LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-823775

Enhancements supporting IC usage of PEM libraries on next-gen platforms

D. F. Richards, B. S. Ryujin, N. Barton, S. Bastea, B. Beauchamp, B. Beck, D. Beckingsale, R. Blake, P. Brantley, P. Brown, J. Burmark, R. Carson, E. Chen, M. Collette, S. Dawson, L. Fried, G. Gert, J. Grondalski, J. Gyllenhaal, B. Hall, R. Haque, B. Isaac, M. Katz, A. Kunen, I. Kuo, H. Le, J. Loffeld, C. Mattoon, M. McFadden, S. McKinley, M. Meraz-Rodriguez, D. Miller, P. Minner, R. Nimmakayala, C. Noble, M. O'Brien, M. Osawe, M. Patel, M. Pozulp, B. Pudliner, V. Rana, R. Rieben, P. Robinson, A. Skinner, D. Slone, B. Stephens, P. Sterne, D. Stevens, T. Stitt, A. Vargas, B. Wayne, K. Weiss, C. White, R. Whitesides, M. Yang, B. Yee

June 22, 2021

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Enhancements supporting IC usage of PEM libraries on next-gen platforms

Edited by
David F. Richards and Brian S. Ryujin

Contributing Authors
Nathan Barton, Sorin Bastea, Brock Beauchamp, Bret Beck, David Beckingsale, Robert Blake, Patrick Brantley, Peter Brown, Jason Burmark, Robert Carson, Evelyn Chen, Mike Collette, Shawn Dawson, Larry Fried, Godfree Gert, John Grondalski, John Gyllenhaal, Burl Hall, Riyaz Haque, Ben Isaac, Max Katz, Adam Kunen, I-F. Will Kuo, Hai Le, John Loffeld, Caleb Mattoon, Marty McFadden, Scott McKinley, Manny Meraz-Rodriguez, Doug Miller, Paul Minner, Rao Nimmakayala, Chad Noble, Matt O'Brien, Maxwell Osawe, Mehul Patel, Mike Pozulp, Brian Pudliner, Verinder Rana, Rob Rieben, Peter Robinson, Aaron Skinner, Dale Slone, Branson Stephens, Philip Sterne, David Stevens, Tom Stitt, Arturo Vargas, Brett Wayne, Kenneth Weiss, Chris White, Russell Whitesides, Max Yang, Ben Yee

June 23, 2021

LLNL-TR-823775

Contents

Introduction and Executive Summary 4

1 PEM Efforts to Support the LLNL ASC Mission 6
- 1.1 LEOS Project . . . 7
- 1.2 MSLib . . . 7
- 1.3 TDF Project . . . 9
- 1.4 Opacity Server . . . 10
- 1.5 GIDIplus Project . . . 11
- 1.6 Cheetah . . . 13
- 1.7 Ares . . . 14
- 1.8 ALE3D . . . 14
- 1.9 MARBL . . . 15
- 1.10 Ardra . . . 16
- 1.11 Mercury . . . 16
- 1.12 Summary . . . 16

2 Progress and Challenges Porting PEM Software 18
- 2.1 LEOS Project . . . 18
- 2.2 MSLib . . . 20
- 2.3 TDF Project . . . 21
- 2.4 Opacity Server . . . 23
- 2.5 GIDIplus Project . . . 24
- 2.6 Cheetah . . . 25
- 2.7 Summary . . . 27

3 Test Problems 29
- 3.1 Hotspot Problem . . . 31
- 3.2 Hotspot Results . . . 32
- 3.3 Shaped Charge Problem . . . 36
- 3.4 Shaped Charge Results . . . 37
- 3.5 Jetting Defect Problem . . . 42
- 3.6 Jetting Defect Results . . . 43
- 3.7 Nonlocal Problem . . . 45
- 3.8 Nonlocal Results . . . 46
- 3.9 Godiva Problem . . . 48
- 3.10 Godiva Results . . . 49
- 3.11 NIF Chamber Problem . . . 56
- 3.12 NIF Chamber Results . . . 57
- 3.13 Barrier Problem . . . 62
- 3.14 Barrier results . . . 64

4 PEM Software Porting Highlights and Gaps 68
- 4.1 LEOS Project . . . 68
- 4.2 MSLib . . . 68
- 4.3 TDF Project . . . 69
- 4.4 Opacity Server . . . 69

4.5 GIDIplus Project 69
4.6 Cheetah 71
4.7 Summary 73

5 Conclusions and Recommendations **74**

Acknowledgments **76**

References **77**

A Program Counter Sampling **79**
A.1 Google Performance Tools PC Sampling 79
A.2 Nvidia Nsight Compute PC Sampling 79
A.3 Alternatives to PC Sampling 81

Introduction and Executive Summary

Integrated multiphysics codes play an important role in the stockpile stewardship mission at LLNL. Preparing these codes for next-generation computing platforms such as Sierra and El Capitan is one of the key responsibilities of the Integrated Codes (IC) and Physics and Engineering Models (PEM) subprograms of the ASC program. This report describes work performed in support of FY21 Milestone #7847: "Enhancements supporting IC usage of PEM libraries on next-gen platforms." It also contains information to satisfy all of the completion criteria. The description and completion criteria of the milestone are:

> **Description:** This milestone reports on the culmination of several years of effort by multiple PEM support software development teams to provide capabilities for use in LLNL-developed integrated codes on next-gen ASC platforms, including GPU support. We will provide a survey of relevant Application Program Interfaces (API) that are required to support LLNL IC code capability on relevant architectures, with a focus on Sierra and El Capitan. We will identify and summarize all dependencies between PEM supported libraries and IC supported physics codes. We will provide an assessment of algorithmic improvements that have been deployed, as well as future developments that are required to complete the GPU porting efforts. This assessment will include a description of programming models adopted by each of the PEM projects, distinct algorithmic challenges for each of the capabilities, and information about sharing GPU memory between the APIs and host codes. We will develop targeted test problems to assess computational performance. Finally, this milestone will result in identification of gaps in our effort to assist the LLNL ASC program in prioritization of effort for porting software to El Capitan.
>
> **Completion Criteria:**
>
> 1. Develop a summary of PEM efforts required to support the LLNL ASC mission on Sierra and El Capitan. This summary will include information about how PEM software is utilized within IC codes.
> 2. Discuss the progress and challenges of porting PEM software integrated into IC software to GPU architectures, including information about programming models employed, GPU memory sharing between PEM libraries and host codes, and a summary of algorithmic challenges.
> 3. Define targeted test problems for assessing computational performance that exercise the PEM capabilities studied in this milestone.
> 4. Highlight porting efforts of PEM software integrated into IC software on Sierra.
> 5. Identify gaps in current effort to enable future prioritization for LLNL PEM-IC integrated development for Sierra and El Capitan.

The scope of this milestone includes 5 IC codes (ALE3D, Ares, MARBL, Mercury, and Ardra) and 6 PEM library collections (LEOS, MSLib, TDF, the Opacity Server, GIDIplus, and Cheetah). The dependencies between codes and libraries is shown in Table 1 on page 6. The outline of this report closely follows the completion criteria.

Section 1 provides a summary of the capabilities of the PEM libraries and their efforts to support the ASC mission and also brief descriptions of the IC codes. The PEM libraries have developed APIs that are designed to meet the needs of IC codes for both CPU and GPU platforms. PEM

library APIs typically contain a variety of functions related to setup or initialization and a set of lookup functions that provide data or model evaluations as a function of problem state. Only the lookup functions need to support GPU execution. Calling context determines whether the library needs to provide scalar or vector (i.e., array) interfaces to the lookup functions. Some PEM libraries such as LEOS and Cheetah that are called in multiple contexts provide both.

Section 2 describes the porting status of the PEM libraries. With the exception of the Opacity Server, all have functional GPU ports that provide at least some of the library's capabilities. Of these, LEOS and TDF are the most mature and are in routine production use. Looking toward El Capitan, few problems are expected. Moving to a new GPU architecture is expected to require much less effort that the initial conversion from CPU to GPU. RAJA and HIP will provide the necessary portability to target AMD GPUs.

Section 3 defines the test problems for this milestone and provides performance results on CPU and GPU platforms. LEOS and TDF both show excellent node-to-node speedups and typically require less than 1% of total problem runtime on both GPU and CPU platforms. MSLib demonstrated speedups of 8.6× and 26× for two different models and Cheetah delivered up to 16× speedup at typical GPU problem sizes. These results demonstrate that high-fidelity physics models can obtain speedups similar to pure hydro, making them practical for use on GPU architectures. Measured GIDI load times are short compared to large 3D simulations, but can be a bottleneck for quick 1D simulations. The GPU port of MCGIDI appears to perform adequately compared to other parts of the Monte Carlo algorithm, but this evaluation is very uncertain due to difficulties obtaining fine-grained performance measurements on the GPU. Finally, the desirability of a GPU port for the Opacity Client library is demonstrated.

Section 4 collects the highlights and gaps observed while running the test problems. With the exception of the Opacity Server, all of the PEM libraries have demonstrated GPU capabilities with encouraging speedups. The fraction of runtime used by the PEM libraries on GPUs is as good as on CPUs, indicating that the libraries will not be a performance bottleneck. Still, there is considerable work to be done to increase the range of features that are ported and optimized for GPUs, as well as testing and hardening code for production.

Finally, Section 5 considers the milestone effort as a whole to provide overall conclusions and recommendations.

1 PEM Efforts to Support the LLNL ASC Mission

Completion criteria #1 of the milestone is:

1. Develop a summary of PEM efforts required to support the LLNL ASC mission on Sierra and El Capitan. This summary will include information about how PEM software is utilized within IC codes.

This section satisfies the criteria by providing a description of each of the PEM libraries and IC codes that are covered by the milestone. Table 1 shows the dependencies between the 6 PEM libraries and 5 IC codes that define the scope of the milestone. The library summaries include:

- A description of the modeling capabilities provided
- Which IC codes are supported
- Programming language(s) used in the library
- Approximate size of the library
- Level of developer effort
- A high-level description of the library API, with attention given to which functions must support the GPU.
- Any unique features or capabilities of the library

The section also includes brief descriptions of each of the IC codes.

	ALE3D	Ares	MARBL	Mercury	Ardra
LEOS	X	X	X		
MSLib	X	X	X		
TDF		X	X		
Opacity Server		X	X		
GIDIplus				X	X
Cheetah	X	X			

Table 1: Dependencies between PEM Libraries and IC codes.

1.1 LEOS Project

The LEOS project provides a set of data access and interpolation routines to facilitate the accurate and efficient use of equation of state (EOS) data by application programs, including all major LLNL ASC multi-physics design codes as well as many support codes. EOS data is an essential component in hydrodynamic and radiation-hydrodynamic simulations, providing energy, pressure, sound speeds and related data that are needed for the solution of conservation equations, such as the Navier Stokes equations. Hence, LEOS is on the critical path for hydrocodes to run on GPUs. Functionality is broken into two libraries, libleos commonly referred to as LEOS, and liblip (the Livermore Interpolation Package) or LIP. Host codes call the LEOS library which, in turn, calls LIP to perform interpolations. LEOS and LIP are written in C++ and contain approximately 150k lines of code. Total developer effort is about 1.5–2.0 FTE/yr.

The LEOS API includes functions for problem setup (file reading, calculation of coefficients, memory placement, etc.) and EOS lookup. The lookup interface supports scalars (one density-temperature point), arrays (multiple density-temperature points using C style pointers), or C++ std::vectors as arguments. Most hydrocodes will use either the array or std::vector interface, but LEOS is also used in some applications and libraries which use the scalar interface. For example, MSLib makes calls into LEOS using the the scalar interface. While the LEOS library is written in C++, it also provides optional interfaces for use by C, Fortran, and Python programs.

Codes call LEOS to perform an initial setup by reading the data from a file and determining the coefficients for efficient interpolation. Setup is performed once per EOS and is done on the CPU to facilitate file access. LEOS includes capabilities to allow host codes to modify EOS data during setup. Filters modify the EOS data that is read in from disk as part of setup, creating a new or modified EOS. After setup, the interpolation coefficient data is unchanged for the duration of the run. Setup tasks are typically not performance critical and there is no need to port them to the GPU.

In contrast, EOS lookup functions must run on the GPU and must be very fast since they occur in 3–4 loops per time step and account for most of the time spent in LEOS. Lookups can specify hooks that allow modification of the input density-temperature values before and after interpolation, as well as the resulting interpolation value. Since hooks are part of the lookup itself, they can execute on the GPU, in conjunction with the LIP interpolation step. Efficient forward lookups using (density, temperature) points are required. As some codes use density and energy as their independent variables, an efficient lookup of temperature is provided via a precomputed (from the energy table) "tcalc" table. The temperature from tcalc is then used with the density in forward lookups on other tables.

Data management is a key consideration when dealing with EOS data. Some memory reduction mechanisms are available, in the form of partial interpolation setup and lossy data compression, allowing trade-off between memory requirements and computational speed should the host code require.

LEOS uses several additional libraries for datafile access (e.g., HDF5) and other operations, many of which run only on the CPU. Some packages, like the zfp compression package, run on both CPU and GPU. LEOS also can use the Umpire and ASCMemory libraries for memory management, helping facilitate memory usage and data transfer between CPUs and GPUs.

1.2 MSLib

MSLib is a library of constitutive modeling capabilities based on continuum mechanics concepts and specialized for utilization in a hydrocode setting. The constitutive model has two related

jobs: it provides the stress tensor that goes into the balance of momentum, and it updates the evolving state of the material. This state can be tracked by variables for the stress itself, measures of accumulated deformation such as the equivalent plastic strain, number densities of atomic-scale defects, orientations of sub-scale physical features such as crystal lattices, and so forth. MSLib is invoked at all of the host code integration points every time step as part of the solution of the balance of momentum. Excluding comments, the main part of MSLib has roughly 110k lines of code (mostly C with some C++) and there are another 40k lines in the `evpc` Fortran module for crystal-mechanics-based models. The `evpc` module is no longer actively developed. There are other models in MSLib that were added for assessment or research purposes and that are neither actively developed nor commonly used in production simulations. Development effort has been less than 0.5 FTE/yr in most recent years, with an increase to roughly 1 FTE/yr during the effort to complete the initial (fat-kernel) GPU port.

In some use cases, MSLib in turn calls either LEOS or Cheetah to obtain EOS information as part of the overall constitutive response. These EOS calls may be made inside of an iteration loop. In more elaborate models, such as those for certain mixtures, the MSLib model may perform LEOS calls for more than one component material at a given host code location. For use with quasi-static formulations, the constitutive models also provide a stiffness matrix.

In some cases, the constitutive model sub-problem can be phrased as a system of non-linear ODEs. In practice, the ODEs are stiff enough that an implicit or semi-implicit time stepping algorithm is used. Thus the problem becomes the solution of a non-linear system of equations. Many constitutive models are simplified to the point where a single non-linear equation is solved iteratively (the rate-dependent radial return algorithm).

While MSLib includes implementations of some of the simplest constitutive model types, it is often used for more elaborate model types, such as those involving porosity evolution or for materials that undergo phase transformations.

Many different sub-model combinations are possible in MSLib, and the particular combination to be used for a given material is established during model initialization. After model initialization, the model data are constant and the API calls for which computational performance is a concern are of the `getResponse` and `map` families of functions, with the less compute-intensive `map` type functions being used for fixed-state evaluation and post-advection "fixup." It is only these `getResponse` and `map` type functions that have been ported for use on GPUs. The rest of the MSLib API is implemented only on the CPU. The preferred input parsing pathway for the C++ interface to MSLib uses MatProp. The initialization phase also includes calls by which the host code is provided information about history variable requirements. The host code is then responsible for the initialization and storage of history variables as fields over the appropriate subset of the mesh. A typical model may have roughly 10 history variables, but the number can vary significantly depending on the details. Some history variables are for output only and are not needed to track the material state—so that it is possible for the host code to reduce persistent memory requirements by choosing not to store these outputs.

The data associated with a given MSLib model itself is typically less than 400 numbers, not including any associated data requirements for LEOS or Cheetah. Even when the C++ interface is being used, most of the data for the model are kept in a C struct, and those data are constant after initialization.

To various degrees, MSLib has interfaces for C++, C, Python, and Fortran.[1] MSLib was originally developed as part of ALE3D and, due to an incomplete transition of some of the input

[1]The C++ interface used by MSLib can also be used by other constitutive modeling libraries, such as the GeoDyn material library.

parsing, some models remain usable only through the C interface in ALE3D. Over time, more capabilities have migrated to being available through the C++ interface. A vectorized[2] API for the material evaluation calls and some of the newer models is available only through the C++ interface. Like ALE3D, Ares uses a combination of the C and C++ interfaces. The Fortran interface is used by Diablo. The C++ interface is currently being added to MARBL. And the Python interface is used by MIDAS and for testing and development purposes.

For the vectorized API available through the C++ interface, most arrays are currently required to be unit-stride. There is an index array for history variables, given that history variables may be stored on only a relevant subset of the locations on the mesh.

In addition to the above-mentioned direct dependencies, MSLib makes use of SNLS for some of its non-linear system solves. The SNLS library is LLNL-developed and open-source [19]. The acronym SNLS stands for Small Non-Linear Solver, with "small" referring to size of the systems being solved locally to each host code integration point.

1.3 TDF Project

The Thermonuclear Data File (TDF) system generates the data related to the Maxwellian-averaged thermonuclear reaction rates of various light nuclear reactions and makes it available to the downstream applications codes Ares, Kull, and Marbl. This is achieved with the two independent modules that are called TDFgen and TDFlib. Both TDFgen and TDFlib are written in C and contain about 2,300 and 7,200 lines of code (respectively). Work on these libraries is performed only as needed and total developer effort is minimal. Total developer effort has been about one month/year over the last five years and was exclusively focused on the GPU porting effort. Porting work for Sierra required less than 0.5 FTE of effort. Prior to Sierra porting work, the libraries have been mostly untouched since 2008.

TDFgen calculates plasma reactivities, mean kinetic energies, and the final state distributions at a discrete set of energies. This information is combined with the necessary interpolation data and stored in ASCII files for use in downstream applications. These downstream applications use TDFlib as the interface with the TDFgen output ASCII files. TDFgen is not called directly by application codes. Hence, only TDFlib needs to run on GPUs to support the LLNL ASC mission on Sierra and El Capitan.

TDFlib divides its API info five categories:

Version Routines (1 function) Returns the version number of the library.

Database Routines (5 functions) Open .tdf file databases and answer simple queries about the data they contain.

Reaction Info Routines (9 functions) Answer queries about individual reactions, such as the number of reactants and products.

