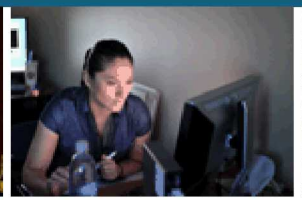
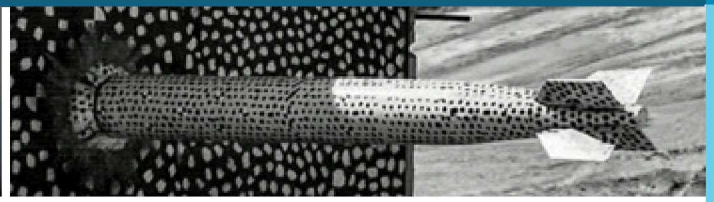
SAND2019-4836PE

Software Resilience using Kokkos Ecosystem





PRESENTED BY

Jeff Miles, Nicolas M. Morales, Keita Teranishi, Christian Trott

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.

Background

DOE Applications must be robust and resilient to hardware failures

- Reduce cost attributed to failed application runs
- Reduce cost attributed to incorrect results
- Minimize catastrophic failure occurrences

Resilience methodology

- Checkpoint / Restart
  - Periodically save execution state and restart from last good state when failure is detected.
  - Also used for pipelining and visualization
- Redundant Execution paths
  - Spawn multiple execution paths and choose the correct results from collection

Checkpoint/Restart

Leverage Kokkos View data structure and deep_copy semantics to enable transfer of data to and from persistent storage

Memory Space for each persistent storage architecture

- HDF5Space, StdFileSpace
- Data cannot be accessed/indexed via view directly.
- Deep_copy is the only way to store and access data

Checkpoint mirror - maintain a list of views which are checkpoint candidates.

Checkpoint operation transfers all candidates to persistent storage

Restore operation can be collective or a single view.

Checkpoint Example

```
typedef Kokkos::LayoutLeft           Layout;
typedef Kokkos::DefaultExecutionSpace::memory_space
                                     defaultMemSpace; // default device
typedef Kokkos::FileMemSpace         fileSystemSpace; // file system

fileSystemSpace fs;
typedef Kokkos::View<double*, Layout, defaultMemSpace> local_view_type;

local_view_type A("view_A", N);  local_view_type B("view_B", N);
local_view_type::HostMirror h_A = Kokkos::create_mirror_view(A);
local_view_type::HostMirror h_B = Kokkos::create_mirror_view(B);

auto F_A = Kokkos::create_chkpt_mirror(fs, h_A);
auto F_B = Kokkos::create_chkpt_mirror(fs, h_B);

fileSystemSpace::restore_all_views();  // restart from existing…
for ( ; ; ) {
   Kokkos::deep_copy(A, h_A);  Kokkos::deep_copy(B, h_B);
   Kokkos::parallel_for (N, KOKKOS_LAMBDA(const int j) {
      A(j) = j;  B(j) = j*2;
     });
   Kokkos::deep_copy(h_A, A);  Kokkos::deep_copy(h_B, B);

   if !consistency_check() {
      fileSystemSpace::restore_view("view_A"); // restore data
      fileSystemSpace::restore_view("view_B"); // restore data
   } else {
      fileSystemSpace::checkpoint_views();  // save result
   }
}
```

Automatic Checkpointing

Leveraging scalable checkpointing libraries such as VeloC https://github.com/ECP-VeloC/VELOC

- Provide automatic determination of whether a checkpoint is needed by finding the last restore point

We are interested in automatically checkpointing Kokkos views:

```
Kokkos::Experimental::checkpoint( "test_checkpoint", iter, [view, dim0, dim1](){
  Kokkos::parallel_for( dim0, KOKKOS_LAMBDA( int i ) {
    for ( int j = 0; j < dim1; ++j )
      view( i, j ) = 3.0;
  } );
}, checkpoint );
```

This will generate a checkpoint with the name "test_checkpoint" for iteration iter

- If the checkpoint exists it will not execute the lambda and instead restart from the checkpoint for that iteration
- All views capture by value in the lambda will be checkpointed
- Supports grouping multiple parallel executions into one checkpoint

Resilient Execution Model

Duplicate Parallel Execution (x3)

- Duplicate Captured views prior to executing
- Recombine results when complete.
- Const data views are captured normally (views all point back to same const data)

