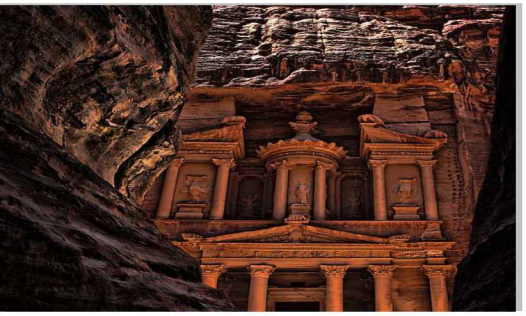
SAND2019-13073PE



Purging reliance on UVM from Tpetra & downstream solvers

Mark Hoemmen

Sandia National Laboratories

24 Oct 2019

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Y'all want to purge UVM reliance

Sandia National Laboratories

- What is UVM?
- Advantages & disadvantages of UVM
- When, why, & how do Tpetra-based solvers rely on UVM?
- Removing UVM reliance in Tpetra: pitfalls & opportunities
- Strategies for downstream solvers
- Kokkos & Trilinos tools to help
- Best practices

UVM: Unified Virtual Memory

Sandia National Laboratories

- NVIDIA calls it “Unified Memory” now
- UVM feature is ALWAYS ON in CUDA hardware & runtime, but users opt into using it at memory allocation time
- CUDA has 3 kinds of allocations that GPU can access
  - cudaMalloc (Kokkos::CudaSpace): CPU cannot access
  - cudaHostAlloc (Kokkos::CudaHostPinnedSpace): CPU can access
  - cudaMallocManaged (Kokkos::CudaUVMSpace): CPU can access

UVM uses virtual memory pages

Sandia National Laboratories

- User just accesses UVM allocation on CPU or GPU
- CUDA runtime uses page faults to trigger copies
- Copies happen PER MEMORY PAGE, not per allocation
- Post-copy, page behaves like any other memory (e.g., cached)
- Host-pinned memory uses a different mechanism
  - Memory lives in CPU memory & stays there
  - Pages “locked” to fixed physical (non-virtual) addresses
  - This lets GPU access memory directly (Direct Memory Access), without needing to ask CPU to convert virtual to physical addresses
  - Non-CUDA uses: e.g., Network adapters use it to make MPI faster

Disadvantages of relying on UVM

Sandia National Laboratories

- Must avoid concurrent CPU & GPU access
  - UVM can't resolve "diffs" between same page on CPU & GPU
  - Nondeterministic, context-dependent bugs
  - Trilinos strategy: Set CUDA_LAUNCH_BLOCKING=1
    - Lose benefit of asynchronous GPU kernel launches
    - Can't overlap GPU kernels with CPU work or GPU-CPU copies
  - Less costly strategy: Fence (force GPU kernel completion) before accessing data on CPU, if you expect GPU to be accessing it
  - CUDA_LAUNCH_BLOCKING does something different from this
    - We (& apparently NVIDIA) don't really know what it does

Disadvantages of UVM (2 of 2)

Sandia National Laboratories

- Context-dependent hidden data movement costs
- Limit on max # of allocations (2^{16} perhaps)
- Will MPI correctly recognize UVM allocations?
  - Or crash?
  - Or think they are CPU memory & access them slowly on host, thus forcing subsequent GPU kernels to page them back?
    - This is why Tpetra only uses CudaSpace or HostSpace for MPI buffers
- Will upcoming computer architectures support it?
  - AMD (Frontier) has had unified memory for longer than NVIDIA
  - But, expect more architecture variety in the future
  - Also don’t expect that other vendors’ UM will be more reliable

Advantages of using UVM

Sandia National Laboratories

- Can allocate more memory than GPU physically has
- Only copy pages, not entire allocations
  - Kokkos::DualView::sync* copies entire allocations
  - DualView not the right tool for synchronizing sparse updates
- Above: Good for sparse graph algorithms
- Simplifies gradually porting code to run on GPUs

Why not use host-pinned memory?