Look-up Routines (10 functions) Return reactivity and energy information by interpolating from stored data values. These are the main functions that will be GPU-callable.

On-the-fly Routines (6 functions) Return similar information as look-up routines, but for non-thermal distributions, where the distributions are inputs. The calculations involve quadratures.

To support calling of query operations within client GPU kernels TDFlib API routines which may be invoked from either the CPU or the GPU are marked as __host__ __device__ functions. These

[2]While the interface to MSLib has been vectorized, as described in Section 2.2, in the current fat-kernel GPU port vectorization is not yet pushed down the call stack.

device-callable functions accept scalar arguments. Applications have not expressed any need for an array-based API. The necessary data structures are made available on the GPU through the use of unified memory (e.g., cudaMallocManaged). Because the look-up routines have little mathematical intensity, and because TDF data is typically migrated to the GPU only once per application run, there is very little cost in terms of flops or data movement from TDFlib itself.

A typical application code will first open a TDF database (database routine), getting back a C-struct handle to a database struct. It will then ask for the individual reactions it is interested in (database routines) to get back each reaction as a handle to a reaction C struct. It will then set itself up to use the reactions by asking each reaction for the numbers of reactions and products and their clyde numbers (reaction info routines). Finally, in the main loop of its code, it will ask for reactivities and energies at particular temperatures (look-up routines). Alternatively, it will call the on-the-fly variants of those routines instead.

During file reading, all data is read on MPI rank 0, and the internal reaction data structures (C structs) are built and filled. To populate other ranks, the data is serialized, broadcast to all other processors over MPI, and deserialized. This initial setup runs entirely on the CPU. Once it is complete, client codes call TDF strictly locally and no further interprocess communication is needed. In typical production runs the total size of TDF related data structures is approximately 11 MB per MPI rank.

The underlying data for the reactions is mostly in the form of probability distribution functions. The functions are represented as tables of values at discrete points. Almost all queries, such as obtaining thermal reactivity at some temperature, require interpolation on the tabular data to find values intermediate to the stored points. TDFlib also provides accumulations or quadratures on data, such as when computing cumulative distribution functions. The remaining operations are bookkeeping lookups of basic data, such as how many products are in a reaction. In summary, most of the computational routines are linear combinations (weighted sums) over data points held in tables. The operations are therefore very low cost.

At present, TDFlib depends only on standard system libraries and it uses CMake and BLT to configure and build. It has no runtime dependence on the nuclear data libraries but it has to be compiled against a legacy utility library that provides access to an error handling method.

1.4 Opacity Server

The Opacity Server processes raw opacity data of individual chemical elements into average opacity tables for arbitrary user-specified mixtures. The server is an Apache-based web server maintained and operated by WCI IT. The supporting client library is used by virtually all WCI radiation hydrodynamics codes to submit requests to the server, read server generated data tables, and look up opacity data. Only the client library is linked into host codes. The server contains approximately 35k lines of source code in Python and Yorick. The client library is also about 35k lines of code and is written in C++ using an object-oriented framework for flexibility. In addition to the C++ interface, it also includes C and Python interfaces. The developer effort for this project is about 1.0 FTE.

The client API provides functions in three categories: submitting requests to the Opacity Server for a specific mixture, reading server-generated opacity tables from disk, and performing lookups at arbitrary density and temperature. Historically, host codes used the client only to submit requests to the server. Each host code maintained its own “native” capability to read opacity files, store opacity data, and perform lookups. However, usage of the full range of client capabilities, including the lookup features of the client API is growing steadily. At present the client has no GPU capabilities. The functions to make server requests and read files are used only during problem set

up and are expected to run on the CPU. Hence, only the lookup functions need to be ported to the GPU.

Lookup functions allow users to specify either an individual density and temperature point (scalar interface) or group of points (array interface), as well as whether the point(s) are logarithmic or linear. The initial GPU port will support only the array interface. Users can also specify different off-table behavior (OTB) for each direction. Different choices of OTB are required due to the complex dependencies of opacities with respect to density, temperature, and energy groups. A lookup works by calculating the bounding box of each point passed to the lookup function and interpolating between the values. If a point is off-table in any direction, the specified OTB for that direction is used instead.

A server generated file can be read and processed by the client library. By default, each server generated file contains a metadata string, which instructs the client library how to properly read the data. The client library only loads and processes the data specified in the metadata string; these data are stored in a general OPAC::Opacity_Data object. The OPAC::Opacity_Data object supports several fundamental types of data; data of the same type are stored and can be retrieved from a C++ std::map. In general, each set of tabulated data is stored in an OPAC::Table_Data object, which can hold an arbitrary number of axes. Each of the axes is represented by an OPAC::Axis_Data_Abstract object, and these objects can be retrieved directly from the OPAC::Table_Data object. The OPAC::Table_Data interface allows users to read their data as logarithmic or linear, change the axis ordering of the table, and specify which axes belong in the dependent or independent space. The independent space determines which axes are used for the interpolation, and the dependent space are the values at each point being interpolated. Interpolation of data is provided via the OPAC::Lookup_Table object, which contains a reference to a OPAC::Table_Data object. OTBs are specified independently for each axis and is represented by derivations of the OPAC::OTB object. OPAC::Lookup_Table contains all the logic for finding the bounding box of the points passed to the lookup function to perform interpolation.

The client library relies on a number of third party libraries including Pact, PDBLite/Silo, HDF5, zlib, and ASCMemory. Most of these are needed on the CPU only. Umpire and RAJA dependencies will be added when GPU lookups are implemented.

1.5 GIDIplus Project

GIDIplus is a collection of C and C++ APIs that support loading and sampling libraries of evaluated and processed nuclear physics data, including nuclear reaction cross sections, energy and angle distributions for outgoing reaction products, transfer matrices (for use in deterministic transport), particle masses and nuclear decay properties. These data are necessary for solving the Boltzmann transport equation to determine the flux of neutrons, photons and charged particles in transport applications. The main users of GIDIplus at LLNL are the deterministic radiation transport code Ardra and the Monte Carlo radiation transport code Mercury. Data are stored in the Generalized Nuclear Database Structure (GNDS) format, which has been developed over the past decade by an international collaboration led by LLNL. GIDIplus contains approximately 30k lines of code divided across several related libraries as described below. Developer effort is about 0.3 FTE.

GIDIplus currently coexists with an older set of nuclear data libraries including libNDF, libM-CAPM and libNuclear. These libraries are part of an aging infrastructure that was built around the Evaluated Nuclear Data Library (ENDL) format, which has been used at LLNL for over 50 years. These libraries are still commonly used for multiple applications at LLNL, but they are no longer being actively maintained. The older libraries are being replaced by GIDIplus and GNDS, and development work is focused on these new products.

GIDIplus is designed to be 'particle agnostic': it handles nuclear reactions for any combination of projectile and target, as well as some reactions with atoms and molecules (e.g. photon scattering off of atomic electrons). However, the code can only provide whatever data is present in GNDS libraries. Currently those libraries include nuclear reaction information for incident neutrons, photons and light charged particles (up to alpha particles), atomic interactions for incident photons and electrons and in some special cases interactions between low-energy incident neutrons and molecular targets.

One of the main challenges to the design of GIDIplus is the large volume of nuclear data needed by transport applications. For example, the LLNL ENDL2009.4 nuclear reaction data library includes information about over 3600 different combinations of projectile and target, including 585 evaluations for incident neutrons. Each evaluation may store multiple reactions (typically 30–50 for incident neutrons) and lists cross sections and outgoing product distributions for each reaction. These libraries are organized through the use of 'map' files which list all available evaluations for each projectile and target.

Nuclear data libraries for incident neutrons are typically Doppler broadened to multiple temperatures to account for changes in the cross section (and to some degree the outgoing distributions) due to thermal motion in the target. Doppler broadening is computationally expensive, so it is done as part of generating the processed library. Processing also requires generating transfer matrices for each reaction, temperature and outgoing product. The transfer matrices are used to transform an incident particle flux into an outgoing flux as a function of energy and angle. Processed GNDS files list cross sections and transfer matrices at 23 different target temperatures ranging from room temperature up to 100 keV/kB (approximately 1.2 GK). These files store both 'continuous energy' and 'multigroup' versions of cross sections and distributions, with the continuous energy data typically being used for Monte Carlo transport and the grouped data being used for deterministic transport. Altogether the processed ENDL2009.4 library (in GNDS format) takes up approximately 99 GB on disk, the bulk of which (96 GB) is consumed by data for neutron-induced reactions.

GIDIplus consists of several related libraries:

- GIDI, the General Interaction Data Interface. GIDI is primarily responsible for loading nuclear data from GNDS data files into an internal class hierarchy and for performing some basic operations such as collapsing multi-group data down to coarser group structures while applying flux-weighting spectra. GIDI consists of approximately 10,000 lines of code, with an additional 3200 lines of test code.
- PoPI, Properties of Particles Interface. PoPI is responsible for loading information about particles such as nuclei, excited nuclear states, baryons, leptons and photons. These data are stored in 'PoPs' files, which are also defined as part of the GNDS standard. When GIDI encounters particle information while parsing nuclear reaction data, it delegates responsibility to PoPI for reading that particle data. PoPI consists of approximately 1500 lines of code with an additional 300 lines of test code.
- MCGIDI, Monte-Carlo GIDI. MCGIDI is designed to be integrated into a Monte Carlo transport code such as Mercury, and provides stochastic sampling of nuclear reaction and decay data. MCGIDI does not include any direct file I/O capability. Instead it relies on GIDI to load in data, then performs some simple transformations to optimize performance while sampling the data. MCGIDI consists of approximately 9500 lines of code, with an additional 3000 lines of test code.
- HAPI, the Hierarchical data API. HAPI is a recently developed compatibility layer designed to help speed up GIDI file load times, and will be discussed further in section 4.5.

The test problems in this milestone focus on the GIDI and MCGIDI components of GIDIplus.

Two of the most important components of the GIDI API are the GIDI::Map::Map and the GIDI::Protare classes. The Map class typically serve as the main point of entry for application codes using GIDI. The Map reads in a map file, which stores information about all available projectile / target / evaluation combinations in a library. These combinations are also called Protares (PROjectile + TARget + Evaluation). The Map class provides a method for reading in specific files to generate GIDI::Protare instances. The GIDI::Protare (and its subclasses) contain methods for extracting all of the nuclear data terms required by deterministic transport codes.

Monte Carlo codes also use the Map and GIDI::Protare classes to read in data, and then use the GIDI::Protare to construct an MCGIDI::Protare. The MCGIDI::Protare class contains a smaller subset of the nuclear data, and is enhanced with sampling capabilities. For example, MCGIDI provides the total reaction cross section for each material in a transport problem. Once the transport code determines (using the total cross section) that a reaction has occurred on one of the materials in the problem, the MCGIDI::Protare corresponding to that material is responsible for sampling 1) which reaction occurred, 2) what reaction products are emitted and 3) the outgoing energy and angle of each reaction product.

GIDI depends on only a few third party libraries including Umpire, HDF5, and pugixml. Of these only Umpire has any relevance to GPU performance.

1.6 Cheetah

The Cheetah high explosive code provides advanced high explosive equation of state and burn kinetic models for the Ares and ALE3D multiphysics codes. Cheetah calculates chemical equilibrium between reacting chemical species by solving stiff nonlinear equations using backtracking Newton solvers. Chemical kinetics is solved by solving ordinary differential equations with partial chemical equilibrium, which requires nonlinear equation solving at every step of the differential equation integrator.

Cheetah is written in C and C++, and contains about 360,000 lines of code. The code functions as both a material library for multiphysics code, and as a stand-alone application used to model energetic material energy delivery. The Cheetah stand-alone application is supported by the Department of Defense through the DoD/DOE Joint Munitions Program, and is released to over 500 users. The stand-alone application can be used with a traditional command line interface, or with a Java graphical interface implemented in roughly 50k lines of Java code through a Java/C native interface. Total code developer (as opposed to model development and calibration) effort has historically been about 1.0 FTE/yr. However, that effort is currently elevated to about 3.0 FTEs to provide the extra resources needed for GPU porting.

Cheetah provides a C API consisting of functions providing per-problem initialization, per-material initialization, equation of state evaluation, chemical kinetic evaluation, and parallel communication between MPI tasks. Cheetah provides to the multiphysics code an evaluation of equation of state parameters such as temperature and pressure, transport parameters such as viscosity, and solves chemical kinetic equations associated with high explosive burn. The parameters returned by Cheetah can be specified by the calling code, which allows for efficient adaptation to an array of possible host code environments.

The original API, designed for CPUs, relied on a single function call per hydrodynamic finite element (typically called a zone). This scalar interface is not well suited for advanced GPU architectures that efficiently process long vectors of operations. To improve Cheetah's performance on GPU architectures, the Cheetah team developed a vector interface, which allows millions of zones to be processed together. The vector interface is invoked from the CPU, but calls device kernels to

process long arrays of zones.

A central issue in Cheetah is the use of partial chemical equilibrium. Partial chemical equilibrium allows Cheetah to track many more chemical species than are exposed to the hydrodynamic code, which greatly improves efficiency and reduces memory usage. Partial chemical equilibrium, however, requires stiff nonlinear equations to be solved for every evaluation of the chemical reaction rate. In addition, the nonlinear chemical equilibrium equations are based on sophisticated statistical mechanical theories of dense fluids. The statistical mechanical equations are highly complicated in form, making the solution of the nonlinear chemical equilibrium equations numerically intensive from the viewpoint of a multiphysics code. It is common for the stand-alone application to take 0.01–1 second to solve the equations, which is up to 6 orders of magnitude slower than requirements for a successful hydrodynamic simulation at a typical application scale.

A unique feature of Cheetah is its ability to greatly improve performance using a database or cache of solutions that is constructed on-the-fly. As chemical equilibrium calculations are completed, they are stored in a sparse equation-of-state database that is used to answer future queries and reduce the number of chemical equilibrium solutions that need to be executed. Therefore Cheetah functions as an on-demand physics code, where numerically expensive physics equations are solved adaptively as the multiphysics simulation explores new phase space. Use of the cache database increases the speed of the HE detonation response function evaluations 10,000 times over direct numerical solution of the physics equations, making a highly sophisticated model computationally affordable.

Cheetah relies on a number of third-party libraries when linked to a multiphysics code. These include RAJA, Umpire, HDF5, zlib, and dmalloc or tcmalloc. Cheetah also uses a customized version of the LLNL CVODE solver, which is part of the Sundials package and a customized version of the donlp2 nonlinear programming library (www.netlib.org) translated into C and used for quadratic and general nonlinear constrained optimization. Additional dependencies arise for the graphical interface of the stand-alone code, which will not be discussed here.

1.7 Ares

Ares is a multi-dimensional, massively parallel, multi-physics code developed to support the stockpile stewardship mission. Its primary use is to support the design and analysis of programmatic experiments supporting that mission. This includes experiments to better understand basic material properties important to DOE and DoD national security missions, such as the behavior of high explosives (HE) under a range of conditions as well as strength, melt and vaporization properties of inert materials. The variety of experimental platforms that must be modeled includes the Z-machine at SNL, which necessitates the need for magnetohydrodynamic (MHD) capabilities, the Omega (LLE) and NIF (LLNL) platforms, which require laser modeling capability and high energy density physics (HEPD) capabilities including local and non-local thermodynamic equilibrium (NLTE) models, separate electron, ion and radiation temperatures, and radiation diffusion and transport. In order to support the design and analysis of experiments aimed at understanding turbulent mix, Ares has a wide collection of mix models for micro-jetting at surfaces, hydrodynamic, and plasma-phase mixing.

1.8 ALE3D

ALE3D is a multi-physics numerical simulation software tool utilizing arbitrary-Lagrangian-Eulerian (ALE) techniques. The code is written to address both two-dimensional (2D plane and axisymmetric) and three-dimensional (3D) physics and engineering problems using a hybrid finite element and

finite volume formulation to model fluid and elastic-plastic response of materials on an unstructured grid. ALE3D is a single code that integrates many physical phenomena through an operator splitting approach. Additional ALE3D features include heat conduction, chemical kinetics, species diffusion, incompressible flow, a wide range of material models, chemistry models, multi-phase flow, and magnetohydrodynamics, which can be used in numerous combinations for long (implicit) to short (explicit) time-scale applications. ALE3D also makes heavy use of Smoothed Particle Hydrodynamics (SPH) via Spheral and embedded grids.

ALE3D operates on a wide variety of platforms, ranging from laptops to the world's largest supercomputers. ALE3D has native implementations for Windows and Mac workstations for smaller scale problem sets, and it is portable to virtually any Unix-based machine with C++/C and Fortran compilers available. The code will also run in parallel on multi-processor Windows and Mac machines. While most users will be interested in Linux-based versions of the code, it has also been ported to several other lightweight kernel operating systems.

1.9 MARBL

MARBL is a next-generation multiphysics code that addresses the modeling needs of the high energy density physics community for simulating high-explosive, magnetic or laser driven experiments such as inertial confinement fusion (ICF), pulsed-power magnetohydrodynamics, equation of state and material strength studies as part of the NNSA's stockpile stewardship program .

MARBL 's design centers around its modular computer science, physics, and math infrastructure. A foundational component of MARBL is the Axom computer science (CS) toolkit which provides infrastructure for the development of modular, performance portable, multi-physics application codes [15]. Axom's *Sidre* (**Si**mulation **d**ata **re**pository) component provides capabilities to centralize data management in high-performance computing (HPC) applications for efficient coordination and sharing of data across physics packages and other libraries in integrated applications, and between applications and tools that support file I/O, in situ visualization and analysis. Sidre leverages Conduit's *Mesh Blueprint* conventions to share mesh-based simulation data between physics packages.

In addition, MARBL's performance portability abstractions enable performance portability and heterogeneous memory management. RAJA is an open source set of C++ abstractions for writing single-source, portable loop kernels which supports multiple back-ends including sequential, SIMD, OpenMP (CPU, target), CUDA and AMD HIP. Umpire is an open source API for managing heterogeneous memory resources including memory operations for integrated applications. MARBL has integrated these into its usage of finite element classes via MFEM's built-in *Device* and *Memory* abstractions.

MARBL is designed from inception to support multiple diverse algorithms, including Arbitrary Lagrangian-Eulerian (ALE) and direct Eulerian methods for solving the conservation laws associated with its various physics packages. A distinguishing feature of MARBL is the use of advanced, high-order numerical discretizations such as high-order finite element ALE and high-order finite difference Eulerian methods. High-order numerical methods were chosen because they have higher resolution/accuracy per unknown compared to standard low-order finite volume schemes and because they have computational characteristics which play to the strengths of current and emerging HPC architectures. Specifically, they have higher FLOP/byte ratios meaning that more floating-point operations are performed for each piece of data retrieved from memory. This leads to improved strong parallel scalability, better throughput on GPU platforms and increased computational efficiency. To achieve the necessary meshing flexibility, high-order discretization capabilities and high performance including both on- and off-node parallel scalability, MARBL makes extensive

use of the modular, open source finite element library MFEM as well as the scalable linear solvers library, Hypre.

