actively developed) largely to support the Energy Exascale Earth System Model's (E3SM's) efforts under the U.S. Department of Energy Exascale Computing Project. YAKL is a relatively simple and small C++ library for portability to CPUs, Nvidia GPUs, AMD GPUs, and Intel GPUs that specializes in allowing user-level code that maintains much of the look and feel of the dominant style of Fortran code currently used by the E3SM.

YAKL was first motivated by the absence of other C++ portability frameworks being able to support AMD GPUs, which was eventually remedied in the other frameworks. Its current motivation is the ability to quickly port Fortran code to a portable C++ library and to support development practices that are familiar to Fortran domain science developers such as multi-dimensional arrays, frequent allocation and deallocation of arrays, and use of simple reduction intrinsics such as `sum` and `minval`. YAKL currently supports CPU targets, CPU threading (through the OpenMP specification), Nvidia GPUs (through the CUDA language), AMD GPUs (through the HIP language), and Intel GPUs (through the SYCL specification). Importantly, SYCL can also produce code for Nvidia and AMD GPUs, and HIP can also produce code for Nvidia GPUs. YAKL has already been used by a number of published studies [7–10]. Particularly with a kernel whose algorithm is well-suited to floating point operation ("flop") capable hardware, YAKL was shown to be capable of achieving up to 80% of peak single precision flops per second (flop/s) on an Nvidia Tesla V100 GPU [7].

Fortran developers less familiar with C++ might find it useful to first read Sect. 3.1. A brief overview of C++ portability is given in Sect. 2. Then an overview of YAKL's main features is given in Sect. 3. The YAKL hardware targets are described in Sect. 4. Finally, concluding remarks and future work are detailed in Sect. 5.

2 A Brief Description of the C++ Portability Approach

It is important to note that C++ portability is not a language extension but rather just a library fully specified within the C++ language that uses additional hardware-specific languages such as CUDA, HIP, SYCL, and OpenMP to target specific hardware under

[10] https://github.com/kokkos/kokkos/.

[11] https://github.com/LLNL/RAJA.

[12] https://www.khronos.org/sycl/.

[13] https://en.cppreference.com/w/cpp/language/lambda.

[14] https://github.com/mrnorman/YAKL.

```
// All this template expression means is that any class object can be passed in
template <class F>
void launch_on_cpu_with_threads( int loop_size , F const &f ) {
  #pragma omp parallel for
  for (int i=0; i < loop_size; i++) {
    f(i);
  }
}
int constexpr n = 128;
float a[n], b[n], c[n];
// initialize a and b
auto code_as_class_object = [] (int i) { c[i] = a[i] + b[i]; };
launch_on_cpu_with_threads( n , code_as_class_object );
// The launch_on_cpu_with_threads function gives the same result as the following loop
for (int i=0; i < n; i++) {
  c[i] = a[i] + b[i];
}
```

Fig. 1 An example of encapsulating code as an object using a lambda expression and then passing it to a function to launch it with CPU threading using the OpenMP specification

the hood. Therefore, portable C++ libraries benefit fully from the many available mature C++ compilers.

2.1 Code as an Object

The core of C++ portability is the ability to express code as an object that can be passed to functions as parameters. This is most conveniently performed using a lambda expression, which creates an anonymous class object (meaning it has no class name), either copies by value whatever variables are needed or references them, and then wraps a section of code inside an overloaded `operator()` operator. The class object created by the lambda expression can then be passed as a parameter to a function that can then execute the code inside that class object on any desired hardware target. See Fig. 1 as an example. Instead of running the lambda inside a `for` loop with OpenMP threading, one could call it inside a CUDA kernel or a SYCL parallel dispatch to run on Nvidia GPUs and Intel GPUs, respectively.

2.2 Shallow-Copy Data Structures and Copy-by-Value Lambda Expressions

Another common thread among C++ portability libraries is that they seek to allow developers to run on accelerated hardware that has its own distinct memory space. The complication of this is that for machines that do not page data automatically between host and device memory spaces, pointers and references to host data are invalid in device memory (and vice versa). Even when machines do page data, it is typically more efficient to move and manage the data manually.

By default, lambda expressions will capture data by reference. However, since host references are invalid in device memory, this is no longer acceptable. Therefore, in most C++ portability libraries, lambdas must capture data by value, meaning the lambda syntax is typically: `[=] (parameters...) {...}`. The exception to this is with SYCL's use of `Buffer` objects because, in SYCL, one obtains an appropriate accessor

that is already valid in the desired memory space. Therefore, lambdas that use these accessors may pass them by reference.

When copying data structures by value into a lambda expression, it is common for those data structures to use what are called shallow-copy semantics, which means that the actual underlying data pointer is valid already on the device it is being launched on, and only a small amount of metadata of the data structure is copied explicitly, while the pointer is used as is. This way, large amounts of data are not copied every time a kernel is launched. This can, however, lead to confusion for developers used to Fortran assignment semantics, which are deep-copy semantics. In most C++ portability frameworks, to copy the underlying data, one most use an explicit "deep copy" routine rather than assigning to an object directly. This will be explained in further detail later. Therefore, keep in mind that in C++ portability libraries, assigning one array object to another is more akin to pointer assignment – it does not copy or duplicate the underlying data.

2.3 Some Subtleties of C++ Portability Libraries

A complication that is more subtle is that C++ lambda expressions only capture by value variables in *local* scope. Therefore, variables at the class scope, global scope, and namespace scope are still captured by reference inside the resulting class object. To avoid this behavior, one must pull all variables used inside the lambda expression's code that live in class, global, or namespace scope into local scope before creating the lambda expression. This can be conveniently done with `auto &varname = ::varname` or `auto &varname = this->varname`, though YAKL has its own syntax for this called `YAKL_SCOPE`, which is covered later.

Another subtlety in C++ portability libraries is calling member functions from your own class from `parallel_for` kernels. As a general rule, it is best practice to make all class methods that are called from device kernels `static`, meaning they belong to the class itself, not any particular instance of that class. The reason is that `static` methods have no `this->` reference (which is generally only valid in host memory). One *can* capture the class by value in lambda expressions, making the `this->` reference valid inside `parallel_for` kernels (meaning you can you class methods and data inside that `parallel_for` call). However, member functions called by `parallel_for` kernels still cannot use class member data or functions because those will still invoke `this->` references that are only valid in host memory.

