LA-UR-23-26959

Approved for public release; distribution is unlimited.

Title: An Introduction to Kokkos

Author(s): Maginot, Peter Gregory

Intended for: Parallel school lecture

Issued: 2023-06-27

Los Alamos NATIONAL LABORATORY

Los Alamos
NATIONAL LABORATORY

An Introduction to Kokkos

Pete Maginot
Eulerian Applications Project- Deputy for Physics

XCP Parallel Workshop Lecture
June 30, 2023

Overview

- Who am I?
- What is a performance abstraction layer?
  - Why would you do this to yourself?
- Major Kokkos Ideas
- Some gotchas / tricks

Overview

- **Who am I?**
- What is a performance abstraction layer?
  - Why would you do this to yourself?
- Major Kokkos Ideas
- Some gotchas / tricks

I am old

- From St. Louis, MO
- Ten years at Texas A&M
  - Did graduate more than once
  - BS, MS, and PhD in Nuclear Engineering
    - Dissertation topic: high-order methods for S_N grey radiative transfer equations
  - DOE CSGF Fellow 2010-2014
- Strongly suggest trying out different locales and jobs
  - Environmental Health Physics Tech (St. Louis, MO; 2007)
  - DNFSB (Washington D.C.; 2008)
  - ORNL (Oak Ridge, TN 2009)
  - LANL (TA-3-390; 2010)
  - KAPL (Schenectady, NY; 2012)

I've spent 8 years in the Weapons Complex

- 3.5 years at LLNL
  - Postdoc, WSC Deterministic transport project
    - ISCB spatial discretization profiling in Kripke proxy-app
    - Lumping for HO Mixed Finite Element Transport
    - Documentation of UCB in Teton
  - Staff-member, WSC Deterministic transport project
    - Co-PI LDRD on HO Transport on HO Grids
    - Librarization of Teton deterministic x-ray transport code for multiple multiphysics codes
- 4.5 years at LANL
  - Staff-member, XCP-2, supporting Eulerian Application Project (xRAGE, Cassio, …)
    - Edge infrastructure, geometry setup, timestep controls, grey diffusion porting, user support
    - TITANS (3+ year program on weapons physics)
    - Weapon outputs / simulations of a novel design class

Why work at LANL?

- Meaning in the work
  - Support national security of the United States
  - Work is used and applied, work is not designing “paper reactors”
- Challenging work
  - Never ending supply of new things to learn
  - Opportunities to become “the” expert
- National Lab atmosphere
  - Everyone is self-motivated
    - Though still a relaxed atmosphere (Dr. not needed if everyone in the room is Dr.)
- Location
  - 15 minutes to skiing, no S.A.D. winters
  - 28 years of humidity more than enough for a lifetime

Overview

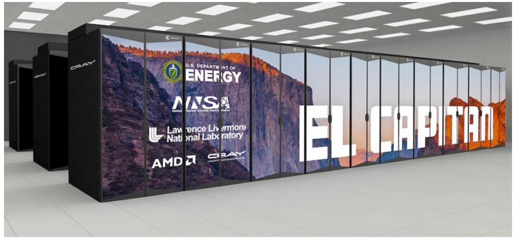
- Who am I?
- What is a performance abstraction layer?
  - **Why would you do this to yourself?**
- Major Kokkos Ideas
- Some gotchas / tricks

ASC codes *MUST* run on a lot of platforms!

- xRAGE nightly regression tests currently on:
  - Snow (CTS), Rocinante (SPR), RZAnsel (Power9), Trinitite (Haswell), Trinitite (KNL)
  - Fire/Ice/Cyclone (CTS), Trinity (Haswell), Trinity (KNL), Sierra (Power9)
  - RZVernal (EAS-3), Tioga (EAS-3), and Venado (G+H) in progress / coming soon
- xRAGE is a *BIG* code with a relatively small # of developers
  - O(500K) SLOC
  - Budget of < 14 FTE / year for all activities
    - Deployment, user support, new features, code maintenance
- If a platform requires specialized coding for performance it is intractable for EAP to implement this given the number of machines we must support
  - Enter the **Performance Abstraction Layer**

Do you really need to target all those machines?