1.10 Ardra

Ardra is a multi-dimensional, scalable, massively parallel code for performing deterministic, discrete ordinates (SN) neutron and radiation transport calculations. It supports high resolution in all aspects of the solution phase space (space, energy, direction), employs a highly efficient, scalable MG-DSA (MultiGrid-Diffusion Synthetic Acceleration) preconditioned solver, and supports both CPU and GPU architectures. The modeling of the transport of neutral particles (e.g., neutrons and photons) is of importance to many scientific and engineering activities, including but not limited to reactor and shielding design, development of medical radiation treatment, nuclear well logging applications, and nonproliferation applications. In particular, Ardra is an important tool at LLNL for neutron criticality, shielding, and dosage calculations in support of DOE and DoD missions.

1.11 Mercury

Mercury is a production Monte Carlo particle transport code under development at LLNL for over twenty years [9]. Mercury can transport neutrons, photons, and light element (hydrogen and helium) charged particles. Both fixed source and criticality problems are treated. Mercury models problem geometry using either a constructive solid geometry and/or a mesh representation. Mercury can use either continuous energy cross section data or a hybrid approach in which multigroup cross sections are sampled from a histogram while the collision kinematics uses continuous energy. Nuclear/atomic cross section data access and particle collision physics are provided by the legacy Monte Carlo All Particle Method (MCAPM) library [10] or the modern GIDIplus project [7] containing the GIDI/MCGIDI libraries. Mercury is parallelized via domain decomposition and domain replication with dynamic load balancing. Mercury uses MPI parallelism across compute nodes and MPI or OpenMP threading on-node to target CPU cores and CUDA to target GPUs. Mercury is written in C++ with a Python user interface and runs efficiently on CTS massively parallel computing platforms. Mercury has been ported to the ATS-2 Sierra GPU architecture; however, obtaining good GPU performance is a challenging and ongoing research effort.

1.12 Summary

The PEM libraries are actively supporting the execution of IC codes on GPU platforms by providing APIs and implementations that are suitable for GPU execution. Section 2 contains more specific information about the GPU ports of each library. Table 2 summarizes the characteristics of the PEM libraries discussed in this section.

	Lines of code	Language	Effort Level	API Type
LEOS	150,000	C++	1.5–2.0 FTE	Scalar and Array
MSLib	150,000	C/C++/ Fortran	1 FTE	Array
TDF	10,000	C	< 0.5 FTE	Scalar
Opacity Server	35,000	C++	1 FTE	Array
GIDIplus	30,000	C/C++	0.3 FTE	Scalar
Cheetah	360,000	C/C++	1–3 FTE	Scalar and Array

Table 2: Code characteristics of PEM libraries

2 Progress and Challenges Porting PEM Software

Completion criteria #2 of the milestone is:

> 2. Discuss the progress and challenges of porting PEM software integrated into IC software to GPU architectures, including information about programming models employed, GPU memory sharing between PEM libraries and host codes, and a summary of algorithmic challenges.

This section satisfies the criteria by describing the porting progress and challenges for PEM libraries in the scope of this milestone. These descriptions include information about state of the GPU port, the GPU programming and memory model employed, algorithmic challenges and restructuring, and data size and memory sharing considerations.

2.1 LEOS Project

The GPU ports of LEOS and LIP are now sufficiently complete to allow application codes to run most problems of interest. Some specialized and rarely-used capabilities (e.g., support for some multitable features) still need to be completed. The porting proceeded in stages: forward and inverse array lookups, tcalc using a composite method with two forward array lookups, single-point lookups for scalar data, a native tcalc method, and finally multitable forward, inverse, and tcalc lookups. The missing pieces are single point lookups for functions of multitable materials, some setup methods for interpolants, and using the std::vector interface for lookups. The latter is constrained by the need to allocate the std::vector's data in GPU memory. If that is done, possibly using the CHAI library, then it may be possible to extract the pointers to device memory and use the array interface for the evaluation.

The LIP library performs the bulk of the computations. At the start of the porting process, LIP was a C library which had grown organically over the years. As a result, the interface was inconsistent, and the addition of new interpolation methods resulted in significant code duplication. The library was therefore not suitable for immediate porting to a GPU-based environment. We had discussed the need to rewrite LIP in C++, purely based on code maintainability considerations, before the advent of GPU machines; this accelerated the process. We decided to rewrite in C++ using template-based generic programming so that we could plug in different policies for interpolation to offer as much flexibility as possible in the model we needed for GPU ports. We avoided polymorphism and virtual functions because of known challenges with this approach on GPUs. The use of policy classes for the different interpolation schemes allowed us to concentrate initially on a few most commonly used interpolation schemes: bihermite, bimonotonic hermite, and bilinear. Once the framework was in place to handle these classes, it was straightforward to add classes for other interpolation schemes, or variants of existing schemes. We also added (lookup) policy classes to the framework, to control how input values are located in the axes.

As originally coded, a LIP interpolation call had three major loops: 1) compute the indices of the input density values into the density axis, 2) compute the indices of the input temperature into the temperature axis, and 3) use the indices in a loop over the input (density, temperature) pairs to evaluate the output values. We decided to adopt RAJA for_alls to replace the for statements, as this allowed us to make minimal changes to the existing source code. Instead of using raw RAJA coding, we used a set of wrapper classes on top of RAJA developed for the ASC code Ares which turn into regular for statements if RAJA is not used to compile the code base. As the porting process continued and the growing need to support single point lookups on the GPU became evident (e.g., when LEOS is called from MSlib), the coding in the policy classes was refactored so that the bodies

of the RAJA for_alls were turned into device callable functions. These functions are then reused for single point interpolation.

We made two significant algorithmic changes for GPUs. The first was to add a full binary search lookup policy for use on the GPUs. This allows the lookups to run independently on each GPU thread. The old lookup policy was similar, but it contained a legacy serial dependency since it used the result of the last lookup to seed the search. The second change, which occurred after the scalar interface was implemented for GPUs, was to remove the temporary arrays created to hold the indices found by the lookups and to use single point calls of the lookup policies in the for_alls with scalar values for the indices. This did not reduce CPU efficiency since the CPU coding passes the indices found on the previous iteration into the lookup calls.

Memory management is very important for code stability and efficiency. In the standard use case of full setup, the interpolation policies precompute interpolation coefficients and store them for later use. Depending on the problem, this permanent memory can be 400–500 MB in size. LEOS also uses temporary memory for arrays used in evaluation calls. The amount of temporary memory needed can be anywhere from zero to 10x the size of the input array. On Sierra this could be up to 100–200 MB of memory for problems with 1–2 million zones per GPU. For good performance, both permanent and temporary memory need to be located where the code will be executed. Additionally, if the host code is using memory pools, LEOS will benefit from them as well. Thus, the host code needs to control selection of memory locations. The original C LIP coding has some macros for using CPU shared memory (sharing memory between multiple MPI ranks on the same node) for the permanent data, but these were insufficient for the needs of the GPU code.

A new LIP::MemoryManager class was added to the C++ version of LIP to handle controlling the location of memory allocations. The initial implementation wrapped malloc/free, CUDA memory calls, and the CNMEM library. Selection of location was controlled by a new flag which was passed into LIP via new members in the LIP setup and lookup option objects. An LEOS::MemoryManager class which simply wraps LIP::MemoryManager calls was added to support the memory needs of LEOS. The LEOS options objects were also modified to pass in flags for memory control which are then propagated into LIP. The managers also support CPU shared memory via the ASCMemory library for permanent memory.

As the Umpire library gained both features and popularity, the memory manager coding was extended to use Umpire for memory management. The flags for memory control in the option objects were changed from an enumeration type to ints so that Umpire allocator IDs could be passed into the MemoryManager classes. Additional data members and functions to get/set the IDs were added to the MemoryManager classes to control how Umpire is used. Currently, three modes are supported when Umpire is compiled into the code base: 1) Pass in Umpire IDs and use them to get Umpire allocators to use, 2) pass in values from the enumerated type and use basic Umpire allocators (e.g., "HOST", "DEVICE", "UM"), and 3) pass in enumeration values and do not use Umpire. Using case 1 allows LEOS to use Umpire memory pools, if the host has created them. Case 3 is currently the only way to use CPU shared memory.

A new interpolant class was added to efficiently handle GPU evaluation of phase fractions for flattened multitable materials. This class, which combines continuous interpolation in 2D space with a third discrete axis, can also be used in the future to support other data objects with similar structure, such as multigroup opacity tables.

In LEOS, the changes needed were fairly modest. The for loops in the Prehooks and Posthooks (42 simple loops) were converted to RAJA for_alls. Also, 50 loops were converted for multitable and tcalc lookups. The new LEOS::MemoryManager class was used to manage memory for the temporary arrays needed when hooks are used. The option objects were modified to handle the memory flags, and to pass those flags to LIP.

Originally, a compile time choice determined where LEOS and LIP would execute (CPU or GPU). This has been changed at the request of one of the ASC hydrocodes to allow the choice to be made at runtime for each individual setup call and each individual lookup call. A new flag was added to the option classes to specify where the call should execute. The RAJA wrapper classes were modified to select the correct implementation for the location specified by the new flag.

Initial work for El Capitan has started. So far, only two places have been found where changes needed to be made. First, the memory managers were modified to support either CUDA memory calls or the HIP equivalents. Second, the implementation of some functions needed to be moved into header files to prevent link time errors with hipcc. These straightforward changes have been merged into the develop branch and will be included in the LEOS 8.4 release. Full support for El Capitan will require updated versions of RAJA and Umpire that include HIP support, but no additional LEOS or LIP source code are anticipated.

2.2 MSLib

A fat-kernel port of a subset of MSLib sub-model options has been completed. This work was meant to get initial GPU capability in place, particularly for more commonly used models, to mitigate the performance penalty associated with the use of MSLib on next-gen platforms. Details of the supported sub-models are captured in the "GPU Port" area of the MSLib confluence page, and the `Get_GPUSupported` function can be called on a MSLib model object to determine whether it can execute[3] `getResponse` and `map` type calls[4] on the GPU. Notably, MSLib models that make use of LEOS, Cheetah, or the EOS callback feature do not yet work on GPUs. Capabilities for GPU utilization of LEOS are in progress, making use of device-callable single-point functions (Section 2.1). Other than the utilization of LEOS, all of the most commonly used sub-models within MSLib have been ported in the fat-kernel approach.

In the fat-kernel approach there is a loop high in the call stack, and vectorization over zones is not propagated down the call stack. Given the depth of the call stack and the lack of clear hot-kernels on which to focus attention, this fat-kernel approach was an expeditious means of porting key models with the modest effort available. In some cases, the call stack has a depth of ∼10 below the loop over zones from the host-code. Functions were reworked to be device-callable, and some cleanup and restructuring was required.

For many legacy models there are no near-term plans to perform a GPU port. One of the crystal-mechanics-based models in the Fortran `evpc` module was rewritten in templated C++ as part of the ECMech library (which uses the same interface as MSLib and will for purposes of this discussion be considered as part of MSLib). The initial GPU port into ECMech was supported by the Exascale Computing Project (17-SC-20-SC) and the incorporation for hydrocode used was supported by ASC/PEM and the Joint DoD/DOE Munitions Program. Note that while the core of ECMech is open-source [5] the hydrocode-specific interface is not.

The SNLS solver is templated on the size of the non-linear system, but in one of the porosity-based sub-models the system size is not known until runtime. For that sub-model, we use a switch block for the instantiation of the templated solver.

In the initial implementation for stand-alone testing, MSLib managed local host/device memory operations with memory allocations via `cudaMalloc` and `cudaMemcpy`. Subsequently, there were

[3] To some degree, execution on the CPU or the GPU can be controlled at runtime using `setExecutionStrategy`; with this feature facilitating timing comparisons and debugging.

[4] While the principal path to GPU utilization is through the vectorized C++ calls, there is some exploratory work in which the MSLib C interface is used to call `MS_MaterialModels_D` (instead of `MS_MaterialModels`) inside of a host-code kernel.

improvements to allow device pointers to be passed directly to vectorized API, eliminating most data motion costs.

For performance improvement, we are exploring pushing vectorization down the call stack as an alternative to the fat-kernel approach for a subset of the sub-model options. This would be done using some combination of RAJA, Umpire, and either CHAI or CARE. Exploratory work has been performed on vectorization of SNLS, with ECMech being used as a testbed. For the full vectorization, initial focus would be on capabilities that work with `ysmodel` 115 (YSMODEL_J2_VOID). This sub-model captures some complexity but is still much simpler than, for example, a porosity-mechanics-based model. Even with this narrowed focus, vectorization is a significant undertaking given that most of the call stack is in C and would require migration to new class structures in C++ as part of this effort. This porting effort to C++ will also improve functionality and maintainability of MSLib. Once the initial port is finished, future sub-models should require less effort to port due to many of the shared internal models having already been ported.

Finally, error reporting on GPUs remains incomplete. On CPUs, the error reporting had been based either on exceptions or on `MPI_Abort`. Currently hard errors in MSLib have become soft errors when running on the GPU (meaning that an error message is printed), with potentially unexpected results in the execution after the soft error. In the future, we plan to rework the error checking so that there is an array of status codes. This array would then be checked within MSLib on the host.

MSLib does not use data tables and the memory needed by MSLib model parameters is typically only a few kilobytes or less.

Currently, a CUDA call is used for sending the MSLib model data to the device. In the migration to El Capitan, this will have to be refactored to work with HIP. As part of this rework, we are looking at replacing these raw CUDA calls with Umpire and RAJA. Umpire will also enable the use of temporary memory pools already designated by host codes. There are also plans to explore whether CARE can replace the current MSLib-specific loop abstraction and memory management constructs with a community-supported solution.

2.3 TDF Project

As explained in Section 1.3, TDF consists of two libraries, TDFgen and TDFlib, but only the TDFlib library is used by application codes. Aside from CPU-only setup functions, applications typically call TDFlib functions inside loops (or, on GPUs, inside kernels). Hence the process of porting TDFlib to GPUs was driven by the need to make reaction-info and look-up functions device-callable. It was also necessary to place data tables in unified memory to make them GPU accessible.

A CUDA-based port of TDFlib is sufficiently complete to support all current production use cases of all supported host codes. Four of the nine reaction info functions and eight of the ten look-up functions have been made device-callable and all necessary reaction data tables are device-accessible. Although there may be a need to port additional routines in the future, the current port has been vetted with current needs of multiple applications.

Most device-called routines in TDFlib are simple interpolation or lookup routines. In most cases, porting these functions to CUDA required little effort beyond marking them as __host__ __device__ functions. Function bodies typically remained unchanged.

The main challenge of porting TDFlib to CUDA was moving the hierarchy of C structs representing the TDF data to the GPU. Because the reaction data is represented by a hierarchy of C structs connected by pointers, copying the the reaction data from CPU to GPU at setup time requires effectively a “deep memory copy” of the reaction data.

A GPU accessible version of the reaction data is created using serialize/deserialize routines separate from those used to broadcast the reaction data over MPI. This process converts the original CPU data structures with over 100 smaller memory allocations into a single large pool (per reaction) that is big enough to hold the entire set of TDF data.

After reaction data is read in from disk, if compiling for the GPU, a single allocation per reaction accessible on both CPU and GPU is created using cudaMallocManaged and data is copied from the original CPU structures. After the data is copied, the original memory (with the 100+ separate allocations) is entirely freed, and both the host CPU and device GPU access the version in managed memory.

Currently, the MPI broadcast of the TDF data from MPI process 0 to all other MPI processes takes place after the above copy. During the broadcast, rank 0 first sends the size needed for the single allocation the other ranks, which allocate managed memory. The TDF data is then sent and the data is copied into managed memory as described above. Thus all MPI ranks work with the same type and layout of managed memory, with the addressses set as offsets which are correct for their MPI process.

It should be noted that the MPI broadcast currently involves an additional copy. The use of memory allocated with cudaMallocManaged with MPI communication is not guaranteed to work correctly unless specific job submission (jsrun or lrun) options are specified by the user. To avoid this requirement (and difficult to diagnose errors if the user fails to supply the correct flags) the data is copied into regular host memory for the communication, and freed when done. Since this extra copy appears only during setup, the cost is acceptable for portability and usability goals.

In 2020, the GPU port was reviewed and updated based on some observations of leaked memory and segmentation faults under some scenarios. Extensive debug code was added to check that the memory copies are correct. This included routines to do a detailed comparison of the original memory to the single pool of managed memory to verify the data matches after the copy. Also in 2020, more routines were made device-callable as a new code required additional TDFib functions on the GPU. Some of these functions were refactored to avoid the use of certain functions from the libc c-string family of functions that are not supported on the GPU. The library was refactored to perform such work on the host only. This work is reflected in the unofficial 2.3.62 version of the TDF library in the git repository.

The memory footprint of TDF reaction data is very small, only about 11 MB. Because this footprint is so small there is no need to coordinate memory allocation with host codes or with other libraries.

The El Capitan port of TDFlib has not progressed past the earliest planning stages. However, considering the simplicity of the CUDA code in TDFlib and the broad compatibility between the HIP and CUDA programming models, we anticipate very few issues creating an implementation that will be portable between Nvidia and AMD hardware. The planned implementation of managed memory on El Capitan also has all of the features necessary to support the current design of TDFlib. We also plan to investigate using Umpire pooled allocators to significantly simplify copying of reaction data into managed memory. It should be possible to avoid the current copy-before-broadcast approach and only serialize the CPU version of the reaction data structures for MPI broadcast. The deserialize function can then unpack the data into memory obtained from an appropriate Umpire allocator. This would completely eliminate the separate copy functions that are currently used.

2.4 Opacity Server

The Opacity Client library currently has no support for GPUs. The conventional wisdom has been that the time spent in the client library will be negligible compared to the total run time of physics packages that require opacity data, (e.g., Teton SN transport). Hence, developing the ability to perform lookups on the GPU has not been a high priority. However, as various physics packages have started to obtain good speedups on the GPU it has become clear that leaving the lookup code on the CPU will not meet performance goals. To understand why, consider a physics package that runs 10x faster on 4 GPUs (1 Sierra node) than on 40 CPUs (also 1 Sierra node). Speeding up the physics makes the CPU-only lookup code 10x more expensive relative to physics. The problem is made worse by the fact that host codes running on GPUs typically run only 1 rank per GPU. Hence, the lookup code (which is not threaded) will be limited to 4 CPUs per node imposing another factor of 10x slowdown compared to the CPU-only baseline. With the additional cost of moving data from GPU to CPU, the fraction of time spent in lookups can grow by over 100x and times that were once negligible can become quite painful. This section highlights the GPU development plan for the Opacity Client and the key steps in making the library work on the GPUs.