In short: (1) it is a best practice to make all class member functions that are called from device kernels `static` and to pass any required class data by parameter; and (2) it is a best practice to pull all class, global, and namespace scoped data into local scope before creating the lambda expression.

3 An Overview of YAKL's Main Features

The YAKL C++ 17 library, with currently around 11K lines of core code (not counting unit test code), is geared toward simplicity, clarity, and readability with a particu-

lar niche application of allowing Fortran-like behavior in the user-level code. Much, though not all, of YAKL's API is patterned after the kokkos API. Notable simplifications in YAKL compared to other C++ portability libraries include allowing only allowing two memory spaces (host and device), only allowing basic `Array` slicing of contiguous chunks and whole dimensions, and only supporting basic scalar reductions: `minval`, `maxval`, and `sum`, though there are other simplifications as well.

While the example below highlights YAKL's ability to enable user-level code that looks like Fortran, please note that YAKL also allows C-style behavior as well.

3.1 An Example of Fortran Code Compared to Fortran-Style YAKL Code

It is helpful to start with an example of what a given snippet of Fortran code would look like once converted to YAKL. The examples in Figs. 2 and 3 are computing the maximum stable time step over a Cartesian grid Shallow-Water model. Figure 2 shows the Fortran code example using OpenACC directives as an example of parallelization on GPUs. Figure 3 shows the corresponding Fortran-style YAKL code that does the same calculation. The **`using`** and **`typedef`** statements at the top of Fig. 3 would typically be placed in a header file somewhere and reused by all of the YAKL code so that the main code is more readable.

The features that allow a developer used to coding in Fortran to feel more comfortable in C++ are: (1) avoiding the need to reverse the indexing order, (2) avoiding the need to change to a zero-based indexing strategy, and (3) being able to allocate and deallocate arrays at any place in the program without worrying about performance penalties (enabled by the YAKL pool allocator). While this small example would not be too arduous to move to a C-style indexing strategy, managing many thousands of lines of code with arbitrary lower indexing bounds in the arrays and potentially complex integer arithmetic based on the chosen indexing strategy becomes a difficult task to manage.

As evident from this example, most of the code looks the same, particularly in the calculations themselves. There are some changes when moving to YAKL, however. First, the developer will need to become familiar with a C++ function syntax, which is unavoidable when moving from Fortran to C++. Here, the **`intent`**`(in)` arrays are declared with the type defined `realConst2d` type. The `size` and `epsilon` Fortran intrinsic routines remain identical in syntax when using YAKL. Those can be used inside kernels as well. Since YAKL does not support in-kernel reductions, the intermediate values must be stored in a new array `dt2d` first and then reduced with a call to `minval`.

The starkest change between Fortran and Fortran-style YAKL, however, is how the looping is expressed. The syntax was made as minimal as possible when designing YAKL, but some level of change of this nature is unavoidable. By commenting out the corresponding **`do`** loops, it can be seen what looping is implied by the `parallel_for` call. More information on the behavior of `parallel_for` is given in Sect. 3.2.1.

In brief, though, the syntax is as follows:

```
function max_stable_dt(height, u, v, cfl, grav, dx, dy)  result(dt)
  real(8), dimension(:,:), intent(in) :: height, u, v
  real(8)                , intent(in) :: cfl, grav, dx, dy
  real(8)                             :: dt
  integer                             :: i, j, nx, ny
  real(8)                             :: gw, dtloc, eps
  nx = size(height,1)
  ny = size(height,2)
  dt = huge(height)       ! Initialize to a large value
  eps = epsilon(height)  ! To avoid division by zero
  !$acc parallel loop collapse(2) present(height,u,v) reduction(min:dt)
  do j = 1 , ny
    do i = 1 , nx
      gw = sqrt(grav * height(i,j))  ! Speed of gravity waves
      ! Compute local maximum stable time step
      dtloc = min( cfl * dx / ( abs(u(i,j)) + gw + eps ) , &
                   cfl * dy / ( abs(v(i,j)) + gw + eps ) )
      ! Compute global minimum of the local maximum stable time steps
      dt = min( dt , dtloc )
    enddo
  enddo
endfunction max_stable_dt
```

Fig. 2 Example Fortran code with OpenACC directives to compute the maximum stable time step of a 2-D Shallow-Water Model

```
parallel_for( LABEL , Bounds<N>(D1,D2,...) , YAKL_LAMBDA (int i1, int i2,
...) { CODE });,
```

where `LABEL` is a string label for the kernel; `N` is the number of loops; `D1`, `D2` and so on are the loop bounds ($\{1, \ldots, D1\}$, $\{1, \ldots, D2\}$, etc. for Fortran-style loop bounds) where the left-most bound is for the outermost loop and the right-most bound is for the innermost loop; and `i1`, `i2`, and so on are the loop index variables with the left-most variable being for the outermost loop and the right-most variable being for the innermost loop.

The allocation in line 16 of Fig. 3 should be thought of as the equivalent of the Fortran code: **`real, allocatable ::`** `dt2d(:,:)` followed by **`allocate`**`(dt2d(nx,ny))`. Also note that allocations like this in YAKL are very cheap because they are done using a pool allocator (see Sect. 3.9).

Finally, note in line 20 of Fig. 3 that the `parallel_for` call includes a string label for the kernel launch. While a label is not required, it is helpful when it comes to automatic timing of all of the kernels in the code as well as labeling kernel launches clearly in GPU profiling tools such as Nvidia's nvprof and nsight tools. It also enables automated "printf" debugging of the code to dump to one file per process every action that occurs in YAKL, including labeled calls to `parallel_for`.

