Experiences Porting a Multiphysics Code to GPUs

COE Performance Portability Meeting

Brian Ryujin

August 22, 2017

LLNL-PRES-737000

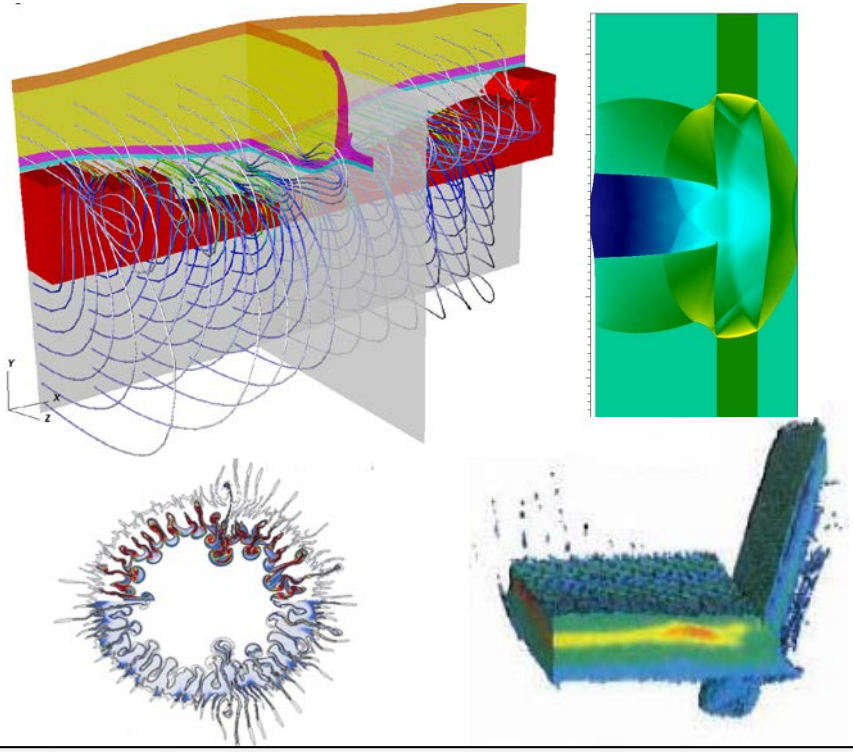
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

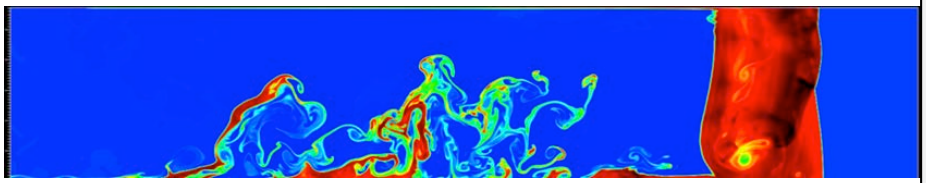
Lawrence Livermore National Laboratory

Ares is a massively parallel, multi-dimensional, multi-physics code

Physics Capabilities:

- ALE-AMR Hydrodynamics
- High-order Eulerian Hydrodynamics
- Elastic-Plastic flow
- 3T plasma physics
- High-Explosive modeling
- Diffusion, SN Radiation
- Particulate flow
- Laser ray-tracing
- MHD
- Dynamic mixing
- Non-LTE opacities





Applications:

- ICF modeling
- Pulsed power
- NIF Debris
- High-Explosive experiments

Porting a large, existing code comes with additional constraints

- Ares is 21 years old and ~800k lines of code
  - C/C++ with 60+ libraries in C, C++ and Fortran
  - Some libraries are full physics capabilities, such as SN Radiation
- Ares is used daily by an active user base on our current supercomputers
  - Cannot break or slow down current functionality
  - Must be able to continue to add new functionality that users are requesting throughout the process
- Code overall has ~5,000 mesh loops with limited hotspots
  - Lagrange hydro problem runs 80+ kernels
  - Grey radiation diffusion problem runs 250+ kernels
  - ALE hydro problem runs 450+ kernels
- Code is mostly bandwidth bound

We can only maintain a single code base, but must effectively utilize all HPC platforms

