Kokkos

The Programming Guide

Version 1.0 May 2015

Sandia National Laboratories

SAND2015-4178 O

Authors:

Christian R. Trott (crtrott@sandia.gov)
Mark Hoemmen (mhoemm@sandia.gov)
Simon D. Hammond (sdhammo@sandia.gov)
H. Carter Edwards (hcedwar@sandia.gov)

Version:

1.0 May 2015

This document constitutes the initial version of the Kokkos Programming Guide. As such we ask readers to submit feedback about usability, readability and outright errors. We expect this document to grow over time and include more comprehensive coverage of Kokkos's features as well as more examples to guide users. But we hope that it will proof useful in its current state.

Contents

1 Introduction

The field of High Performance Computing (HPC) is on the verge of entering a new era. The need for a fundamental change comes from many angles including the growing acceptance that rates of pure computation (often called FLOP/s) are a poor single optimization goal for scientific workloads, as well as practical challenges in the form of producing energy efficient and cost efficient processors. Since the convergence on the Message-Passing Interface (MPI) standard in the mid 1990s, application developers have enjoyed a seemingly static view of the underlying machine – that of a distributed collection of homogeneous nodes executing in collaboration. However, after almost two decades of domaince, the sole use of MPI to derive parallelism is acting as a potential limiter to greater future performance. While we expect MPI to continue to function as the basic mechanism for communication between compute nodes for the immediate future, additional parallelism is likely to be required on the computing node itself if high performance and efficiency goal are to be realized.

In reviewing potential options for the computing nodes of the future the reader might fall upon three broad categories of computing device: (1) the multi-core processor with powerful serial performance, optimized to reduce operation latency; (2) many-core processors with low to medium powered compute cores that are designed to extend the multi-core concept toward throughput based computation and, finally, (3), the general-purpose graphics processor unit (GP-GPU, or often, GPU) which is a much more numerous collection of low powered cores designed to tolerate long latencies but provide performance through a much higher degree of parallelism and computational throughput. Any combination of these options might also be combined in the future.

The diversity of the options available for processor selection raises as interested question as to how these should be programmed. In part due to their heritage, but also due to their optimized designs, each of these hardware types offers a different programming solution and a different set of guidelines to write applications to for highest performance. Options available today include a number of shared memory approaches such as OpenMP, Cilk+, Thread Building Blocks (TBB) as well as Linux p-threads. To target both contemporary multi/many-core processors and GPUs technologies such as OpenMP, OpenACC and OpenCL might be used. Finally, for highest performance on GPUs a programming model such as CUDA may be selected. Such a variety of options poses a problem to the application developer of today which is reminiscent of the challenges before MPI became the default communication library – which model should be selected to provide portability across hardware solutions, as well as provide high performance across each class of processor and protect algorithm investment into the future. None of the models listed above have been able to provide practical solutions to these questions.

The Kokkos programming model described in this programming guide seeks to address these concerns by providing an abstraction of both computation and application data allocation and layout. These abstraction layers are designed specifically to isolate software developers from fluctuation and diversity in hardware details yet provide portability and high levels of performance across many architectures.

This guide provides an introduction to the motivations of developing such an abstraction library, a coverage of the programming model and its implementation as an embedded C++ library requiring no additional modifications to the base C++ language. As such it should be seen as an introduction for new users and as a reference for application developers who are already employing Kokkos in their applications. Finally, supplementary tutorial examples are included as part of the Kokkos software release to help users experiment and explore the programming model through a gradual series of steps.

2 Machine Model

After reading this chapter you will understand the abstract model of a parallel computing node which underlies the design choices and structure of the Kokkos framework. The machine model ensures the applications written using Kokkos will have portability across architectures while being performant on a range of hardware.

The machine model has two important components:

- *Memory spaces*, in which data structures can be allocated
- *Execution spaces*, which execute parallel operations using data from one or more memory spaces

2.1 Motivations

Kokkos is comprised of two orthogonal aspects. The first of these is an underlying *abstract machine model* which describes fundamental concepts required for the development of future portable and performant high performance computing applications; the second is a concrete instantiation of the programming this model written in C++, which allows programmers to write to the concept machine model. It is important to treat these two aspects to Kokkos as distinct entities because the underlying model being used by Kokkos could, in the future, be instantiated in additional languages beyond C++ yet the algorithmic specification would remain valid.

2.1.1 Kokkos Abstract Machine Model

Kokkos assumes an *abstract machine model* for the design of future shared-memory computing architectures. The model (shown in Figure 2.1) assumes that there may be multiple execution units in a compute node. For a more general discussion of abstract machine models for Exascale computing the reader should consult reference [**?**]. In the figure shown here, we have elected to show two different types of compute units – one which represents multiple latency-optimized cores, similar to contemporary processor cores, and a second source of compute in the form of an off die accelerator. Of note is that the processor and accelerator each have distinct memories, each with unique performance properties, that are accessible across the node (*i.e.* the memory is reachable or *shared* by all execution units. The specific layout shown in Figure 2.1 is an instantiation of the Kokkos abstract machine model used to describe the potential for multiple types of compute engines and memories within a single node. In future systems there may be a range of execution engines which are used in the node ranging from a single type of core as in many/multi-core processors found today through to an range of execution units where many-core processors may be joined to numerous types of accelerator cores. In order to ensure portability to the potential range of nodes an abstraction of the compute engines and available memories are required.

2.2 Kokkos Execution Spaces

Kokkos uses the term *execution spaces* to describe a logical grouping of computation units which share an indentical set of performance properties. An execution space

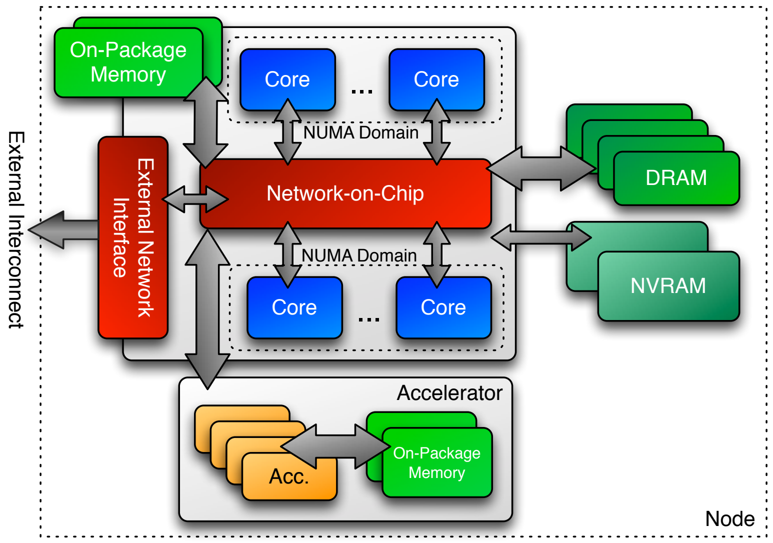

Figure 2.1: Conceptual Model of a Future High Performance Computing Node

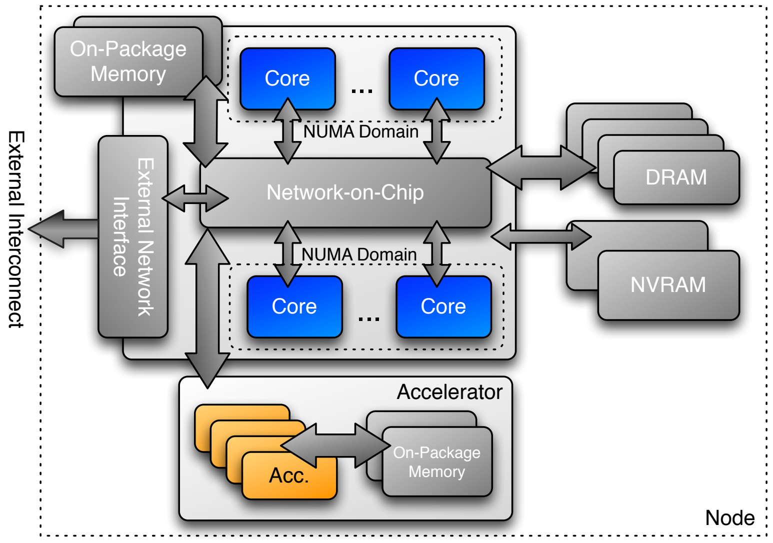

Figure 2.2: Example Execution Spaces in a Future Computing Node

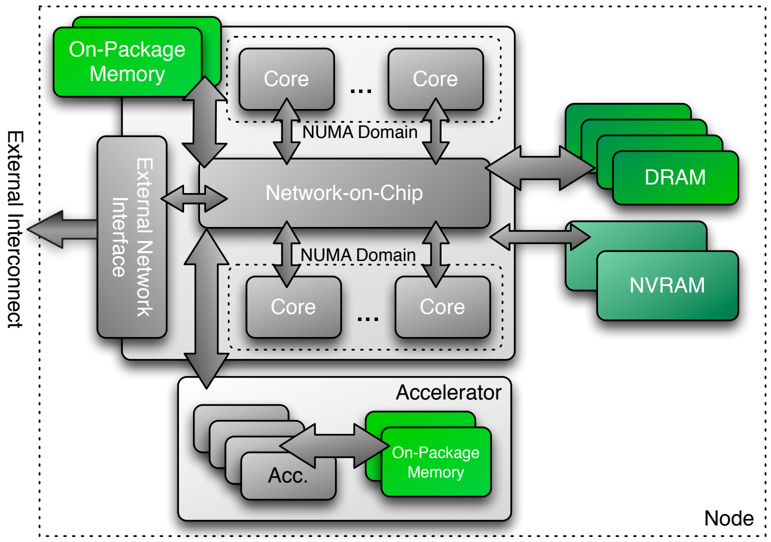

Figure 2.3: Example Memory Spaces in a Future Computing Node

provides a set of parallel execution resources which can be utilized by the programmer using several types of fundamental parallel operation. For a list of the operations available see Chapter 7.

2.2.1 Execution Space Instances

An *instance* of an execution space is a specific instantiation of an execution space to which a programmer can target parallel work. By means of example, an execution space might be used to describe a multi-core processor. In this example, the execution space contains several homogeneous cores which share some logical grouping. In a program written to the Kokkos model, an instance of this execution space would be made available on which parallel kernels could be executed. As a second example, if we were to add a GPU to the multi-core processor a second execution space type is available in the system, the application programmer would then have two execution space instances available to select from. The important consideration here is that the method of compiling code for different execution spaces and the dispatch of kernels to instances is abstracted by the Kokkos framework allowed application programmers to be free from written algorithms in hardware specific languages or requiring the programmer to understand the restrictions required for parallel execution in the execution space.

2.2.2 Kokkos Memory Spaces

The multiple types of memory which will become available in future computing nodes are abstracted by Kokkos through *memory spaces*. Each memory space provides a finite storage capacity at which data structures can be allocated and accessed. In order to provide portable allocations and performant access, Kokkos requires data to be allocated as a k-dimensioned array, where k must be greater than equal to zero (a zero array being a scalar). Each dimension of the array can be either fixed in size at compile time or dynamically sized with size resolution performed during execution.

2.2.3 Instances of Kokkos Memory Spaces

In much the same way execution spaces have specific instaniations through the availablilty of an *instance* so do memory spaces. An instance of a memory space provides a concrete method for the application programmer to request data storage allocatons. Returning to the examples provided for execution spaces, the multi-core processor may have multiple memory spaces available including on-package memory, slower DRAM and additional a set of non-volatile memories. The GPU may also provide an additional memory space through its local on-package memory. The programmer is free to decide where each data structure may be allocated by requesting these from the specific instance associated with that memory space. Kokkos provides the appropriate abstraction of the allocation routines and any associated data management operations including releasing the memory and returning it for future use, as well as copy operations.

Atomic accesses to Memory in Kokkos In cases where multiple executing threads attempt to read a memory address, complete a computation on the item and write it back to same address in memory, an ordering collision may occur. These situations, known as *race conditions* (because the data value stored in memory are the threads complete is dependent on which thread completes its memory operation last), are often the cause of non-determinism in parallel programs. A number of methods can be employed to ensure that race conditions do not occur in parallel programs including the use of locks (which allow only a single thread to gain access to data structure at a time), critical regions (which allow only one thread to execute a code sequence at any point in time) and *atomic* operations. Memory operations which are atomic guarantee that a read, simple computation, and write to memory are completed a single unit. This might allow application programmers to safely increment a memory value for instance, or, more commonly, to safely accumulate values from multiple threads into a single memory locations.

Although atomic operations are logically an operation on memory, they are performed by compute units and so are a property of execution spaces in the Kokkos framework. Execution spaces are not guaranteed to provide atomic operations support but attempts to use them on spaces which do not support atomic access will result in a compile time error.

Memory Consistency in Kokkos Memory consistency models are a complex topic in and of themselves and usually rely on complex operations associated with hardware caches or memory access coherency (for more information see reference [**?**]. Kokkos, does not *require* caches to be present in hardware and so assumes an extremely weak memory consistency model. In the Kokkos model, the programmer should not assume any specific ordering of memory operations being isused by a kernel. This has the potential to create race conditions between memory operations if these are not appropriately protected. In order to provide a guaranteed that memory are completed, Kokkos provides a *fence* operation which forces the compute engine to complete all outstanding memory operations before any new ones can be issued. With appropriate use of fences, programmers are thereby able to ensure that guarantees can be made as to when data will *definitely* have been written to memory.