- YES!
  - Users require we run on the bread and butter machines (Snow, Fire/Ice/Cyclone)
  - Users ***should*** want us to target the others
- If FLOPS dictate how awesome your simulation is:
  - Fire/Ice/Cyclone: 1.3 PFLOPS
    - [**P**eta(10^{15}) **FL**oating (double precision) point **O**perations **P**er **S**econd] each
  - Trinity: 42 PFLOPS
    - KNL partition: 30 PFLOPS
  - Crossroads: 44 PFLOPS
  - Sierra: 125 PFLOPS
    - 120 PFLOPS on GPUs
  - El Capitan: >2000 PFLOPS (predicted)
    - EAS-3: 5.76 PFLOPS GPU / 5.824 PFLOPS Total



Are FLOPS everything? Is TOP500 *the* list?

- Probably not / this is up for debate
- HPCG
  - Sparse linear algebra as compared to HPL[inpack]'s dense linear algebra
    - Can be argued that this is much closer to our codes' behavior than LINPACK
  - Sierra 1.8 PFLOPS on HPCG
  - Trinity 0.5 PFLOPS on HPCG
- Maybe we're concerned with power consumption
  - Enter Green500:
    - Sierra: 12.723 [GFLOPS/Watt]
    - Trinity: 2.66 [GFLOPS/Watt]

Regardless of metric, running [well] on GPU machines is needed to take advantage of ASC resources!!

Can’t you just run the codes on the GPUs?

- No
- GPU processors are very different than CPU processors
  - Lots and lots of “dumb” processors vs. a few very talented multi-taskers
  - Slow clock speed vs. faster clock speed
  - Small vs. large cache
- Distinct memory spaces
  - This is becoming less true, but is important for many current systems
- GPUs are typically programmed in vendor specific code
  - Allows for control of advanced hardware features distinct from CPUs

Overview

- Who am I?
- **What is a performance abstraction layer?**
  - Why would you do this to yourself?
- Major Kokkos Ideas
- Some gotchas / tricks

What is a performance abstraction layer?

- A set of C++ widgets that ideally lets physics/code be performant on multiple platforms with a single set of source code
  - Likely benefits from a small companion header file that modifies platform specific template parameters
  - **Might** ease maintenance burden
    - Abstraction layers are non-trivial
- Focus on on-node performance
- Requires comfort with template parameters / template programming
- Made possible by lambdas functionality of C++ 11

What is a performance abstraction layer?

- Is there a performance cost? Maybe
  - Getting access to all of the features may or may not be necessary
  - May or may not negate the generality of the abstraction layer
- Is there a code debt cost? Yep
  - A third-party library will now be deeeeeply integrated into your code base
- Abstraction layers not required to utilize GPUs / advanced platforms
  - OpenMP4.5
  - Native hardware languages
  - Vectorization intrinsics
- Are performance abstraction layers used and enjoyed by all ASC projects?
  - No, not all need it / benefit from it
  - Challenges to adoption: FORTRAN, limited # of “hotspots”, heavily OO code
    - Some of these things make GPU programming hard
    - Some of these things make vectorization easy

Are there different performance abstraction layers?

- Two concepts require abstraction
  - Memory locations / data movement
  - Loop abstractions
- Two mainline DOE products / projects

1. SNL (Sandia) solution
   - Kokkos: Addresses both in one project
2. LLNL (Livermore) solution
   - RAJA: Loop abstraction
   - Chai: Data movement

Overview

- Who am I?
- What is a performance abstraction layer?
  - Why would you do this to yourself?
- **Major Kokkos Ideas**
- Some gotchas / tricks