Sandia National Laboratories

- More limited resource
  - Physical CPU memory
  - Used by network adapters; use may grow w/ # MPI processes
- Very slow allocation
  - 10-100x CUDA
  - CUDA allocation already very slow (~ MPI communication)
- Does not use GPU cache ➔ cannot exploit locality

How to get (non-)UVM Kokkos::View

Sandia National Laboratories

```
using Kokkos::Device; using Kokkos::Cuda; using Kokkos::View;

// Explicitly specify memory space

using non_uvm_cuda = Device<Cuda, Kokkos::CudaSpace>;
using my_non_uvm_view = View</* args */, non_uvm_cuda>;

using uvm_cuda = Device<Cuda, Kokkos::CudaUVMSpace>;
using my_uvm_view = View</* args */, uvm_cuda>;
```

Kokkos space defaults?

Sandia National Laboratories

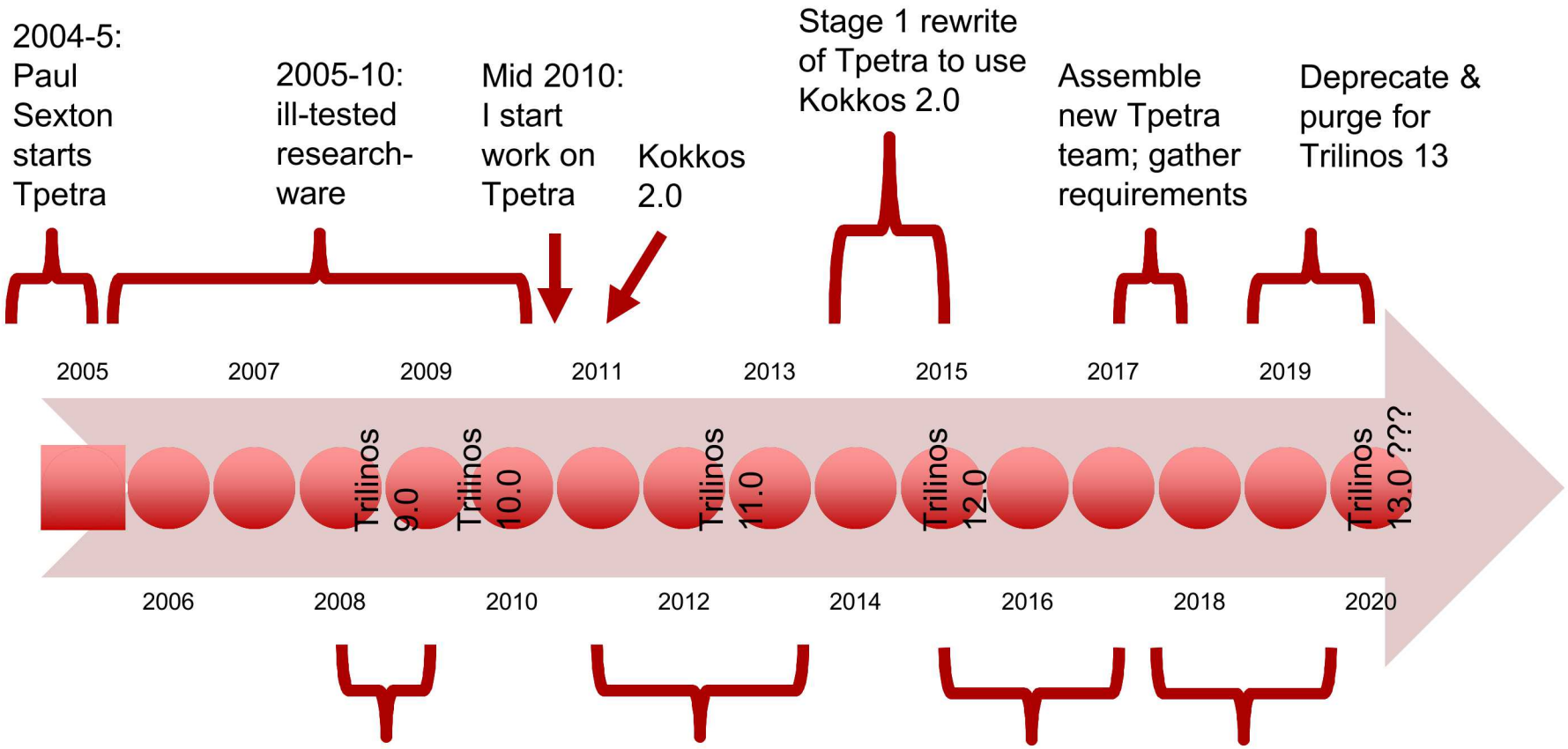
- All Kokkos::Device & {execution,memory} spaces have aliases
  - device_type
  - memory_space
  - execution_space
- This lets users use execution space in Kokkos::View
  - View<double*, Kokkos::Cuda> my_view;
  - View uses memory_space typedef in Kokkos::Cuda
- What's Cuda::memory_space?
  - Kokkos CMake option: either CudaSpace or CudaUVMSpace
  - Trilinos sets it to CudaUVMSpace, which is NOT Kokkos' default
- What's CudaSpace::execution_space::memory_space?