2.3 Program execution

It is tempting to try to define formally what it means for a processor to execute code. None of us authors have a background in logic or what computer scientists call "formal methods," so our attempt might not go very far! We will stick with informal definitions and rely on Kokkos' C++ implementation as an existence proof that the definitions make sense.

Kokkos lets users tell execution spaces to execute parallel operations. These include parallel for, reduce, and scan (see Chapter 7) as well as View allocation and initialization (see Chapter 6). We name the class of all such operations *parallel dispatch*.

From our perspective, there are three kinds of code:

1. Code executing inside of a Kokkos parallel operation
2. Code outside of a Kokkos parallel operation that asks Kokkos to do something (e.g., parallel dispatch itself)
3. Code that has nothing to do with Kokkos

The first category is the most restrictive. Section **??** above explains restrictions on inter-team synchronization. In general, we limit the ability of Kokkos-parallel code to invoke Kokkos operations (other than for nested parallelism; see Section **??** above and Chapter 8). We also forbid dynamic memory allocation (other than from the team's scratch pad) in parallel operations. Whether Kokkos-parallel code may invoke operating system routines or third-party libraries depends on the execution and memory spaces being used. Regardless, restrictions on inter-team synchronization have implications for things like filesystem access.

Kokkos threads are for computing in parallel, not for overlapping I/O and computation, and not for making graphical user interfaces responsive. Use other kinds of threads (e.g., operating system threads) for the latter two purposes. You may be able to mix Kokkos' parallelism with other kinds of threads; see Section 2.3.1. Kokkos' developers are also working on a task parallelism model that will work with Kokkos' existing data-parallel constructs.

Reproducible reductions and scans Kokkos promises *nothing* about the order in which the iterations of a parallel loop occur. However, it *does* promise that if you execute the same parallel reduction or scan, using the same hardware resources and run-time settings, then you will get the same results each time you run the operation. "Same results" even means "with respect to floating-point rounding error."

Asynchronous parallel dispatch This concerns the second category of code, that calls Kokkos operations. In Kokkos, parallel dispatch executes *asynchronously*. This means that it may return "early," before it has actually completed. Nevertheless, it executes *in sequence* with respect to other Kokkos operations on the same execution or memory space. This matters for things like timing. For example, a `parallel_for` may return "right away," so if you want to measure how long it takes, you must first call `fence()` on that execution space. This forces all functors to complete before `fence()` returns.

2.3.1 Thread safety?

Users may wonder about "thread safety," that is, whether multiple operating system threads may safely call into Kokkos concurrently. Kokkos' thread safety depends on both its implementation, and on the execution and memory spaces that the implementation uses. The C++ implementation has made great progress towards (non-Kokkos) thread safety of View memory management. For now, however, the most portable approach is for only one (non-Kokkos) thread of execution to control Kokkos. Also, be aware that operating system threads might interfere with Kokkos' performance, depending on the execution space that you use.

Thread safety with respect to Kokkos' threads is a different matter. View (see Chapter 6) promises safety of its memory management. That is, you may safely access Views inside of Kokkos' parallel operations. Kokkos provides some synchronization constructs as well; see the following section.

3 Programming Model

The programming model Kokkos is characterized by 6 core abstractions: Execution Spaces, Execution Patterns, Execution Policies, Memory Spaces, Memory Layout and Memory Traits. These abstraction concepts allow the formulation of generic algorithms and data structures which can then be mapped to different types of architectures. Effectively they allow for compile time transformation of algorithms to allow for adaptions of varying degrees of hardware parallelism as well as of the memory hierarchy.

3.1 Execution Spaces

An Execution Space is the place *Where* code can actually be executed. For example on current Hybrid GPU / CPU systems there are two types of execution spaces: the GPU cores and the CPU cores. In the future this could include Processing in Memory (PIM) modules or different core types on a heterogenous CPU. In principle this can also be used to introduce remote memory spaces, e.g. the capability of sending work to a different node. Execution Spaces thus give an application developer the means to target different parts of a heterogenous hardware architecture. This corresponds directly to the previously described machine model.

3.2 Execution Patterns

Execution Patterns are the fundamental parallel algorithms in which an application has to be expressed. Examples are `parallel_for`: execute a function in undetermined order a specified amount of times., `parallel_reduce`: which combines a `parallel_for` execution with a reduction operation, `parallel_scan`: which combines a `parallel_for` operation with a prefix or postfix scan on output values of each operation, and `task`: which executes a single function with dependencies on other functions. Expressing an application in these patterns allows the underlying implementation or the used compiler to reason about valid transformations. For example all `parallel_***` patterns allow unspecified execution order, and only promise deterministic results of the reductions themselves. This enables different mapping patterns on different hardware such as assignment of iterations to threads or vector lanes.

3.3 Execution Policies

An Execution Policy determines together with an Execution Pattern *How* a function is executed. Some policies can be nested in others.

3.3.1 Range Policies

The most simple form of execution policies are *Range Policies*. They are used to execute an operation once for each element in a range. There are no prescriptions of order of execution or concurrency, which means that it is not legal to synchronize different iterations.

3.3.2 Team Policies

Team policies are used to implement hierarchical parallelism. For that purpose Kokkos groups threads into *teams*. A *thread team* is a collection of one or more parallel "threads" of execution. Kokkos allows an arbitrary number of teams – the *league size*. Hardware constrains the number of threads in a team – the *team size*. All threads in a team are guaranteed to run concurrently.

Threads in a team can synchronize – they have a "barrier" primitive – and share a "scratch pad" memory which they may use for temporary storage. Note that the not all forms of synchronization mechanism are legal in Kokkos, in particular implementing "spin-locks" for threads in a team may result in dead locks. In Kokkos there is no forward progress guarantee for threads, that means that a single thread spinning on a lock acquired by another thread of the team may use 100% of the compute engines resources. Calling the explicit Kokkos team barrier is the only safe way to synchronize threads in a team.

Scratch pad memory exists only during parallel operations; allocations in it do not persist across kernels. Teams themselves may run in any order, and may not necessarily run all in parallel. For example, if the user asks for T teams, the hardware may choose to run them one after another in sequence, or in groups of up to G teams at a time in parallel.

Users may *nest* parallel operations. Teams may perform one parallel operation (for, reduce, or scan), and threads within each team may perform another, possibly different parallel operation. Different teams may do entirely different things. For example, all the threads in one team may execute a `parallel_for`, and all the threads in a different team may execute a `parallel_scan`. Different threads within a team may also do different things. However, performance may vary if threads in a team "diverge" in their behavior (e.g., take different sides of a branch). Chapter 8 shows how the C++ implementation of Kokkos exposes thread teams.

NVIDIA's CUDA programming model inspired Kokkos' thread team model. The scratch pad memory corresponds with CUDA's per-team "shared memory." The "league / team" vocabulary comes from OpenMP 4.0, and has many aspects in common with our thread team model. We have found that programming to this model results in good performance, even on computer architectures which only implement parts of the full model. For example, most multicore processors in common use for high-performance computing lack "scratch pad" hardware. However, if users request a scratch pad size that fits comfortably in the largest cache shared by the threads in a team, programming as if a scratch pad exists forces users to address locality in their algorithms. This also reflects the common experience that rewriting a code for more restrictive hardware, then porting the code *back* to conventional hardware, tends to improve performance relative to an unoptimized code.

3.4 Memory Spaces

Memory Spaces are the places *Where* data resides. They specify physical location of data as well as certain access characteristics. Different physical locations correspond to things such as high bandwidth memory, on die scratch memory or non-volatile bulk storage. Different logical memory spaces allow for concepts such as UVM memory in the CUDA programming model, which is accessible from Host and GPU. Memory Spaces also can be used to express remote memory locations. Furthermore they encapsulate functionality such as consistency control and persistence scopes.

3.5 Memory Layout

Layouts express the mapping from logical (or algorithmical) indicies to address offset for a data allocation. By adopting appropriate layouts for memort structures an application can optimise data access patterns in a given algorithm. If an implementation provides polymorphic layouts (i.e. a data structure can be instantiated at compile or runtime with different layouts) an architecture dependent optimisation can be performed.

3.6 Memory Traits

Memory Traits specify how a data structure is accessed in an algorithm. Traits express usage scenarios such as atomic access, random access and streaming loads or stores. By putting such attributes on data structures, an implementation of the programming model can insert optimal load and store operations. If a compiler implements the programming model, it could reason about the access modes and use that to inform code transformations.

4 Compiling

This chapter explains how to compile Kokkos, and how to link your application against Kokkos. Kokkos supports two build systems:

- Using the embedded Makefile
- Trilinos' CMake build system

Note that the two explicitly supported build methods should not be mixed. For example, do not include the embedded Makefile in your application build process, while explicitly linking against a pre-compiled Kokkos library in Trilinos. We also include specific advice for building for NVIDIA GPUs, and for Intel Xeon Phi.

4.1 General Information

Kokkos consists mainly of header files. Only a few functions have to be compiled into object files outside of the application's source code. Those functions are contained in `.cpp` files inside the `kokkos/core/src` directory and its subdirectories. The files are internally protected with macros to prevent compilation if the related execution space is not enabled. Thus, it is not necessary to create a list of included object files specific to your compilation target. One may simply compile all `.cpp` files. The enabled features are controlled via macros which have to be provided in the compilation line or in the `KokkosCore_config.h` include file. A list of macros can be found in Table 4.1.

4.2 Using Kokkos' Makefile system

The base of the build system is the Makefile.kokkos, which is designed to be included by application Makefiles. It contains logic to (re)generate the KokkosCore_config.h file if necessary, build the Kokkos library, and provide updated compiler and linker flags.

The system can digest a number of variables which are used to configure Kokkos settings. Generally the variables are parsed for Keywords. This allows for multiple options being given for each variable. The separator doesn't matter as long as it doesn't interact with the Make system. A list of variables, their meaning and options is given in table 4.2.

A word of caution on where to include the Makefile.kokkos: since the embedded Makefiles defines targets it is usually better to include it after the first application target has been defined. Since that target can't use the flags from the embedded Makefiles it should be a meta target:

```
CXX=g++

default: main

include Makefile.kokkos

main: $(KOKKOS_LINK_DEPENDS) $(KOKKOS_CPP_DEPENDS) main.cpp
    $(CXX) $(KOKKOS_CPPFLAGS) $(KOKKOS_CXXFLAGS) \
    $(KOKKOS_LDFLAGS) $(KOKKOS_LIBS) main.cpp -o main
```

Table 4.1: Table of configuration Macros

Macro	Effect	Comment
KOKKOS_HAVE_CUDA	Enable the CUDA execution space.	Requires a compiler capable of understanding CUDA-C. See Section 4.4.
KOKKOS_HAVE_OPENMP	Enable the OpenMP execution space.	Requires the compiler to support OpenMP (e.g., `-fopenmp`).
KOKKOS_HAVE_PTHREADS	Enable the Threads execution space.	Requires linking with libpthread.
KOKKOS_HAVE_Serial	Enable the Serial execution space.	
KOKKOS_HAVE_CXX11	Enable internal usage of C++11 features.	The code needs to be compile with the C++11 standard. Most compilers accept the `-std=c++11` flag for this.
KOKKOS_HAVE_HWLOC	Enable thread and memory pinning via hwloc.	Requires linking with libhwloc.

More example application Makefiles can be found in the tutorial examples under `kokkos/example/tutorial`.

Kokkos provides a script `generate_makefile.bash` which can generate a Makefile for building and installing the library as well as building and running the tests. Please run `generate_makefile.bash --help` for options. Note that paths given to the script must be absolute paths, and the script must be run with the `bash` shell (the script will do it if it is run directly i.e. as `./generate_makefile.bash`.

4.3 Using Trilinos' CMake build system

The Trilinos project (see `trilinos.org`) is an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multiphysics engineering and scientific problems. Trilinos is organized into packages. Even though Kokkos is a stand-alone software project, Trilinos uses Kokkos extensively. Thus, Trilinos' source code includes Kokkos' source code, and builds Kokkos as part of its build process.

Trilinos' build system uses CMake. Thus, in order to build Kokkos as part of Trilinos, you must first install CMake (version `2.8.12` or newer; CMake `3.x` works). To enable Kokkos when building Trilinos, set the CMake option `Trilinos_ENABLE_Kokkos`. Trilinos' build system lets packages express dependencies on other packages or external libraries. If you enable any Trilinos package (e.g., Tpetra) that has a required dependency on Kokkos, Trilinos will enable Kokkos automatically. Configuration macros are automatically inferred from Trilinos settings. For example, if the CMake option `Trilinos_ENABLE_OpenMP` is ON, Trilinos will define the macro `KOKKOS_HAVE_OPENMP`. Trilinos' build system will autogenerate the previously mentioned `KokkosCore_config.h` file that contains those macros.

We refer readers to Trilinos' documentation for details. Also, the `kokkos/config` directory includes examples of Trilinos configuration scripts.

