Los Alamos
NATIONAL LABORATORY

Kokkos Usage in xRAGE

Peter Maginot- EAP DPL for Physics
with Daniel Holladay, Zach Medin, Clell (CJ) Solomon

Kokkos User Group Meeting, December 12-15, 2023

LA-UR-23-33701

Outline

1. What is xRAGE
2. Porting history/strategy
3. FY24 plans
4. Kokkos usage within grey diffusion

xRAGE is a large LANL ASC multiphysics code

- Mostly Fortran 90/95
- O(500k) lines
- Large number of internal users
  - Their day job is to run xRAGE
- O(15-20) developers
- Extensive validation basis
- MPI domain decomposition for parallelism
- Expected to run [performantly] on all large NNSA HPC systems
- Geared toward high energy density applications (NIF, Omega, Z experiments)
  - But still has “cold” physics for HE burn, material strength, etc..

Interested in Kokkos for GPU porting because code base is too large to maintain multiple code paths

xRAGE targeting of GPUs has not been steady

- Significant work in FY19 and FY20
  - Focus on inter-operatbility, FLCL, DualView
  - Start porting physics [folders]
- Impediments
  - COVID
  - Unilateral decision to stop porting
  - LANL ASC re-organization, EAP re-organization
  - Staff departures (Classification Office, Industry, Other projects → Industry)
- Resumption in late FY22 through Today
  - Focus on verifying / demonstrating GPU performance of ported packages

We are not where we could be, but we have a consistent vision and plan to move forward!

Each ported xRAGE physics currently responsible for migrating data to and from the GPU

- The above also implies, “from Fortran to C++”. Currently:
  1. Break apart Fortran derived types into component arrays and scalar
  2. Convert arrays into flcl_ndarry_t objects
  3. Cross the Fortran/C barrier
  4. Convert flcl_ndarry_t* objects into Host Views
  5. Transfer Host Views to Device Views
- Very manual, very large function signatures
- FLCL has caused some tricky to find issues
  - FLCL::HostSpace not really HostSpace (it’s CudaUVM)
  - Correctness of taking a Fortran allocated array and assuming it is in CudaUVM space?
  - flcl_ndarray_t have served their purpose, but new options exist

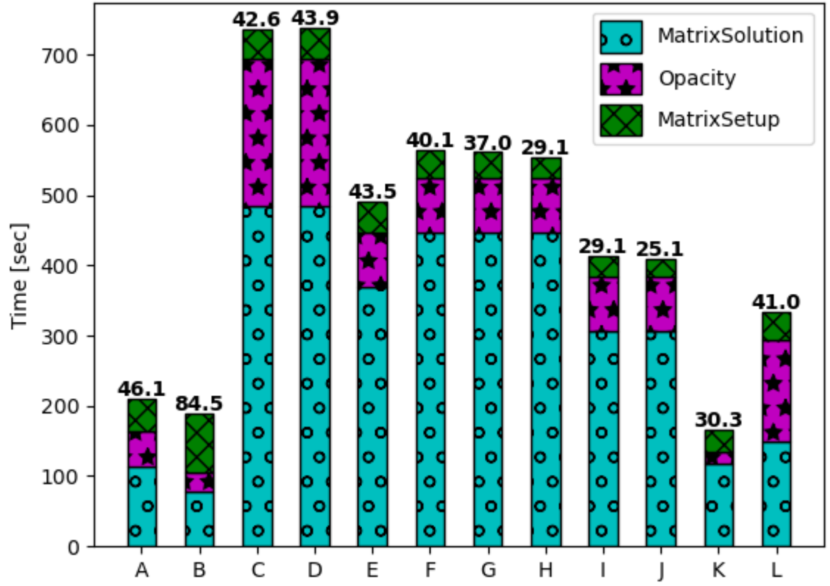
FY23 was “Year of the Machine” FY24 will be xRAGE’s “Year of the GPUs”

- Unsplit hydrodynamics, conduction, high explosives reactive burn, multigroup radiation diffusion to be ported
  - LANL is getting serious since impending Venado and El Cap hardware deliveries suggest GPUs are not going to go away
- Removing FLCL dependency
  - xRAGE mixed compiler build + availability of “newer” GNU compilers on Sierra
  - xRAGE will make use of F2018 “C Descriptors” to replace flcl_ndarray_t
- Streamline steps from Fortran derived types to GPU Views
  - Maintain historical hierarchy of derived type data
  - Consistent method to auto-generate Fortran to C interoperable structs then transform
    - Most developers will write a single Python file per Fortran derived type
    - CMake will run Python files to 1) auto-generate interoperable structs definitions and 2) C++ transforms to get from C descriptors to Views