The ability to serialize tables and move data around in memory already exists and is currently used for sharing tables across MPI processes. This existing code can be refactored to move data to the GPU. The serialization logic will be modified to use Umpire to allocate memory on the GPU. An OPAC::Table_Data object, which contains the opacity data and axes for a specific table, will be serialized and copied to GPU. Once the serialized table is copied to the GPU, a simplified OPAC::Table_Data object will be constructed on the GPU with the minimum interface a lookup needs to access the data in the table. Logic for reading the table from a file and reordering the axes will remain on the CPU only. The OPAC::Lookup_Table object itself will exist on the CPU containing a pointer to the GPU OPAC::Table_Data. The existing lookup logic works by looping over each point passed to the lookup function. For each point, it finds the bounding box from the OPAC::Table_Data by running up each axis. Next, it passes the bounding box points to the interpolation function along with any information about inputs or outputs being logarithmic or linear. Within the interpolation function, the interpolation is done for each group, since tables returned by the server are often group averaged. The lookup function can be parallelized by point or by group. It is likely that both options will be implemented to allow the host code to decide which option works best for them. The OPAC::Lookup_Table class will use RAJA to parallelize the lookup function. Using RAJA to launch kernel functions and Umpire for memory management has the benefit that all code will be CUDA/HIP agnostic for compatibility with El Capitan.

The Livermore Interpolation Package (LIP) is being considered as an alternative interpolation library. It has already implemented various interpolation functions on the GPU and it seems feasible to replace the client interpolation logic with LIP, however, LIP currently lacks several required features. First, the current table implementation assumes an arbitrary number of dimensions, but LIP currently supports up to three dimensions. Although the full generality of an arbitrary number of dimensions is unnecessary, a 4-dimensional table is a minimum requirement. Second and more importantly, LIP does not currently support different off-table behavior (OTB) depending on the direction a lookup is off table. Engagement with the LIP development team is ongoing to determine the best path forward to running on GPUs. The LIP development team is open to collaboration to get both libraries working together.

The memory footprint of opacity data is modest. In typical production problems opacity table data will occupy roughly 5–60 MB per material with most use cases falling at the lower end of that range. This could change in the future with the introduction of NLTE (non-local thermodynamic equilibrium) tables. Such tables can occupy up to 5 GB. For these larger tables is clear that some

kind of memory sharing across MPI ranks would be highly desirable, especially on CPU platforms or even Sierra. However, as we move to El Capitan, all except the largest tables occupy a small fraction of GPU memory so such coordination may be unnecessary.

Early attempts to port the client to GPUs have identified issues stemming from the reliance on inheritance and polymorphism for data classes that must exist on the GPU. The client library supports different table types, which are implemented as different derivations of OPAC::Table_Data. OPAC::Table_Data objects are also responsible for reading and organizing the data by axis when it is read from the file, which occurs only during initialization (and on the the CPU). To handle these issues, the logic that deals with accessing the data after it has been read has been split into an accessor class, which will be created on the GPU. Reliance on smart pointers and other STL classes also causes difficulty. This issue can be resolved by simplifying the classes that will exist on the GPU to explicitly remove these dependencies.

2.5 GIDIplus Project

Of the various nuclear data libraries in use at LLNL, only the GIDIplus project has been ported to GPU-enabled advanced architectures. Supporting only one nuclear data code saves considerable effort, especially since the legacy libraries like NDF and MCAPM are no longer actively maintained. GIDIplus is made of many related libraries, of which only MCGIDI needs to run on the GPU as well as CPU. The other components of GIDIplus, including GIDI, are expected only to run on the CPU.

GIDI is an I/O library that runs only at problem start up and will not benefit from GPUs. However, it still can have significant impact on problem run times. The time required for GIDI to read nuclear data files has been a concern with early versions of GIDIplus. Compared to the older libraries libNDF and libMCAPM, load times in GIDI are significantly longer. Fortunately, recent efforts to improve GIDI load times have been successful, and are expected to be released soon for general use. These efforts are described further in Section 4.5.

The GPU port of MCGIDI is complete. Porting MCGIDI to GPU architectures required three categories of changes: adding CUDA device declarations to functions, replacing the standard template library, and supporting C++ object construction on the GPU.

Functions that need to be callable on the GPU were marked as device functions using a macro that expands to "__host__ __device__". This macro should allow for porting to a HIP-based architecture in the future. Marking functions is a relatively easy task. However, any function running on the GPU device can call only other functions that run on the device. Member functions in a number of commonly used C++ library classes (such as std::string) are not device-callable. Hence, this step required replacing such classes with GPU-compatible versions.

Any C++ classes or functions that used std::string or std::vector had to be modified to use GPU compatible replacements, MCGIDI::String and MCGIDI::Vector, respectively. The code for these routines was borrowed from the Monte Carlo transport group, which faced similar issues. The main difference in between the std:: and the GIDI routines is that internal data pointers are initialized to nullptr and are assumed to be allocated only once. The first assignment or resize performs the allocation. These replacement classes are not thread safe, but once they are initialized they contain only read-only data so thread safety is not a requirement.

This task required most of the porting effort. Coping data associated with MCGIDI objects from CPU to GPU required a deep-copy. Fortunately, a serialize/deserialize infrastructure originally set up for MPI broadcasting was available to help. To support serialization for MPI broadcast, each class implements a serialize() function that takes a mode argument. For each variable in a class, we have a macro expansion conditional behavior based on the mode. Originally, the modes were

Count, Pack, and Unpack.

In the Count mode, the macro expands out to count the number of integers, doubles, characters, and 64-bit integers needed to represent all the data. After allocating data buffers, the Pack mode writes each class variable into a buffer of corresponding type. These buffers can broadcast over MPI or copy to the GPU device. The last of the original modes was Unpack. This unpacks data in buffers and reconstructs the class with all fields. Some memory operations may need to happen in this stage. For example, if a class has a MCGIDI::Vector, it will need to resize this object to the correct size as passed through an int array.

Two additional features had to be added to support copying data to a GPU device. While the original Unpack memory allocation can work on the GPU, it is very slow and inefficient to call malloc() on the device. The first main change was to use placement new operation to specify a memory location to construct the class. Memory pointers were updated based on the size of the class and any data contained within. In addition, the data also had to align on 8-byte boundaries. The Unpack routines were updated to handle memory address pointers with 8-byte alignment. On a GPU device, data array copies were sped up using a threaded copy in chunks of 32 threads (size of a warp). To better support unpacking to a memory block, a new mode called Memory was added to the serialize routine. This calculates the size of a continuous data block that can be unpacked into.

The memory requirement for the cross section tables used by MCGIDI varies considerably depending on the type of cross sections in use. Multi-group tables typically require roughly 100 MB. Continuous energy tables can be considerably larger and could consume 5 GB or more.

Looking toward El Capitan, it should be easy to replace CUDA constructs with HIP to target AMD GPUs. It may be necessary to adjust details such as data alignment, but few difficulties are expected.

2.6 Cheetah

There are numerous challenges to porting a sophisticated advanced physics package such as Cheetah to advanced computer architectures based on GPUs. The original library interface relied on a single function call per zone (hydrodynamic finite element), which was not well suited for advanced GPU architectures that efficiently process long vectors of operations. To improve Cheetah 's performance on GPU architectures, the Cheetah team developed a vector interface, which allows millions of zones to be processed together. The vector interface required partitioning data which varied by zone from data that was common to all zones. This partitioning allowed for efficient memory use in a vector environment. Vector loops are dispatched to the GPU using RAJA to abstract CUDA calls, thereby maintaining code readability and avoiding vendor-specific code as much as possible.

At the present time, Cheetah is fully functional on GPU-based computers and nearly all common algorithms and inputs are supported. Routines that perform lookups in the the equation of state (EOS) database (called cache by the Cheetah team) have been vectorized and optimized for the GPU. In the case of a cache miss, the physics calculations required to fill a database entry currently occur on the CPU and can require substantial wall clock time. However, for typical GPU problem sizes of 10^6 elements or more, the vast majority of lookups are satisfied in cache and the cost of the relatively rare misses is easily amortized. Zones tend to be highly correlated, so as the zone count grows the efficiency of the cache increases. This trend is clearly shown in Figure 13 in Section 3.14.

Unified memory was chosen for use in Cheetah to simplify porting and maintain coherency between equation of state cache data on the CPU and GPU. Unified memory is allocated through Umpire calls. For Ares, the host code provides Cheetah with a pre-existing unified memory allocator, while in ALE3D no such allocator exists, so Cheetah creates its own allocator. Optimizations

were performed to reduce large numbers of small allocations. In particular, each cache entry was a single allocation in the CPU version of Cheetah, while for GPU systems blocked allocation of flattened data structures was used.

It is well known that optimal memory layout often differs between CPU and GPU. Because Cheetah targets both CPU and GPU platforms, it is desirable to retain flexibility in memory layout without forfeiting performance or maintainability. To that end, a set of matrix abstractions was added through a series of policy-based class templates implemented using the features of C++11. The design of these templates was motivated by the multidimensional span (mdspan) ISO C++ standards proposal [1]. A non-owning matrix view is defined with template parameters specifying the scalar data type, the static and dynamic matrix extents, a layout policy, and an accessor policy. The layout policy specifies the mapping of dimensional indices to the index of the corresponding element in vectorized storage; row-major, column-major and arbitrary strided storage orders are supported. The accessor policy specifies whether restrict (assumed non-aliasing) semantics or bounds checking should apply to element access. An owning matrix type is also defined that is additionally parameterized with an allocator, providing support for the various allocation strategies used by the code in different configurations. Compile-time information is leveraged for optimization where possible; for example, storage for the row dimension and stride can be elided when the matrix is statically defined as a row vector. Unsafe implicit conversions, such as assignment of a column-major matrix view to a row-major view, are either not defined or excluded from applicable overload sets. In addition to the significant performance benefits afforded by optimizing memory layout through these matrix abstractions, they also enable scope-based resource management and add robustness by associating memory layout and extents with the data and its references.

For cache lookups, great pains were taken to minimize unnecessary usage of GPU global memory. Manual inlining was used to reduce memory spilling during function calls and reduce the use of temporary data buffers. To further speed up the code, the dimension of the cache was translated from a run-time constant into a compile-time constant through C++ templating and a function dispatch that calls the appropriate concrete template instantiation at run-time. This code transformation had the added benefit of replacing many global memory buffers with register storage and thread local memory. It also allowed the compiler to unroll all loops over the dimension of the cache, leading to fewer memory operations and fewer branching instructions.

In addition to providing equation of state calculations, Cheetah can compute the time evolution of species conversion through kinetic reactions. Because of the disparate time scales between them, the chemical kinetics and hydrodynamics are usually coupled via operator splitting methods where each zone is treated as an independent/isolated system. When leveraging GPU hardware to solve these problems, each zone can be solved by an individual thread and no communication is necessary between threads. However, due to the branching decisions made in time integration algorithms to satisfy error tolerance (or accuracy) requirements, thread divergence can become an issue such that neighboring threads no longer benefit from data parallelism.

To maintain thread coherence, Cheetah 's strategy for solving kinetics on the GPU is to batch together all of the zones to be solved on a GPU as a single time integration problem. Batching eliminates thread divergence, as all zones will be subject to the same integrator logic branches. A trade-off with this approach is that, if treated independently the zones will not all take the same number of kinetics sub-steps, while when batching the number of sub-steps is dictated by the stiffest zone. This means that the total number of kinetics sub-steps on the GPU will be higher than on the CPU, for constant error tolerance. The time overhead resulting from these extra sub-steps remains to be quantified, but is expected be small, and less than the gain from maintaining thread coherence and data-parallelism. The code changes necessary to achieve this strategy involved

creating a call path for computing the right-hand-side (RHS) of the kinetics time derivative for all reacting species and all reacting zones. The resulting function can then be passed to the time integration algorithms which only need to be modified to use RAJA loops to compute future states. Within the RHS function, new functions were created that vectorize all of the rate laws supported in Cheetah. Restructuring was also necessary to efficiently compute EOS information within the kinetics calculation, and to compute and return kinetics state variables (e.g., viscosity or specific heat) requested by the hydrocode. All memory used in these routines is CUDA managed memory and data transfers are minimized by only accessing the vector data within RAJA-CUDA loops (i.e., in CUDA kernels).

The Cheetah team is working on two new capabilities that were not employed for this milestone. They were deemed to be less ready for use at scale in multiphysics applications. The first is the option to completely pre-fill the cache database on the GPU. Cache pre-filling, when deployed, would completely eliminate the possibility of a cache miss. This option is especially attractive for problems with high node counts since the cache filling process is embarrassingly parallel and can easily be spread across all nodes of a calculation. The second is a cache-based approach to accelerate the solution of chemical kinetic equations.

Cheetah's memory use includes 1–100 MB for the EOS cache, depending on input, as well as some amount of memory for the arrays used to process zones. The latter is believed to require a few hundred megabytes per hundred thousand zones. However, this has not been rigorously measured and is likely subject to change as the Cheetah team continues to make improvements in the code.

It is anticipated that the RAJA abstraction will make the transition to El Capitan relatively easy. Cheetah also uses some asynchronous CUDA calls that will need to be ported to HIP in the future.

2.7 Summary

Table 3 summarizes the GPU porting status of the PEM libraries in the scope of this milestone. With the exception of the Opacity Server, all have functional GPU ports that provide at least some of the library's capabilities. Of these, LEOS and TDF are the most mature and are in routine production use. The GPU ports of MSLib, MCGIDI, and Cheetah are still in pre-production. The Opacity Server GPU port has justifiably been deferred as lower priority until now, but plans to create a GPU port are taking shape.

Most of these libraries have well understood and localized data structures that are amenable to explicit copy from CPU to GPU memory. There is a growing appreciation for the role of Umpire in helping to coordinate memory use between host and libraries, especially memory that is needed for temporary storage.

Looking toward El Capitan, few problems are expected. Moving to a new GPU architecture is expected to require much less effort that the initial conversion from CPU to GPU. The libraries that utilize RAJA will be able to take advantage of the portability built into the abstraction to target AMD GPUs. Those that are written in CUDA expect to easily take advantage of AMD's HIP portability strategy.

	Parallel Model	Memory Model	Typical Data Size	Porting Status
LEOS	RAJA	Host-controlled	500 MB	In Production
MSLib	CUDA	Explicit Copy	Nearly Zero	In Development
TDF	CUDA	Explicit Copy	11 MB	In Production
Opacity Server	CUDA	Explicit Copy	5 MB–5 GB	In Development
GIDIplus	CUDA	Explicit Copy	100 MB–5 GB	In Development
Cheetah	RAJA	Managed Memory	1-100 MB	In Development

Table 3: Porting status of PEM libraries

3 Test Problems

Completion criteria #3 of the milestone is:

> 3. Define targeted test problems for assessing computational performance that exercise the PEM capabilities studied in this milestone.

This section satisfies this criteria by providing the required test problem definitions as well as performance results. The IC codes and PEM libraries used by each test problem are shown in Table 4.

The test problems in this section were run on a variety of LLNL computing platforms. The systems used can be divided into two classes: Commodity Technology Systems (CTS-1) and Advanced Technology Systems (ATS-2). CTS-1 refers to the first series of CTS systems delivered starting in late 2016. A second series of CTS systems (CTS-2) are expected to arrive in March 2022. ATS-2 refers to Sierra, which is the second ATS machine delivered to the NNSA labs. (Trinity was ATS-1.) Throughout this section, CTS-1 will refer to CPU-based platforms and ATS-2 will refer to machines that use same node design as Sierra that is based on the Nvidia V100 GPU. Table 5 shows the characteristics of each system that was used for test problems.

Throughout this report, CPU to GPU speedups are reported on a node-to-node basis. That is, by comparing performance on one node of CTS-1 to performance on one node of ATS-2.

	ALE3D	Ares	MARBL	Mercury	Ardra
LEOS		Hotspot	Shape Charge		
MSLib	Jetting, Non-Local				
TDF		Hotspot			
Opacity Server		Hotspot			
GIDIplus				Godiva	NIF Chamber
Cheetah	Barrier				

Table 4: The test problem matrix shows the IC code and PEM libraries used for each test problem.

System	Type	CPU	GPU
RZGenie	CTS-1	Intel Broadwell (36 cores per node)	None
RZTopaz	CTS-1	Intel Broadwell (36 cores per node)	None
RZAnsel	ATS-2	IBM Power 9 (40 cores per node)	Nvidia V100 (4 per node)
Lassen	ATS-2	IBM Power 9 (40 cores per node)	Nvidia V100 (4 per node)

Table 5: Compute systems used for this milestone.

3.1 Hotspot Problem

Name: Hotspot Problem (Igniting ICF capsule)
Size: 18,821,096 zones
Code: Ares
Libraries: TDF, LEOS, Opacity Server
Physics: Hydrodynamics, thermonuclear burn, grey radiation diffusion, electron heat conduction

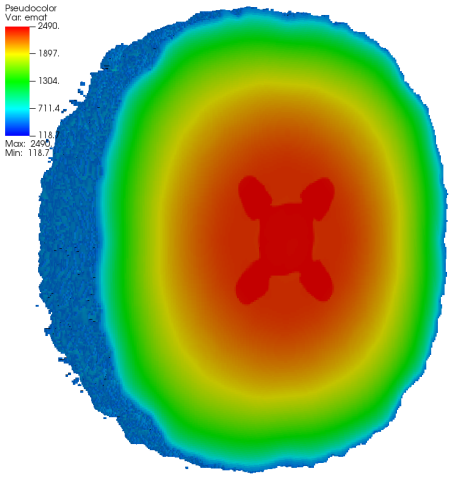


Figure 1: Snapshot of a hotspot simulation showing the electron energy

Description

This problem is used to represent the conditions in the hot spot of an igniting ICF capsule and tests whether the coupled physics that matters in this regime have been modeled adequately. The problem is simplified by initializing the geometry with a spherized version of the capsule implosion at the time of ignition. The conditions of the materials are also initialized to be consistent with what would be observed at that time.

This problem exercises Eulerian hydrodynamics, thermonuclear burn, grey radiation diffusion, electron heat conduction, opacity server opacities, and LEOS equations of state. It contains 18.8 million zones.

Additionally, as a variant, the Opacity Client can be used to calculate the interpolations for the opacity lookups, so that the performance of those lookups can be measured in a realistic setting.