Table 4.2: Variables for the embedded Makefile

Variable	Description
KOKKOS_PATH (IN):	Path to the Kokkos root or install directory. One can either build against an existing install of Kokkos or use its source directly for an embedded build. In the former case the "Input variables" are set inside the embedded Makefile.kokkos and it is not valid to set them differently in the including Makefile.
CUDA_PATH (IN):	Path to the Cuda toolkit root directory.
KOKKOS_DEVICES (IN):	What Execution and Memory Spaces should be enabled.
Options:	OpenMP,Serial,Pthreads,Cuda
Default:	OpenMP
KOKKOS_ARCH (IN):	What backend Architecture to build for.
Options:	KNC,SNB,HSW,Kepler,Kepler30,Kepler35,Kepler37,Maxwell, Maxwell50,ARMv8,BGQ,Power7,Power8
Default:	None – this means no particular architecture flags are set.
KOKKOS_USE_TPLS (IN):	Enable optional third party libraries.
Options:	hwloc,librt
Default:	
KOKKOS_CUDA_OPTIONS (IN):	Enable optional settings specific to CUDA.
Options:	force_uvm,use_ldg,rdc
Default:	
HWLOC_PATH (IN):	Path to the hardware locality library if enabled.
KOKKOS_DEBUG (IN):	Enable debugging.
Options:	yes,no
Default:	no
KOKKOS_CXX_STANDARD (IN):	Set the C++ standard to be used.
Options:	c++11
Default:	c++11
KOKKOS_CPPFLAGS (OUT):	Pre processor flags (include directories and defines). Add this to applications compiler and preprocessor flags.
KOKKOS_CXXFLAGS (OUT):	Compiler flags. Add this to the applications compiler flags.
KOKKOS_LDFLAGS (OUT):	Linker flags. Add this to the applications linker flags.
KOKKOS_LIBS (OUT):	Libraries required by Kokkos. Add this to the link line after the linker flags.
KOKKOS_CPP_DEPENDS (OUT):	Dependencies for compilation units which include any Kokkos header files. Add this as a dependency to compilation targets including any Kokkos code.
KOKKOS_LINK_DEPENDS (OUT):	Dependencies of an application linking in the Kokkos library. Add this to the dependency list of link targets.
CXXFLAGS (IN):	User provided compiler flags which will be used to compile the Kokkos library.
CXX (IN):	The compiler used to compile the Kokkos library.

4.4 Building for CUDA

Any Kokkos application compiled for CUDA embeds CUDA code via template meta-programming. Thus, the whole application must be built with a CUDA-capable compiler. (At the moment, the only such compiler is NVIDIA's NVCC.) More precisely, every compilation unit containing a Kokkos kernel or a function called from a Kokkos kernel has to be compiled with a CUDA-capable compiler. This includes files containing `Kokkos::View` allocations, which call an initialization kernel.

The current version of NVCC has some shortcomings when used as the main compiler for a project, in particular when part of a complex build system. For example, it does not understand most GCC command-line options, which must be prepended by the `-Xcompiler` flag when calling NVCC. Kokkos comes with a shell script, called `nvcc_wrapper`, that wraps NVCC to address these issues. We intend this as a drop-in replacement for a normal GCC-compatible compiler (e.g., GCC or Intel) in your build system. It analyzes the provided command-line options and prepends them correctly. It also adds the correct flags for compiling generic C++ files containing CUDA code (e.g., `*.cpp`, `*.cxx`, or `*.CC`). By default `nvcc_wrapper` calls `g++` as the host compiler. You may override this by providing NVCC's '`-ccbin`' option as a compiler flag. The default can be set by editing the script itself or by setting the environment variable `NVCC_WRAPPER_DEFAULT_COMPILER`.

Many people use a system like Environment Modules (see `http://modules.sourceforge.net/`) to manage their shell environment. When using a module system, it can be useful to provide different versions for different back-end compiler types (e.g., `icpc`, `pgc++`, `g++`, and `clang`). To use the `nvcc_wrapper` in conjunction with MPI wrappers, simply overwrite which C++ compiler is called by the MPI wrapper. For example, you can reset OpenMPI's C++ compiler by setting the `OMPI_CXX` environment variable. Make sure that `nvcc_wrapper` calls the host compiler with which the MPI library was compiled.

5 Initialization

In order to use Kokkos an initialization call is required. That call is responsible for aquiring hardware resources such as threads. Typically this call should be placed right at the start of a program. If you use both MPI and Kokkos, your program should initialize Kokkos right after calling `MPI_Init`. That way, if MPI sets up process binding masks, Kokkos will get that information and use it for best performance. Your program must also *finalize* Kokkos when done using it, in order to free hardware resources.

5.1 Initialization by command-line arguments

The simplest way to initialize Kokkos is by calling the following function:

```
Kokkos::initialize(int& argc, char* argv[]);
```

Just like `MPI_Init`, this function interprets command-line arguments to determine the requested settings. Also like `MPI_Init`, it reserves the right to remove command-line arguments from the input list. This is why it takes `argc` by reference, rather than by value; it may change the value on output.

This function will initialize the default execution space

```
Kokkos::DefaultExecutionSpace;
```

and its default host execution space

```
Kokkos::DefaultHostExecutionSpace;
```

if applicable. These defaults depend on the Kokkos configuration. Kokkos chooses the two spaces using the following list, ordered from low to high:

1. `Kokkos::Serial`
2. `Kokkos::Threads`
3. `Kokkos::OpenMP`
4. `Kokkos::Cuda`

The highest execution space in the list which is actually enabled is Kokkos' default execution space, and the highest enabled host execution space is Kokkos' default host execution space. (Currently, the only non-host execution space is `Cuda`.) For example, if `Kokkos::Cuda`, `Kokkos::OpenMP`, and `Kokkos::Serial` are enabled, then `Kokkos::Cuda` is the default execution space and `Kokkos::OpenMP` is the default host execution space.[1]

Command-line arguments come in "prefixed" and "non-prefixed" versions. Prefixed versions start with the string `--kokkos-`. `Kokkos::initialize` will remove prefixed options from the input list, but will preserve non-prefixed options. Argument options are given with an equals (=) sign. If the same argument occurs more than once, the last one counts. Furthermore, prefixed versions of the command line arguments take precedence over the non-prefixed ones. For example, the arguments

[1]This is the preferred set of defaults when CUDA and OpenMP are enabled. If you use a thread-parallel host execution space, we prefer Kokkos' OpenMP back-end, as this ensures compatibility of Kokkos' threads with the application's direct use of OpenMP threads. Kokkos cannot promise that its Threads back-end will not conflict with the application's direct use of operating system threads.

Table 5.1: Command-line options for `Kokkos::initialize`

Argument	Description
`--kokkos-help`	print this message
`--kokkos-threads=INT`	specify total number of threads or number of threads per NUMA region if used in conjunction with '–numa' option.
`--kokkos-numa=INT`	specify number of NUMA regions used by process.
`--kokkos-device=INT`	specify device id to be used by Kokkos.
`--kokkos-ndevices=INT[,INT]`	used when running MPI jobs. Specify number of devices per node to be used. Process to device mapping happens by obtaining the local MPI rank and assigning devices round-robin. The optional second argument allows for an existing device to be ignored. This is most useful on workstations with multiple GPUs, of which one is used to drive screen output.

```
--kokkos-threads=4 --threads=2
```

set the number of threads to 4, while

```
--kokkos-threads=4 --threads=2 --kokkos-threads=3
```

set the number of threads to 3. Table 5.1 gives a full list of command-line options.

5.2 Initialization by struct

Instead of giving `Kokkos::initialize()` command-line arguments, one may directly pass in initialization parameters, using the following struct:

```
struct Kokkos::InitArguments {
  int num_threads;
  int num_numa;
  int device_id;
  // ... the struct may have more members ...
};
```

The `num_threads` field corresponds to the `--kokkos-threads` command-line argument, `num_numa` to `--kokkos-numa`, and `device_id` to `--kokkos-device`. (See Table 5.1 for details.) Not all parameters are observed by all execution spaces, and the struct might expand in the future if needed.

If you set `num_threads` or `num_numa` to zero or less, Kokkos will try to determine default values if possible or otherwise set them to 1. In particular Kokkos can use the hwloc library to determine default settings, using the assumption that the process binding mask is unique, i.e. that this process does not share any cores with another process. Note that the default value of each parameter is -1.

Here is an example of how to use the struct.

```
Kokkos::InitArguments args;
// 8 (CPU) threads per NUMA region
args.num_threads = 8;
// 2 (CPU) NUMA regions per process
```

```
args.num_numa = 2;
// If Kokkos was built with CUDA enabled,
// use the GPU with device ID 1.
args.device_id = 1;

Kokkos::initialize(args);
```

5.3 Initializing non-default execution spaces

Instead of calling the generic initialization, one can call initialization for each execution space on its own. If the associated host execution space of an execution space is not identical to the latter, it has to be initialized first. For example when compiling with support for pthreads and Cuda, `Kokkos::Threads` has to be initialized before `Kokkos::Cuda`. The initialization calls take different arguments for each of the execution spaces.

If you want to initialize an execution space other than those that Kokkos initializes by default, you *must* initialize it explicitly in code, by calling its `initialize()` method and passing it the appropriate arguments. Furthermore, if the associated host execution space of an execution space is not identical to the latter, the host execution space must be initialized first. For example, if you have set the default execution space to `Kokkos::Serial`, but want to use `Kokkos::Cuda` and `Kokkos::OpenMP`, you must initialize `Kokkos::OpenMP` before `Kokkos::Cuda`. The initialization calls take different arguments for each of the execution spaces.

We do *not* recommend initializing multiple host execution spaces, because the different spaces may fight over threads or other hardware resources. This may have a big effect on performance. The only exception is the `Serial` execution space. You may safely initialize `Serial` along with another host execution space.

5.4 Finalization

At the end of each program Kokkos needs to be shut down in order to free resources. Do this by calling `Kokkos::finalize()`. You may wish to set this to be called automatically at program exit, either by setting an `atexit` hook or by attaching the function to `MPI_COMM_SELF`, so that it is called automatically at `MPI_Finalize`.

6 View: Multidimensional array

After reading this chapter, you should understand the following:

- A Kokkos View is an array of zero or more dimensions
- How to use View's first template parameter to specify the type of entries, the number of dimensions, and whether the dimensions are determined at run time or compile time
- Kokkos handles array deallocation automatically
- Kokkos chooses array layout at compile time for best overall performance, as a function of the computer architecture
- How to set optional template parameters of View for low-level control of execution space, layout, and how Kokkos accesses array entries

In all code examples in this chapter, we assume that all classes in the `Kokkos` namespace have been imported into the working namespace.

6.1 Why Kokkos needs multidimensional arrays

Many scientific and engineering codes spend a lot of time computing with arrays of data. Programmers invest a lot of effort making array computations as fast as possible. This effort is often intimately bound to details of the computer architecture, run-time environment, language, and programming model. For example, optimal array layout may differ based on the architecture, with a large integer factor penalty if wrong. Low-level issues like pointer alignment, array layout, indexing overhead, and initialization all affect performance. This is true even with sequential codes. Thread parallelism adds even more pitfalls, like first-touch allocation and false sharing.

For best performance, coders need to tie details of how they manage arrays, to details of how parallel code accesses and manages those arrays. Programmers who write architecture-specific code then need to mix low-level features of the architecture and programming model into high-level algorithms. This makes it hard to port codes between architectures, or to write a code that still performs well as architectures evolve.

Kokkos aims to relieve some of this burden, by optimizing array management and access for the specific architecture. Tying arrays to shared-memory parallelism lets Kokkos optimize the former to the latter. For example, Kokkos can easily do first-touch allocation, because it controls threads that it can use to initialize arrays. Kokkos' architecture awareness lets it pick optimal layout and pad allocations for good alignment. Expert coders can also use Kokkos to access low-level or more architecture-specific optimizations in a more user-friendly way. For instance, Kokkos makes it easy to experiment with different array layouts.

6.2 Creating and using a View

6.2.1 Constructing a View

A View is an array of zero or more dimensions. Programmers set both the type of entries, and the number of dimensions, at compile time, as part of the type of the View.

For example, the following specifies and allocates a View with four dimensions, whose entries have type `int`:

```
const size_t N0 = ...;
const size_t N1 = ...;
const size_t N2 = ...;
const size_t N3 = ...;
View<int****> a ("some␣label", N0, N1, N2, N3);
```

The string argument is a label, which Kokkos uses for debugging. Different Views may have the same label. The ellipses indicate some integer dimensions specified at run time. Users may also set some dimensions at compile time. For example, the following View has two dimensions. The first one (represented by the asterisk) is a run-time dimension, and the second (represented by [3]) is a compile-time dimension. Thus, the View is an N by 3 array of type double, where N is specified at run time in the View's constructor.

```
const size_t N = ...;
View<double*[3]> b ("another␣label", N);
```

Views may have up to (at least) 8 dimensions, and any number of these may be run-time or compile-time. The only limitation is that the run-time dimensions (if any) must appear first, followed by all the compile-time dimensions (if any). For example, the following are valid three-dimensional View types:

- `View<int***>` (3 run-time dimensions)
- `View<int**[8]>` (2 run-time, 1 compile-time)
- `View<int*[3][8]>` (1 run-time, 2 compile-time)
- `View<int[4][3][8]>` (3 compile-time)

and the following are *not* valid three-dimensional View types:

- `View<int[4]**>`
- `View<int[4][3]*>`
- `View<int[4]*[8]>`
- `View<int*[3]*>`

This limitation comes from the implementation of View using C++ templates. View's first template parameter must be a valid C++ type.