Two central tenants of Kokkos: Views and parallel_<for>

- *Views*: Data management
  - Multidimensional arrays, FORTRAN like (i,j,k) indexing / access operator
  - Defined by:
    - data type (double, int, …)
    - storage location (*CudaSpace*, *HostSpace*, *HipSpace*, *DefaultMemorySpace*, …)
    - rank (1-D to 7-D)
    - data layout (*LayoutLeft* , *LayoutRight* , *LayoutStride*)
      - Which index (in a multi-D View) has data closer in memory from index (i,j)? (i+1,j) or (i,j+1)
      - Strided memory access arises in array slices
- *parallel_for* / *parallel_reduce* / *parallel_scan* : loop abstraction
  - Defined by execution policy
    - Where to do the computation? (*Cuda , Serial, OpenMP, …)*
    - How to iterate? (*RangePolicy*, *MDRangePolicy*, *TeamVectorMDRange*, …)

“Standard” *View* constructor

```
Kokkos::View< DataType **[3], DataLayout, DataLocation>
     my_vector(“my_vector”, nX, nY);
```

- Create a new View that is nX * nY * 3 elements long of type DataType
  - Fixed length dimensions must be last (and enclosed in [ ])
- Allocated in DataLocation MemorySpace
- Memset (initialized) to 0
- DataLocation::DefaultExecutionSpace blocking
  - This View will allocated and ready before anything else is executed
- “my_vector” is for debugging purposes
  - Kokkos may list the “name” of a vector if a runtime issue arises

More Advanced *View* constructor

```
Kokkos::View< DataType **[3], DataLayout,
              DataLocation, Kokkos::MemoryTraits<Kokkos::Atomic>>
     my_vector( Kokkos::view_alloc("my_vector", Kokkos::WithoutInitializing,
                eexecStream, DataLocation() ), nX, nY);
```

- Similar to previous, BUT:
  - `Kokkos::WithoutInitializing:` No memset
  - `Kokkos::MemoryTraits<Kokkos::Atomic>>:` Atomic access enforced when used
  - *execStream :* will be allocated in spaceStream, possibly asynchronously
    - Must call spaceStream.fence() to ensure the *View* is available
    - Non-blocking call (possibly)!

Views from Data Outside of Kokkos' Management

```
Kokkos::View< DataType ***, DataLayout,
              DataLocation, Kokkos::MemoryTraits<Kokkos::Unmanaged>>
    my_vector( data_ptr, nX, nY, 3);
```

- Developers may hand Kokkos a pointer to data allocated outside of Kokkos
- Developer is asserting Location, Layout, rank, and length of data
  - LayoutLeft: FORTRAN allocator
  - LayoutRight: C/C++ malloc
  - LayoutStride: FORTRAN array slice
- Non-allocating

How to create “mirrors” in different memory spaces

- Mirrors- related Views that can transfer data between one another via deep_copy (typically different spaces)

```
auto newView = create_mirror([newViewSpace()], srcView);
```

- Creates a new View in HostSpace or newViewSpace (if given)
  - Always a deep copy!

```
View<newDataType, newLayout, newStorage> newView =
                      create_mirror_and_copy([newViewSpace()], srcView);
```

- Create mirror View and copy data (if necessary)
  - Can change from non-const to const data
  - Can change layout and location in a single call
    - Copying back may then require an intermediate step
  - Shallow copy if possible
- All of the above can take view_alloc() to allow synchronizing, control initialization state, etc.

Kokkos Allows for Array Slicing via *subview*

- Highly recommended to use *auto* to determine type of the result

```
int nx,ny,nz;
View<double***, LayoutLeft, CudaSpace> bigThing(“bigThing”, nx,ny,nz);
auto smallPiece = Kokkos::subview( bigThing,
                                   1, Kokkos::make_pair(2,4), Kokkos::ALL);
parallel_for(“example”,
             MDRangePolicy<execSpace,Rank<3>>({0,0,0},{1,2,nz}),
              KOKKOS_LAMBDA(size_t i, size_t j, size_t k){
                 smallPiece(i,j,k) += 0.25;
             });
```

- Above smallPiece, take the data in row 1; columns 2,3; and all of the next dimension
- Pair indices are read as [closed,open) interval of columns

Generic Anatomy of Kokkos Parallel Constructs

```
parallel_blah("name" , ExecutionPolicy,
                 KOKKOS_LAMBDA( size_t index){
                    … });
```