Motivation

- Although simplified, this problem exercises all of the physics used for ICF capsule problems and should give an accurate context for how PEM libraries are used in this regime.
- Running this problem on Sierra and El Capitan offers increased resolution in the form of higher zone counts. This scaling study coincides with the range of number of zones per GPU that users are running (i.e., 500k–4.7M zones per rank).

- Problems with this set of physics are currently being run in production regularly and are considered state of the practice.

3.2 Hotspot Results

We performed a strong scaling study on RZAnsel (ATS-2) and RZGenie (CTS-1). The code on RZAnsel ran on 1, 2, 4 and 8 nodes using 4 MPI ranks per node. On the RZGenie, we ran it on 4 and 8 nodes for comparison.

For the variant that uses the Opacity Client for lookup functionality, we used only 2 nodes and 8 MPI ranks to simplify analysis and only 10 cycles due to computational cost. This is not the default method of running problems nor is it recommended.

TDF Analysis

Tables 6 through 8 show the performance of TDF on an 18,821,096 zone Ares simulation of the Hotspot Problem on RZGenie and RZAnsel. The total runtime of the simulation, the runtime of TDF in isolation, and the percentage of the total runtime are listed for 1, 2, 4, and 8 node runs. In Table 7, the run times are given for minimum, average, and maximum times across ranks. In Table 8, the runtime is given as a percentage of the total runtime for the maximum TDF runtime.

For the same number of nodes, the total simulation time is reduced by about a factor of ten when going from the CPU-based RZGenie to the GPU-based RZAnsel. TDF is called 19,060 times within the simulation. Its runtime is also accelerated by roughly about the same factor of 10, resulting in TDF remaining a negligible percentage of the runtime on both CPU and GPU systems. In Table 8, we see that the total TDF cost is always within 0.15% of the total runtime even in the maximum (worst) case.

As the problem is strong scaled, we see that the relative variation in runtime between the minimum, average, and maximum cases grows significantly. This is because TDF is only needed for zones with materials involved in TDF reactions, which is not all zones. As the problem is strong scaled, an increasing number of ranks do not contain zones involved in TDF reactions, driving their TDF cost to zero. This results in less than ideal strong scaling of TDF cost. However, if we look at the average cost of the TDF functions over all processors, we see that we are strong scaling very well. Overall, we see that even on ranks for which the TDF cost is maximum, the percentage of TDF cost to overall cost remains negligible, so there is currently no concern about needing to optimize TDF further, even looking forward to El Capitan.

LEOS Analysis

Table 9 shows the performance of LEOS on the Hotspot Problem. We see that at 4 and 8 nodes, we have a 13.3x and 9.7x speedup respectively. We also see that the percentages of overall runtime between the CPU and GPU runs are very close, within 0.1%, and following the same trends. The process-to-process time spent in these routines are fairly consistent, so it doesn't appear that load imbalance is a large factor here.

Since the speedups are tracking the hydrodynamics performance well, and the percent of total runtime is very small, we can say that LEOS is performing well and that we do not anticipate the need for further optimization.

	Total runtime (s)			
	1 Node	2 Nodes	4 Nodes	8 Nodes
RZGenie (CPU)	X	X	109,633	57,006
RZAnsel (GPU)	20,609	11,160	7,516	5,617

Table 6: CPU and GPU total run times on an 18,821,096 zone Hotspot Problem run with Ares.

	TDF *minimum* runtime (s)			
	1 Node	2 Nodes	4 Nodes	8 Nodes
RZGenie (CPU)	X	X	0.00046	0.00026
RZAnsel (GPU)	18.8	10.6	2.6	0.77

	TDF *average* runtime (s)			
	1 Node	2 Nodes	4 Nodes	8 Nodes
RZGenie (CPU)	X	X	33.0	17.0
RZAnsel (GPU)	19.3	10.8	5.9	3.4

	TDF *maximum* runtime (s)			
	1 Node	2 Nodes	4 Nodes	8 Nodes
RZGenie (CPU)	X	X	124	64
RZAnsel (GPU)	19.5	11.1	9.5	7.7

Table 7: CPU and GPU performance of TDF on the same Hostpot Problem of Table 6. The times are listed for the minimum, average, and maximum time recorded across all MPI ranks. There is growing variation across ranks as the problem is strong scaled.

	TDF maximum % of runtime			
	1 Node	2 Nodes	4 Nodes	8 Nodes
RZGenie (CPU)	X	X	0.11%	0.11%
RZAnsel (GPU)	0.10%	0.10%	0.13%	0.14%

Table 8: CPU and GPU performance of TDF on the Hotspot Problem as a percentage of the total runtime from Table 6. The percentage is for the *maximum* cost across MPI ranks. The cost of TDF is reduced on the GPU in proportion with the rest of the simulation, keeping it a negligible percentage of the total cost.

	LEOS max processor runtime (s)			
	1 Node	2 Nodes	4 Nodes	8 Nodes
RZGenie (CPU)	X	X	643	309
RZAnsel (GPU)	126	75.1	48.3	31.9

	LEOS percentage of runtime			
	1 Node	2 Nodes	4 Nodes	8 Nodes
RZGenie (CPU)	X	X	0.59%	0.54%
RZAnsel (GPU)	0.65%	0.70%	0.69%	0.59%

Table 9: LEOS runtime cost. LEOS takes less that 1% of runtime on both CPU and GPU versions of this problem.

	Opacity Client max processor runtime (s)	
	Opacity Client	Native Interpolation
RZGenie (CPU)	642	142
RZAnsel (CPU)	1638	417
RZAnsel (GPU)	1475	0.47

	Opacity Client percentage of cycle time	
	Opacity Client	Native Interpolation
RZGenie (CPU)	28.6	8.1
RZAnsel (CPU)	49.3	20.0
RZAnsel (GPU)	98.7	2.8

Table 10: Opacity Client costs on 2 nodes, 8 MPI ranks, 10 cycles. The Opacity Client requires a large fraction of the runtime, especially when the GPU is used for the rest of the problem. The Ares native interpolation (which runs on the GPU) provides much better performance.

Opacity Server Analysis

Table 10 shows performance using both the native interpolation scheme in Ares and the Opacity Client's interpolation scheme. The RZAnsel CPU run is running the Ares code entirely on the Power9 CPU.

The RZAnsel CPU and GPU builds with the Opacity Client have similar run times, but the GPU runtime is slightly faster. Even in the GPU build, Opacity Client lookups are running on the CPU (the client library is not ported to GPU yet), but the overall runtime benefits from Ares packing and unpacking the data for the Opacity Client call on the GPU. This shows that that even with the overhead of transferring data from CPU to GPU, running parts of the calculation on the GPU is more efficient than running solely on the CPU (with large enough mesh sizes).

Comparing RZAnsel (Power9) to RZGenie (Intel Broadwell) CPU performance, both the native and the Opacity Client lookups perform significantly worse on a per-core basis on RZAnsel compared to the RZGenie, although the Opacity Client suffers from a larger slowdown.

The native interpolation on the GPU is the only run where the interpolation time maintains the fraction of total run time observed on a CPU architecture. Running the Opacity Client lookups on the CPU dominates the runtime. Performance takes a large hit not only because the interpolation isn't getting speed up from GPU execution, but also because in runs with one rank per GPU, only

4 out of the 40 cores on the system are active. Although the opacity interpolations are not a large percentage of runtime on current CPU platforms, if interpolation remains on the CPU while the rest of the code is accelerated on the GPU, interpolation quickly becomes a bottleneck.

Summary

The Hotspot results show that in our default configuration, TDF and LEOS both have speedups that are comparable to the hydrodynamics and are considered to be running well. As we strong scale, we start to see hints of increasing load imbalance, which prevents the libraries from strong scaling perfectly. Overall, this is of little concern due to the very small percentages of the runtime that are utilizing TDF and LEOS.

The variant using the Opacity Client lookup functionality shows that its inability to utilize the GPU will cause the lookups to quickly dominate the overall runtime of the problem if it's used in this way. If codes plan to use the client interpolation on Sierra or El Capitan, it must be ported and tuned in order to keep up with the rest of the code.

3.3 Shaped Charge Problem

Name: BRL81 Shaped Charge (Shaped Charge)
Size: 14M quadrature points
Code: MARBL
Libraries: LEOS
Physics: ALE hydrodynamics, high-explosives, elasto-plasticity

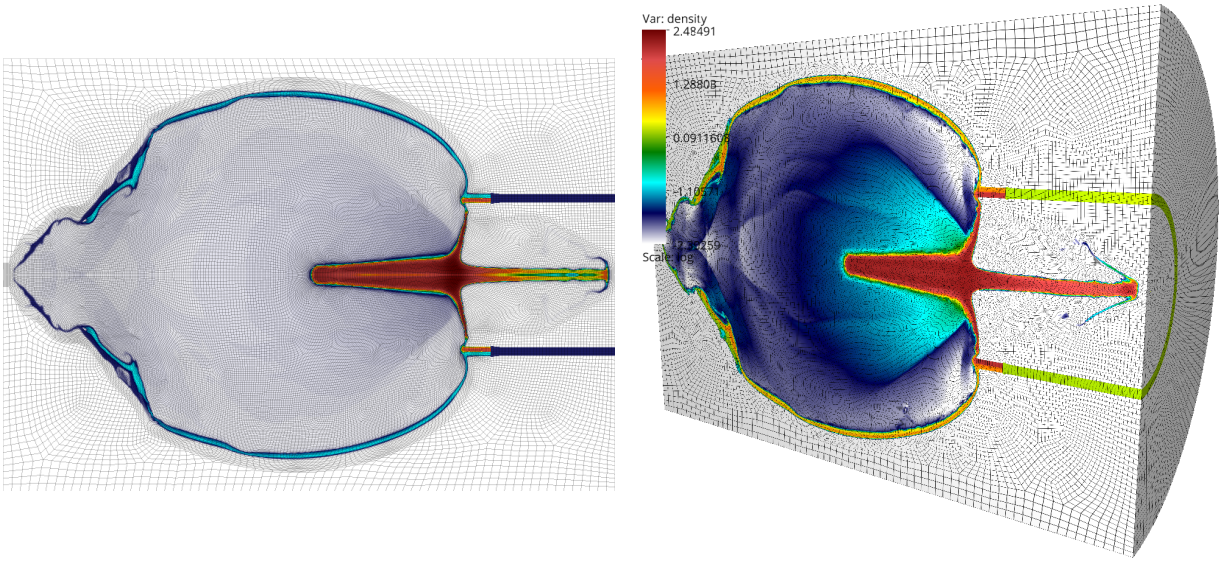


Figure 2: BRL81 Shaped Charge: Medium Resolution 2D Model (left) and BRL81 Shaped Charge with off-axis detonation: Low Res 3D Model (right).

Description

The BRL81a shaped charge is a device which focuses the pressures of a high explosive onto a metal "liner" to form a hyper-velocity jet which can be used in many applications, including armor penetration, metal cutting and perforation of wells for the oil/gas industry [18]. Modeling such a device requires multi-material compressible hydrodynamics with general equations of state for the various materials, high-explosive burn and elasto-plasticity.

The calculation is using ALE for its hydrodynamics. It uses the Jones-Wilkins-Lee (JWL) equation of state for the high explosive, which is implemented as a native model in MARBL. It uses elastic-perfectly plastic and Steinberg-Guinan strength models, which are also natively in the code. Additionally, it uses LEOS tabular EOS for every material, except for the high explosive.

We are interested in three mesh resolutions (low, medium, and high) for both 2D and 3D models. The low-resolution 3D model consists of 14,346,240 quadrature points (224,160 quadratic hexahedral elements with 64 quadrature points each) and can currently be run in MARBL using 12 GPUs (3 nodes of RZAnsel). Running at higher resolutions scales the problem by successive factors of eight: 96 GPUs (24 nodes) for the medium resolution model and 768 GPUs (192 nodes) for the high resolution model.

For MARBL, we employ a high-order finite element based ALE approach to this problem involving three stages:

1. High-order Lagrange phase multi-material hydrodynamics on a moving, unstructured, high-order (NURBS) mesh [14], including use of PEM libraries for equations of state and material constitutive models
2. Non-linear, material adaptive, high-order mesh optimization using the TMOP method
3. High-order continuous (kinematic) and discontinuous (thermodynamic) Galerkin based remap using flux corrected transport (FCT)

All three phases of the ALE algorithm are done on the GPU using high-order finite element operators provided by the MFEM library using the partial assembly (PA) numerical technique, where global matrices are not assembled, but instead have their action on vectors computed on the fly. This is computationally advantageous for GPUs since it involves batched computation of small, dense tensor contractions. The PEM libraries perform their operations in batches of element quadrature points in parallel on the GPU during the Lagrange phase.

Motivation

- This problem is a good stress test of the ALE hydrodynamics in MARBL and running this problem at very high resolutions in 3D is important for studying the hydrodynamics of hyper-velocity jet formation. There are still outstanding questions as to the cause of jet instabilities which are experimentally observed. Having a new code with a high order algorithm model a shaped charge in 3D at very high resolution will help bring a new perspective on this problem.
- We are also interested in applying design optimization techniques to shaped charges to explore means for improving the shape and velocity of the jet. This will require performing large ensembles of 3D calculations to span a parameterized "design space" which will then be used to train a machine learned surrogate for use in optimization.

3.4 Shaped Charge Results

We performed a strong scaling study on LLNL's Lassen (ATS-2) and RZTopaz (CTS-1) clusters on the low-resolution variant of MARBL's 3D Shaped Charge Problem. In the remainder of this section, we will refer to these systems as ATS-2 and CTS-1, respectively.

The problem consists of 14,346,240 quadrature points (64 quadrature points for each of the 224,160 quadratic hexahedra). Our scaling study ran on 3, 6, 12 and 24 nodes using 4 MPI ranks per node on ATS-2 and 36 MPI ranks per node on CTS-1. To ensure that we measured the behavior of the problem at later times, we ran the code to $t = 15$ microseconds (out of the total $t = 35$ microseconds in the full run). We designed and ran our scaling study using Maestro [13], captured run metadata using Adiak [2] and hierarchical performance data using Caliper [8] annotations.

Table 11 and corresponding Figure 3(a) compare the node-to-node strong scaling performance for MARBL's `timeStepLoop` annotation, which includes the runtime for the simulation's time steps, but not its initialization times. For the Shaped Charge Problem, `timeStepLoop` accounts for approximately 97% of the total runtime on ATS-2 and 99.9% of the runtime on CTS-1.

Tables 11 through 13 compare the node-to-node strong scaling performance of LEOS on our 14 million quadrature point MARBL simulation of the Shaped Charge Problem on CTS-1 and ATS-2. As a proxy for the actual runtime within LEOS, we measured the `ComputeMaterialProperties` annotation within MARBL. This includes all calls to LEOS within a hydrodynamics time step,

	Total runtime (s)			
	3 Node	6 Nodes	12 Nodes	24 Nodes
CTS-1	77,218	41,952	20,839	10,326
ATS-2	5,340	2,964	1,766	1,168

Table 11: CPU and GPU run times for the `timeStepLoop` annotation on a 14M quadrature point Shaped Charge Problem run with MARBL.

	LEOS *minimum* runtime (s)			
	3 Node	6 Nodes	12 Nodes	24 Nodes
CTS-1	181.9	86.1	41.9	21.2
ATS-2	37.9	9.5	6.0	4.3

	LEOS *average* runtime (s)			
	3 Node	6 Nodes	12 Nodes	24 Nodes
CTS-1	620.3	299.6	150.7	77.2
ATS-2	57.2	31.6	16.7	11.6

	LEOS *maximum* runtime (s)			
	3 Node	6 Nodes	12 Nodes	24 Nodes
CTS-1	1520.8	740.2	461.1	234.8
ATS-2	73.2	46.4	29.4	23.7

Table 12: CPU and GPU performance of the `ComputeMaterialProperties` annotation, which contains all LEOS calls within a hydrodynamics time step on the same Shaped Charge Problem of Table 11. The times are listed for the minimum, average, and maximum time recorded across all MPI ranks. There is growing variation across ranks as the problem is strong scaled.

	LEOS maximum % of runtime			
	3 Node	6 Nodes	12 Nodes	24 Nodes
CTS-1	1.97%	1.76%	2.21%	2.27%
ATS-2	1.37%	1.56%	1.67%	2.03%

Table 13: CPU and GPU performance of the `ComputeMaterialProperties` function, which contains all calls to LEOS on the same Shaped Charge Problem of Table 11. The percentage is for the *maximum* cost across MPI ranks. The cost of LEOS is reduced on the GPU in proportion with the rest of the simulation, keeping it a relatively low percentage of the total cost.

as well as some additional kernels related to processing the material data. It contains six kernels for each of the seven materials in the Shaped Charge Problem and is called twice within each Lagrange time step in the simulation. About half of runtime within a `ComputeMaterialProperties` annotation takes place within the LEOS (see Figure 4 for a breakdown of kernel calls within a single instance of this annotation in a 3-node run on ATS-2). To ensure that we're properly capturing the timings for kernels that are launched asynchronously, we added explicit `cudaDeviceSynchronize()` calls at the beginning and end of the `ComputeMaterialProperties` function.

These tables list the total runtime of the whole simulation, the runtime of LEOS in isolation, and the percentage of the total runtime for each run in our scaling study. Table 12 lists the minimum, average, and maximum `ComputeMaterialProperties` times across ranks, while Table 13 lists the maximum LEOS runtime as a percentage of the `timeStepLoop` runtime. We observe that `ComputeMaterialProperties` cost is always less than 2.5% of the total runtime, even in the maximum (worst) case. Interestingly, the relative cost of `ComputeMaterialProperties` is lower on ATS-2 than on CTS-1 by around 0.65% at low node counts and by around 0.25% at high node counts.