Note that the above used constructor is not necessarily available for all view types. Specific Layouts or Memory Spaces may require more specialized allocators. This will be discussed later.

6.2.2 What types of data may a View contain?

C++ lets users construct data types that may "look like" numbers in terms of syntax, but do arbitrarily complicated things inside. Some of those things may not be thread safe, like unprotected updates to global state. Others may perform badly in parallel, like fine-grained dynamic memory allocation. Therefore it is strongly advised to use only simple data types inside Kokkos Views. Users may always construct a View whose entries are

- built-in data types ("plain old data"), like `int` or `double`, or
- structs of built-in data types.

While it is in principal possible to have Kokkos Views of arbitrary objects there are a number of restrictions. The `T` type must have a default constructor and destructor. For portability reasons `T` is not allowed to have virtual functions and one is not allowed to call functions with dynamic memory allocations inside of kernel code. Furthermore assignment operators as well as default constructor and deconstructor must be marked with `KOKKOS_[INLINE_]FUNCTION`. Keep in mind that the semantics of the resulting View are a combination of the Views 'view' semantics and the behaviour of the element type.

6.2.3 Const Views

A view can have const data semantics (i.e. its entries are read-only) by specifying a 'const' data type. It is a compile-time error to assign to an entry of a "const View". Assignment semantics are equivalent to a pointer to const data. A const View means the *entries* are const. You may still assign to a const View. `View<const double*>` corresponds exactly to `const double*`, and `const View<double*>` to `double* const`. Therefore it does not make sense to allocate a const View, since you could not obtain a non-const view of the same data and you can not assign to it. You can however assign a non-const view to a const view. Here is an example:

```
const size_t N0 = ...;
View<double*> a_nonconst ("a_nonconst", N0);

// Assign a nonconst View to a const View
View<const double*> a_const = a_nonconst;
// Pass the const View to some read-only function.
const double result = readOnlyFunction (a_const);
```

Const Views often enables the compiler to optimize more aggressively by allowing it to reason about possible write conflicts and data aliasing. For example in a vector update `a(i+1)+=b(i)` with skewed indexing it is safe to vectorize if `b` is a View of const data.

6.2.4 Accessing entries (indexing)

You may access an entry of a View using parentheses enclosing a comma-delimited list of integer indices. This looks just like a Fortran multidimensional array access. For example:

```
const size_t N = ...;
View<double*[3][4]> a ("some␣label", N);
// KOKKOS_LAMBDA macro includes capture-by-value specifier [=].
parallel_for (N, KOKKOS_LAMBDA (const ptrdiff_t i) {
    const size_t j = ...;
    const size_t k = ...;
    const double a_ijk = a(i,j,k);
    /* rest of the loop body */
  });
```

Note how in the above example, we only access the View's entries in a parallel loop body. In general, you may only access a View's entries in an execution space which is allowed to access that View's memory space. For example, if the default execution space is `Cuda`, a View for which no specific Memory Space was given may not be accessed in host code [1]. Furthermore, access costs (e.g., latency and bandwidth) may vary, depending on the View's "native" memory and execution spaces, and the execution space from which you access it. CUDA UVM may work, but it may also be slow, depending on your access pattern and performance requirements. Thus, best practice is to access the View only in a Kokkos parallel for, reduce, or scan, using the same execution space as the View. This also ensures that access to the View's entries respect first-touch allocation. The first (leftmost) dimension of the View is the *parallel dimension*, over which it is most efficient to do parallel array access if the default memory layout is used (e.g. if no specific memory layout is specified).

6.2.5 Reference counting

Kokkos automatically manages deallocation of Views, through a reference-counting mechanism. Otherwise, Views behave like raw pointers. Copying or assigning a View does a shallow copy, and changes the reference count. (The View copied has its reference count incremented, and the assigned-to View has its reference count decremented.) A View's destructor (called when the View falls out of scope or during a stack unwind due to an exception) decrements the reference count. Once the reference count reaches zero, Kokkos may deallocate the View.

For example, the following code allocates two Views, then assigns one to the other. That assignment may deallocate the first View, since it reduces its reference count to zero. It then increases the reference count of the second View, since now both Views point to it.

```
View<int*> a ("a", 10);
View<int*> b ("b", 10);
a = b; // assignment does shallow copy
```

For efficiency, View allocation and reference counting turn off inside of Kokkos' parallel for, reduce, and scan operations. This affects what you can do with Views inside of Kokkos' parallel operations.

6.2.6 Resizing

Kokkos Views can be resized using the `resize` non-member function. It takes an existing view as its input by reference and the new dimension information corresponding to the constructor arguments. A new view with the new dimensions will be created and a kernel will be run in the views execution space to copy the data element by element from the old view to the new one. Note that the old allocation is only deleted if the view to be resized was the *only* view referencing the underlying allocation.

```
// Allocate a view with 100x50x4 elements
View<int**[4]> a( "a", 100,50);

// Resize a to 200x50x4 elements; the original allocation is freed
resize(a, 200,50);

// Create a second view b viewing the same data as a
View<int**[4]> b = a;
```

[1] An exemption is if you specified for CUDA compilation that the default memory space is CudaUVMSpace, which can be accessed from the host.

```
// Resize a again to 300x60x4 elements; b is still 200x50x4
resize(a,300,60);
```

6.3 Layout

6.3.1 Strides and dimensions

Layout refers to the mapping from a logical multidimensional index $(i, j, k, \ldots)$ to a physical memory offset. Different programming languages may have different layout conventions. For example, Fortran uses *column-major* or “left” layout, where consecutive entries in the same column of a 2-D array are contiguous in memory. Kokkos calls this LayoutLeft. C, C++, and Java use *row-major* or “right” layout, where consecutive entries in the same row of a 2-D array are contiguous in memory. Kokkos calls this LayoutRight.

The generalization of both left and right layouts is “strided.” For a strided layout, each dimension has a *stride*. The stride for that dimension determines how far apart in memory two array entries are, whose indices in that dimension differ only by one, and whose other indices are all the same. For example, with a 3-D strided view with strides (s_1, s_2, s_3), entries (i, j, k) and $(i, j+1, k)$ are s_2 entries (not bytes) apart in memory. Kokkos calls this LayoutStride.

Strides may differ from dimensions. For example, Kokkos reserves the right to pad each dimension for cache or vector alignment. You may access the dimensions of a View using the dimension method, which takes the index of the dimension.

Strides are accessed using the stride method. It takes a raw integer array, and only fills in as many entries as the rank of the View. For example:

```
const size_t N0 = ...;
const size_t N1 = ...;
const size_t N2 = ...;
View<int***> a ("a", N0, N1, N2);

int dim1 = a.dimension (1); // returns dimension 1
size_t strides[3]
a.strides (dims); // fill 'strides' with strides
```

You may also refer to specific dimensions without a runtime parameter:

```
const size_t n0 = a.dimension_0 ();
const size_t n2 = a.dimension_2 ();
```

Note the return type of dimension_N() is the size_type of the views memory space. This causes some issues if warning free compilation should be achieved since it will typically be necessary to cast the return value. In particular in cases where the size_type is more conservative than required it can be beneficial to cast the value to int, since signed 32 bit integers typically give the best performance when used as index types. In index heavy codes this performance difference can be significant compared to using size_t since the vector length on many architectures is twice as long for 32 bit values as for 64 bit values, and signed integers have less stringent overflow testing requirements than unsigned integers.

Users of the BLAS and LAPACK libraries may be familiar with the ideas of layout and stride. These libraries only accept matrices in column-major format. The stride between consecutive entries in the same column is 1, and the stride between consecutive entries in the same row is LDA ("leading dimension of the matrix A"). The number of rows may be less than LDA, but may not be greater.

6.3.2 Other layouts

Other layouts are possible. For example, Kokkos has a "tiled" layout, where a tile's entries are stored contiguously (in either row- or column-major order) and tiles have compile-time dimensions. One may also use Kokkos to implement Morton ordering or variants thereof.

6.3.3 Default layout depends on execution space

Kokkos selects a View's default layout for optimal parallel access over the leftmost dimension, based on its execution space. For example, `View<int**, Cuda>` has `LayoutLeft`, so that consecutive threads in the same warp access consecutive entries in memory. This *coalesced access* gives the code better memory bandwidth.

In contrast, `View<int**, OpenMP>` has `LayoutRight`, so that a single thread accesses contiguous entries of the array. This avoids wasting cache lines and helps prevent false sharing of a cache line between threads. In section 6.4 more details will be discussed.

6.3.4 Explicitly specifying layout

We prefer that users let Kokkos determine a View's layout, based on its execution space. However, sometimes you really need to specify the layout. For example, the BLAS and LAPACK libraries only accept column-major arrays. If you want to give a View to the BLAS or LAPACK library, that View must be `LayoutLeft`. You may specify the layout as a template parameter of View. For example:

```
const size_t N0 = ...;
const size_t N1 = ...;
View<double**, LayoutLeft> A ("A", N0, N1);

// Get 'LDA' for BLAS / LAPACK
int strides[2]; // any integer type works in stride()
A.stride (strides);
const int LDA = strides[1];
```

You may ask a View for its layout via its `array_layout` typedef. This can be helpful for C++ template metaprogramming. For example:

```
template<class ViewType>
void callBlas (const ViewType& A) {
  typedef typename ViewType::array_layout array_layout;
  if (std::is_same<array_layout, LayoutLeft>::value) {
    callSomeBlasFunction (A.ptr_on_device (), ...);
  } else {
    throw std::invalid_argument ("A␣is␣not␣LayoutLeft");
  }
}
```

6.4 Managing Data Placement

6.4.1 Memory spaces

Views are allocated by default, in the default execution space's default memory space. You may access the View's execution space via its `execution_space` typedef, and its memory space via its `memory_space` typedef. You may also specify the memory space explicitly as a template parameter. For example, the following allocates a View in CUDA device memory:

```
View<int*, CudaSpace> a ("a", 100000);
```

and the following allocates a View in "host" memory, using the default host execution space for first-touch initialization:

```
View<int*, HostSpace> a ("a", 100000);
```

Since there is no bijective association between execution spaces and memory spaces, Kokkos provides a way to explicitly provide both to a View as a `Device`.

```
View<int*, Device<Cuda,CudaUVMSpace> > a ("a", 100000);
View<int*, Device<OpenMP,CudaUVMSpace> > b ("b", 100000);
```

In this case `a` and `b` will live in the same memory space, but `a` will be initialized on the GPU while `b` will be initialized on the host. The `Device` type can be accessed as a views `device_type` typedef. A `Device` has only three typedef members `device_type`, `execution_space`, and `memory_space`. The `execution_space`, and `memory_space` typedefs are the same for a view and the `device_type` typedef.

It is important to understand that accessibility of a View does not depend on its execution space directly. It is only determined by its memory space. Therefore both `a` and `b` have the same access properties. They differ only in how they are initialized, as well as in where parallel kernels associated with operations such as resizing or deep copies are run.

6.4.2 Initialization

A View's entries are initialized to zero by default. Initialization happens in parallel, for first-touch allocation over the first (leftmost) dimension of the View using the execution space of the View.

You may allocate a View without initializing. For example:

```
View<int*> x (ViewAllocateWithoutInitializing (label), 100000);
```

This is useful in situations where your dominant use of the View exhibits a complicated access pattern. In this case it is best to run the most costly kernel directly after initialization to execute the first touch pattern and get optimal memory affinity.

6.4.3 Deep copy and HostMirror

Copying data from one view to another, in particular between views in different memory spaces, is called deep copy. Kokkos never performs a hidden deep copy. To do so a user has to call the `deep_copy` function. For example:

```
View<int*> a ("a", 10);
View<int*> b ("b", 10);
deep_copy (a, b); // copy contents of b into a
```

Deep copies can only be performed between views with an identical memory layout and padding. For example the following two operations are not valid:

```
View<int*[3], CudaSpace> a ("a", 10);
View<int*[3], HostSpace> b ("b", 10);
deep_copy (a, b); // This will give a compiler error

View<int*[3], LayoutLeft, CudaSpace> c ("c", 10);
View<int*[3], LayoutLeft, HostSpace> d ("d", 10);
deep_copy (c, d); // This might give a runtime error
```

The first one will not work because the default layouts of `CudaSpace` and `HostSpace` are different. The compiler will catch that since no overload of the `deep_copy` function exists to copy view from one layout to another. The second case will fail at runtime if padding settings are different for the two memory spaces. This would result in different allocation sizes and thus prevent a direct memcopy.

The reasoning for allowing only direct bitwise copies is that a deep copy between different memory spaces would otherwise require a temporary copy of the data to which a bitwise copy is performed followed by a parallel kernel to transfer the data element by element.