- ExecutionPolicy can have varying levels of verbosity
  - *nEL*; implies 1D RangePolicy(0,nEl) on DefaultExecutionSpace
  - *nStart,nEnd* ;implies 1D RangePolicy(nStart, nEnd) on DefaultExecutionSpace
  - RangePolicy<EXEC_SPACE>(0, nEnd); 1D RangePolicy(0, nEnd) on EXEC_SPACE
- KOKKOS_LAMBDA
  - A macro that handles the lambda capture syntax / decorating for device as necessary
  - Not strictly required, but is convenient to include
- *parallel_blah* are generally non-blocking
  - Work will be started, but other kernels can also be launched.
  - Requires use of fence() and/or streams for to respect data dependencies

Some ExecutionPolicy Examples

- 1-D iteration : RangePolicy

```
parallel_for("name", RangePolicy<EXEC_SPACE>(a,b),
                    KOKKOS_LAMBDA(const size_t index){});
```

- Multi-Dimension iteration: MDRangePolicy

```
parallel_for("name",
        MDRangePolicy<EXEC_SPACE,Kokkos::Rank<3>>({0,0,0},{a,b,c}),
        KOKKOS_LAMBDA(const size_t i, const size_t j, const size_t k)
        {});
```

- Above do not pass in execStream
  - Use of execStream allows for multiple kernels working at once

```
parallel_for("name", RangePolicy<EXEC_SPACE>(execStream1, a,b),
               KOKKOS_LAMBDA(const size_t index){});
parallel_for("name", RangePolicy<EXEC_SPACE>(execStream2, a,b),
               KOKKOS_LAMBDA(const size_t index){});
```

parallel_for()

```
parallel_for(“name” , ExecutionPolicy,
                KOKKOS_LAMBDA( size_t index){
                    f(index) = ….
});
```

- Simplest to think of- “Do a thing for all elements of the ExecutionPolicy”

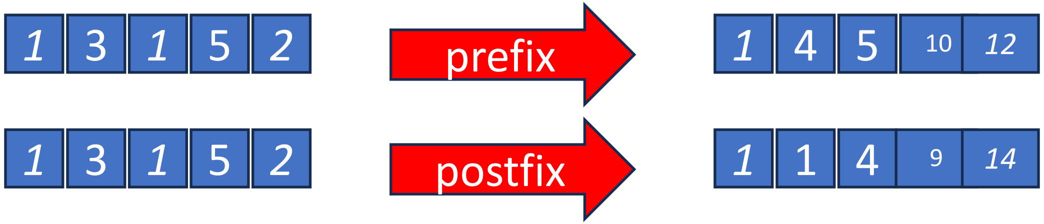
parallel_reduce()

```
Data globalVal;
parallel_reduce("name" , ExecutionPolicy,
              KOKKOS_LAMBDA( size_t i, Data& localVal){
                  localVal = operation(localVal, val(i));
}, Kokkos::Operation<Data>(globalVal) );
```

- Kokkos has several built in Reduction operations
  - *Max, Min, MaxLoc, MinLoc, Sum, …*
- Can also create your own, or do a defined reduction on a user-defined type

parallel_scan()

- prefix or postfix operations, e.g. sum




```
parallel_scan(“prefixSum” , ExecutionPolicy,
    KOKKOS_LAMBDA( size_t i, Data& localVal, bool is_final){
    const int val_before_sum = x(l);
    sum_of_need_to_update += val_before_sum;
    if(is_final){
        prefix_sum(l) = sum_of_need_to_update;
    }});
```

How to achieve program flow control given asynchronous resources

- Unless explicitly stated, it should be assumed that Kokkos idioms are non-blocking!
  - This means that another statement may start executing before the previous action is completed
  - There are rules to this, but be advised that creativity or over-reliance on what you think “should” be happening can lead to horrific debug challenges
- Program flow is controlled by *fence()*-ing in Kokkos
  - *`Kokkos::fence();`* Blocking until all work in all execution spaces completes before proceeding
  - *`ExecutionSpace::fence();`* Blocking until all work in this execution space completes
  - *`streams[i].fence();`* Blocking until all work in this execution *stream* completes
    - Even if *`streams[i]`* is of type ExecutionSpace, *`ExecutionSpace::fence()`* will NOT fence stream[i]