How does Trilinos rely on UVM?

Sandia National Laboratories

- Explicitly at configure time, via Kokkos CMake option
- Implicitly semantically, by
  - accessing UVM allocations on host
- Implicitly syntactically, by
  - assuming Cuda::memory_space == CudaUVMSpace
  - assuming View<…>::HostMirror::memory_space == View<…>::memory_space
  - assuming DualView::t_dev::memory_space == DualView::t_host::memory_space
- Why? History lesson…

Over 15 years of Tpetra



Tpetra objects: Global, w/ local data

Sandia National Laboratories

- "Global": Distributed over 1 or more MPI processes
- Have local data on each MPI process
- Either Tpetra or users may modify local data
  - e.g., A.apply(X,Y) writes result of sparse mat-vec into Y
- UVM question relates to
  - Storage of Tpetra objects' local data (do they use UVM allocations?)
  - Implementation of Tpetra functions (do they rely on UVM?)
  - User interface & users' access to Tpetra objects' local data

Tpetra in 2009: No UVM

Sandia National Laboratories

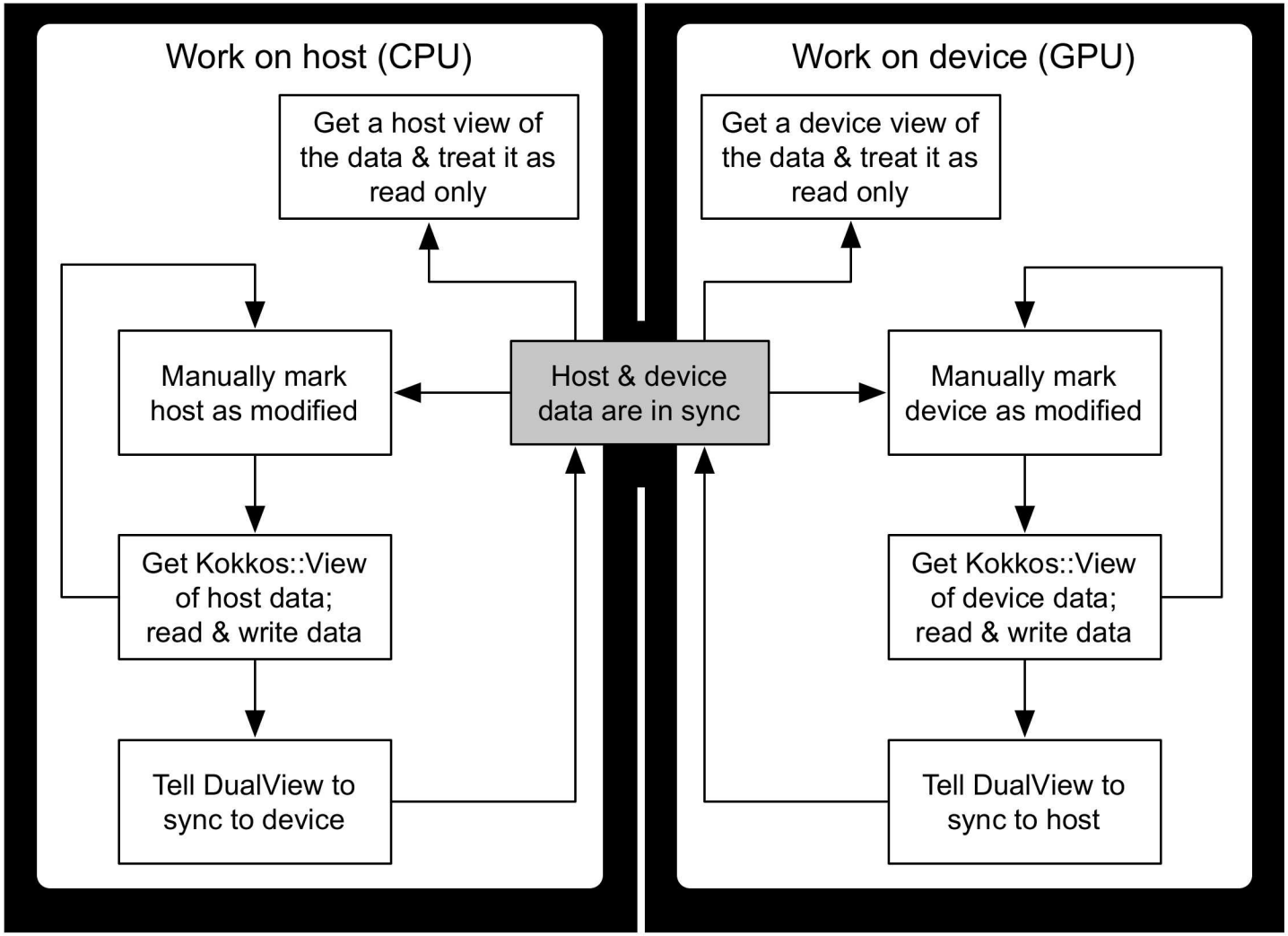
- Baker 2009: No UVM (it didn't exist)
  - Access local data on CPU only
  - GPU as accelerator for a few kernels
- Tpetra classes copy CPU-to-GPU at specific explicit points
  - CrsMatrix: resumeFill, CPU-only changes, fillComplete copies to GPU
  - MultiVector: "generalized view" (not in Kokkos::View sense)
    - e.g., get1dViewNonConst returns Teuchos::ArrayRCP of CPU memory
    - Read+Write copies back to GPU on destruction; read-only does not
- No Kokkos::View yet; use ArrayRCP to manage GPU memory
  - Memory ownership models: ref counting, or nonowning
    - Generalized views could have benefited from unique_ptr
- Failing builds & tests habitually ignored