3.2 `parallel_for`, `Bounds`, `YAKL_LAMBDA`, and `fence`

YAKL achieves parallel dispatch using the following possible function definitions:

```
template <class F, int N> void parallel_for( char const *label , Bounds
<N> bounds ,
                                             F const &code );
template <class F, int N> void parallel_for( Bounds<N> bounds ,
F const &code );
```

```
// The lines before the function would be placed in a header file somewhere else
// using and typedef make code more readable, hiding template expressions and namespaces
using yakl::fortran::Bounds;
using yakl::fortran::parallel_for;
using yakl::intrinsics::minval;
using yakl::Array;
using yakl::memDevice;
using yakl::styleFortran;
typedef double real;
typedef Array<real const,2,memDevice,sytleFortran> realConst2d; // intent(in)
typedef Array<real      ,2,memDevice,sytleFortran> real2d;      // intent(inout)

// Here begins the main user-level YAKL code:
real max_stable_dt(realConst2d height, realConst2d u, realConst2d v,
                   real cfl, real grav, real dx, real dy) {
  int nx = size(height,1);
  int ny = size(height,2);
  real eps = epsilon(height); // To avoid division by zero
  real2d dt2d("dt2d",nx,ny);  // Allocate array to store the local max stable time steps
                              // This is similar to a Fortran automatic array
  // do j = 1 , ny
    //   do i = 1 , nx
  parallel_for( "Max stable timestep" , Bounds<2>(ny,nx) , YAKL_LAMBDA (int j, int i) {
    real gw = sqrt(grav * height(i,j));  // Speed of gravity waves
    // Compute local maximum stable time step
    dt2d(i,j) = min( cfl * dx / ( abs(u(i,j)) + gw + eps ) ,
                     cfl * dy / ( abs(v(i,j)) + gw + eps ) );
  });
  // With the local max stable time steps stored, compute the minimum among all of them
  return minval( dt2d );
}
```

Fig. 3 Example C++ portable code with Fortran-style YAKL to compute the maximum stable time step of a 2-D Shallow-Water Model. This corresponds to converting Fig. 2 to Fortran-style YAKL

The string label is optional, though very highly recommended. When there are a lot of `parallel_for` calls, and it is hard to come up with meaningful names, the `YAKL_AUTO_LABEL()` macro function is available, which simply inserts the filename (with the path removed) augmented with the line number. This way, it is readily known where the call lives in the code.

The `parallel_for` functions are defined in a `yakl::c::` namespace and a `yakl::fortran::` namespace, where each namespace has that language's behavior. In the `c` namespace, if you pass a scalar, `N` as the loop bounds, it is assumed to iterate over `{0,1,...,N-1}`; whereas in the `fortran` namespace, it is assumed to iterate over `{1,2,...,N}`

The Bounds class comes in a `yakl::c::` namespace and a `yakl::fortran::` namespace, and it describes the looping implied in the parallel kernel launch. The Bounds class accepts an integer template parameter that determines how many tightly nested loops are being dispatched. Bounds in the `c` namespace default to a lower bound of zero, and bounds in the `fortran` namespace default to a lower bound of one. Each loop in the `Bounds` class constructor's parameters is described either by an integer, an initializer list of two parameters that describes the inclusive lower and upper bounds, or an initializer list of three values that describes the inclusive lower bound, the inclusive upper bound, and the stride. Negative strides are not currently supported, and this is partially to protect the user from attempting to use `parallel_for` for work that

depends on the order in which loop indices are processed (e.g. prefix sums and other general loop-carried dependencies). If a loop cannot be cast with a positive stride, then it contains a loop-carried dependency. Examples of specifying loop bounds are as follows:

```
yakl::c::Bounds<2>(ny,nx); // Outer loop
 from 0 to ny-1, inner loop from 0 to nx-1
yakl::fortran::Bounds<1>(nx); // Loop from 1 to nx
yakl::c::Bounds<1>({-1,nx+1,2});  // for (int i=-1; i <= nx+1; i+=2)
yakl::fortran::Bounds<1>({1-hs,nx+hs});  // do i = 1-hs , nx+hs
```

Finally, the code is recommended to be passed to the `parallel_for` function by using a C++ lambda expression via the `YAKL_LAMBDA` macro. On the CPU, this maps simply to `[=]`, meaning variables used are passed by value. While the CPU could technically pass data by reference, since it isn't possible to pass data by reference on GPUs (as CPU references are not valid in GPU memory in general), it was deemed wise to use copy-by-value behavior even on the CPU. In the CUDA and HIP backends, it maps to `[=] __host__ __device__`.

When creating the lambda expression, the parameters passed to the lambda expression are the looping indices assigned by the `parallel_for` call in the hardware backend. For instance, if `Bounds<3>(nz,ny,nx)` is passed to the `parallel_for` function, then the lambda expression must accept exactly three parameters to accept indices for each of these loop:

```
// do k = 1 , nz
//   do j = 1 , ny
//     do i = 1 , nx
parallel_for( yakl::fortran::Bounds<3>(nz,ny,nx) , YAKL_LAMBDA
 (int k, int j, int i) { ... }
```

All calls to `parallel_for` are asynchronous with respect to host code. However, the order of `parallel_for` calls on the device is respected, meaning that a subsequent call to `parallel_for` call will not start until all previous device work has been completed. To synchronize after any asynchronous call in YAKL, use the `yakl::fence()` routine, which synchronizes the host code with respect to all existing asynchronous work on the device. All calls to `parallel_for` are assumed to be run on the device for which the YAKL code is targeted. YAKL can only target one device at a time.

There is an optional `LaunchConfig` parameter to the `parallel_for` function. This object contains two template parameters: an integer vector length that determines the number of threads in a "block" in CUDA and HIP, and a boolean bit-for-bit flag that defaults to false, which determines whether the launched kernel should be run in serial on the CPU instead of on the GPU whenever the C Pre-Processor (CPP) macro `YAKL_B4B` is defined (there is more on this behavior in Sect. 3.11).

3.2.1 Handling Loops that are not Tightly Nested

"Tightly nested" loops as used here means: (1) all loops appear consecutively with no work in between and (2) inner loop bounds do not depend on outer loop indices. There are some common approaches to handling situations where these are not both true.