Resilient CUDA Execution space + Resilient CUDA Memory Space

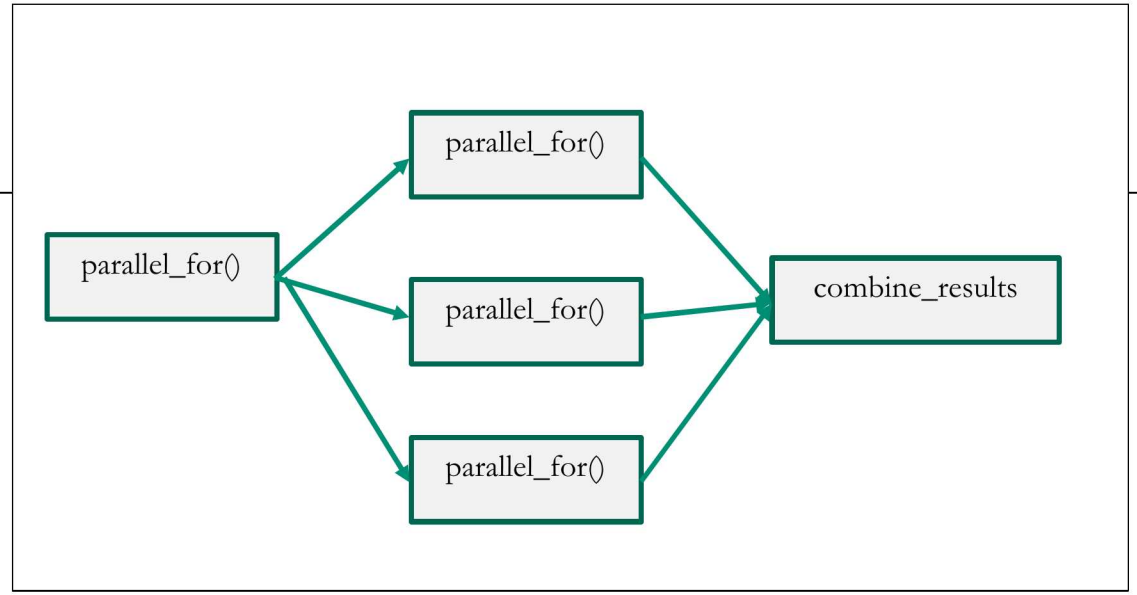
- Use streams to separate duplicate execution patterns
- Use Single parallel_for to recombine the results
- Recombine operation takes any 2 results that are the same and writes back to original view
- Comparison operation is type sensitive
- Types are not pre-defined, but user must “enabled” types used (via Kokkos macros)