Kokkos provides the following way to work around those limitations. First views have a `HostMirror` typedef which is a view type with compatible layout inside the `HostSpace`. Additionally there is a `create_mirror` and `create_mirror_view` function which allocate views of the `HostMirror` type of a view. The difference between the two is that `create_mirror` will always allocate a new view, while `create_mirror_view` will only create a new view if the original one is not in `HostSpace`.

```
View<int*[3], MemorySpace> a ("a", 10);
// Allocate a view in HostSpace with the
// layout and padding of a
typename View<int*[3], MemorySpace>::HostMirror b =
    create_mirror(a);
// This is always a memcopy
deep_copy (a, b);

typename View<int*[3]>::HostMirror c =
    create_mirror_view(a);
// This is a no-op if MemorySpace is HostSpace
deep_copy (a,c)
```

6.4.4 How do I get the raw pointer?

We discourage access to a View's “raw” pointer. This circumvents reference counting. That is, the memory may be deallocated once the View's reference count goes to zero, so holding on to a raw pointer may result in invalid memory access. Furthermore, it may not even be possible to access the View's memory from a given execution space. For example, a View in the `Cuda` space points to CUDA device memory. Also using raw pointers would normally defeat the usability of polymorpic layouts and automatic padding. Nevertheless, sometimes you really need access to the pointer. For such cases, we provide the `ptr_on_device()` method. For example:

```
// Legacy function that takes a raw pointer.
extern void legacyFunction (double* x_raw, const size_t len);
```

```
// Your function that takes a View.
void myFunction (const View<double*>& x) {
  // DON'T DO THIS UNLESS YOU MUST
  double* a_raw = a.ptr_on_device ();
  const size_t N = x.dimension_0 ();
  legacyFunction (a_raw, N);
}
```

A user is in most cases also allowed to obtain a pointer to a specific element via the usual `&` operator. For example

```
// Legacy function that takes a raw pointer.
void someLibraryFunction (double* x_raw);

KOKKOS_INLINE_FUNCTION
void foo(const View<double*>& x) {
  someLibraryFunction(&x[3]);
}
```

This is only valid if a Views reference type is an `lvalue`. That property can be queried statically at compile time from the view through its `reference_is_lvalue` member.

6.5 Access traits

Another way to get optimized data accesses is to specify a memory trait. These traits are used to declare intended use of the particular view of an allocation. For example a particular kernel might use a view only for streaming writes. By declaring that intention Kokkos can insert the appropriate store intrinsics on each architecture if available. Access traits are specified through an optional template parameter which comes last in the list of parameters. Multiple traits can be combined with binary "or" operators:

```
View<double*, MemoryTraits<SomeTrait> > a;
View<const double*, MemoryTraits<SomeTrait | SomeOtherTrait> > b;
View<int*, LayoutLeft, MemoryTraits<SomeTrait | SomeOtherTrait> > c;
View<int*, MemorySpace, MemoryTraits<SomeTrait | SomeOtherTrait> > d;
View<int*, LayoutLeft, MemorySpace, MemoryTraits<SomeTrait> > e;
```

6.5.1 Atomic access

The `Atomic` access trait lets you create a View of data , such that every read or write to any entry uses an atomic update. Kokkos supports atomics for all data types independent of size. Restrictions are that you are (i) not allowed to alias data for which atomic operations are performed, and the results of non atomic accesses (including read) to data which is at the same time atomically accessed is not defined.

Performance characteristics of atomic operation depend on the data type. Some types (in particular integer types) are natively supported and might even provide asynchronous atomic operations. Others (such as 32 bit and 64 bit atomics for non integer types) are often implemented using CAS loops of integers. Everything else is implemented with a locking approach, where an atomic operations aquires a lock based on a hash of the pointer value of the data element.

Types for which atomic access are performed must support the necessary operators such as =,+=,-=,+,- etc. as well as have a number of `volatile` overloads of functions such as assign and copy constructors defined.

```
View<int*> a("a" , 100);
View<int*, MemoryTraits<Atomic> > a_atomic = a;

a_atomic(1) += 1; // This access will do an atomic addition
```

6.5.2 Random Access

The `RandomAccess` trait declares the intent to access a View irregularly (in particular non consecutively). If used for a const View in the `CudaSpace` or `CudaUVMSpace`, Kokkos will use texture fetches for accesses when executing in the `Cuda` execution space. For example:

```
const size_t N0 = ...;
View<int*> a_nonconst ("a", N0); // allocate nonconst View
// Assign to const, RandomAccess View
View<const int*, RandomAccess> a_ra = a_nonconst;
```

If the default execution space is `Cuda`, access to a `RandomAccess` View may use CUDA texture fetches. Texture fetches are not cache coherent with respect to writes, which is why you must use read-only access. The texture cache is optimized for noncontiguous access, since it has a shorter cache line than the regular cache.

While `RandomAccess` is valid for other execution spaces, currently no specific optimizations are performed. But in the future a view allocated with the `RandomAccess` attribute might for example use a larger page size, and thus reducing page faults in the memory system.

6.5.3 Standard idiom for specifying access traits

The standard idiom for View is to pass it around using as few template parameters as possible. Then, assign to a View with the desired access traits only at the "last moment," when those access traits are needed just before entering a computational kernel. This lets you template C++ classes and functions on the View type, without proliferating instantiations. Here is an example:

```
// Compute a sparse matrix-vector product, for a sparse
// matrix stored in compressed sparse row (CSR) format.
void
spmatvec (const View<double*>& y,
  const View<const size_t*>& ptr,
  const View<const int*>& ind,
  const View<const double*>& val,
  const View<const double*>& x)
{
  // Access to x has less locality than access to y.
  View<const double*, RandomAccess> x_ra = x;
  typedef View<const size_t*>::size_type size_type;

  parallel_for (y.dimension_0 (), KOKKOS_LAMBDA (const size_type i) {
      double y_i = y(i);
      for (size_t k = ptr(i); k < ptr(i+1); ++k) {
        y_i += val(k) * x_ra(ind(k));
      }
      y(i) = y_i;
    });
}
```

6.5.4 Unmanaged Views

It's always better to let Kokkos control memory allocation, but sometimes you don't have a choice. You might have to work with an application or an interface that returns a raw pointer, for example. Kokkos lets you wrap raw pointers in an *unmanaged View*. "Unmanaged" means that Kokkos does *not* do reference counting or automatic deallocation for those Views. The following example shows how to create an unmanaged View of host memory. You may do this for CUDA device memory too, or indeed for memory allocated in any memory space, by specifying the View's execution or memory space accordingly.

```
// Sometimes other code gives you a raw pointer, ...
const size_t N0 = ...;
double* x_raw = malloc (N0 * sizeof (double));
{
  // ... but you want to access it with Kokkos.
  //
  // malloc() returns host memory, so we use the
  // host memory space HostSpace.  Unmanaged
  // Views have no label, because labels work with
  // the reference counting system.
  View<double*, HostSpace, MemoryTraits<Unmanaged> >
    x_view (x_raw, N0);

  functionThatTakesKokkosView (x_view);

  // It's safest for unmanaged Views to fall out of
  // scope, before freeing their memory.
}
free (x_raw);
```

7 Parallel dispatch

You probably started reading this Guide because you wanted to learn how Kokkos can parallelize your code. This chapter will teach you different kinds of parallel operations that Kokkos can execute. We call these operations collectively *parallel dispatch*, because Kokkos "dispatches" them for execution by a particular execution space. Kokkos provides three different parallel operations:

- `parallel_for` implements a "for loop" with independent iterations.
- `parallel_reduce` implements a reduction.
- `parallel_scan` implements a prefix scan.

Kokkos gives users two options for defining the body of a parallel loop: functors and lambdas. It also lets users control how the parallel operation executes, by specifying an *execution policy*. Later chapters will cover more advanced execution policies that allow nested parallelism.

Important notes on syntax:

- Use the `KOKKOS_INLINE_FUNCTION` macro to mark a functor's methods that Kokkos will call in parallel
- Use the `KOKKOS_LAMBDA` macro to replace a lambda's capture clause when giving the lambda to Kokkos for parallel execution

7.1 Specifying the parallel loop body

7.1.1 Functors

A *functor* is one way to define the body of a parallel loop. It is a class or struct[1] with a public `operator()` instance method. That method's arguments depend on both which parallel operation you want to execute (for, reduce, or scan), and on the loop's execution policy (e.g., range or team). For an example of a functor, see the section in this chapter for each type of parallel operation. In the most common case of a `parallel_for`, it takes an integer argument which is the for loop's index. Other arguments are possible; see Chapter 8 on "hierarchical parallelism."

The `operator()` method must be const, and must be marked with the `KOKKOS_INLINE_FUNCTION` macro. If building with CUDA, this macro will mark your method as suitable for running on the CUDA device (as well as on the host). If not building with CUDA, the macro is unnecessary but harmless. Here is an example of the signature of such a method:

```
KOKKOS_INLINE_FUNCTION void operator() (...) const;
```

The entire parallel operation (for, reduce, or scan) shares the same instance of the functor. However, any variables declared inside the `operator()` method are local to that iteration of the parallel loop. Kokkos may pass the functor instance by "copy," not by pointer or reference, to the execution space that executes the code. In particular, the functor might need to be copied to a different execution space than the host. For this

[1]A "struct" in C++ is just a class, all of whose members are public by default.

reason, it is generally not valid to have any pointer or reference members in the functor. Pass in Kokkos Views by copy as well; this works by shallow copy. The functor is also passed as a const object, so it is not valid to change members of the functors. (However, it is valid for the functor to change the contents of, for example, a View or a raw array which is a member of the functor.)

7.1.2 Lambdas

The 2011 version of the C++ standard ("C++11") provides a new language construct, the *lambda*, also called "anonymous function" or "closure." Kokkos lets users supply parallel loop bodies as either functors (see above) or lambdas. Lambdas work like automatically generated functors. Just like a class, a lambda may have state. The only difference is that with a lambda, the state comes in from the environment. (The name "closure" means that the function "closes over" state from the environment.) Just like with functors, lambdas must bring in state by "value" (copy), not by reference or pointer.

By default, lambdas capture nothing (as the default capture specifier `[]` specifies). This is not likely to be useful, since parallel for generally works by its side effects. Thus, we recommend using the "capture by value" specifier `[=]` by default. You may also explicitly specify variables to capture, but they must be captured by value. We prefer that for the outermost level of parallelism (see Chapter 8), you use the `KOKKOS_LAMBDA` macro instead of the capture clause. If CUDA is disabled, this just turns into the usual capture-by-value clause `[=]`. That captures variables from the surrounding scope by value. Do NOT capture them by reference! If CUDA is enabled, this macro may have a special definition that makes the lambda work correctly with CUDA. Compare to the `KOKKOS_INLINE_FUNCTION` macro, which has a special meaning if CUDA is enabled. If you do not plan to build with CUDA, you may use `[=]` explicitly, but we find using the macro easier than remembering the capture clause syntax.

It is a violation of Kokkos semantics to capture by reference `[&]` for two reasons. First Kokkos might give the lambda to an execution space which can not access the stack of the dispatching thread. Secondly, capturing by reference allows the programmer to violate the const semantics of the lambda. For correctness and portability reasons lambdas and functors are treated as const objects inside the parallel code section. Capturing by reference allows a circumvention of that const property, and enables many more possibilities of writing non-threads-safe code.

When using lambdas for nested parallelism (see Chapter 8 for details) using capture by reference can be useful for performance reasons, but the code is only valid Kokkos code if it also works with capturing by copy.

7.1.3 Should I use a functor or a lambda?

Kokkos lets users choose whether to use a functor or a lambda. Lambdas are convenient for short loop bodies. For a much more complicated loop body, you might find it easier for testing to separate it out and name it as a functor. Lambdas by definition are "anonymous functions," meaning that they have no name. This makes it harder to test them. Furthermore, if you would like to use lambdas with CUDA, you must have a sufficiently new version of CUDA (at the time of writing Lambdas are not supported yet supported by the release 7.0). Finally, the "execution tag" feature, which lets you put together several different parallel loop bodies into a single functor, only works with functors. (See Chapter 8 for details.)

7.1.4 Specifying the execution space

If a functor has an `execution_space` public typedef, a parallel dispatch will only run the functor in that execution space. That's a good way to mark a functor as specific to an execution space. If the functor lacks this typedef, `parallel_for` will run it in the default execution space, unless you tell it otherwise (that's an advanced topic; see "execution policies"). Lambdas do not have typedefs, so they run on the default execution space, unless you tell Kokkos otherwise.

7.2 Parallel for

The most common parallel dispatch operation is a `parallel_for` call. It corresponds to the OpenMP construct `#pragma omp parallel for`. Parallel for splits the index range over the available hardware resources and executes the loop body in parallel. Each iteration is executed independently. Kokkos promises nothing about the loop order or the amount of work which actually runs concurrently. This means in particular that not all loop iterations are active at the same time. Consequently, it is not legal to use wait constructs (e.g., wait for a prior iteration to finish). Kokkos also doesn't guarantee that it will use all available parallelism. For example, it can decide to execute in serial if the loop count is very small, and it would typically be faster to run in serial instead of introducing parallelization overhead. The `RangePolicy` allows you to specify minimal chunk sizes in order to control potential concurrency for low trip count loops.