If there is work in between loop bounds, the two common approaches are:

1. Push that work down into inner loops and simply duplicate the processing of that line of code.
2. Perform that work before entering the tightly nested loops and store to a temporary array that is used in the tightly nested loops.

There are also many cases where one loop's bounds will depend upon another. For instance, in many ocean models, not all vertical levels are active for every horizontal grid point. In these cases, the number of vertical levels depends upon the element index. To alleviate the situation, the typical practice is to have the inner loop over vertical levels iterate to the maximum number of vertical levels and place an if-statement inside the innermost loop.

3.3 Multi-dimensional Dynamically Allocated `Array` Classes

The next most important feature of the YAKL library is the dynamically allocated multi-dimensional `Array` class, which takes four template parameters: (1) data type, (2) number of dimensions, (3) memory space, and (4) style. The data type is templated and can be any type (e.g. **`float`**, **`int const`**, or **`bool`**). The memory space can be either `yakl::memHost` or `yakl::memDevice`. For all hardware targets with separate device memory spaces (i.e., most if not all GPU targets), host-space `Array` objects cannot be (portably) used on the device, and vice versa. The exception is when the CPP macro `YAKL_MANAGED_MEMORY` is specified, in which case `yakl::memDevice Array` objects can be accessed on the host. Finally, the style parameter can either be `yakl::styleC` or `yakl::styleFortran`. C-style `Array` objects have row-major index ordering (meaning the right-most index varies the fastest) and zero-based indexing. Fortran-style `Array` objects have column-major index ordering (meaning the left-most index varies the fastest) and by default one-based indexing, though the lowest index of a given dimension can be any integer.

All YAKL `Array` objects have contiguous indexing. When creating an `Array` object in device memory, it is not recommended to use a data type that has a constructor because device memory is nearly always allocated with the hardware backend's version of `malloc`, which does not call the constructor. However, when creating an `Array` object in host memory, the C++ **`new`** operator is used, meaning any data type should be suitable.

All `Array` objects have debugging capabilities (when turned on) to detect things like out of bounds indexing, indexing with the wrong number of dimensions, and using data in the wrong memory space (device data on the host or host data on the device). There are also YAKL flags that cause all allocations to be performed with Managed or Shared memory to allow device data to be used on the host.

3.3.1 Shallow and Deep Copy

As mentioned earlier, YAKL `Array` objects use shallow copy semantics for assignments, meaning if `a` and `b` are `Array` objects, then `a = b` will copy the metadata from `b` to `a`, but then they will each share the same data pointer. Thus, changes to one

will affect the other, similar to pointer assignment or the **equivalence** statement in Fortran. In order to maintain separate data pointers and copy the data itself between them, a "deep copy" is needed, which is achieved via the `deep_copy_to()` member function: e.g., `b.deep_copy_to(a)`.

One can only deep copy between `Array` objects of the same data type and total number of elements. The developer assumes responsibility for maintaining proper indexing when performing a deep copy between C-style and Fortran-style `Array` objects or between `Array` objects with differing numbers of dimensions.

3.3.2 Memory Space Management

YAKL's `Array` objects have convenient functions to transfer the data between memory spaces. The `createHostCopy()` and `createDeviceCopy()` member functions will create a separate copy of the `Array` object in host and device memory spaces, respectively, and deep copy the data. Even if the object is already in that memory space, a separate object with a separate data pointer is created with a full deep copy to avoid any semantics confusion when using the routine. If the user expect an object with a separate data pointer and the data pointer is, instead, shared, this would lead to code bugs that might be difficult to track down. If the user wishes to simply create a similar object in host or device memory without copying the underlying data, the `createHostCopy()` and `createDeviceCopy()` member functions are also available, respectively.

If the array objects already exist, then one can deep copy the data between different memory spaces with the `deep_copy_to` member function described in the previous section.

3.3.3 Automatic and Manual `Array` Deallocation

YAKL `Array` deallocation works similarly to Fortran's automatic deallocation semantics. Whenever an `Array` object falls out of scope, it is automatically deallocated. YAKL `Array` objects count the number of references to the same data pointer. As soon as the number of references drops to zero, the data is deallocated. Therefore, if an `Array` object is created and allocated inside function, as long it is not shallow copied to a global object or returned from the function, it will be deallocated as soon as the function ends.

The highest likelihood for memory leaks (in all contexts) is with statically scoped `Array` objects that technically do not fall out of scope until the program ends. While this should never lead to memory usage that grows unbounded over the executable's runtime, it can nonetheless lead to errors and bad behavior. In these cases, one can deallocate the `Array` object manually in one of two ways: (1) explicitly call the `deallocate()` member function; or (2) replace the `Array` object with an empty `Array` object via shallow copy assignment, e.g., `arr = real2d();` using the **typedef** from Fig. 3.

3.4 Multi-dimensional Statically-Sized `SArray` Classes

YAKL also has statically sized multi-dimensional array objects via the `SArray` (for C-style indexing) and `FSArray` (for Fortran-style indexing) classes. These are the multi-dimensional equivalent of simple arrays in C with the dimension size known at compile time, e.g., `float data[128]`. These are placed in the stack of whatever context they are defined. They can be defined inside `parallel_for` kernels as small thread-private arrays. With YAKL's debugging turned on, the indices are checked during runtime. While some host architectures allow runtime-sized stack arrays, this is *not* allowed in YAKL because most accelerator devices do not allow this behavior. The dimension sizes must be known at compile time.

3.5 Handling Parallel Data Races

Reductions: YAKL allows handling kernel-wide reduction operations via intrinsic functions, which mimic Fortran intrinsic syntax: `sum`, `minval`, and `maxval`, `minloc`, and `maxloc`. These are based on vendor provided libraries for optimal performance.

Atomic Instructions: Whenever multiple parallel threads might read/write to the same data location, one needs to use atomic instructions. YAKL supports these at a low level with the following three functions: `atomicAdd(a,b)`, `atomicMin(a,b)`, and `atomicMax(a,b)`. These correspond to serial equivalents of `a=a+b`, `a=min(a,b)`, and `a=max(a,b)`, respectively.