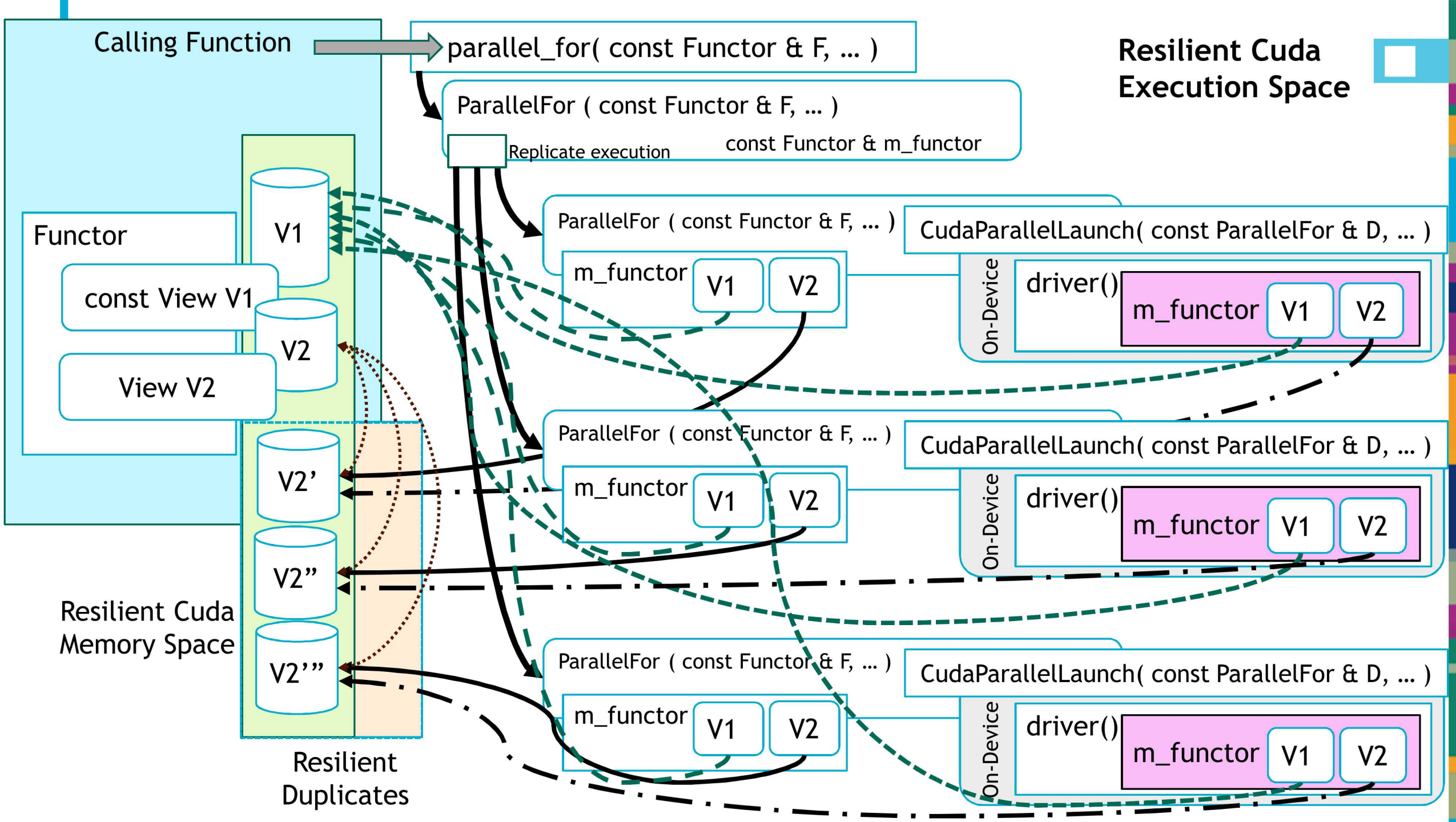
Replicate – Execute - Combine

```
typedef Kokkos::View< double*, Kokkos::ResCudaSpace > view_type;
view_type m_data ( "data", N );
typename view_type::HostMirror v = Kokkos::create_mirror_view(m_data);
Kokkos::RangePolicy<Kokkos::ResCuda> rp (0,N);

Kokkos::parallel_for(rp,KOKKOS_LAMBDA(const int i){
    m_data(i)=i;
});
Kokkos::fence();

Kokkos::deep_copy(v, m_data);
```




Calling Function
parallel_for(const Functor & F, …)
Resilient Cuda
Execution Space
ParallelFor (const Functor & F, …)
Replicate execution
const Functor & m_functor
Functor
const View V1
View V2
V1
V2
V2’
V2”
V2’”
Resilient Cuda
Memory Space
Resilient
Duplicates
ParallelFor (const Functor & F, …)
m_functor
V1
V2
CudaParallelLaunch(const ParallelFor & D, …)
On-Device
driver()
m_functor
V1
V2
ParallelFor (const Functor & F, …)
m_functor
V1
V2
CudaParallelLaunch(const ParallelFor & D, …)
On-Device
driver()
m_functor
V1
V2
ParallelFor (const Functor & F, …)
m_functor
V1
V2
CudaParallelLaunch(const ParallelFor & D, …)
On-Device
driver()
m_functor
V1
V2

Resilient Cuda Execution Space (RCES)

Kokkos::parallel_for invokes RCES ParallelFor

RCES ParallelFor replicates the Cuda Execution Space ParallelFor

- Use the same Execution Policy for each duplicate
- Associate different stream with each instance
- m_functor in RCES is a reference to the original (not a copy) – copies are contained/created in Cuda Execution Space ParallelFor
- Non-const views are duplicated with Functor copy construction (managed similar to ref-counts)
- Internal list of duplicated views maintained for recombination step

On-Device access to non-const views point to the duplicated memory locations

After execution is complete – Combine Resilient Memory space duplicates into originals, and free memory associated with duplicates

Conclusions

Checkpointing through Kokkos View adds a seem-less point of data integration

- Serialize data to persistent storage
- Publish data to workflow interface
- Kokkos provides automatic or manual interface giving users selective control over implementation
- Able to integrate with existing checkpoint libraries

Kokkos execution resilience able to minimizes execution cost and data latency

- takes advantage of available hardware concurrency through streams
- provide automatic data replication and recombination using parallel execution when possible

Section Break Slide