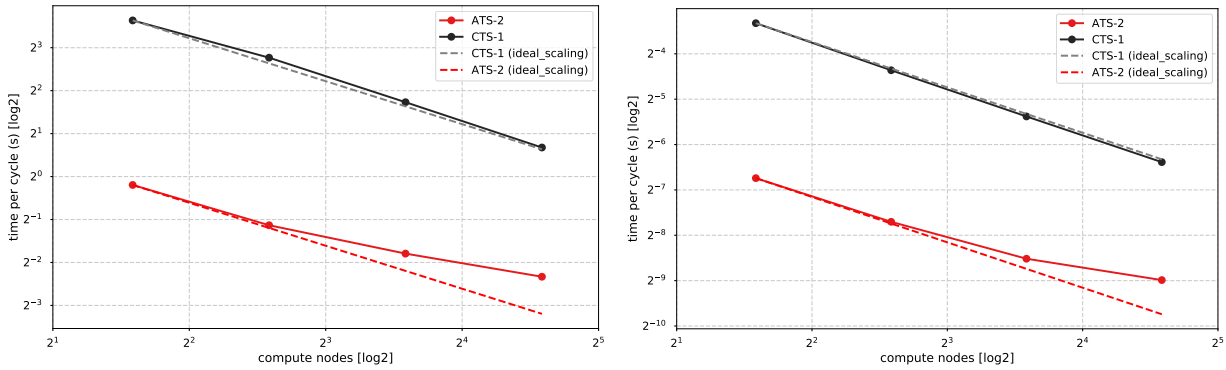
Comparing the strong scaling across architectures, we observe that for the same number of nodes, the total simulation time is reduced by about a factor of 15 for low node counts and 9 for high node counts when going from the CPU-based CTS-1 to the GPU-based ATS-2. As can be seen in Figure 3, the runtime for `ComputeMaterialProperties` (right chart) tracks nicely with the observed strong scaling for the overall simulation (left chart). Its runtime is also accelerated by similar (but slightly lower) strong scaling: about a factor of 11 at low node counts to a factor of 7 at high node counts. This load imbalance for LEOS at higher node counts is somewhat expected for this problem due to how the materials are distributed across the nodes. In particular, the HE material uses an analytic JWL EOS model, while the other materials use LEOS.

Interestingly, in running this study, the MARBL team observed a previously unidentified strong-scaling slowdown in its LEOS usage. Although MARBL uses Umpire's pooled allocators for sharing memory with its third-party libraries, it had not used this for its LEOS setup, and was instead directly allocating and deallocating memory within `ComputeMaterialProperties`, yielding only around a 2–3× speedup for ATS-2 when compared to CTS-1 at the highest node count. Switching to pooled allocators for temporary memory increased the code's relative performance within `ComputeMaterialProperties` by more than a factor of 3 for our ATS-2 runs, as can be seen in Figure 5.

Given that the speedups are tracking the hydrodynamics performance well even at relatively high node counts where the GPU is no longer saturated, and that the percent of total runtime is relatively small, we can say that LEOS is performing well and that we do not anticipate the need for further optimization.

Summary

The Shaped Charge Problem has shown that LEOS's speedups are comparable to those observed in other parts of the simulation and is considered to be running well. We see that as we strong scale, we can potentially see increasing load imbalance, which prevents the libraries from strong scaling perfectly. However, the MARBL team is not concerned about this due to the relatively small percentages of the runtime that are utilizing LEOS and the expected load imbalance in this problem's setup.



(a) Overall strong scaling

(b) `ComputeMaterialProperties` strong scaling

Figure 3: Node-to-node strong scaling for MARBL's Shaped Charge Problem on CTS-1 and ATS-2. For each platform, we plot the time per cycle against the number of nodes on a log2-log2 plot (however, please note the differences in y-axis ranges for these plots). (a) The `timeStepLoop` annotation includes simulation cycles, but omits initialization times. (b) The `ComputeMaterialProperties` annotation includes all calls to the LEOS library. It contains six kernel invocations for each of the problem's seven materials.

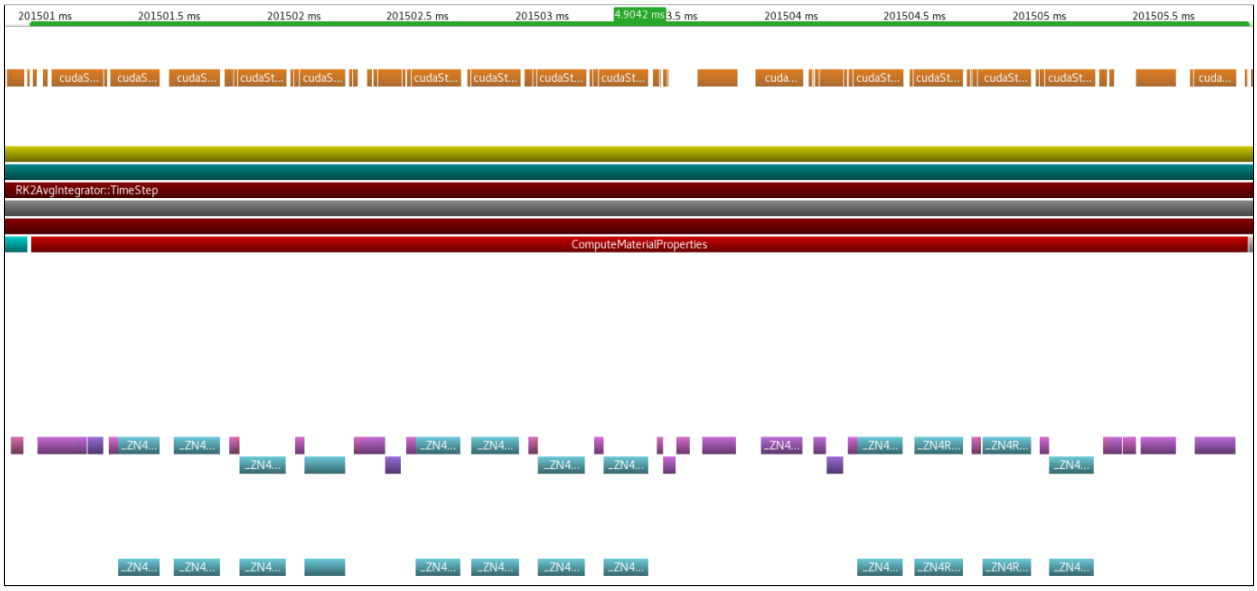


Figure 4: Screenshot from an Nvidia visual profiler (nvvp) session highlighting the breakdown of kernel calls within a single `ComputeMaterialProperties` annotation (red) in a 3-node run of the Shaped Charge Problem. The LEOS calls within this 5 millisecond interval are primarily in the blue blocks on the bottom row as well a subset of the smaller purple blocks. The larger purple blocks within this annotation include calls to MARBL's material strength library Leilak.

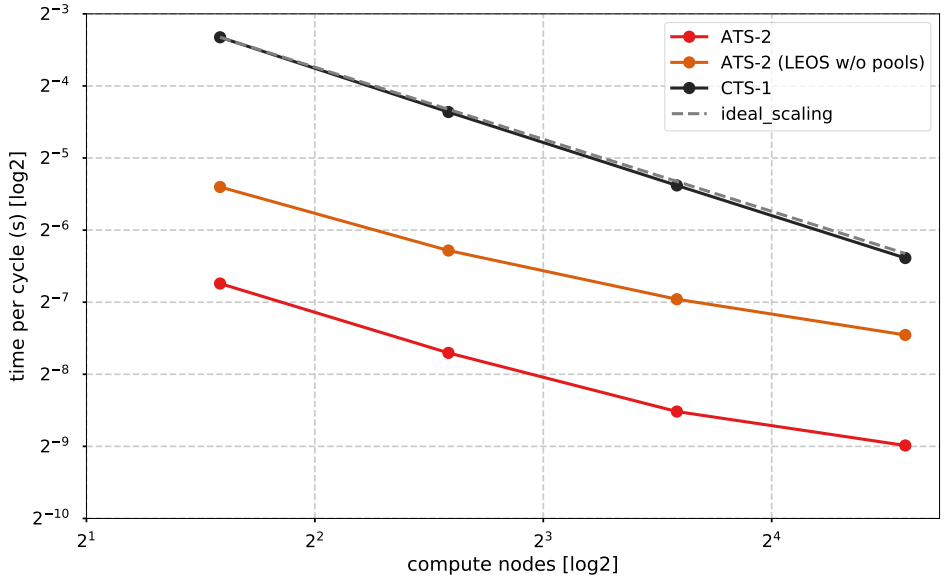


Figure 5: Sharing memory pools between the codes and the PEM libraries is essential for performance at scale. This chart adds an additional scaling study run 'ATS-2 (LEOS w/o pools)' (orange) to the data from Figure 3(b). The latter was run on the same MARBL code, but did not use LEOS's Umpire-based memory pools to share memory resources between MARBL and LEOS. This imposed around a 3× runtime cost relative to the runs that utilized this feature.

3.5 Jetting Defect Problem

Name: Jetting Defect Problem (Focused physics experiment)
Size: 600k–5.7M zones
Code: ALE3D
Libraries: MSLib, LEOS
Physics: Hydro, MSLib, LEOS

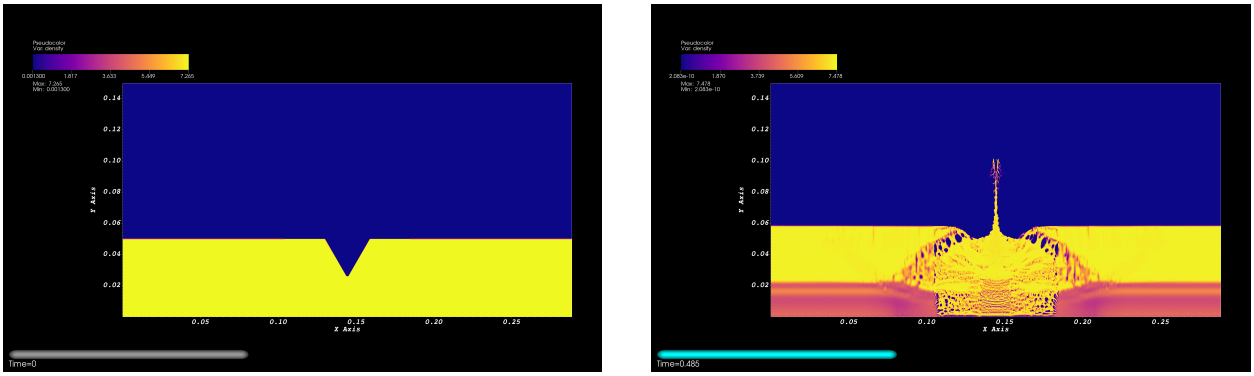


Figure 6: Jetting Defect Problem setup: the setup represents an experiment in which a target plate of Sn with a groove is impacted by a Sn flyer plate. This "symmetric impact" is captured by the Sn target plate impacting a symmetry plane at half of the speed of the Sn flyer plate. The plots show density, with (left) the initial condition and (right) a condition roughly 0.5 microseconds after impact. A jet has formed from the groove, and there is significant porosity formation (low density material) in the target plate due to release wave interactions producing tensile conditions.

Description

A shock wave is driven into a plate of tin with a groove (surface defect), and the interaction of the shock with the groove produces a jet of material. The conditions are akin to those examined both in gas gun plate impact experiments and in laser-driven surface defect experiments, and features of the jet formation and breakup are of interest. This problem involves significant advection as the jet is formed and streams through the mesh, and it uses the Eulerian mode in ALE3D. Overall, this is a relatively simple problem in terms of the number of physics packages active.

MSLib is used in the tin target material and a simple gamma-law gas is used in the surrounding material. Within MSLib, the sub-models are given in Table 14.

sub-model type	number	model name
shear modulus	12	curve-analytic
yield surface	115	j2-void
strength	209	PTW
EOS	310	LEOS
damage	400	Johnson-Cook

Table 14: MSLib sub-models used in the Jetting Defect test problem

Motivation

- Flow instabilities such as Rayleigh-Taylor and Richtmyer-Meshkov instabilities are of general interest. The formation of jets from surface defects offer a means of examining the influence of material response across a range of conditions explored by varying the shock strength and geometric features of the defect.
- In 2D, this problem does not require GPUs or Sierra. However, 3D features of the jet formation and breakup are of interest and well-resolved versions of such 3D calculations would require extensive resources; hence making them suitable problems for the GPUs.
- For the graphics provided above, the simulation is run in 2D plane strain with 2,058,120 zones and 144 ranks on RZTopaz. The resolution is controllable on the command-line and a coarse version of the problem can capture gross features of the jet formation using $\sim 10^5$ zones.
- The material model used for this simulation is considered state of the art. Using LEOS table 503, phase fractions are provided for the beta, gamma, and liquid phases of tin.
  These phase fractions are then used in a mixture theory within the material model, with distinct behaviors for the three phases. A simpler Mie-Gruneisen EOS version of this problem is also available for comparison, and in that model the melting transition is idealized as being abrupt.
- Aspects of the material model are described in [3] and in [16].

3.6 Jetting Defect Results

We performed a strong scaling study on RZAnsel (ATS-2) and RZGenie (CTS-1), with results shown in Tables 15, 16, and 17. As noted in Section 2.2, at the time of the execution of the timing runs shown here, work is still in progress to have MSLib call LEOS on GPUs, using device-callable single-point LEOS functions. Thus the timing runs are instead conducted using a Mie-Grüneisen EOS within MSLib.

For the 1-node results shown in the table, there is a speedup of roughly 8.6× in going from execution on the CPU-based CTS-1 node to the GPU-based ATS-2 node. While further CPU and GPU performance enhancements to MSLib would be of significant benefit, this speedup is consistent with expectations for this kind of hydro problem with advection. With an increase in the number of nodes on ATS-2, the speedup is less than the ideal value, but this is not surprising given that on 8 nodes there are only ∼175k zones/GPU. On CTS-1, with an increase in the number of nodes the speedup is somewhat better than ideal. This may be due to the memory requirements being a better fit for a larger number of nodes on the CPU-based CTS-1 platform.

More detailed profiling results are also given in the table, but they provide somewhat limited information given that they are collected only for rank 0 of the parallel job. Refinement in how ALE3D collects these profiling data would be helpful in forming a more complete picture. It is however worth noting that the `map` calls take more time than expected on the GPUs. The `map` calls involve less computational work than the `getResponse` calls, and on the CPU `map` calls take about 30% of the time required for the `getResponse` calls. One the GPU, the `map` calls take as much or even more time than the `getResponse` calls. This may point to a problem with memory management or the like, and the issue is worth further investigation.

Total runtime (s)				
	1 Node	2 Nodes	4 Nodes	8 Nodes
CTS-1	49709	25177.6	12356	5992.5
ATS-2	5776	3773	3048.7	2382

Table 15: CPU and GPU timing results on a 5,704,200 zone Jetting Defect Problem run with ALE3D, indicating a ∼8.6× speedup for the 1-node case in which there are roughly 1.4M zones/GPU.

MSLib `getResponse` runtime (s), rank 0				
	1 Node	2 Nodes	4 Nodes	8 Nodes
CTS-1	10932	5213	4902	2327
ATS-2	1232	710	388	205

MSLib `getResponse` % of runtime, rank 0				
	1 Node	2 Nodes	4 Nodes	8 Nodes
CTS-1	22.0 %	20.7 %	39.7 %	38.8 %
ATS-2	21.3 %	18.8 %	12.7 %	8.6 %

Table 16: Timing data for constitutive model update (`getResponse`) calls on rank 0. In the 1-node case the `getResponse` calls consume roughly the same fraction of the wall clock time on CTS-1 and ATS-2 architectures.

MSLib `map` runtime (s), rank 0				
	1 Node	2 Nodes	4 Nodes	8 Nodes
CTS-1	3653	1743	1527	688
ATS-2	1133	622	457	284

MSLib `map` % of runtime, rank 0				
	1 Node	2 Nodes	4 Nodes	8 Nodes
CTS-1	7.3 %	6.9 %	12.3 %	11.5 %
ATS-2	19.6 %	16.5 %	15.0 %	11.9 %

Table 17: Timing data for post-advection "fixup" (`map`) calls on rank 0. The `map` calls are consuming an unexpectedly large fraction of the wall clock time on the ATS-2 architecture, suggesting that there may be an issue that requires further attention. These `map` calls are not computationally intensive.

3.7 Nonlocal Problem

Name: Double Edged Notched Tension (DENT) Problem (Material Failure)
Size: 7,853,734 zones
Code: ALE3D
Libraries: MSLib
Physics: Hydro, MSLib, non-local machinery

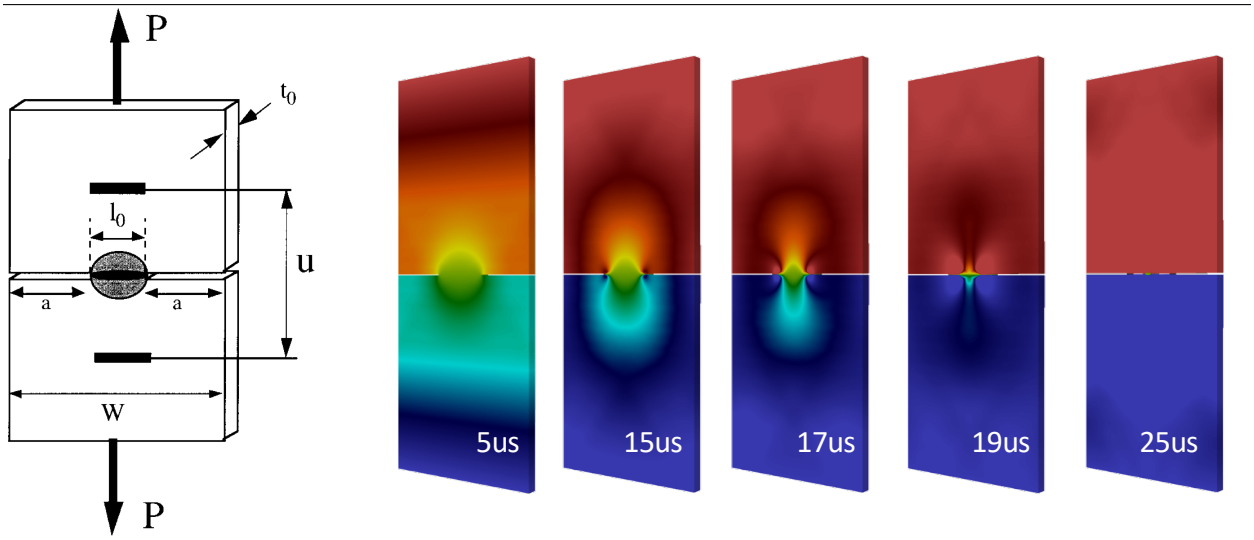


Figure 7: Non-local problem setup: a schematic of the Double Edged Notched Tension from [17] and simulation results for the axial component of velocity, showing the progression to complete failure.

Description

In the 3D Double Edged Notched Tension (DENT) Problem, a plate of finite thickness is pulled in the direction perpendicular to notches located on either side of the plate. The problem is single-material and uses Lagrangian hydrodynamics. As shown in the graphic above, the extension is in the vertical direction. The schematic is from [17]. Stress concentrations at the notch tips drive crack growth and failure of the plate. In the figure above, which shows the axial component of the velocity, we see progressive localization of the velocity gradient into the central portion of the plate through which the cracks propagate, culminating in the complete failure of the plate as the cracks meet in the middle. The 3D nature of the problem manifests in the finite thickness of the plate influencing the stress distributions that drive crack growth.