3.6 "Scalar Live-Out"

There are cases where a scalar value is written to inside a device `parallel_for` kernel and it must be read on the host after the kernel has finished. Since all variables inside a `parallel_for` call are copied by value, this means the scalar data must actually be allocated on the device beforehand if it is to be accessed afterward. To make this process easier, YAKL has a `ScalarLiveOut` class where room for a single scalar value is allocated on the device in the constructor, the initial value can be assigned in the constructor, the variable can be written to with a simple `operator=` assignment, and it can be read subsequently on the host with the `hostRead()` member function. In the rare case that it is necessary, one can get the device reference for modification with the `operator()` overload.

The most common need for a scalar live-out situation is in device-resident error checking routines where a boolean value is used to determine if, for instance, the data is within physical bounds or not.

3.7 Limited Fortran Intrinsics Library

There is a limited (but growing) library of Fortran intrinsic routines in YAKL. These currently include size, shape, lbound, ubound, allocated, associated, epsilon, tiny, huge,

sign, mod, merge, minval, minloc*, maxval, maxloc*, sum, product*, any, matmul*, transpose*, count, and pack*. Routines with a superscript asterisk are only available for `SArray` and `FSArray` objects and do not invoke parallel kernels. The routines minval, maxval, sum, any, and count will invoke `parallel_for` device kernels whenever operating on dynamically allocate `Array` objects, but they use simple inline looping for `SArray` and `FSArray` objects. The reason for this is that statically sized arrays are intended to be relatively small and dynamically sized arrays are intended to be larger. This removes ambiguity in terms of what behavior to expect when calling one of these intrinsics.

Therefore, any of these routines may be called on an `SArray` or `FSArray` object anywhere in the code, but routines that invoke a `parallel_for` kernel should not be called inside another `parallel_for` call.

3.7.1 Componentwise Operator Library

YAKL also has a library of componentwise operators that can be performed on YAKL `Array` objects in the `yakl::componentwise` namespace. These include unary operators and binary operators between two arrays or between an array and a scalar. Each of these operators launches a `parallel_for` in the default stream. These are largely to make error checking code more convenient to write (e.g., `if (any(arr < 0)) {...}`).

3.8 `YAKL_INLINE` and Calling Functions from `parallel_for` Kernels

When calling a function from a `parallel_for` kernel, it is recommended to use the `YAKL_INLINE` macro, which gives it modifiers for the appropriate hardware backend to run on the device. For instance, in the CUDA and HIP backends, `YAKL_INLINE` maps to `__forceinline__ __host__ __device__`. It is *very highly recommended* that any class member function decorated with `YAKL_INLINE` *also* be decorated with `static` to avoid any potential use of the class's `this->` pointer. While YAKL does have a `YAKL_CLASS_LAMBDA` macro that captures `*this`, the user can still run into erroneous situations when trying to use a class's own on the GPU from inside `YAKL_INLINE` functions.

3.9 A Transparent and Efficient Pool Allocator

As mentioned earlier, it is a common practice in many Fortran codes to use automatic arrays in functions and subroutines. To enable efficient frequent allocations and deallocations on the device in C++, YAKL enables a transparent pool allocator under the hood for all YAKL allocations, including `Array` objects. Allocations on the host are typically not that expensive, but on GPU devices, they can be prohibitively expensive. A pool allocator allocates an initial large block of memory and then hands out chunks quickly upon request. Even for host allocations, YAKL has proven to often be faster than system `malloc` calls – likely due to how closely packed arrays are in memory

when using the pool compared to using `malloc`. The pool allocator is intended for use outside `parallel_for` calls, not inside them.

YAKL's pool allocator, "Gator", is based on a simple linear allocation mechanism, which traverses linearly through existing allocations to find space for a requested allocation. This is certainly not the fastest allocation method, but it is beneficial in terms of reducing segmentation and improving memory locality. Further, since pool allocations typically overlap with device kernel executions, the relatively small increase in searching time is typically unnoticeable compared to faster allocation mechanisms that use the available space less efficiently. Particularly in the presence of allocations from multiple parallel CPU threads in indeterminant order, the projects that use YAKL found it important to use the available pool space as efficiently as possible.

Whenever a pool runs out of space, an additional pool is allocated and remains in place until YAKL's `finalize()` routine is called. The user can control the initial size of the pool (in MB) with the `GATOR_INITIAL_MB` environment variable, can control the size of additional pools with the `GATOR_GROW_MB` environment variable, and can disable the pool with the `GATOR_DISABLE` environment variable. YAKL also has an `InitConfig` class to control this with runtime information as well.

YAKL exposes Fortran hooks to the pool allocator through a Fortran module `gator_mod`. Fortran codes can pass contiguous pointers to these routines to allocate data from the Fortran side. This is advantageous for porting Fortran codes because one can allocate efficiently through the pool allocator using "Managed" or "Shared" memory, and those arrays are accessible on the host and device in both C++ and Fortran.

The "Gator" class is also available to the user to use on their own for pools the may need to manage for other purposes. Note that YAKL's pool allocator is not intended to quickly manage things like CUDA "Shared Memory" within kernels. It is only meant to manage device resident allocations from the host.

3.10 `YAKL_SCOPE` and Using Non-Local Data Inside `parallel_for` Kernels

There are case when the code launched by a `parallel_for` call uses data that isn't immediately in local scope but is rather in global or namespace scope or in the class scope. In these cases, as mentioned earlier, C++ lambda expressions will not capture out-of-local-scope data by value. Therefore references to these data are invalid on the device and will lead to invalid memory address errors or segmentation faults in the best case scenarios.

The `YAKL_SCOPE` macro function is intended to help in this case to bring that data into local scope *before* creating the lambda. For instance, for global scope and class scope data, the call would be `YAKL_SCOPE(data,::data);` and `YAKL_SCOPE(data,this->data)`, respectively. After these calls, the variable `data` can safely be used in the code wrapped in a `YAKL_LAMBDA` expression and launched by a `parallel_for` call.