Ares strategy for Sierra

- We tried to adhere to some basic guiding principles
  - Keep strategies relatively simple
  - Leverage existing capabilities and infrastructures
  - Keep concepts familiar to developers
- Overarching approach:
  - Use RAJA to get code to run on the GPU
  - Use Unified Memory to get mesh data onto the GPU
  - 1 MPI task per GPU
  - Keep all data resident on the GPU to avoid data motion

We believe our approach follows our principles, but the devil is in the details…

We use RAJA for our execution model

- RAJA is an abstraction layer for on-node parallelization
- Facilitates using multiple programming model backends(CUDA, OpenMP, etc)
- Designed for existing codes; allows for incremental adoption
- Built on top of standard C++11 constructs only

C-style for-loop

```
double* x ; double* y ;
double a;
for ( int i = begin; i < end; ++i )  {
    y[i] += a * x[i] ;
}
```

RAJA Transformation

RAJA-style loop

```
double* x ; double* y ;
double a ;
RAJA::forall< exec_policy > ( begin, end, [=] (int i) {
    y[i] += a * x[i] ;
} );
```

RAJA Concepts

Pattern

(forall, reduction, scan, etc.)

Execution Policy

(how loop runs: PM backend, etc.)

See Adam Kunen's talk on Wednesday for more details

Ares's use of RAJA

- Implemented a code specific layer above RAJA
  - Improves readability by giving loops additional context
  - Provides an easy place to put hooks in for all our loops
  - Lets us map descriptive loop classifications into machine appropriate policies

C-style for-loop

```
double* x ; double* y ;
double a;
Int* ndx = domain->Zones;
for ( int i = begin; i < end; ++i )  {
    int zone = ndx[i];
    y[zone] += a * x[zone] ;
}
```

RAJA Transformation

RAJA-style loop

```
double* x ; double* y ;
double a ;
Domain_t* domain;
for_all_zones< parstream > ( domain, [=] (int zone) {

    y[zone] += a * x[zone] ;
} );
```

Ares Concepts

Pattern

Code specific iteration pattern

Execution Policy

High level classification of loop

See Olga Pearce's talk on Thursday for an example

RAJA provides us with additional flexibility at little cost to code maintainability and familiarity

- After the port to RAJA, the code still looks very similar
  - Lambda semantics also helped in the porting process to some degree
  - There's still a learning curve for developers to change how they're coding, but largely overcome after about a week or two of use
- Over 98% of our loops could be ported in a straightforward manner
  - Remaining loops are serial loops that need to be reworked for performance
- RAJA's ability to use multiple backends is an invaluable tool
  - Very easy to switch between different backends (Serial, OpenMP3, CUDA)
  - Serial performance is comparable to non-RAJA code
  - We use CPU thread analysis tools to help track down race conditions we see on the GPU by using the OpenMP3 execution policy
  - Still waiting on some development to compare OpenMP4.5 with CUDA
- Code benefits directly from performance improvements in RAJA
  - Platform specific optimizations are hidden within RAJA's primitives
  - Upgrading to new RAJA versions has been straightforward

Memory management strategies in Ares

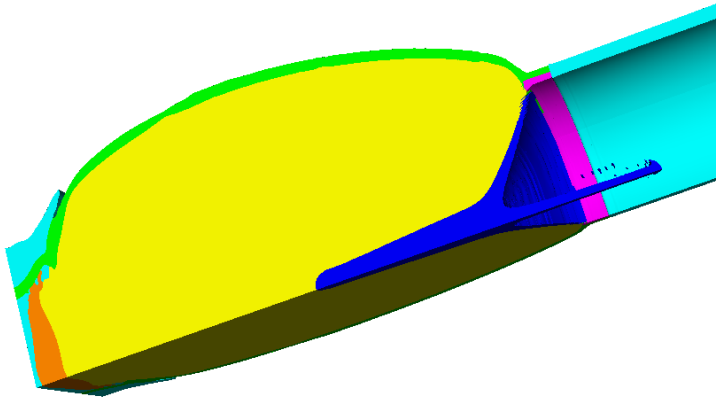
- Ares has ~5000 malloc calls that are each wrapped by a macro
  - A call to malloc does not provide enough context to effectively use GPUs
  - Some context can be inferred by existing state (e.g., allocating temporaries)
  - Some context cannot be inferred (e.g., location of data usage)
- For performance, we now have a three tiered system
  - Originally used cudaMallocManaged for everything
    - Worked correctly, but performed poorly
  - Allocating different memory has vastly different costs
    - Malloc – CPU control code
    - cudaMallocManaged (UM) – For mesh data (accessed on CPU and GPU)
    - cudaMalloc (cnmem memory pools) – Temporary GPU data
  - Switching from a naïve single tier system to a three tier system gave a 14x speedup on current hardware
- Plan to switch the allocation to use Umpire