The lambda or the `operator()` method of the functor takes one argument. That argument is the parallel loop "index." The type of the index depends on the execution policy used for the `parallel_for`. It is an integer type for the implicit or explicit `RangePolicy`. The former is used if the first argument to `parallel_for` is an integer.

7.3 Parallel reduce

Kokkos' `parallel_reduce` operation implements a reduction. It is like `parallel_for`, except that each iteration produces a value, and the values are accumulated into a single value with a user-specified associative binary operation. It corresponds to the OpenMP construct `#pragma omp parallel reduction`, but with fewer restrictions on the reduction operation.

The lambda or the `operator()` method of the functor takes two arguments. The first argument is the parallel loop "index." The type of the index depends on the execution policy used for the `parallel_reduce`. If you give `parallel_reduce` an integer range as its first argument, or use `RangePolicy` explicitly, then the first argument of the lambda or `operator()` method is an integer index. Its second argument is a nonconst reference to the type of the reduction result.

7.3.1 Example using lambda

Here is an example reduction using a lambda, where the reduction result is a `double`.

```
const size_t N = ...;
View<double*> x ("x", N);
// ... fill x with some numbers ...
double sum = 0.0;
// KOKKOS_LAMBDA macro includes capture-by-value specifier [=].
```

```
parallel_reduce (N, KOKKOS_LAMBDA (const int i, double& update) {
    update += x(i); }, sum);
```

This version of `parallel_reduce` is easy to use, but it imposes some assumptions on the reduction. For example, it assumes that it is correct for threads to join their intermediate reduction results using binary `operator+`. If you want to change this, you must either implement your own reduction result type with a custom binary `operator+`, or define the reduction using a functor instead of a lambda.

7.3.2 Example using functor

The following example shows a reduction using the *max-plus semiring*, where `max(a,b)` corresponds to addition and ordinary addition corresponds to multiplication:

```
class MaxPlus {
public:
  // Kokkos reduction functors need the value_type typedef.
  // This is the type of the result of the reduction.
  typedef double value_type;

  // Just like with parallel_for functors, you may specify
  // an execution_space typedef.  If not provided, Kokkos
  // will use the default execution space by default.

  // Since we're using a functor instead of a lambda,
  // the functor's constructor must do the work of capturing
  // the Views needed for the reduction.
  MaxPlus (const View<double*>& x) : x_ (x) {}

  // This is helpful for determining the right index type,
  // especially if you expect to need a 64-bit index.
  typedef View<double*>::size_type size_type;

  KOKKOS_INLINE_FUNCTION void
  operator() (const size_type i, value_type& update) const
  { // max-plus semiring equivalent of "plus"
    if (update < x_(i)) {
      update = x_(i);
    }
  }

  // "Join" intermediate results from different threads.
  // This should normally implement the same reduction
  // operation as operator() above.  Note that both input
  // arguments MUST be declared volatile.
  KOKKOS_INLINE_FUNCTION void
  join (volatile value_type& dst,
        const volatile value_type& src) const
  { // max-plus semiring equivalent of "plus"
    if (dst < src) {
      dst = src;
    }
  }

  // Tell each thread how to initialize its reduction result.
  KOKKOS_INLINE_FUNCTION void
  init (value_type& dst) const
  { // The identity under max is -Inf.
    // Kokkos does not come with a portable way to access
    // floating-point Inf and NaN.  Trilinos does, however;
    // see Kokkos::ArithTraits in the Tpetra package.
#ifdef __CUDA_ARCH__
    return -CUDART_INF;
#else
    return strtod ("-Inf", (char**) NULL);
#endif // __CUDA_ARCH__
  }

private:
  View<double*> x_;
};
```

This example shows how to use the above functor:

```
const size_t N = ...;
View<double*> x ("x", N);
// ... fill x with some numbers ...

//  Trivial reduction in max-plus semiring is -Inf.
double result = strtod ("-Inf", (char**) NULL);
parallel_reduce (N, MaxPlus (x), result);
```

7.3.3 Example using functor with default join and init

If your functor does not supply a `join` method with the correct signature, Kokkos will supply a default `join` that uses binary `operator+`. Likewise, if your functor does not supply an `init` method with the correct signature, Kokkos will supply a default `init` that sets the reduction result to zero.

Here is an example of a reduction functor that computes the sum of squares of the entries of a View. Since it does not implement the `join` and `init` methods, Kokkos will supply defaults.

```
struct SquareSum {
  // Specify the type of the reduction value with a "value_type"
  // typedef.  In this case, the reduction value has type int.
  typedef int value_type;

  // The reduction functor's operator() looks a little different than
  // the parallel_for functor's operator().  For the reduction, we
  // pass in both the loop index i, and the intermediate reduction
  // value lsum.  The latter MUST be passed in by nonconst reference.
  // (If the reduction type is an array like int[], indicating an
  // array reduction result, then the second argument is just int[].)
  KOKKOS_INLINE_FUNCTION
  void operator () (const int i, value_type& lsum) const {
    lsum += i*i; // compute the sum of squares
  }
};

// Use the above functor to compute the sum of squares from 0 to N-1.
const size_t N = ...;
```

```
sum = 0;
// parallel_reduce needs an instance of SquareSum,
// so we invoke its constructor with ().
parallel_reduce (N, SquareSum ());
```

This example has a short enough loop body that it would be better to use a lambda:

```
// Compute the sum of squares from 0 to N-1.
int sum = 0;
parallel_reduce (N, KOKKOS_LAMBDA (const int i, int& lsum) {
  lsum += i*i;
});
```

7.3.4 Reductions with an array of results

Kokkos lets you compute reductions with an array of reduction results, as long as that array has a (run-time) constant number of entries. This currently only works with functors. Here is an example functor that computes column sums of a 2-D View.

```
struct ColumnSums {
  // In this case, the reduction result is an array of float.
  // Note the C++ notation for an array typedef.
  typedef float value_type[];

  typedef View<float**>::size_type size_type;

  // Tell Kokkos the result array's number of entries.
  // This must be a public value in the functor.
  size_type value_count;

  View<float**> X_;

  // As with the above examples, you may supply an
  // execution_space typedef.  If not supplied, Kokkos
  // will use the default execution space for this functor.

  // Be sure to set value_count in the constructor.
  ColumnSums (const View<float**>& X) :
    value_count (X.dimension_1 ()), // # columns in X
    X_ (X)
  {}

  // value_type here is already a "reference" type,
  // so we don't pass it in by reference here.
  KOKKOS_INLINE_FUNCTION void
  operator() (const size_type i, value_type sum) const {
    // You may find it helpful to put pragmas above
    // this loop to convince the compiler to vectorize it.
    // This is probably only helpful if the View type
    // has LayoutRight.
    for (size_type j = 0; j < value_count; ++j) {
      sum[j] += X_(i, j);
    }
  }

  // value_type here is already a "reference" type,
  // so we don't pass it in by reference here.
  KOKKOS_INLINE_FUNCTION void
  join (volatile value_type dst,
        volatile value_type src) const {
    for (size_type j = 0; j < value_count; ++j) {
      dst[j] += src[j];
    }
  }

  KOKKOS_INLINE_FUNCTION void init () const {
    for (size_type j = 0; j < value_count; ++j) {
      sum[j] += 0.0;
    }
  }
};
```

We show how to use this functor here:

```
  const size_t numRows = 10000;
  const size_t numCols = 10;

  View<float**> X ("X", numRows, numCols);
  // ... fill X before the following ...
  ColumnSums cs (X);
  float sums[10];
  parallel_reduce (X.dimension_0 (), cs, sums);
```

7.4 Parallel scan

Kokkos' `parallel_scan` operation implements a *prefix scan*. A prefix scan is like a reduction over a 1-D array, but it also stores all intermediate results ("partial sums"). It can use any associative binary operator. The default is `operator+`, and we call a scan with that operator a "sum scan" if we need to distinguish it from scans with other operators. The scan operation comes in two variants. An *exclusive scan* excludes (hence the name) the first entry of the array, and an *inclusive scan* includes that entry. Given an example array $(1,2,3,4,5)$, an exclusive sum scan overwrites the array with $(0,1,3,6,10)$, and an inclusive sum scan overwrites the array with $(1,3,6,10,15)$.

Many operations that "look" sequential can be parallelized with a scan. To learn more, see e.g., "Vector Models for Data-Parallel Computing," Guy Blelloch, 1990 (the book version of Prof. Blelloch's PhD dissertation).

Kokkos lets users specify a scan by either a functor or a lambda. Both look like their `parallel_reduce` equivalents, except that the `operator()` method or lambda takes three arguments: the loop index, the "update" value by nonconst reference, and a `bool`. Here is a lambda example where the intermediate results have type `float`.

```
View<float*> x = ...; // assume filled with input values
const size_t N = x.dimension_0 ();
parallel_scan (N, KOKKOS_LAMBDA (const int& i,
          float& upd, const bool& final) {
    if (final) {
      x(i) = upd; // only update array on final pass
    }
    // For exclusive scan, change the update value after
    // updating array, like we do here. For inclusive scan,
    // change the update value before updating array.
    upd += x(i);
  });
```

Kokkos may use a multiple-pass algorithm to implement scan. This means that it may call your `operator()` or lambda multiple times per loop index value. The `final` Boolean argument tells you whether Kokkos is on the final pass. You must only update the array on the final pass.

For an exclusive scan, change the `update` value after updating the array, as in the above example. For an inclusive scan, change `update` *before* updating the array. Just as with reductions, your functor may need to specify a nondefault `join` or `init` method if the defaults do not do what you want.

7.5 Function Name Tags

When writing class based applications it often is useful to make the classes themselves functors. Using that approach allows the kernels to access all other class members, both data and functions. An issue coming up in that case is the necessity for multiple parallel kernels in the same class. Kokkos supports that through function name tags. An application can use optional (unused) first arguments to differentiate multiple operators in the same class. Execution policies can take the type of that argument as an optional template parameter. The same applies to init, join and final functions.

```
class Foo {
  struct BarTag {};
  struct RabTag {};

  void compute() {
     Kokkos::parallel_for(RangePolicy<BarTag>(0,100), *this);
     Kokkos::parallel_for(RangePolicy<RabTag>(0,1000), *this);
  }

  KOKKOS_INLINE_FUNCTION
  void operator() (const BarTag&, const int& i) const {
    ...
    foobar();
    ...
  }

  KOKKOS_INLINE_FUNCTION
  void operator() (const RabTag&, const int& i) const {
    ...
    foobar();
    ...
  }

  void foobar() {
    ...
  }
};
```

This approach can also be used to template the operators by templating the tag classes which is useful to enable compile time evaluation of appropriate conditionals.

8 Hierarchical Parallelism

This chapter explains how to use Kokkos to exploit multiple levels of shared-memory parallelism. These levels include thread teams, threads within a team, and vector lanes. You may nest these levels of parallelism, and execute `parallel_for`, `parallel_reduce`, or `parallel_scan` at each level. The syntax differs only by the execution policy, which is the first argument to the `parallel_*` operation. Kokkos also exposes a "scratch pad" memory which all threads within a team may share.

8.1 Motivation

Node architectures on modern high-performance computers are characterized by ever more *hierarchical parallelism*. A level in the hierachy is determined by the hardware resources which are shared between compute units at that level. Higher levels in the hierarchy also have access to all resources in its branch at lower levels of the hierarchy. This concept is orthogonal to the concept of heterogeneity. For example, a node in a typical CPU-based cluster consists of a number of multicore CPUs. Each core supports one or more hyperthreads, and each hyperthread can execute vector instructions. This means there are 4 levels in the hierarchy of parallelism:

1. CPU sockets share access to the same memory and network resources,
2. cores within a socket typically have a shared last level cache (LLC),
3. hyperthreads on the same core have access to a shared L1 (and L2) cache and they submit instructions to the same execution units, and
4. vector units execute a shared instruction on multiple data items.

GPU-based systems also have a hierarchy of 4 levels:

1. Multiple GPUs in the same node share access to the same host memory and network resources,
2. core clusters (e.g. the SMs on an NVIDIA GPU) have a shared cache and access to the same high bandwidth memory on a single GPU,
3. threads running on the same core cluster have access to the same L1 cache and scratch memory and they are
4. grouped in so called Warps or Wave Fronts within which threads are always synchronous and can collaborate more closely for example via direct register swapping.

Kokkos provides a number of abstract levels of parallelism, which it maps to the appropriate hardware features. This mapping is not necessarily static or predefined; it may differ for each kernel. Furthermore, some mapping decisions happen at run time. This enables adaptive kernels which map work to different hardware resources depending on the work set size. While Kokkos provides defaults and suggestions, the optimal mapping can be algorithm dependent. Hierarchical parallelism is accessible through execution policies.

8.2 Thread teams

Kokkos' most basic hierarchical parallelism concept is a thread team. A *thread team* is a collection of threads which can synchronize, and which share a "scratch pad" memory (see Section8.3).

Instead of mapping a 1-D range of indices to hardware resources, Kokkos' thread teams map a 2-D index range. The first index is the *league*, the index of the team. The second index is the thread index within a team. In CUDA this is equivalent to launching a 1-D grid of 1-D blocks. The league size is arbitrary – that is, it is only limited by the integer size type – while the team size must fit in the hardware constraints. As in CUDA, only a limited number of teams are actually active at the same time, and they must run to completion before new ones are executed. Consequently it is not valid to use inter thread-team synchronization mechanisms such as waits for events initiated by other thread teams.