With reference to the above schematic, for this problem we have $a = 2$ cm, $W = 6$ cm, and $t_0 = 0.4$ cm.

For treatment of material failure, the MSLib constitutive model uses a porosity-mechanics-based approach with aspects of the model described in [4]. The model includes non-local computations, which amount to performing a convolution integral, and are performed within ALE3D each time step and are provided to MSLib as auxiliary data.

The single material in the problem is parameterized for Al6061-T6, and the associated MSLib

sub-model type	number	model name
shear modulus	6	RING
yield surface	124	NP
strength	209	PTW
EOS	304	LEOS
damage	499	dummy

Table 18: MSLib sub-models used in the DENT test problem.

sub-models are given in Table 18. We note that, unlike in Section 3.5, a damage model is not used because the yield surface model includes a treatment of porosity.

Transitions in zone size along the axial extension direction are used to concentrate zones in the vicinity of the propagating cracks. This induces variations in the number of zones within the cutoff radius in the non-local calculations, and thus in the workload per zone. The domain decomposition does not currently account for these differences in workload, and load imbalance is thus expected in this test problem. However, this load imbalance is not a salient feature of common use cases in fragmentation problems given that a refined mesh is needed throughout the fragmentation problems to capture the localization behaviors that are precursors to crack formation.

Motivation

- There is significant programmatic interest in modeling material failure and associated phenomena such as fragmentation. In the progression to failure, say by the nucleation and growth of porosity, the material becomes weaker. While physically motivated, this constitutive behavior produces mesh-dependence in numerical implementations due to the tendency of deformation to localize into narrower regions with mesh refinement. This results in a reduction in energy dissipation with mesh refinement and non-convergent behavior. One means of mitigating this effect is to introduce a length scale, which can be associated with the characteristic spacing of relevant microstructural features. Here we will make use of a non-local implementation that involves a convolution integral to compute the non-local values to be used in the constitutive model. This approach introduces a length scale associated with the width of the kernel function in the convolution integral. Additional details on this type of approach to mitigating mesh dependence can be found, for example, in [6].
- This particular simulation does not require GPUs or Sierra to run. However, because of the expense of the non-local method and the resolutions required to study material failure adequately, 3D fracture/fragmentation problems will need the use of GPUs to run these calculations efficiently. Furthermore, we ultimately want the non-local MSLib machinery and models to work with embedded grids and element erosion. We recently accomplished this goal, but we cannot run embedded grid problems on the GPUs until the FEusion library has been successfully ported to the GPUs and have undergone significant testing to gain confidence it is working correctly as it required a rewrite of the library from Fortran to C++.
- This overall modeling capability is currently state of the art and a work in progress.

3.8 Nonlocal Results

We performed a strong scaling study on RZAnsel (ATS-2) and RZGenie (CTS-1), with results shown in Table 19. Timing data are collected from simulations spanning 1 microsecond of simulation time.

	Total runtime (s)		
	2 Nodes	4 Nodes	8 Nodes
CTS-1	29161	13293	6413
ATS-2	1123	658	339

Table 19: CPU and GPU timing results on a 7,853,734 zone DENT problem run with ALE3D. Deviations from perfect strong scaling on the GPU is expected due to the relatively small node count.

None-to-node GPU speedup is approximately 26× for the 2-node case in which each GPU is responsible for approximately 982k zones. With an increase in the number of nodes on ATS-2 (RZAnsel), the speedup is less than the ideal value, but this is not surprising given that on 8 nodes there are only ∼245k zones/GPU. As in the test problem in Section 3.6, on CTS-1 (RZGenie) the speedup is somewhat better than ideal over this range of nodes.

While the GPU port shows good speedup, the use of the non-local capabilities increases runtime by ∼16× for this test problem (as compared to a simulation that also uses MSLib but without non-local capabilities active). That is, the non-local calculation dominates the overall simulation time, and further improvements in computational performance are desirable. The relative cost of running non-local depends on the details, especially the characteristic element size compared to the non-local cutoff radius. In future work, we will examine the use of a different kernel function in the convolution integral that is, for the same characteristic length scale, amenable to a shorter cutoff radius.

The primary performance bottleneck in the neighbor evaluation on the GPUs is access to global memory. The use of shared memory in CUDA to reduce this bottleneck is being investigated. This involves restructuring the kernel to reduce the number of global memory accesses. In the current implementation, we see only 10–20% GPU compute utilization. Initial testing of the reworked kernel shows close to 50% compute utilization.

3.9 Godiva Problem

Name: Godiva Criticality Calculations (Criticality k-eigenvalue calculation)
Size: 1 CSG cell or 8,000 mesh cells, 10^7–10^8 Monte Carlo particles
Code: Mercury
Libraries: GIDI/MCGIDI
Physics: Neutron transport with continuous energy or multigroup neutron cross section data

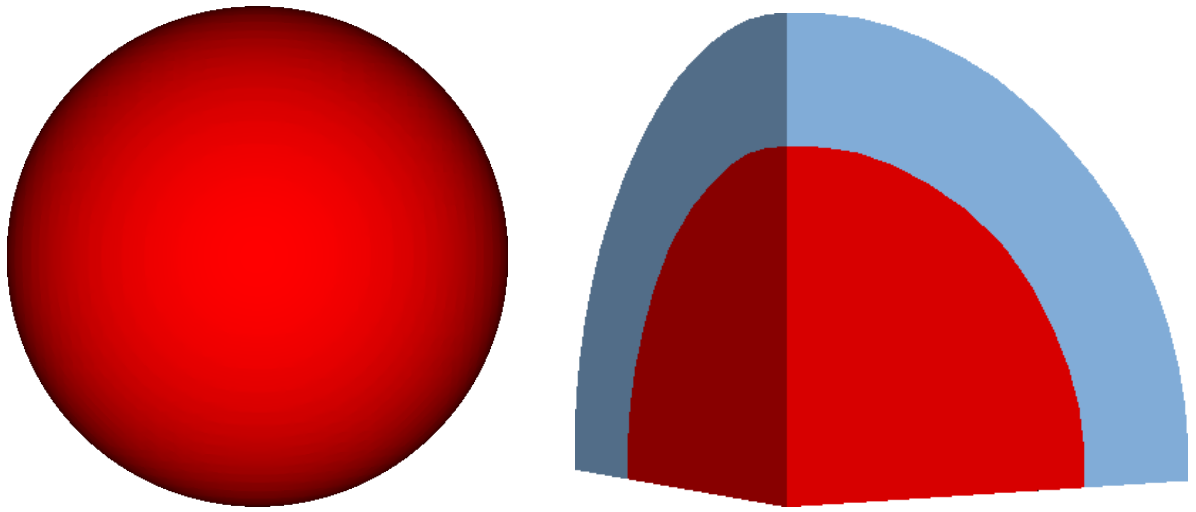


Figure 8: The Godiva sphere of highly enriched uranium (left) is a common neutron transport benchmark problem. Godiva in water (right) includes water surrounding the uranium.

Description

Godiva Sphere: The Godiva sphere [11] is a common neutron transport benchmark problem used to test neutron transport codes and to assess nuclear cross section data. The Godiva sphere is a bare sphere of highly-enriched uranium composed of a small number of uranium isotopes and is documented as problem HEU-MET-FAST-001 as part of the International Criticality Safety Benchmark Evaluation Project (ICSBEP). The test problem is spatially homogeneous and involves a "fast" neutron spectrum since it only contains uranium. This test problem can be simply modeled using a single constructive solid geometry (CSG) sphere. The goal of the calculation is to compute the k-eigenvalue for the test problem to determine the criticality of the system. Although geometrically simple, the test problem requires correct neutron transport physics and nuclear data to obtain the correct results. Because the problem is geometrically simple, this test problem focuses on the collision physics implementation in the GIDI/MCGIDI PEM libraries used by Mercury. The Godiva Problem was simulated using either continuous energy or multigroup neutron cross section data and 10^8 Monte Carlo particles.

Godiva In Water: The Godiva in Water test problem [12] is a simple modification of the Godiva Sphere test problem to include a sphere of water surrounding the sphere of uranium. The presence of the water makes the problem spatially heterogeneous and introduces additional neutron physics into the simulation as neutrons from fissions in the uranium now slow down (thermalize) in

the water and re-enter the uranium sphere. In addition, this test problem uses a mesh instead of a single constructive solid geometry cell. As a result, the Godiva in Water test problem includes introduces more complexity in the geometric representation as well as more complex neutron physics. This test problem serves to investigate GIDI/MCGIDI performance in a simulation with more balanced Monte Carlo algorithmic characteristics. The Godiva in Water problem was simulated using a mesh with 8,000 cells, either continuous energy or multigroup neutron cross section data, and 10^7 Monte Carlo particles. We are not investigating thermal neutron scatter law (TNSL) or unresolved resonance (URR) physics for these test problems.

Motivation

- Correctly and efficiently computing criticality is programmatically important. Collections of criticality benchmark problems such as the Godiva Problem help validate neutron transport codes and nuclear data against experiment.
- These test problems are useful for assessing MCGIDI performance as they exercise a range of neutron transport physics and can be readily scaled to different numbers of nodes on CTS and Sierra machines by changing the number of Monte Carlo particles. The accurate computation of the global eigenfunction for these types of criticality calculations, for example to obtain an accurate prediction of the neutron flux at a diagnostic foil location, can require a significantly larger number of particles than the calculation of the eigenvalue alone.
- Because the Godiva Sphere is represented by a single CSG element, the particle tracking kernel is dominated by collision events (since there are no mesh facet crossings to calculate). In contrast, Godiva in Water employs a fine mesh so that particle tracking will be dominated by facet crossings. Analyzing collision-dominated vs. facet-dominated problems allows comparison of these two regimes of performance.

3.10 Godiva Results

Both Godiva test problems (Godiva Sphere and Godiva in Water) were run on one node of the CTS-1 system RZGenie and one node of the ATS-2 system RZAnsel. We report Tracking Time, which is the time in seconds that Mercury spent simulating the movement of particles through the problem material. Tracking is the only portion of the calculation affected by MCGIDI and GIDI data load times are negligible for the problems considered. We also report Segments, which is the number of times a particle was moved; Collisions, which is the number of times that a particle movement resulted in a nuclear reaction; Seg/s, which is the quotient Segments/Tracking Time; k, which is the k-eigenvalue; and σ, which is the standard deviation of k.

We ran Godiva Sphere as a single constructive solid geometry cell with 100 million particles for 20 cycles. We ran Godiva In Water as a mesh with 8000 zones with 10 million particles for 20 cycles. On CTS-1 we ran with history-based tracking on 1 node with 36 MPI ranks. On ATS-2 we ran with event-based tracking on 1 node with 4 MPI ranks. All ranks had full descriptions of the problem geometry (i.e. "full-replication"). We ran Mercury 5.26.2 using GIDIplus 3.19.68. We compiled with the Intel Compiler version 19.1.0 on CTS-1 and the Nvidia CUDA Compiler version 11.2.0-beta on ATS-2.

Comparing MCGIDI to legacy MCAPM on CTS-1

To ensure that MCGIDI performs on par with the legacy MCAPM library, on CTS-1, Godiva Sphere and Godiva in Water simulations were performed using Mercury with both MCGIDI and

MCAPM. As shown in Tables 20 and 21, the run times differed by less than 8%. Confirming that MCGIDI runs well on CTS-1 provides important context examining its performance on ATS-2; if MCGIDI was 10x slower than MCAPM on CTS-1 then a 10x speedup on ATS-2 would be worthless.

Evaluating MCGIDI performance on CPUs vs. GPUs

Evaluating the performance of MCGIDI in Mercury is complicated by a challenge not encountered by other PEM libraries in this milestone. Mercury has internal timers to report the amount of time spent in various code blocks. Unfortunately, none of these timers specifically isolate the time spent in MCGIDI. This is due to the fact that MCGIDI is called inside of loops or kernels that also perform other tasks. Unfortunately, there is no practical way to add timers to collect information within a GPU kernel. Analysis is further complicated by the fact that when Mercury calls MCGIDI to perform collision physics, the function arguments include a functor that MCGIDI uses to call back into Mercury.

Performance data can also be obtained with program counter (PC) sampling tools such as the Google Performance Tools profiler (for CPUs) and a newly added sampling capability in Nvidia's Nsight Compute (for GPUs). The Google profiler worked well for CPU PC sampling on the CTS-1 platform. While Nsight Compute can perform PC sampling on GPU code, we had difficulties collecting the data in a reasonable way on the ATS-2 platform. Nvidia support was very responsive and worked with us in a codesign effort to apply existing capabilities of the tool in a new analysis methodology. Significant progress was made, but unfortunately due to the complexity of the application, Nsight Compute is not yet fully ready to perform the PC sampling analysis using our desired methodology. We have reported the issue to Nvidia for follow up engineering work. Appendix A contains more information about attempts to use Nsight Compute.

Performance for the Godiva Problems on CPU and GPU are shown in Tables 22 and 23. Note that the fraction of Segments corresponding to Collisions (i.e., Collisions / Segments) is 0.83 for Godiva Sphere but only 0.14 for Godiva In Water. This verifies that Godiva Sphere is collision dominated, while Godiva in Water is dominated by other events such as facet crossing. This fraction also serves as an estimate of the tracking time spent in MCGIDI relative to the rest of the tracking code.

The collision-dominated Godiva Sphere is ∼6–7x faster on GPUs while the facet-dominated Godiva in Water is only ∼3x faster on GPUs. Because the performance of collisions depends heavily on MCGIDI we conclude that MCGIDI has better GPU speedup than other parts of Mercury. This indicates that MCGIDI does not appear to be a bottleneck that prevents Mercury from achieving better GPU performance, at least for these problems.

Tables 24 and 25 provide further support for the assertion that the number of collision segments is a proxy for the amount of time spent in MCGIDI. These tables show the amount of time spent in MCGIDI on CTS-1 as reported by the Google profiler and demonstrate that Godiva Sphere spends at least twice as much time in MCGIDI compared to Godiva In Water. These tables also show that the overhead of PC sampling on CPUs is very small—in the range of 2.6–8%.

Link-time optimization (LTO)

Link-time optimization (LTO) is a form of whole program optimization that performs additional optimizations across compilation units at link time. Nvidia provides LTO capability in the CUDA 11 toolchain. To evaluate the impact of LTO on the performance of Mercury and MCGIDI, the Godiva Sphere and Godiva in Water test problems were run on ATS-2 using Mercury compiled with and without LTO. Tables 26 and 27 show the results. LTO provided speedups in all cases;

the maximum speedup was 27.1% for Godiva Sphere, with Seg/s increased from about 80 million to over 100 million. The downside of LTO is an increase in link-time of about 200x: Mercury link-time increases from 8 seconds to 30 minutes with LTO. The link-time slowdown is acceptable for production builds given the speedups that LTO provides.

Collision Library	Cross Sections	Tracking Time (s)	Segments (billions)	Collisions (billions)	Seg/s (millions/s)	k	σ
GIDIplus	CE	455	6.486	5.362	14.3	1.00012	4.88e-05
GIDIplus	MG	413	6.489	5.364	15.7	1.00054	2.45e-05
MCAPM	CE	485	6.493	5.370	13.4	1.00052	3.15e-05
MCAPM	MG	393	6.498	5.375	16.5	1.00074	5.16e-05

	GIDIplus Tracking Speedup
CE	1.067
MG	0.952

Table 20: For Godiva Sphere on CTS-1, Mercury runs 6.7% faster with MCGIDI than with the legacy MCAPM library with continuous energy (CE) cross-sections, and 4.8% slower with multi-group (MG).

Collision Library	Cross Sections	Tracking Time (s)	Segments (billions)	Collisions (billions)	Seg/s (millions/s)	k	σ
GIDIplus	CE	344	8.455	1.162	24.5	1.11381	1.32e-04
GIDIplus	MG	328	8.446	1.160	25.8	1.11202	8.31e-05
MCAPM	CE	370	8.451	1.164	22.8	1.11435	6.14e-05
MCAPM	MG	330	8.439	1.162	25.6	1.11259	1.15e-04

	GIDIplus Tracking Speedup
CE	1.075
MG	1.007

Table 21: For Godiva In Water on CTS-1, Mercury runs 7.5% faster with MCGIDI than with the legacy MCAPM library with continuous energy (CE) cross-sections, and 0.7% faster with multigroup (MG).

System	Cross Sections	Tracking Time (s)	Segments (billions)	Collisions (billions)	Seg/s (millions/s)	k	σ
CTS-1	CE	455	6.486	5.362	14.3	1.00012	4.88e-05
CTS-1	MG	413	6.489	5.364	15.7	1.00054	2.45e-05
ATS-2	CE	64	6.486	5.362	101.9	1.00012	4.88e-05
ATS-2	MG	62	6.489	5.364	105.0	1.00054	2.45e-05

	ATS-2 Tracking Speedup
CE	7.147
MG	6.683

Table 22: For Godiva Sphere, Mercury runs 7.15x faster on ATS-2 than on CTS-1 with continuous energy (CE) cross-sections, and 6.68x faster with multigroup (MG).

System	Cross Sections	Tracking Time (s)	Segments (billions)	Collisions (billions)	Seg/s (millions/s)	k	σ
CTS-1	CE	344	8.455	1.162	24.5	1.11381	1.32e-04
CTS-1	MG	328	8.446	1.160	25.8	1.11202	8.31e-05
ATS-2	CE	105	8.455	1.162	80.7	1.11381	1.32e-04
ATS-2	MG	103	8.446	1.160	82.2	1.11202	8.30e-05

	ATS-2 Tracking Speedup
CE	3.290
MG	3.191

Table 23: For Godiva In Water, Mercury runs 3.29x faster on ATS-2 than on CTS-1 with continuous energy (CE) cross-sections, and 3.19x faster with multigroup (MG).

Profiled?	Cross Sections	Tracking Time (s)	PC Samples in GIDI (%)	Total number of PC samples	k	σ
No	CE	455	X	0	1.00012	4.88e-05
No	MG	413	X	0	1.00054	2.45e-05
Yes	CE	466	24.4	529895	1.00012	4.88e-05
Yes	MG	435	16.7	498123	1.00054	2.45e-05

	Profiled Tracking Slowdown
CE	1.026
MG	1.052

Table 24: For Godiva Sphere on CTS-1, Mercury spends 24.4% of its runtime in MCGIDI with continuous energy (CE) cross-sections, and 16.7% with multigroup (MG). The overhead of profiling is 2.6% for CE and 5.2% for MG.