3.11 Bitwise Floating Point Reproducibility

There are many projects for which bitwise floating point reproducibility is important. For instance, in climate, initially small (even machine precision) differences in floating point values will diverge rapidly into distinct weather states in a matter of only a week or two of simulation time. This chaotic behavior makes determining acceptable and unacceptable answer changes difficult. Therefore, having a bitwise reproducible answer (even if only during testing) is important if only to understand when the answer *could have* changed.

The issue largely comes in regarding floating point mathematical operations that can occur in a non-deterministic order, since floating point arithmetic is not generally commutative. The reduction libraries used by YAKL are deterministic. The `atomicAdd` instructions, however, are not. Therefore, for any kernel that contains an `atomicAdd` instruction, the user can place an optional parameter, `yakl::DefaultLaunchConfigB4b` at the end of the `parallel_for` call to ensure that whenever the CPP macro `YAKL_B4B` is defined during compilation, that kernel is run in serial on the CPU.

When the user defines `YAKL_B4B`, YAKL automatically turns on "Managed" memory for CUDA and HIP and "Shared" memory for SYCL. This allows the device data in the kernel to be accessible on the host so that kernels with the `yakl::DefaultLaunchConfigB4b` parameter at the end can be run successfully in serial on the CPU. Since these are run serially when `YAKL_B4B` is defined, floating point determinism is maintained. Whenever `YAKL_B4B` is not defined, those kernels still run in parallel efficiently on the GPU with non-deterministic results.

3.12 Hierarchical Parallelism

YAKL supports two levels of parallelism on the GPU: one intended for threading inside a vector unit (e.g., "Streaming Multiprocessor" for Nvidia GPUs or "Compute Unit" for AMD GPUs); and one for threading across multiple vector units. The functions to launch on these are called `parallel_inner` and `parallel_outer`, respectively. When `parallel_outer` is called, it creates an object called an `InnerHandler`, which must be accepted after the loop indices in the lambda function passed to it. The `parallel_inner` routine then accepts this `InnerHandler object` as a parameter. This data structure holds internal YAKL data to manage the two-level parallelism. Technically, this is only required because of SYCL, which requires this kind of behavior. CUDA and HIP would not require this object. Since the goal is single source portability, though, it is required for all contexts.

There are two other functions also defined that will be commonly used in this context. A `single_inner` function ensures only one inner thread performs the work inside, and a `fence_inner` function synchronizes kernel work within an inner loop until all previous inner loop threads have completed. For example, this maps to `__syncthreads()` in CUDA and HIP. Both `single_inner` and `fence_inner` both require the `InnerHandler` object to satisfy SYCL requirements. Further, all lambda functions passed to `parallel_inner` and

`single_inner` should be standard C++ pass-by-reference lambdas rather than using `YAKL_LAMBDA`: i.e.,
`[&] ( ... ) { ... }`.

3.13 "Streams"

YAKL supports multiple parallel "streams" (using CUDA terminology). In SYCL, these are called "queues." A stream / queue should be thought of as a first-in, first-out queue into which all device operations are enqueued and completed in the order they were enqueued. By default, YAKL uses a single default stream / queue for all operations. The user can, however, use multiple queues by defining the CPP macro `YAKL_ENABLE_STREAMS` at compile time. YAKL's `create_stream()` routine returns a YAKL `yakl::Stream` object. Streams can be used to record `yakl::Event` objects, streams can wait on event objects, and the host can wait on event or stream objects to be completed. `parallel_for`, `parallel_outer`, and intrinsics that launch kernels all take optional stream arguments that default to the default stream.

One thing to be aware of, however, is that YAKL's default use of a non-blocking pool allocator for all device allocations creates a potential aliasing problem when using multiple streams simultaneously. If the user deallocates and allocates during runtime, then multiple `Array` objects will likely be aliasing overlapping memory address ranges at the same time from the host's perspective. This is fine when using a single stream because device work is guaranteed to be performed in-order. It is advantageous, even, because it reduces device memory usage compared to allocating all variables at the same time at program initialization.

When using multiple streams at the same time, however, there is generally no guarantee in what order the work will be completed. This means these `Array` objects aliasing the same memory range might end up running at the same time, producing indeterminate and incorrect results. To avoid this, the user has two options. First, the user can disable the pool at initialization during runtime using the `yakl::InitConfig` class or using the `export GATOR_DISABLE=1` shell environment variable. This will then use expensive device allocation routines every time an `Array` object is allocated, and the performance hit might be unacceptably large. The alternative is for the user to manage the dependencies of data on streams themselves.

This is done via the `Array` class's `add_stream_dependency(yakl::Stream stream)` member function. If you know that your `Array` object, `arr`, is used by kernels launched in `stream1`, then you can declare that stream dependence with `arr.add_stream_dependency(stream1)`. Then, whenever that array is deallocated, either with an explicit deallocate call or by falling out of scope, events will be declared in all streams that array is dependent upon, and it will not *actually* be released from the pool until those events are completed. This removes any possibility of `Array` objects aliasing the same memory range being used at the same time on the device. If the user desires to use multiple streams and the pool allocator simultane-

ously, they take responsibility for declaring stream dependencies for data used during runtime.

3.14 Debugging and Profiling Capabilities

YAKL has a number of debugging capabilities, the vast majority of which are turned on with the CPP macro variable `YAKL_DEBUG` defined at compile time. This flag turns on checks that ensure the following things among others:

- The Array constructor used has the correct dimensionality
- Non-owned Array objects are not wrapping **`nullptr`**
- Array indices are in bounds and have the correct dimensionality
- Arrays are not indexed before they are allocated
- Host Arrays are not indexed inside device kernels
- Device Arrays are not indexed on the host unless Managed / Shared memory is turned on via the CPP macro variable `YAKL_MANAGED_MEMORY`
- Deep copies are only between Array objects of the same type and total element count
- Array slices are performed appropriately
- Array reshaping maintains the total element count
- No intrinsic routine or `Create[Host | Device]Copy` routine is called on an unallocated array.
- All entries are freed from the memory pool before calling `yakl::finalize()`
- `Bounds` objects passed to `parallel_for` have appropriate bounds (strides are positive, and upper bounds are greater than or equal to lower bounds).