Tag Views, parallel regions, and add profiling regions

```
Kokkos::Profiling::pushRegion("descriptive Name");
….
Kokkos::Profiling::popRegion();
```

- Above are Kokkos specific annotations
- Need to be added in pairs (push/pop)
- No compiler warning if there are un-matched push/pop
  - Some kokkos-tools will segfault if there are mismatches
  - new Kokkos::Profiling::ScopedRegion will give compiler warnings
    - But must upgrade to Kokkos 4.1 (or later)

Profiling and debugging tools

- Kokkos has a companion (and separate git repo) of plugin tools, *kokkos-tools*
- Connector tools available
  - Vendor specific profiler adaptors (Intel VTune, NVTX, ROCTX…)
  - Memory Usage (allocations by memory type, high-water marks, …)
  - Timers (hierarchical time, simple kernel timers, …)
- To use:
  - 1) Download kokkos-tools, 2) ‘make’ within the subfolder of the tool you want to us 3) Set the environment variable KOKKOS_TOOLS_LIBS to the path of the shared library file that was created
- Note:
  - Makefiles may need hand editing to account for how your specific environment looks
    - Ex:. CUDA_ROOT vs. CUDA_PATH
  - CMake builds exist, but generally work less well than Makefile path

How to overlap work, data transfers, etc.

```
auto streams = Kokkos::Experimental::partition_space(
                      EXEC_SPACE(), weights);
auto streams = Kokkos::Experimental::partition_space(
                      EXEC_SPACE(), 2,1,2…);
```

- Unless told otherwise, all Kokkos operations are launched on the Default stream/queue of the ExecutionSpace that work was set to work on
  - Operations complete in order
- It is desirable to overlap multiple things working at once to maximize hardware utilization
  - Ex: moving data onto the GPU while doing work on the CPU and/or GPU
- Kokkos permits this through the *partition_space* idea

partition_space allows for asynchronous work

- *weights* are used by OpenMP backend to allocate relative resource levels
- With Cuda and other GPU backends, numeric value of weights is ignored
  - One new stream for each *weights* entry
- Each element of *partition_space* is distinct
  - only respects a fence on itself or a global *Kokkos::fence()*

Overview

- Who am I?
- What is a performance abstraction layer?
  - Why would you do this to yourself?
- Major Kokkos Ideas
- **Some gotchas / tricks**

Be verbose, incrementally increase complexity only after verifying a given implementation is working

- Be explicit in where memory is located and where you think work should occur
  - Kokkos will default a lot of things for you
  - Leads to lower initial entry barrier, but steeper rise to perfection
- Do not attempt to maximize throughput from the start
  - fence() if you’re not sure
  - Overlapping work can lead to a bookkeeping nightmare
- Ensure you can default everything to Serial and HostSpace via changing / redefining one or two template parameters
  - Kokkos allows for not only logic error in your algorithm, but also errors in memory and execution space location management!

Seek information from a variety of sources

- Great to have a friend or colleague that is ahead of the Kokkos game for you
  - Thanks danl@lanl
- Kokkos documentation
  - Both the current and deprecated have useful information
  - Documentation is not perfect, best consumed with source code access
- Kokkos Slack channel
  - The Kokkos equivalent to crestone_support@lanl.gov
- Kokkos tutorials / workshops (online)
- Do it to learn it
  - Kokkos knowledge really only imprinted after exercising it

Caveats / Things I skipped

- SIMD operations in Kokkos
  - Befikir and Yasuki should have some good knowledge soon
- Hierarchical parallelism
  - Haven't used it yet in xRAGE, but I'd like to learn
- Profiling
  - Required to effectively direct incremental improvement in your code
- The compiler is right
  - I tried to reproduce things correctly in here, but I am prone to mistakes

Questions?