Tpetra 2013-2015 refactor

Sandia National Laboratories

- Hoemmen & Trott late 2013-2015: Port to use Kokkos 2.0
- Refactor in place ("change engine while truck is rolling")
  - Use Kokkos data structures esp. View as much as possible
  - Parallelize more; remove sequential bottlenecks, esp. in setup
  - Minimize downstream changes & preserve host interface
- LAMMPS strategy: Kokkos::DualView
  - LAMMPS: physics plugins can opt into GPU execution
  - Clear boundary between "run on GPU" & "run on CPU"
  - DualView maintains separate copies of data on CPU & GPU
  - Manually mark modified on either side & call sync
- Tpetra adopted LAMMPS strategy
  - Classes (well, just MultiVector) imitate Kokkos::DualView

Kokkos::DualView behavior



2014-5: UVM introduced into Tpetra

Sandia National Laboratories

- In theory, Tpetra interface makes CPU-GPU copy explicit
- In practice, GPU tests ignored up to this time
  - ➔ Interface boundary untested & not respected by users
- UVM made downstream code "just work" unchanged
- UVM gave us a quick path to Kokkos-ize Crs{Graph,Matrix}
  - Too many states – too hard to get non-UVM working
  - StaticProfile resumeFill storage lives on GPU
  - CPU-only access interface (get*Row*, replace*, sumInto*, …) accesses GPU memory through UVM
- Remember that Christian & I had to stand up kokkos-kernels
  - Historically a Tpetra product & responsibility
  - Tpetra relied on Baker's Kokkos 1.0 kernels

Changed Tpetra interface meaning

Sandia National Laboratories

- Crs{Graph,Matrix}: StaticProfile resumeFill storage on GPU
- Generalized view methods rely on sync/modify, not copy back
- Users have direct access to GPU data
  - When running either on CPU or on GPU
  - ➔ Impossible for Tpetra to know if user has changed data
- UVM means that CPU & GPU storage are the same
  - Kokkos::DualView<T*, CudaUVMSpace> “lies”
    - dv.d_view.data() == dv.h_view.data() (same pointer)
    - Both CPU & GPU Views have same memory space
    - DualView doesn’t fence the GPU on sync (it used to!)
  - No enforcement for correct use of sync/modify