8.2.1 Creating a Policy instance

Kokkos exposes use of thread teams with the `Kokkos::TeamPolicy` execution policy. To use thread teams you need to create a `Kokkos::TeamPolicy` instance. It can be created inline for the parallel dispatch call. The constructors requires two arguments: a league size and a team size. As with the `Kokkos::RangePolicy` a specific execution tag and a specific execution space can be given as optional template arguments.

```
// Using default execution space and launching
// a league with league_size teams with team_size threads each
Kokkos::TeamPolicy<>
        policy( league_size, team_size );

// Using  a specific execution space to
// run a n_worksets x team_size parallelism
Kokkos::TeamPolicy<ExecutionSpace>
        policy( league_size, team_size );

// Using a specific execution space and an execution tag
Kokkos::TeamPolicy<SomeTag, ExecutionSpace>
        policy( league_size, team_size );
```

8.2.2 Basic kernels

The team policy's `member_type` provides the necessary functionality to use teams within a parallel kernel. It allows access to thread identifiers such as the league rank and size, and the team rank and size. It also provides team-synchronous actions such as team barriers, reductions and scans.

```
using Kokkos::TeamPolicy;
using Kokkos::parallel_for;

typedef TeamPolicy<ExecutionSpace>::member_type member_type;
// Create an instance of the policy
TeamPolicy<ExecutionSpace> policy (league_size, team_size);
// Launch a kernel
parallel_for (policy, KOKKOS_LAMBDA (member_type team_member) {
    // Calculate a global thread id
    int k = team_member.league_rank () * team_member.team_size () +
            team_member.team_rank ();
    // Calculate the sum of the global thread ids of this team
    int team_sum = team_member.reduce (k);
    // Atomicly add the value to a global value
    a() += team_sum;
  });
```

The name "TeamPolicy" makes it explicit that a kernel using it constitutes a parallel region with respect to the team.

8.3 Team scratch pad memory

Each Kokkos team has a "scratch pad." This is an instance of a memory space accessible only by threads in that team. Scratch pads let an algorithm load a workset into a shared space and then collaboratively work on it with all members of a team. The lifetime of data in a scratch pad is the lifetime of the team. In particular, scratch pads are recycled by all logical teams running on the same physical set of cores. During the lifetime of the team all operations allowed on global memory are allowed on the scratch memory. This includes taking addresses and performing atomic operations on elements located in scratch space. Team-level scratch pads correspond to the per-block shared memory in Cuda, or to the "local store" memory on the Cell processor.

Kokkos exposes scratch pads through a special memory space associated with the execution space: `execution_space::scratch_memory_space`. You may allocate a chunk of scratch memory through the `TeamPolicy` member type. You may request multiple allocations from scratch, up to a user-provided maximum. The maximum is provided either through a function in the functor which returns a potentially team-size dependent value, or it can be specified as an argument to the constructor of the TeamPolicy. It is not valid to provide both values at the same time. The argument to the TeamPolicy can be used to set the shared memory size when using functors. One restriction on shared memory allocations is that they can not be freed during the lifetime of the team. This avoids the complexity of a memory pool, and reduces the time it takes to obtain an allocation (which currently is a few tens of integer operations to calculate the offset).

```
template<class ExecutionSpace>
struct functor {
  typedef ExecutionSpace execution_space;
  typedef execution_space::member_type member_type;

  KOKKOS_INLINE_FUNCTION
  void operator() (member_type team_member) const {
    size_t double_size = 5*team_member.team_size()*sizeof(double);

    // Get a shared team allocation on the scratch pad
    double* team_shared_a = (double*)
      team_member.team_shmem().get_shmem(double_size);

    // Get another allocation on the scratch pad
    int* team_shared_b = (int*)
      team_member.team_shmem().get_shmem(160*sizeof(int));

    // ... use the scratch allocations ...
  }
  // Provide the shared memory capacity.
  // This function takes the team_size as an argument,
  // which allows team_size dependent allocations.
  size_t team_shmem_size (int team_size) const {
    return sizeof(double)*5*team_size +
           sizeof(int)*160;
  }
};
```

Instead of simply getting raw allocations in memory, users can also allocate Views directly in scratch memory. This is achieved by providing the shared memory handle as the first argument of the View constructor. Views also have a static member function which return their shared memory size requirements. The function expects the runtime dimensions as arguments, corresponding to View's constructor. Note that the view must be unmanaged (i.e. it must have the `Unmanaged` memory trait).

```
tyepdef Kokkos::DefaultExecutionSpace::scratch_memory_space
  ScratchSpace;
// Define a view type in ScratchSpace
typedef Kokkos::View<int*[4],ScratchSpace,Kokkos::MemoryTraits<Kokkos::Unmanaged>

// Get the size of the shared memory allocation
size_t shared_size = shared_int_2d::shmem_size(team_size);
Kokkos::parallel_for(Kokkos::TeamPolicy<>(league_size,team_size),
                     KOKKOS_LAMBDA ( member_type team_member) {
  // Get a view allocated in team shared memory.
  // The constructor takes the shared memory handle and the
  // runtime dimensions
  shared_int_2d A(team_member.team_shmem(), team_member.team_size());
  ...

});
```

8.4 Nested parallelism

Instead of writing code which explicitly uses league and team rank indices, one can use nested parallelism to implement hierarchical algorithms. Kokkos lets the user have up to three nested layers of parallelism. The team and thread levels are the first two levels. The third level is *vector* parallelism.

You may use any of the three parallel patterns – for, reduce, or scan – at each level. You may nest them and use them in conjunction with code that is aware of the league and team rank. The different layers are accessible via special execution policies: `TeamThreadLoop` and `ThreadVectorLoop`.

8.4.1 Team and vector loops

The first nested level of parallel loops splits an index range over the threads of a team. This motivates the policy name `TeamThreadRange`, which indicates that the loop is executed once by the team with the index range split over threads. The loop count is not limited to the number of threads in a team, and how the index range is mapped to threads is architecture dependent. It is not legal to nest multiple parallel loops using the `TeamThreadRange` policy. However, it is valid to have multiple parallel loops using the `TeamThreadRange` policy follow each other in sequence, in the same kernel. Note that it is not legal to make a write access to POD data outside of the closure of a nested parallel layer. This is a conscious choice to prevent difficult to debug issues related to thread private, team shared and globally shared variables. A simple way to enforce this is by using the "capture by value" clause with lambdas. With the lambda being considered as `const` inside the `TeamThreadRange` loop, the compiler will catch illegal accesses at compile time as a `const` violation.

The simplest use case is to have another `parallel_for` nested inside a kernel.

```
using Kokkos::parallel_for;
using Kokkos::TeamPolicy;
using Kokkos::TeamThreadRange;

parallel_for (TeamPolicy<> (league_size, team_size),
                    KOKKOS_LAMBDA (member_type team_member)
{
  Scalar tmp;
  parallel_for (TeamThreadRange (team_member, loop_count),
    [=] (int& i) {
      // ...
      // tmp += i; // This would be an illegal access
    });
});
```

The `parallel_reduce` construct can be used to perform optimized team-level reductions:

```
using Kokkos::parallel_reduce;
using Kokkos::TeamPolicy;
using Kokkos::TeamThreadRange;
parallel_for (TeamPolicy<> (league_size, team_size),
                 KOKKOS_LAMBDA (member_type team_member) {
    // The default reduction uses Scalar's += operator
    // to combine thread contributions.
    Scalar sum;
    parallel_reduce (TeamThreadRange (team_member, loop_count),
      [=] (int& i, Scalar& lsum) {
        // ...
        lsum += ...;
      }, sum);

    // You may provide a custom reduction as another
    // lambda together with an initialization value.
    Scalar product;
    Scalar init_value = 1;
    parallel_reduce (TeamThreadRange (team_member, loop_count),
      [=] (int& i, Scalar& lsum) {
        // ...
        lsum *= ...;
      }, product, [=] (Scalar& lsum, Scalar& update) {
        lsum *= update;
      }, init_value);
  });
```

The third pattern is `parallel_scan` which can be used to perform prefix scans.

8.4.2 Restricting execution to a single executor

As stated above, a kernel is a parallel region with respect to threads (and vector lanes) within a team. This means that global memory accesses outside of the respective nested levels potentially have to be protected against repetitive execution. A common example is the case where a team performs some calculation but only one result per team has to be written back to global memory.

Kokkos provides the `Kokkkos::single(Policy,Lambda)` function for this case. It currently accepts two policies:

- `Kokkos::PerTeam` restricts execution of the lambda's body to once per team
- `Kokkos::PerThread` restricts execution of the lambda's body to once per thread (that is, to only one vector lane in a thread)

The `single` function takes a lambda as its second argument. That lambda takes zero arguments, so its body must perform side effects in order to have an effect. It must always be correct for the lambda to capture variables by value (`[=]`, not `[&]`). Thus, if the lambda captures by reference, it must *not* modify variables that it has captured by reference.

```
using Kokkos::parallel_for;
using Kokkos::parallel_reduce;
using Kokkos::TeamThreadRange;
using Kokkos::ThreadVectorRange;
using Kokkos::PerThread;

TeamPolicy<...> policy (...);
typedef TeamPolicy<...>::member_type team_member;

parallel_for (policy, KOKKOS_LAMBDA (const team_member& thread) {
  // ...

  parallel_for (TeamThreadRange (thread, 100),
    KOKKOS_LAMBDA (const int& i) {
      double sum = 0;
      // Perform a vector reduction with a thread
      parallel_reduce (ThreadVectorRange (thread, 100),
        [=] (int i, double& lsum) {
          // ...
          lsum += ...;
        }, sum);
      // Add the result value into a team shared array.
      // Make sure it is only added once per thread.
      Kokkos::single (PerThread (), [=] () {
          shared_array(i) += sum;
        });
    });

  double sum;
  parallel_reduce (TeamThreadRange (thread, 99),
    KOKKOS_LAMBDA (int i, double& lsum) {
      // Add the result value into a team shared array.
      // Make sure it is only added once per thread.
      Kokkos::single (PerThread (thread), [=] () {
        lsum += someFunction (shared_array(i),
                              shared_array(i+1));
      });
  }, sum);

// Add the per team contribution to global memory.
Kokkos::single (PerTeam (thread), [=] () {
    global_array(thread.league_rank()) = sum;
  });
});
```

9 Subviews

After reading this chapter, you should understand the following:

- A *slice* of a multidimensional array behaves as an array, and is a view of a structured subset of that array
- A *subview* is a slice of an existing Kokkos View
- A subview has the same reference count as its parent View
- Use C++11 type inference (`auto`) to let Kokkos pick the subview's type

9.1 A subview is a slice of a View

In Kokkos, a *subview* is a slice of a View. A *slice* of a multidimensional array behaves as an array, and is a view of a structured subset of the original array. "Behaves as an array" means that the slice has the same syntax as an array should; one can access its entries using array indexing notation. "View" means that the slice and the original array point to the same data. That is, the slice sees changes to the original array, and vice versa. "Structured subset" means a cross product of indices along each dimension, as for example a plane or face of a cube. If the original array has dimensions $(N_0, N_1, \dots, N_{k-1})$, then a slice views all entries whose indices are $(\alpha_0, \alpha_1, \dots, \alpha_{k-1})$, where α_j is an ordered subset of $\{0, 1, \dots, N_j - 1\}$.

Array slices are handy for encapsulation. A slice looks and acts like an array, so you can pass it into functions that expect an array. For example, you can write a function for processing boundaries (as slices) of a structured grid, without needing to tell that function properties of the entire grid.

Programming languages like Fortran 90, Matlab, and Python have a special "colon" notation for representing slices. For example, if `A` is an $M \times N$ array, then

- `A(:, :)` represents the whole array,
- `A(:, 3)` represents the fourth column (if the language has zero-based indices, or the third column, if the language has one-based indices),
- `A(4, :)` represents the fifth row,
- `A(2:4, 3:7)` represent the sub-array of rows 3-4 and columns 4–7 (different languages differ on whether the ranges are inclusive or exclusive of the last index – Kokkos, like Python, is exclusive), and
- `A(3, 4)` represents a "zero-dimensional" slice which views the entry in the fourth row and fifth column of the matrix.

These languages may have more elaborate notation for expressing sets of indices other than contiguous ranges. These may include "strided" subsets of indices, like `3:2:9` = $\{3, 5, 7, 9\}$, or even arbitrary sets of indices.

9.2 How to take a subview

To take a subview of a View, use the `Kokkos::subview` function. This function is overloaded for all different kinds of Views and index ranges. For example, the following code is equivalent to the above example `A(2:4, 3:7)`:

```
const size_t N0 = ...;
const size_t N1 = ...;
View<double**> A ("A", N0, N1);

auto A_sub = subview (A, make_pair (2, 4), make_pair (3, 7));
```

In the above example and those that follow in this chapter, we assume that `Kokkos::View`, `Kokkos::subview`, `Kokkos::ALL`, `std::make_pair`, and `std::pair` have been imported into the working C++ namespace.