YAKL also has an automated "printf" debugging capability enabled by defining the CPP macro `YAKL_VERBOSE_FILE` at compile time. This will dump one file per process containing all activity in the YAKL library, flushing after each line printed, to enable the user to determine where each MPI task fails in a failed run.

YAKL has built-in timers as well as hooks to switch to other timing libraries. Using `yakl::timer_start(char const *label)` and `yakl::timer_stop(char const *label)`, YAKL keeps track of the runtime between those calls for each CPU thread and MPI task (using a `fence()` operation before each to ensure GPU work has completed). YAKL will print out the timers in human readable fixed column width format for the main task to `stdout` at the end of the run by default, but the user can override this behavior if they desire. Timers are enabled by specifying the CPP macro `YAKL_PROFILE`. If the user wishes to automatically generate timers for all YAKL kernel launches (including internal ones), the CPP macro `YAKL_AUTO_PROFILE` will accomplish this. All `parallel_for` calls without labels will stated as unlabeled with no distinctions between them.

4 A Look at the YAKL Hardware Targets

4.1 Nvidia GPUs with CUDA

YAKL's CUDA hardware target is used to target Nvidia GPUs. Memory allocations and frees are performed with `cudaMalloc` and `cudaFree`. If the C Pre-Processor (CPP) macro `YAKL_MANAGED_MEMORY` is defined, then `cudaMallocManaged` is used to allow device allocations to be used on the host. If the CPP macro `_OPENACC` is defined along with `YAKL_MANAGED_MEMORY`, then all allocated memory is run through `acc_map_data` to ensure the OpenACC runtime does not automatically create data statements for those pointer address ranges. Similarly, if the CPP macro `_OPENMP45` is defined, then managed allocations are run through `omp_target_associate_ptr` to ensure the OpenMP runtime leaves data within those address ranges alone. Memory transfers are performed with `cudaMemcpyAsync`.

CUDA has hardware atomic functions for addition, minimum, and maximum operators, and they are used when possible. When the CUDA compute capability is too low for a given data type and operation, then a compare and swap (CAS) implementation is used instead.

CUDA makes the following definitions:

```
#define YAKL_LAMBDA [=] __host__ __device__
#define YAKL_DEVICE_LAMBDA [=] __device__
#define YAKL_CLASS_LAMBDA [=, *this] __host__ __device__
#define YAKL_INLINE __host__ __device__ __forceinline__
#define YAKL_DEVICE_INLINE __device__ __forceinline__
#define YAKL_SCOPE(a,b) auto &a = b
#define YAKL_SEPARATE_MEMORY_SPACE
#define YAKL_CURRENTLY_ON_HOST() (! defined(__CUDA_ARCH__))
#define YAKL_CURRENTLY_ON_DEVICE() (defined(__CUDA_ARCH__))
```

The latter macro functions are useful for determining whether a section of code is currently being executed in the host compiler pass or the device compiler pass in order to hide host-only code from device compilation. This is how YAKL handles managed data structures (which contain host-only reference counting and allocation / free calls) being passed to device kernels without warnings and errors.

For kernel launches, the "chevron syntax" is used, and the total amount of threading, `nIter` is decomposed into `vectorSize` threads within a CUDA "block" and `ceil(nIter / vectorSize)` threads distributed across CUDA blocks, where `nIter` represents the total number of threads being distributed in parallel. `vectorSize` defaults to 128 but can be changed in the `parallel_for` call. All CUDA kernels and `cudaMemcpy` calls are performed in the CUDA default stream (which is the same as stream "0") unless the user specifies a different stream. One aspect of CUDA not experienced with other hardware backends is that the kernel launch parameter size has a limit of typically 4 Kb. Whenever a function is called with more than this, the kernel has to first be loaded into a temporary buffer in device memory and then launched from device memory with a dereferencing of the functor.

For reductions, the Nvidia "CUB" library[15] is used, and for FFTs, the Nivdia "cuFFT" library is used.[16]

For reductions, the Nvidia CUB is used.

4.2 AMD GPUs with HIP

The HIP backend is used to target AMD GPUs, and it is unsurprisingly very similar to the CUDA backend. The following macros differ from CUDA's

```
#define YAKL_CURRENTLY_ON_HOST() (! defined(__HIP_DEVICE_COMPILE__))
#define YAKL_CURRENTLY_ON_DEVICE() defined(__HIP_DEVICE_COMPILE__)
```

Reductions use the "hipCUB" library,[17] and FFTs use the rocFFT library.[18] Also, with HIP, the default vector length is 256.

4.3 Intel GPUs with SYCL

The SYCL backend is used to target Intel GPUs. The SYCL backend is different from the CUDA and HIP backends in a number of ways, though the workflow is still similar. In this backend, the software abstractions of CUDA/HIP streams maps to a SYCL `queue` object. While CUDA and HIP enqueue tasks to the default stream respectively for each, the SYCL workflow is intended to explicitly create a SYCL queue during initialization for enqueueing tasks. Because there is some static initialization in the SYCL runtime, one cannot simply create a global SYCL `queue` object because initialization order of static data is not guaranteed in C++. Therefore, a "singleton" C++ pattern was used so that upon first access, the SYCL queue is created and used thereafter when referenced. YAKL's SYCL backend makes the following macro definitions:

```
#define YAKL_LAMBDA [=]
#define YAKL_DEVICE_LAMBDA [=]
#define YAKL_CLASS_LAMBDA [=, *this]
#define YAKL_INLINE __inline__ __attribute__((always_inline))
#define YAKL_DEVICE_INLINE __inline__ __attribute__((always_inline))
#define YAKL_SCOPE(a,b) auto &a = std::ref(b).get()
#define YAKL_CURRENTLY_ON_HOST() (! defined(__SYCL_DEVICE_ONLY__))
#define YAKL_CURRENTLY_ON_DEVICE() (defined(__SYCL_DEVICE_ONLY__))
```

The Unified Shared Memory, or USM from SYCL 2020 specifications was used to manage host and device pointers. SYCL is typically used with a "buffer" model in which host and device pointers are managed inside a buffer data structure. SYCL kernel launches are traditionally performed two fold: one call to submit work to a queue and another call to access either host or device handles from buffer objects for reading or writing. In YAKL, however, the functors are launched directly to avoid this workflow, which is not compatible with YAKL's usage or with other specifications like CUDA or HIP.