Tpetra-based solvers assumed UVM

Sandia National Laboratories

- Ifpack2, MueLu, Amesos2 all took shape 2010-11
  - Try to comprehend how much functionality that is
- Xpetra started as Python-generated copies of Tpetra classes
  - Baked into MueLu a dependency on Tpetra's host-only data access interface ~ 2010 (despite early discussions about "Kokkos matrix")
  - MueLu originally didn't intend to keep Xpetra – it let them get started when Tpetra wasn't yet fully functional
- No Trilinos CUDA deliverables until >= 2016-7
  - It took a while for solvers to start using Kokkos

De-UVMing Tpetra: Pitfalls (1 of 2)

Sandia National Laboratories

- Code implicitly couples typedefs between Tpetra classes
  - e.g., assumes CrsMatrix::device_type == MultiVector::device_type
  - Hinders fixing one Tpetra class at a time
- kokkos-kernels may assume UVM or have syntactic coupling
- Crs(Graph,Matrix) currently has no DualView-like interface
  - StaticProfile storage is always UVM, & host-only interface accesses it
  - Most complicated code in Tpetra – Sparse matrix-matrix multiply & other MueLu setup code – tied to UVM

De-UVMing Tpetra: Pitfalls (2 of 2)

Sandia National Laboratories

- Current Tpetra classes' interface far too permissive
  - Always (even CrsGraph!) return local data as View of nonconst
    - ➔ Tpetra can't distinguish access mode (e.g., read-only, read-write)
  - Users can always get Views & they are owning
    - ➔ Users may subvert sync/modify or resumeFill/fillComplete interface
    - ➔ Tpetra may clobber users' data or vice versa
  - Multiple ways to modify: e.g., host methods (sumInto*) or local matrix
    - Handy for e.g., finite-element method, to run assembly on GPU but use host-only globally indexed interface for Dirichlet boundary conditions
    - ➔ users must remember to sync/modify, etc.
- UVM optimizes sparse access on one side, dense on other
  - e.g., above boundary conditions example: host access is sparse
  - No UVM ➔ must sync entire matrix to host & back again

Minimal fix to Crs(Graph,Matrix)

Sandia National Laboratories

- Rely on resumeFill / fillComplete interface
  - Assume users don't modify on both sides in a single resumeFill
- Pre-first-fillComplete is a solvable special case
  - Constructor choice implies pre-1st-fillComplete access mode
    - Does it take a Kokkos::View or KokkosSparse::CrsMatrix?
    - Or, does it allocate storage without setting any data?
- Make getLocalMatrix behavior context dependent
  - Make getLocalMatrix throw before first fillComplete
  - getLocalMatrix post resumeFill implies read-write GPU access
    - Host-only interface checks modified-on-device flag & throws
    - Sync to CPU at fillComplete
  - getLocalMatrix post fillComplete implies read-only GPU access
    - Changes on CPU (already) forbidden post fillComplete

De-UVMing Tpetra: Opportunities

Sandia National Laboratories

- We're breaking backwards compatibility at Trilinos 13 anyway
  - Let's purge everything that causes trouble
- Interfaces simplifying access to local data
  - withLocalAccess (see other slide deck): explicitly declare
    - Where to access data (memory space)
    - What execution space accesses them
    - How they are accessed: read-only, read-write, write-only
    - When (the scope within which) users may access data
  - transform & for_each (Tpetra overloads)
    - Work like std::{transform,for_each} but take Tpetra objects
    - Take optional Kokkos execution space instance ("run here")
    - transform_reduce is next
- Opportunity for interesting programming models

withLocalAccess slides

Sandia National Laboratories

What about downstream solvers?

Sandia National Laboratories

- Expect massive downstream build & test breakage
- Be more explicit about data access
  - Where to access data (memory space)
    - Use sync/modify or resumeFill/fillComplete
  - What execution space accesses them
    - Do NOT assume default Kokkos execution space
    - Kokkos::RangePolicy<execution_space, index_type>(0,N)
  - How they are accessed: read-only, read-write, write-only
    - Use existing DualView idioms for declaring access intent
  - When – don't keep Views beyond their scope of use