See David Beckingsale's talk on Umpire

We are seeing good speedups on our EA systems

Problem	Zones (Mzones)	Speedup 4 GPU vs 20 Power8 CPU cores
3D Sedov	1	10X
3D Sedov + LEOS	1	12X
2D Sedov	1	6X
3D ALE test	1.1	7X
3D Rad Diffusion test	1.1	8X
3D Shaped Charge (ALE, Strength, Slides)	1.3	6X

EA Node
20 Power 8+ CPU cores
4 NVIDIA P100 GPUs
16 GB Memory/GPU
NVLINK 1.0

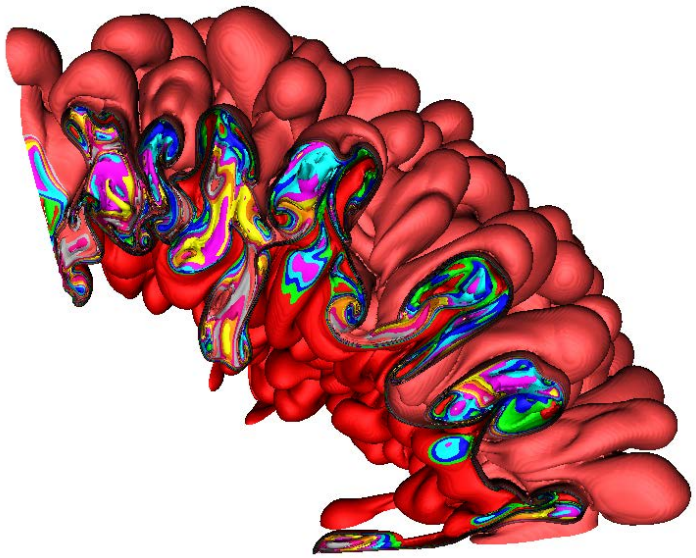


Some packages are more amenable to GPUs. Increasing zone count provides a better speedup

GPU performance is showing great promise for ALE hydro problems

CTS1 (Broadwell) vs. EA (Power8 + P100 GPUs)

Resources	# of Nodes	Runtime (min)	Relative speedup
576 CPU cores	16	2825	1
1152	32	1407	2
2304	**64**	**811**	**3.5**
4608	128	400	7
16 GPUs	**4**	**816**	**3.5**
32 GPUs	8	514	5.5
64 GPUs	16	398	7.1



RT Mixing Layer in a Convergent Geometry

- Octant, 37.3M zones
- 300,000 cycles
- LEOS EOS (with RAJA)
- Eulerian

With nominal zone counts, it takes 15X more CTS1 nodes to match a single EA node

Conclusion and Future Work

- RAJA and Unified Memory is a viable strategy for some degree of performance portability
  - Over 98% of the loops port cleanly
  - Remaining loops may require algorithmic changes for parallelization
- GPU performance tracks about what we'd expect compared to CPU
- Kernel launch overhead can become a significant portion of runtime
  - Packing and unpacking MPI buffers for ghost zone exchanges is dominated by kernel launch overhead, and can take over 25% of the runtime in moderately sized problems
  - Packing and unpacking code is organized with polymorphic objects, which makes kernel merging difficult
  - Investigating techniques to queue the packing/unpacking calls without breaking the objected oriented organization
- Dealing with our large physics libraries will be challenging
  - Management of the limited amount of memory between packages
  - GPU strategies for libraries must be compatible
- Running with much larger MPI tasks (30k zones/rank -> 1M zones/rank) will force us to relearn our intuition on the code's scaling properties

Acknowledgements

- Jason Burmark
- Mike Collette
- Burl Hall
- Rich Hornung
- Holger Jones
- Jeff Keasler
- Olga Pearce
- Brian Pudliner
- George Zagaris

Lawrence Livermore
National Laboratory