[Grey] radiation diffusion is our most ported physics

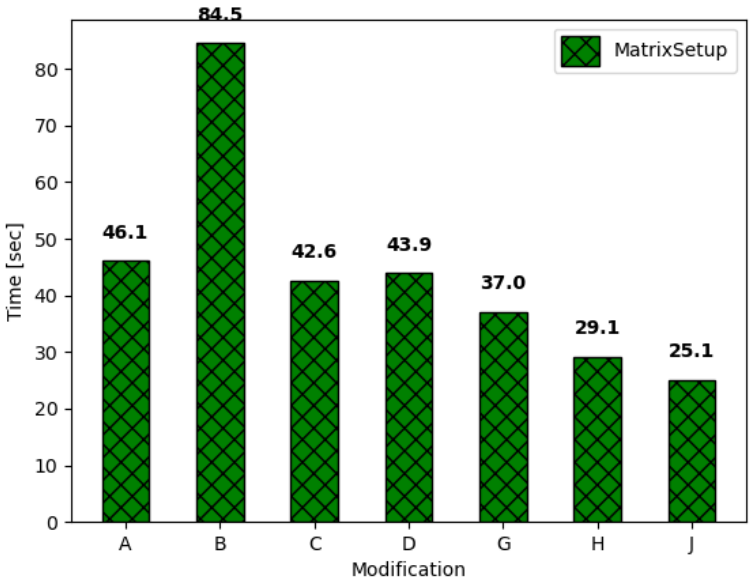
- Amongst the original three “physics” packages [folders] identified in 2018
- Three pieces for complete grey diffusion solve
  - Opacity (data) lookup
  - Matrix setup
  - Matrix solve
- Matrix setup is effectively the piece of xRAGE physics we can control
  - Results suggest we might wish to consider controlling more pieces
- Since it was most ported, radiation diffusion has been our Kokkos testing ground
  - What features of Kokkos might we want to use for performance?
  - Allows us to explore how do multiple GPU chefs in the kitchen work?
    - xRAGE Kokkos, CUDA Fortran, CUDA C, TPLs that use Kokkos all must build and link together in harmony

Despite following later, xRAGE is experiencing much of others' progression in performance



	CPU	Improvement
A	36-CTS	Legacy Fortran
B	40-P9	C++ (forAllFaces)
C	4-P9	no MPS
D	4-P9	forAllFaceType
E	4-P9	lag opacity
F	4-P9	hypre 2.26 (bad settings)
G	4-P9	`partition_space` streams
H	4-P9	`cudaMallocAsync`
I	4-P9	hypre 2.26 (better settings)
J	4-P9	unmanaged memory
K	36-CTS	Final C++ (lag opacity)
L	36-CTS	Final C++ (no opacity lagging)

Focusing on what we are actually speeding up paints a fairer picture



	CPU	Improvement
A	36-CTS	Legacy Fortran
B	40-P9	GPU + MPS
C	4-P9	no MPS
D	4-P9	forAllFaceType splitting
G	4-P9	`partition_space` streams
H	4-P9	`cudaMallocAsync`
J	4-P9	unmanaged memory

We can achieve modest speed-up of a low FLOP, high data transfer routine via code re-writing and less common Kokkos strategies.

xRAGE iteration over faces needed to evolve for GPUs

- Faces in xRAGE are categorized by type and facing direction
  1. Lo boundary
  2. Hi boundary
  3. Interior at 1:1 interface
  4. Interior fine [lo] / coarse [hi]
  5. Interior coarse [hi] / fine [lo]
- Face-based data stored in 2-D arrays
  - Global face index not a concept
  - Indexed within a dimension
  - <10% over allocation (maxint*numdim)

maxint = max(max_f_x, max_f_y, max_f_z)

Fortran Iteration Pattern

```
do dim=1,numdim
  do loop=1, n_types_of_faces(dim)
    face_type = face_id(loop,dim)
    if face_type == 1
      n_lo = face_lo(loop,dim)
      n_hi = face_hi(loop,dim)
      do n=n_lo,ni
        cell_hi = face_local(n,HI,dim)
        face_data(n,dim) = func(data(cell_hi))
      enddo
    else if face_type == 2
    ! ... omitted for brevity

  enddo
enddo
```

Natively would require 15 kernel launches!