Tools Kokkos does / will offer

Sandia National Laboratories

- Mirror views, deep_copy, & Kokkos::DualView
- "Fake CUDA": Debug memory & execution spaces
  - Debug w/out CUDA hardware or builds (that take forever)
  - Test memory access correctness (e.g., no CudaSpace access on host)
  - Test correctness when kernel launch is asynchronous
    - Maybe if we had this, we wouldn't have so much UVM trouble
  - See David Hollman's CppCon 2019 talk: https://youtu.be/sFfRxjAvxhc
  - In progress: See Kokkos PR #2307
- Clang compiler plug-ins (also in progress)
  - Requires Clang CUDA builds
  - Trilinos needs to build w/ Clang CUDA: See #1543, #3702

Nathan Ellingwood's DualView slides

Sandia National Laboratories

Declare access intent; limit scope

Sandia National Laboratories

```
Tpetra::Vector<…> v_gbl (/* constructor arguments */);
// ... intervening code …
{ // read-only access on host
  v_gbl.sync_host(); // sync only as needed
  auto v_lcl = v_gbl.getLocalViewHost();
  some_function_doing_read_only_host_access(v_lcl);
} // v_lcl disappears here
{ // read-and-write access on device
  v_gbl.sync_device(); // sync only as needed
  v_gbl.modify_device();
  auto v_lcl = v_gbl.getLocalViewDevice();
  some_function_doing_read_write_device_access(v_gbl);
} // v_lcl disappears here
```

Write-only access works too

Sandia National Laboratories

```
{ // write-only access on host
  v_gbl.clear_sync_state();
  auto v_lcl = v_gbl.getLocalViewHost();
  some_function_doing_write_only_host_access(v_lcl);
}

{ // write-only access on device
  v_gbl.clear_sync_state();
  auto v_lcl = v_gbl.getLocalViewDevice();
  some_function_doing_write_only_device_access(v_lcl);
}
```

Tools Trilinos have or can write

Sandia National Laboratories

- Integrate CMake options w/ Kokkos tools
  - e.g., debug builds can use debug Kokkos spaces by default
- Run-time test if memory is CUDA or not
  - Tpetra_Details_check{Pointer,View}.hpp
  - Tpetra functions call cudaPointerGetAttributes
  - Tpetra checks some input pointers in debug mode, to avoid bugs like creating a Vector with a “lying” CUDA View that wraps CPU memory
- Forbid CudaUVMSpace at compile time (static_assert)
  - Tpetra::Distributor already does this for MPI communication buffers
  - Would make Trilinos build more robust to Kokkos CMake options

General best practices

Sandia National Laboratories

- DON'T
  - expose an object's DualView directly
  - assume default memory spaces
  - expose owning Views of local data
- DO
  - limit access to local data to a well-defined small scope
  - write interfaces that make clear where data are accessed
  - use C++ idioms to express data access permissions & intent

Brief tutorial: Pointers & ownership

Sandia National Laboratories

- Views, containers, & pointers
- Allocators & Deleters
- Pointer to const != const pointer
- Memory lifetime / ownership models

Views, containers, & pointers

Sandia National Laboratories

- View: “a range type that has constant time copy, move, and assignment operators” ([range.view], C++20 draft)
  - “Range” is a generalization of (begin, end) iterator pair
- Container: deep-copies
  - “Deep-copies”: Copying a container copies its elements
  - ➔ NOT a view (dep. on # of elements, so not constant time)
  - e.g., std::vector, Teuchos::Array
- Pointer: “dumb” (T*) or “smart” (e.g., std::shared_ptr)
  - “Shallow-copies”: Copying pointer does NOT copy data
  - e.g., T*, std::shared_ptr, std::unique_ptr, Teuchos::{(Array)RCP, Ptr, ArrayView}, Kokkos::View
  - We say “pointers view their data” -- not technically views, but this conveys the idea of constant-time (copy,) move(, & assignment)

Allocators & Deleters

Sandia National Laboratories

- Containers can take arbitrary Allocators
  - Allocator generalizes new / delete (malloc / free)
  - Pre-C++17 allocators are stateless; C++17 permits instances (PMR)
  - Compare to Kokkos memory spaces
- Smart pointers can take arbitrary Deleter function or object
  - std::{shared,unique}_ptr, Teuchos::(Array)RCP
  - Can do an arbitrary action in addition to / instead of deallocating
  - e.g., Tpetra::MultiVector::get1dViewNonConst returns Teuchos::ArrayRCP<Scalar>, whose Deleter would copy back from CPU to GPU before freeing the CPU allocation
  - Caller responsible for allocation (unless they use {make,allocate}_shared or make_unique (C++14))