The Kokkos equivalent of a contiguous index range `3:7` is `pair<size_t, size_t>(3, 7)`. The Kokkos equivalent of `:` (a colon by itself; the whole index range for that dimension) is `ALL()` (an instance of the `ALL` class, which Kokkos uses only for this purpose). Kokkos does not currently have equivalents of the strided or arbitrary index sets.

A subview has the same reference count as its parent View, so the parent View won't be deallocated before all subviews go away. Every subview is also a View. This means that you may take a subview of a subview.

9.2.1 C++11 type deduction

Note the use of the C++11 keyword `auto` in the above example. A subview may have a different type than its parent View. For instance, if `A` has `LayoutRight`, `A_sub` has `LayoutStride`. A subview of an entire row of `A` has `LayoutRight` as well, but a subview of an entire column of `A` has `LayoutStride`. If you assign the result of the `subview` function to the wrong type, Kokkos will emit a compile-time error. The best way to avoid this is to use the C++11 `auto` keyword to let the compiler deduce the correct type for you.

9.2.2 Dimension of a subview

Suppose that a View has k dimensions. Then, when calling `subview` on that View, you must pass in k arguments. Every argument that is a single index – that is, not a pair or `ALL()` – reduces the dimension of the resulting View by 1.

9.2.3 Degenerate Views

Given a View with k dimensions, we call that View *degenerate* if any of those dimensions is zero. Degenerate Views are useful for keeping code simple, by avoiding special cases. For example, consider a MPI (Message-Passing Interface) distributed-memory parallel code that uses Kokkos to represent local (per-process) data structures. Suppose that the code distributes a dense matrix (2-D array) in block row fashion over the MPI processes in a communicator. It could be that some processes own zero rows of the matrix. This may not be efficient, since those processes do no work yet participate in collectives, but it might be possible. In this case, allowing Views with zero rows would reduce the number of special cases in the code.

10 Interoperability and Legacy Codes

One goal of Kokkos is to support incremental adoption in legacy applications. This facilitates a step by step conversion allowing for continuous testing of functionality (and in certain bounds) of performance. One feature of this is full interoperability with the underlying backend programming models. This also allows for target specific optimizations written directly in the backend model in order to achieve maximal performance.

After reading this chapter, you should understand the following:

- Restriction on interoperability with raw OpenMP and Cuda.
- How to handle external data structures.
- How to incrementally convert legacy data structures.
- How to call non-Kokkos third party libraries safely.

In all code examples in this chapter, we assume that all classes in the `Kokkos` namespace have been imported into the working namespace.

10.1 OpenMP, Pthreads and CUDA interoperability

Since the implementation of Kokkos is achieved with a C++ library it provides full interoperability with the underlying backend programming models. In particular it allows for mixing of OpenMP, CUDA and Kokkos code in the same compilation unit. This is true for both the parallel execution layers of Kokkos as for the data layer.

It is important to recognize that this does not lift certain restrictions. For example it is not valid to allocate a view inside of an OpenMP parallel region, the same way as it is not valid to allocate a View inside a `parallel_for` kernel. Indeed there are things which are slightly more cumbersome when mixing the models. Assigning one view to another inside a Cuda kernel or an OpenMP parallel region is only possible if the destination view is unmanaged. During dispatch of kernels with `parallel_for` all Views referenced in the functor or lambda are automatically switched into unmanaged mode. This would not happen when simply entering an OpenMP parallel region.

10.1.1 Cuda interoperabiltiy

The most important thing to know for Cuda interoperability is that the provided macro `KOKKOS_INLINE_FUNCTION` evaluates to `__host__ __device__ inline`. This means that calling a pure `__device__` function (for example Cuda intrinsics or device functions of libraries) must be protected with the `__CUDA_ARCH__` pragma.

```
__device__ SomeFunction(double* x) {
  ...
}

struct Functor {
  typedef Cuda execution_space;
  View<double*,Cuda> a;
  KOKKOS_INLINE_FUNCTION
  void operator(const int& i) {
    #ifdef __CUDA_ARCH__
    int block = blockIdx.x;
    int thread = threadIdx.x;
    if (thread == 0)
      SomeFunction(&a(block));
    #endif
  }
}
```

The RangePolicy start a 1D grid of 1D thread blocks so that the index i is calculated as blockIdx.x * blockDim.x + threadIdx.x. For the TeamPolicy the number of teams is the grid dimension, while the number of threads per team is mapped to the Y-dimension of the Cuda thread-block. The optional vector length is mapped to the X-dimension. For example TeamPolicy<Cuda>(100,12,16) would start a 1D grid of size 100 with block-dimensions (16,12,1) while TeamPolicy<Cuda>(100,96) would result in a grid size of 100 with block-dimensions of (1,96,1). The restrictions on the vector length (power of two and smaller than 32 for the Cuda execution space) guarantee that vector loops are performed by threads which are part of a single warp.

10.1.2 OpenMP

One restriction on OpenMP interoperability is that it is not valid to increase the number of threads via omp_set_num_threads() after initializing Kokkos. This restriction is caused due to Kokkos allocating internal per thread book keeping data structures. It is however valid to ask for the thread ID inside a Kokkos parallel kernel compiled for the OpenMP execution space. It is also valid to use OpenMP constructs such as OpenMP atomics inside a parallel kernel or functions called by it. However it is undefined what happens when mixing OpenMP and Kokkos atomics, since those will not necessarily map to the the same underlying mechanism.

10.2 Legacy data structures

There are two principal mechanism to facilitate interoperability with legacy data structures: (i) Kokkos allocates data and raw pointers are extracted to create legacy data structures and (ii) unmanaged views can be used to view externally allocated data. In both cases it is mandatory to fix the Layout of the Kokkos view to the actual layout used in the legacy data structure. Note that the user is responsible for insuring proper access capabilities. For example a pointer obtained from a view in the CudaSpace may only be accessed from Cuda kernels, and a View constructed from memory acquired through a call to new will typically only be accessible from Execution spaces which can access the HostSpace.

10.3 Raw allocations through Kokkos

A simple way to add support for multiple memory spaces to a legacy app is to use kokkos_malloc, kokkos_free and kokkos_realloc. The functions are templated on the memory space and thus allow targeted placement of data structures:

```
// Allocate an array of 100 doubles in the default memory space
double* a = (double*) kokkos_malloc<>(100*sizeof(double));
```

```
// Allocate an array of 150 int* in the Cuda UVM memory space
// This allocation is accessible from the host
int** 2d_array = (int**) kokkos_malloc<CudaUVMSpace>
                           (150*sizeof(int*));

// Fill the pointer array with pointers to data in the Cuda Space
// Since it is not the UVM space you can access 2d_array[i][j]
// only inside a Cuda Kernel
for(int i=0;i<150;i++)
  2d_array[i] = (int*) kokkos_malloc<CudaSpace>(200*sizeof(int));
```

A common usage scenario of this capability is to allocate all memory in the CudaUVMSpace when compiling for GPUs. This allows all allocations to be accessible from legacy code sections as well as from parallel kernels written with Kokkos.

10.3.1 External memory management

When memory is managed externally, for example because Kokkos is used in a library which is given pointers to data allocations as input, it can be necessary or convenient to wrap the data into Kokkos views. If the library anyway receives the data to create a copy it is straight forward to allocate the internal data structure as a view and copy the data in a parallel kernel element by element. Note that you might need to first copy into a host view before copying to the actual destination memory space:

```
template<class ExecutionSpace>
void MyKokkosFunction(double* a, const double** b, int n, int m) {
  // Define the host execution space and the view types
  typedef HostSpace::execution_space host_space;
  typedef View<double*,ExecutionSpace> t_1d_device_view;
  typedef View<double**,ExecutionSpace> t_2d_device_view;

  // Allocate the view in the memory space of ExecutionSpace
  t_1d_device_view d_a("a",n);
  // Create a host copy of that view
  typename t_1d_device_view::HostMirror h_a = create_mirror_view(a);
  // Copy the data from the external allocation into the host view
  parallel_for(RangePolicy<host_space>(0,n),
    KOKKOS_LAMBDA (const int& i) {
    h_a(i) = a[i];
  });
  // Copy the data from the host view to the device view
  deep_copy(d_a,h_a);

  // Allocate a 2D view in the memory space of ExecutionSpace
  t_2d_device_view d_b("b",n,m);
  // Create a host copy of that view
  typename t_2d_device_view::HostMirror h_b = create_mirror_view(b);

  // Get the member_type of the team policy
  typedef TeamPolicy<host_space>::member_type t_team;
  // Run a 2D copy kernel using a TeamPolicy
  parallel_for(TeamPolicy<host_space>(n,m),
    KOKKOS_LAMBDA (const t_team& t) {
    const int i = t.team_rank();
    parallel_for(TeamThreadRange(t,m), [&] (const int& j) {
      h_b(i,j) = b[i][j];
    });
  });
  // Copy the data from the host to the device
  deep_copy(d_b,h_b);
}
```

Alternatively one can create a view which directly references the external allocation. If that data is a multi dimensional view it is important to specify the Layout explicitly. Furthermore all data must be part of the same allocation.

```
void MyKokkosFunction(int* a, const double* b, int n, int m) {
  // Define the host execution space and the view types
  typedef View<int*, DefaultHostExecutionSpace, Unmanaged> t_1d_view;
  typedef View<double**[3],LayoutRight, DefaultHostExecutionSpace,
               Unmanaged> t_3d_view;

  // Create a 1D view of the external allocation
  t_1d_view d_a(a,n);

  // Create a 3D view of the second external allocation
  // This assumes that the data had a row major layout
  // (i.e. the third index is stride 1)
  t_3d_view d_b(b,n,m);
}
```

10.3.2 Views as the fundamental data owning structure

An other option is to let Kokkos handle the basic allocations using Views, and then constructing the legacy data structures around them. Again it is important to fix the Layout of the Views to whatever the layout of the legacy data was.

```
// Allocate a 2D view with row major layout
View<double**,LayoutRight,HostSpace> a("A",n,m);

// Allocate an array of pioneers
double** a_old = new double*[n];

// Fill the array with pointers to the rows of a
for(int i=0; i<n; i++)
  a_old[i] = &a(i,0);
```

10.3.3 std::vector

One of the most common data objects in C++ codes is `std::vector`. Its semantics are unfortunately not compatible with Kokkos requirements and it is thus not well supported. A major problem is that functors and lambdas are passed as const objects to the parallel execution. This design choice was made to (i) prevent a common cause of race conditions and (ii) allow the underlying implementation more flexibility in how to share the functor and where to put it. In particular this leaves the choice open for the implementation to give each thread an individual copy of the functor or place it in read only cache structures.

The semantics of `std::vector` would in this case prevent a kernel from modifying its entries since a const `std::vector` is read only. Furthermore creating multiple copies of the functor would in deed replicate the vector data, since it has copy semantics.

Other issues with `std::vector` are its unrestricted support for resize as well as push functionality. In a threaded environment support for those capabilities would bring massive performance penalties. In particular access to the `std::vector` would require locks in order to prevent one thread to deallocate the view while another accesses its content. And last but not least `std::vector` is not supported on GPUs and thus would prevent portability.

Kokkos provides a drop in replacement for `std::vector` with `Kokkos::vector`. Outside of parallel kernels its semantics are mostly the same as that of `std::vector`, for example assignments performs deep copies and resize and push functionality are provided. One important difference is that it is valid to assign values to the elements of a const vector.

Inside of parallel sections `Kokkos::vector` switches to view semantics. That means in particular that assignments are shallow copies. Certain functions will also throw runtime errors when called inside a parallel kernel. This includes resize and push.

```
// Create a vector of 1000 double elements
Kokkos::vector<double> v(1000);
// Create another vector as a copy of v;
// This allocates another 1000 doubles
Kokkos::vector<double> x = v;

parallel_for(1000, KOKKOS_LAMBDA (const int& i) {
   // Create a view of x; m and x will reference the same data.
   Kokkos::vector<double> m = x;
   x[i] = 2*i+1;
   v[i] = m[i] - 1;
});

// Now x contains the first 1000 uneven numbers
// v contains the first 1000 even numbers
```

10.4 Calling non-Kokkos libraries

There are no restrictions on calling non-Kokkos libraries outside of parallel kernels. However due to the polymorphic layouts of Kokkos views it is often required to test layouts for compatibility with third party libraries. The usual Blas interface for example, expects matrixes to be laid out in column major format (i.e. LayoutLeft in Kokkos). Furthermore it is necessary to test that the library can access the memory space of the view.

```
template<class Scalar, class Device>
Scalar dot(View<const Scalar* , Device> a,
           View<const Scalar*, Device> b) {
// Check for Cuda memory and call cublas if true
#ifdef KOKKOS_HAVE_CUDA
  if(std::is_same<typename Device::memory_space,
                           CudaSpace>::value ||
     std::is_same<typename Device::memory_space,
                           CudaUVMSpace>::value) {
```

```
    return call_cublas_dot(a.ptr_on_device(), b.ptr_on_device(),
                           a.dimension_0() );
  }
#endif

// Call CBlas on the host otherwise
  if(std::is_same<typename Device::memory_space,HostSpace>::value) {
    return call_cblas_dot(a.ptr_on_device(), b.ptr_on_device(),
                          a.dimension_0() );
  }
}
```