Radiation diffusion first unrolled common pattern once and created auxiliary iteration structures

- Did not collect timing data prior to transition
  - Code did not work on GPUs prior to a MR that did too many things ☺
- Pros
  - Single kernel launch
  - Logical “forAllFaces” of a Fortran pattern
- Cons:
  - Increased memory footprint
  - Retains face_type checking logic
  - Atomics needed for reductions to cell data

forAllFaces Iteration Pattern

```
parallel_for("calc_a_thing",
  RangePolicy<EXEC_SPACE>(0,f_idata.n_faces_tot),
  KOKKOS_LAMBDA (const size_t f_idx) {
    cell_lo = f_data.cell_lo_of_face(f_idx);
    cell_hi = f_data.cell_hi_of_face(f_idx);
    face_type = f_data.type_of_face(f_idx);
    dim = f_data.dim_of_face(f_idx);
    idx = f_data.idx_of_face(f_idx);
    if (face_type == 2){
      face_data(n,dim) = func(data(cell_lo))
    }
    // ... omit other face_types for brevity
});
```

forAllFaces has been split into 3 separate kernel launches

- Eliminated if checking
  - FaceIterationData doubles in memory footprint
- Small latency slowdown without `Kokkos::Experimental::partition_space`
- Streams allowed for
  - simultaneous kernels
  - less cudaDeviceSynchronize calls
- Streams increased complexity
  - As code exits a function, leave streams in flight
  - Manual process for book keeping

forAllFaceTypes Iteration Pattern

```
parallel_for("calc_a_thing_interior",
  RangePolicy<EXEC_SPACE>
  (streams[0],0,f_idata.n_faces_int),
  KOKKOS_LAMBDA (const size_t f_idx) {
    // ...
});
parallel_for("calc_a_thing_typ1",
  RangePolicy<EXEC_SPACE>
  (streams[1],0,f_idata.n_faces_typ1),
  KOKKOS_LAMBDA (const size_t f_idx) {
    // ...
});
parallel_for("calc_a_thing_typ2",
  RangePolicy<EXEC_SPACE>
  (streams[2],0,f_idata.n_faces_typ2),
  KOKKOS_LAMBDA (const size_t f_idx) {
    // ...
});
```

xRAGE operator splitting severely limits the amount of overlapping computation that can occur

Streaming of View allocation + enabling cudaMallocAsync resulted in the single largest speed-up of run_diff_cycle

- xRAGE is used to “free” allocations with CPU+DDR
  - More than 120 allocations within run_diff_cycle sized proportional to spatial DOF
- Streams permitted early computations to overlap with View creation on device
  - Further complicated code flow / readability
    - Juggling 11(!) streams
- Change in behavior from 3.7.01 to 4.0.01
  - Our data lookup functions are all on the CPU

```
create_mirror_view_and_copy(view_alloc(stream[0], HOST(), WithoutInitializing, dev_view)
```

- Requests
  - Please no more static assert failures that don’t give a line number
  - Spack variant in Kokkos maintained package.py for cudaMallocAsync

cudaMallocAsync still noticeable when calling 100+ times

- Hand rolled a memory pool for exploratory purposes
  - 120+ allocations to 6
- Umpire will replace manual pointer math
  - Can Kokkos handle Umpire allocators being used to evict / transfer data?
  - Can underlying pointer of Unmanaged views be swapped?

```
n_big_alloc += local_count + mirror_count + copy_in_count;
View<double*> big_alloc(view_alloc(STORE(), "big_alloc" , stream[1])
,                        n_big_alloc);
stream[1].fence();
double* big_alloc_ptr = big_alloc.data();
size_t big_alloc_used = 0;
double* tev_ptr = big_alloc_ptr + big_alloc_used;
View<double*,MemoryUnmanaged> tev (tev_ptr, hv_tev.size());
big_alloc_used += tev.size();
deep_copy(stream[1],tev, hv_tev);
```

xRAGE's path to performance is porting more physics

- Grey diffusion matrix setup has been a testbed for Kokkos concepts to improve performance
  - `Kokkos::Experimental::partition_space`
  - `cudaMallocAsync`
  - `Kokkos::MemoryUnmanaged`
- Would like to see `partition_space` come out of `Experimental`
- Considering moving to `4.1` for profiling concurrent kernels
- Interested in seeing if others have
  - Explored/use memory pools and/or whether it is a priority for Kokkos development
  - Have Power9 + V100 results comparing ScatterView vs. atomics for reductions

Questions / Comments / Advice?