Pointer to const != const pointer

Sandia National Laboratories

- Pointer to const
  - const T*, View<const T*>, RCP<const T>
- const pointer
  - T* const, const View<T*>, const RCP<T>
- Pointers (to const or nonconst) may be passed by reference
  - Nonconst ref: T*& (or const T*&)
    - RCP<T>& (or RCP<const T>&)
  - Const ref: T* const& (or const T* const&)
    - const RCP<T>& (or const RCP<const T>&)
- EVERY TRILINOS DEVELOPER MUST UNDERSTAND THIS
  - ➔ Concisely self-documenting code
  - See Ross Bartlett's "Teuchos Memory Management Classes"

e.g., Kokkos::DualView func arg

Sandia National Laboratories

- Kokkos::DualView<T*, …>& (by nonconst ref):
  - Caller will see resize or reassignment (output argument)
  - DualView<const T*>&: immutable output argument
- Kokkos::DualView<T*, …> (by value):
  - Caller will NOT see resize or reassignment
  - Caller will see syncs & sync flag changes
- const Kokkos::DualView<T*, …>&: (by const ref)
  - Callee may NOT resize, reassign, or sync
  - Callee may change sync flags
- Kokkos::DualView<const T*, …> (by value or const ref)
  - Callee may NOT modify data
- See Trilinos 13 Tpetra::DistObject

Lifetime / ownership models

Sandia National Laboratories

- "When does pointer deallocate?"
  - Trilinos uses 2 models; there are others incl. in C++11
  - General goal: Declare lifetime by construction, syntactically
- Reference counting
  - All (shallow) copies share ownership ("peer to peer")
  - Last copy's destructor deallocates (when ref count -> 0)
  - std::shared_ptr; owning Teuchos::(Array)RCP or Kokkos::View
- Nonowning
  - All (shallow) copies view a "master" allocation
  - Master may disappear any time, invalidating view
  - T*, std::weak_ptr, std::span, std::string_view
  - Teuchos::{ArrayView,Ptr}; nonowning (Array)RCP or Kokkos::View

Other ownership models

Sandia National Laboratories

- Unique: At most one owner
  - May transfer ownership – giver gets null
  - Last owner's destructor deallocates
  - std::unique_ptr, any move-only class
  - "Move" refers to C++11 move construction & move assignment, e.g., unique_ptr(unique_ptr<U>&&)
- "Deferred"
  - For nodes in a possibly cyclic graph, sharing a common heap
  - Herb Sutter's deferred_ptr (see his CppCon 2016 talk)
- Point: Don't automatically reach for reference counting
  - Use self-documenting idioms
  - Kokkos::MemoryTraits<Kokkos::Unmanaged>

Avoid exposing owning Views

Sandia National Laboratories

- BAD: Tpetra returns local data as OWNING Kokkos::View
  - Tpetra objects are global, w/ local data on each MPI process
  - Tpetra stores local data as Kokkos::(Dual)View
  - Tpetra classes have methods to get local data as Kokkos::View
- Bad because owning Views have unlimited scope
  - Reference-counted, so only go away when user & Tpetra let go
  - (Tpetra,user) can't know when (user,Tpetra) reads/writes View
  - Prevents correct use of sync/modify
- Bad even without CUDA
  - User's class might store Tpetra object, then get & store its View too
  - If mesh changes & Tpetra object gets resized, owning View still valid memory, but it won't point to the right object.
  - Nonowning View will be invalid: can detect with tools

Make Views to local data nonowning

Sandia National Laboratories

- Tpetra: Make classes return syntactically nonowning View
  - Kokkos::MemoryTraits<Kokkos::Unmanaged> (nonowning)
  - Or, Kokkos::AnonymousSpace
    - "Union of all memory spaces"; its Views are always nonowning
  - Use C++11 templated type aliases to make syntax concise
  - template<class T, class DeviceType> using local_nonowning_vector = View<T*, LayoutLeft, DeviceType, MemoryTraits<Unmanaged>>;
  - local_nonowning_vector<T> X_lcl = /* function call */;
- Users: Limit access to local data to narrow scope
  - Don't keep View; get it on demand from Tpetra object
  - See examples in next 2 slides

Thanks!

Sandia National Laboratories