[15] https://github.com/NVIDIA/cub.

[16] https://docs.nvidia.com/cuda/cufft/index.html.

[17] https://github.com/ROCmSoftwarePlatform/hipCUB.

[18] https://github.com/ROCmSoftwarePlatform/rocFFT.

A recent development of SYCL is support for the `is_device_copyable` C++ type trait. YAKL currently overloads this for all functors small enough that the total parameter space for the functor being launched is less than 2,048 bytes (a current Intel hardware limitation). Then, SYCL accepts the functor as a device copyable structure, copies it to the device, and launches it, even though it is not "trivially copyable". Whenever the size of the functor is too large to achieve this, it is manually copied to a device memory buffer similar to large CUDA functors.

In the SYCL backend, reductions are performed using the SYCL 2020 specification. Atomic instructions are achieved with the `fetch_[min|max|add]` member functions of the `relaxed_atomic_ref` SYCL class. SYCL `parallel_for` launches are performed directly from the default queue object created upon first instantiation of a class with a Singleton pattern, and the SYCL backend synchronizes with the queue's `wait()` function. FFTs are compute using the Intel MKL library.[19]

SYCL `parallel_for` calls use the `sycl::nd_range` approach for specifying the total number of threads as well as the "local size" to use (an analogue of block size in CUDA). YAKL currently defaults to a size of 128.

4.4 CPU Threading with OpenMP

The CPU threading in YAKL is implemented with OpenMP pragma statements implemented directly inline with the serial CPU for loops, where the OpenMP `collapse` clause is used to collapse all loops into a single level of threadable parallelism.

5 Concluding Remarks and Future Work

This paper has introduced the Yet Another Kernel Launcher (YAKL) C++ portability library, which seeks to enable user-level code that looks like Fortran code for scientific developers who are comfortable in that context. YAKL's features are explored, and examples of its use are provided. The hardware backends are described in detail.

From the authors' experiences, while Fortran is a helpful language in many ways, it is not always simple to run Fortran code on accelerators. Directives-based runtime implementations have varying levels of difficulty when modern features of the Fortran language are used such as classes, type-bound procedures, and non-contiguous pointers. Fortran's module-based structure can make inlining code more difficult, which can lead to difficulties in calling routines from device kernels. Further, the feature sets supported from directives-based specifications can vary widely from one compiler to another, making portability across many compilers more difficult. There are many codes for which Fortran with directives works quite well, and there are some for which the implementations are less reliable. This was the one of the motivations for moving to C++ for the E3SM-MMF project. The authors' experiences have been more predictable and less prone to compiler bugs when using C++ portability.

However, using a C++ portability library is not a decision to be taken lightly, and it is not necessarily the right decision for all projects. Converting Fortran code to a C++

[19] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

portability library is a daunting task, though that is one of the main tasks that the YAKL library seeks to make easier by allowing Fortran-like behavior in the resulting user-level C++ code. For a more complete description of YAKL, please see the tutorial-style and API documentation located at https://github.com/mrnorman/YAKL/wiki.

Regarding future work, there is ongoing investigation into the degree to which YAKL can be built on top of the Kokkos library. Particularly, the parallel dispatch seems to be the most straightforward aspect of YAKL to use kokkos as a backend for. Other than this, the main additional functionality that is planned for inclusion into YAKL includes implementation of additional vendor library provided routines to act on YAKL Array objects, such as scan operations, batched reductions, sorting routines, argmin, and argmax – as well as operations at the "inner" parallelism level for hierarchical parallelism applications.

Acknowledgements This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.
This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
The authors wish to thank Daniel Arndt from Oak Ridge National Laboratory, who helped inform best practices for using SYCL `is_device_copyable`.

References

1. Gropp, W., Gropp, W.D., Lusk, E., Skjellum, A., Lusk, A.D.F.E.E.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press (1999)
2. Wienke, S., Springer, P., Terboven, C., et al.: Openacc-first experiences with real-world applications. In: European Conference on Parallel Processing, pp. 859–870. Springer (2012)
3. Sommer, L., Korinth, J., Koch, A.: Openmp device offloading to fpga accelerators. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 201–205. IEEE (2017)
4. Edwards, H.C., Trott, C.R.: Kokkos: enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (xsw 2013), pp. 18–24. IEEE (2013)
5. Beckingsale, D.A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A.J., Pearce, O., Robinson, P., Ryujin, B.S., Scogland, T.R.: Raja: Portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in Hpc (p3hpc), pp. 71–81. IEEE (2019)
6. Ashbaugh, B., Bader, A., Brodman, J., Hammond, J., Kinsner, M., Pennycook, J., Schulz, R., Sewall, J.: Data parallel c++ enhancing sycl through extensions for productivity and performance. In: Proceedings of the International Workshop on OpenCL, pp. 1–2 (2020)

7. Norman, M., Larkin, J.: A holistic algorithmic approach to improving accuracy, robustness, and computational efficiency for atmospheric dynamics. SIAM J. Sci. Comput. **42**(5), 1302–1327 (2020)
8. Norman, M.R.: A high-order weno-limited finite-volume algorithm for atmospheric flow using the ader-differential transform time discretization. Q. J. R. Meteorol. Soc. **147**(736), 1661–1690 (2021)
9. Lyngaas, I., Norman, M., Kim, Y.: Sam++: Porting the e3sm-mmf cloud resolving model using a c++ portability library. Int. J. High Perform. Comput. Appl. 10943420211044495 (2021)
10. Norman, M.R., Bader, D.A., Eldred, C., Hannah, W.M., Hillman, B.R., Jones, C.R., Lee, J.M., Leung, L., Lyngaas, I., Pressel, K.G., et al.: Unprecedented cloud resolution in a gpu-enabled full-physics atmospheric climate simulation on olcf's summit supercomputer. Int. J. High Perform. Comput. Appl. **36**(1), 93–105 (2022)