Profiled?	Cross Sections	Tracking Time (s)	PC Samples in GIDI (%)	Total number of PC samples	k	σ
No	CE	344	X	0	1.11381	1.32e-04
No	MG	328	X	0	1.11202	8.31e-05
Yes	CE	374	11.7	401302	1.11381	1.32e-04
Yes	MG	349	4.9	383690	1.11202	8.31e-05

	Profiled Tracking Slowdown
CE	1.085
MG	1.065

Table 25: For Godiva In Water on CTS-1, Mercury spends 11.7% of its runtime in MCGIDI with continuous energy (CE) cross-sections, and 4.9% with multigroup (MG). The overhead of profiling is less than 8% for CE and less than 7% for MG.

LTO?	Cross Sections	Tracking Time (s)	Segments (billions)	Collisions (billions)	Seg/s (millions/s)	k	σ
No LTO	CE	76	6.486	5.362	85.2	1.00012	4.88e-05
No LTO	MG	79	6.489	5.364	82.6	1.00054	2.45e-05
LTO	CE	64	6.486	5.362	101.9	1.00012	4.88e-05
LTO	MG	62	6.489	5.364	105.0	1.00054	2.45e-05

	LTO Tracking Speedup
CE	1.195
MG	1.271

Table 26: For Godiva Sphere on ATS-2, Mercury runs 19.5% faster when compiled with link-time optimization (LTO) with continuous energy (CE) cross-sections, and 27.1% faster with multigroup (MG).

LTO?	Cross Sections	Tracking Time (s)	Segments (billions)	Collisions (billions)	Seg/s (millions/s)	k	σ
No LTO	CE	112	8.455	1.162	75.4	1.11381	1.32e-04
No LTO	MG	107	8.446	1.160	78.9	1.11202	8.30e-05
LTO	CE	105	8.455	1.162	80.7	1.11381	1.32e-04
LTO	MG	103	8.446	1.160	82.2	1.11202	8.30e-05

	LTO Tracking Speedup
CE	1.071
MG	1.043

Table 27: For Godiva In Water on ATS-2, Mercury runs 7.1% faster when compiled with LTO with continuous energy (CE) cross-sections, and 4.3% faster with multigroup (MG).

3.11 NIF Chamber Problem

Name: NIF Chamber Problem (Source-detector modeling)
Size: 87 groups, 80 angles, 350k–56M zones
Code: Ardra, Mercury
Libraries: GIDIplus
Physics: Linear particle transport

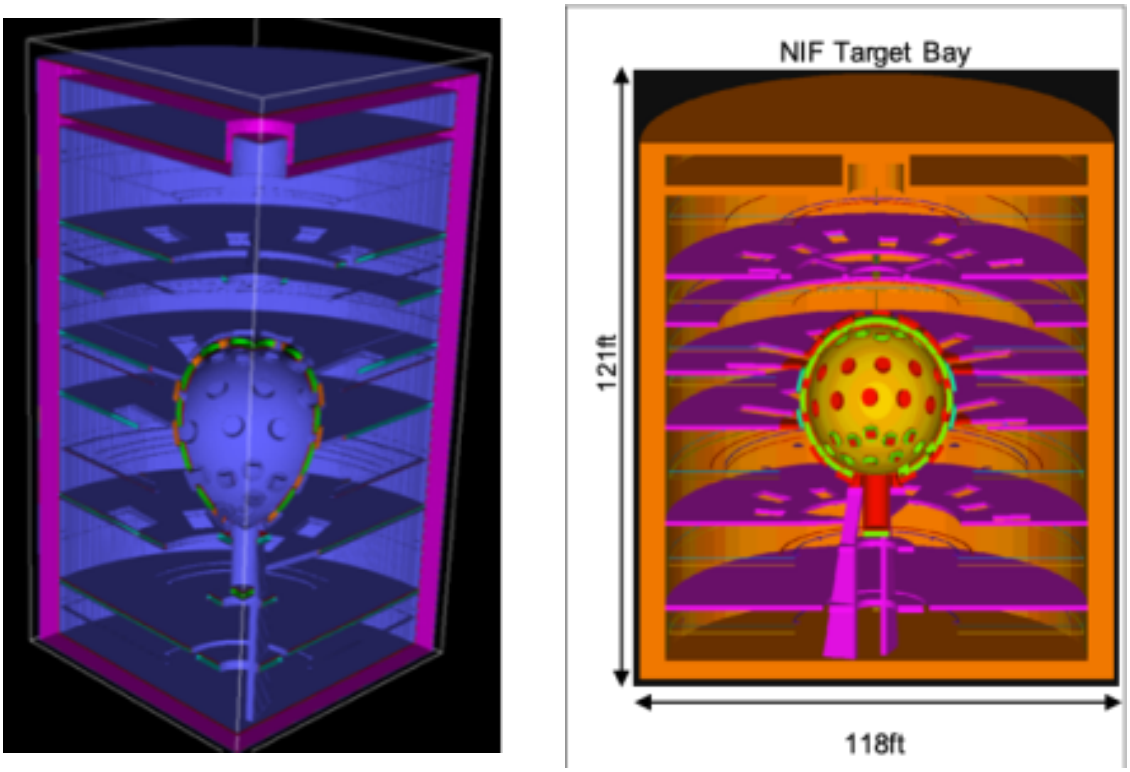


Figure 9: Cross-sectional, perspective view of the NIF target bay from VisIt, with slices along the $x = 0$ and $y = 0$ planes (left). Cross-sectional view of the NIF target bay from COG, sliced along the $x = 0$ plane (right).

Description

This is a source driven time-dependent problem being run in a 3D model of the NIF target bay, the chamber, and the surrounding building, but with no equipment in it. It calculates the radiation dose deposited in various parts of the chamber when an experiment is conducted. This problem is driven by a small spherical 14 MeV neutron source in the middle of the problem. Alternatively, we can also use a point-source instead, which would help eliminate ray effects. The problem uses 87 energy groups for the neutrons and 80 angles. The problem has 359 unique isotopes, which is a relatively large number of unique isotopes and is the primary challenge for the GIDI library.

For this study, we run with 350k to 56M zones, but are interested in higher resolutions. The problem was translated to COG from a TART input file created by Jeff Latkowski.

Motivation

- As noted in the description, the primary purpose of this problem is to serve as a stress test for loading nuclear data. 359 is a fairly large number of unique isotopes; problems of interest are typically in the 50–150 range.
- There are currently two options for loading nuclear data in Ardra: the legacy NDF option, and the newer GIDIplus option. We are interested in using GIDI instead of NDF for data loading because of the improved physics offered by GIDI and GNDS libraries.
- We have seen that GIDI can take much longer than NDF. The difference in the load times is a burden for both simulations run by code users and debugging efforts by code developers. For example, in parametric studies of 1D problems, GIDI can take several minutes per simulation, comprising 50% of the total runtime.
- The porting of Ardra to newer architectures such as Sierra or El Capitan poses an additional challenge. With simulations on Sierra-like systems, fewer MPI ranks per node are used, limiting the parallelization of the data loading in GIDI. Nonetheless, there have been significant recent improvements in GIDI. We hope to assess these recent improvements and to re-evaluate whether GIDI would still be a significant bottleneck in similar problems of interest, especially on Sierra-like systems.
- On systems like Sierra and El Capitan we would continue to increase zonal resolution, as the current resolutions are not yet converged.

3.12 NIF Chamber Results

We performed a "weak" scaling study of the NIF Chamber Problem on two different platforms: Lassen (ATS-2) and RZTopaz (CTS-1). The problem is weak-scaled in the number of spatial zones, but it is effectively strong scaled in the number of isotopes (359) since this number does not change with the number of zones or nodes. The nuclear data load time is a fixed *problem initialization* cost; the amount of data that needs to be loaded does not change with the number of nodes or zones. The runtime of 150 cycles of this problem are shown in Table 28 to provide context for the nuclear data load times. Load times using GIDIplus (version 3.19.67) in Ardra are shown in Table 29. For comparison, load times using the legacy NDF library in Ardra are shown in Table 30. All runs were performed using 87 groups, 80 angles, and the ENDL2009.4 nuclear data set. The single-temperature runs assume that everything is at room temperature; this is the more commonly employed option for Ardra user simulations.

A separate, smaller study was done using GIDIplus (version 3.19.68) in Mercury on RZAnsel (ATS-2) and RZGenie (CTS-1). Though the NIF Chamber Problem itself was not run in Mercury, the cost of loading the same isotopes into Mercury was studied using a problem designed for that purpose. The data loaded into Mercury is temperature-dependent with 23 different temperatures. Load times from this study are shown in Table 31.

GIDIplus-Ardra Analysis

The important features of the results from Tables 29 and 30 are:

(1) GIDIplus scales better than NDF with the number of cores.

(2) On a small number of MPI ranks, GIDIplus is much slower (up to an order of magnitude) than NDF.

(3) On a large number of MPI ranks (e.g., on the same order of magnitude as the number of isotopes), GIDIplus load times are comparable to those of NDF.

(4) The differences in performance for low MPI rank counts is less noticeable if temperature-dependent data is loaded.

The reason for (1) is that GIDIplus loading is divided into one file per isotope, whereas NDF loading is divided into 1 file per temperature. The loading of each file can be assigned to a different MPI rank, and thus, there is more I/O parallelism inherent in the GIDIplus approach. Because there is only one temperature in this problem, NDF effectively has no parallelism in Ardra, and its load time does not improve with the number of ranks. (In general, there are two options for temperature effects in Ardra: single temperature and 22 temperatures. For problems using the latter option, there is more parallelism, but it is still limited compared to parallelism over the number of isotopes.)

Features (1)–(3) lead to the following conclusions about the performance of GIDIplus:

(a) GIDIplus is *not* a bottleneck in problems with a sufficient number of MPI ranks (i.e., similar to the number of isotopes) or high fidelity simulations (i.e., problems with a large number of unknowns or time steps). In the former case, its performance is comparable to NDF. In the latter case, the overall simulation time dwarfs the GIDI load time.

(b) GIDIplus can be a bottleneck when the number of MPI ranks is small relative to the number of isotopes and/or the problem is relatively small (1D or coarsely discretized). The impact is felt most strongly when a small problem needs to be run many times (e.g., 1D parametric studies, debugging).

(c) The bottleneck posed by GIDIplus is worse on GPU architectures such as ATS-2 because fewer ranks per node (4 instead of 36) are used in standard practice compared to CTS-1 architectures.

In general, whether the GIDIplus load times shown represent a bottleneck is highly problem-dependent. If the problem is big enough, tens or hundreds of seconds is not significant. If the problem can be parallelized over enough ranks, the GIDI load times can drop below 6 seconds. However, if a user is performing a parametric study that involves hundreds or thousands of small simulations (e.g., 10 minute 1D simulations), then a GIDIplus load time of ∼120–140 seconds is a significant bottleneck.

Conclusion (c) is particularly evident in the ATS-2 runs in Table 29. For the 1-node simulation, the GIDIplus load time is over 10% of the total wall time. However, in the 256-node simulation, it is a negligible 0.6% of the total wall time. We note that these percentages are also impacted by successes in porting Ardra to GPUs: the problem runs more than 10x faster on a per-node basis in ATS-2 than in CTS-1. Because GIDIplus loading is entirely on the CPU and there are fewer MPI ranks for parallelism, the GIDIplus run times do not improve on a node-to-node basis compared to the rest of the Ardra code which has been ported to the GPU.

GIDIplus-Mercury Analysis

Before comparing the Mercury data to the Ardra data, it is important to note the differences between the two codes:

- Mercury loads continuous energy data, while Ardra loads multigroup transfer matrices.
- Mercury load times include conversion of GIDI Protares objects to MCGIDI Protares objects.
- Mercury has a different parallelization approach that requires more memory use and extra

runtime to broadcast the data between ranks. The memory use scales with the number of MPI ranks and Mercury ran out of memory with 8 nodes and 32 ranks on CTS-1.

The differences in the parallelization and data needs of Mercury and Ardra are demonstrated by the results in Table 31. For the same number of nodes, the load times in Mercury are higher than those in Ardra. In Mercury, several copies of data exist in memory as they are copied and broadcast to each MPI rank. This limits the number of MPI ranks per node that can be used to load GIDI data, thus limiting the parallelization and GIDI load times.

	Number of Zones, Total wall times (s)							
	1 Node		8 Nodes		64 Nodes		256 Nodes	
	Zones	Wall time	Zones	Wall time	Zones	Wall time	Zones	Wall time
CTS-1	700k	13042	6M	11729	48M	11817	N/A	
ATS-2	350k	1188	1.8M	903	14M	881	56M	941

Table 28: Wall times in **Ardra** for the NIF Chamber Problem with 359 isotopes. All CTS-1 runs had 36 MPI ranks per node, while ATS-2 runs had 4 MPI ranks per node. The nuclear data for these runs is single-temperature.

		GIDI load time (s)					
	# of Temperatures	1 Node	4 Nodes	8 Nodes	32 Nodes	64 Nodes	256 Nodes
CTS-1	1	32.9	18.0	13.6	N/A	13.7	N/A
ATS-2	1	119.9	N/A	20.8	5.3	5.3	5.3
CTS-1	22	41.2	24.8	18.1	N/A	12.1	N/A
ATS-2	22	140.8	N/A	24.3	9.1	6.0	5.7

		GIDI % of total runtime			
	# of Temperatures	1 Node	8 Nodes	64 Nodes	256 Nodes
CTS-1	1	0.25 %	0.12 %	0.12 %	N/A
ATS-2	1	10.1 %	2.30 %	0.60 %	0.56 %
CTS-1	22	0.32 %	0.15 %	0.10 %	N/A
ATS-2	22	11.9 %	2.69 %	0.68 %	0.61 %

Table 29: GIDIplus 3.19.67 load times in **Ardra** for the NIF Chamber Problem with 359 isotopes. All CTS-1 runs had 36 MPI ranks per node, while ATS-2 runs had 4 MPI ranks per node.

		NDF load time (s)			
	# of Temperatures	1 Node	8 Nodes	64 Nodes	256 Nodes
CTS-1	1	7.5	14.2	21.3	N/A
ATS-2	1	12.6	12.5	12.8	12.7
CTS-1	22	124	41.2	29.7	N/A
ATS-2	22	94.6	18.1	20.7	20.9

		NDF % of total runtime			
	# of Temperatures	1 Node	8 Nodes	64 Nodes	256 Nodes
CTS-1	1	0.06 %	0.12 %	0.18 %	N/A
ATS-2	1	1.06 %	1.38 %	1.45 %	1.21 %
CTS-1	22	0.95 %	0.35 %	0.25 %	N/A
ATS-2	22	7.96 %	2.00 %	2.34 %	2.22 %

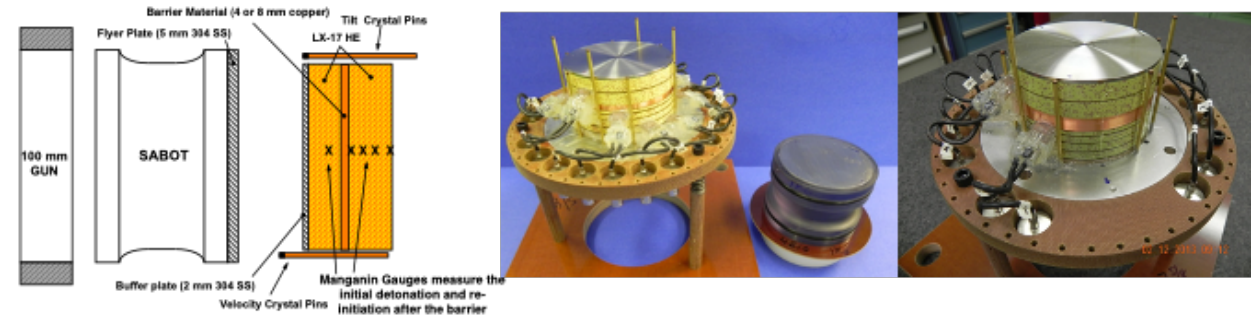
Table 30: NDF load times in **Ardra** for the NIF Chamber Problem with 359 isotopes. All CTS-1 runs had 36 MPI ranks per node, while ATS-2 runs had 4 MPI ranks per node. The increase in the load times with the number of ranks for the CTS-1 is likely due to an issue with the MPI reduction in Ardra.

	GIDI load time (s)		
	1 Node, 2 Ranks	8 Nodes, 16 Ranks	8 Nodes, 32 Ranks
CTS-1	373	101	N/A
ATS-2	N/A	N/A	73

Table 31: GIDIplus 3.19.68 load times in **Mercury** for the NIF Chamber Problem with 359 isotopes.

3.13 Barrier Problem

Name: Barrier Problem (LX-17 Detonation Physics)
Size: 600k–8M zones
Code: ALE3D
Libraries: Cheetah
Physics: Hydro, High Explosive Reactive Flow



Expt	Date	Velocity (km/s)	Flyer	Buffer Plate	Donor LX-17	Barrier	Acceptor LX-17	Backer
4810	12/7/12	2.006 km/s	304 SS, 5 mm	304 SS, 2 mm	20 mm (10, 10 mm)	OFHC Cu, 4 mm	20 mm (3, 5, 6, 6 mm)	Teflon, 10 mm
4811	2/13/13	2.014 km/s	304 SS, 5 mm	302 SS, 2 mm	20 mm (10, 10 mm)	OFHC Cu, 8 mm	20 mm (3, 5, 6, 6 mm)	Teflon, 10 mm

Figure 10: A series of 2 barrier experiments were performed by placing either a 4 mm or an 8 mm barrier of OFHC copper between LX-17 charges. The donor LX-17 charge was shock initiated with a 100 mm diameter sabot with a 304 SS flyer impacting the 90 mm diameter target faced with a thin 304 SS buffer plate. Pressure gauges record the arrival time and peak pressure of the detonation wave.

Description

This problem exercises ALE hydrodynamics with partial Eulerian relaxation (known as backup relaxation) and high explosive reactive flow through Cheetah to simulate the results of a barrier experiment. The problem is typical of many applications in that use of Cheetah was combined with many other materials, including air, copper, teflon, lexan, and stainless steel. All of these materials used an LEOS tabular equation of state. Strength of metals was treated with the Steinberg-Guinan model, while simpler models or no strength was used for other materials. The simulations were run using a range of resolutions, from 600k to 8M zones.

Motivation

- The Barrier Problem is a good validation problem for shock initiation and detonation of insensitive high explosive (IHE). This particular experiment used LX-17 for the high explosive. This problem can be simulated using 2D axisymmetry. The original simulations were used to evaluate various high explosive models: Cheetah, JWL++, CREST, and Ignition & Growth (see Figure 12). The unique part of this simulation is that it evaluates how well our models

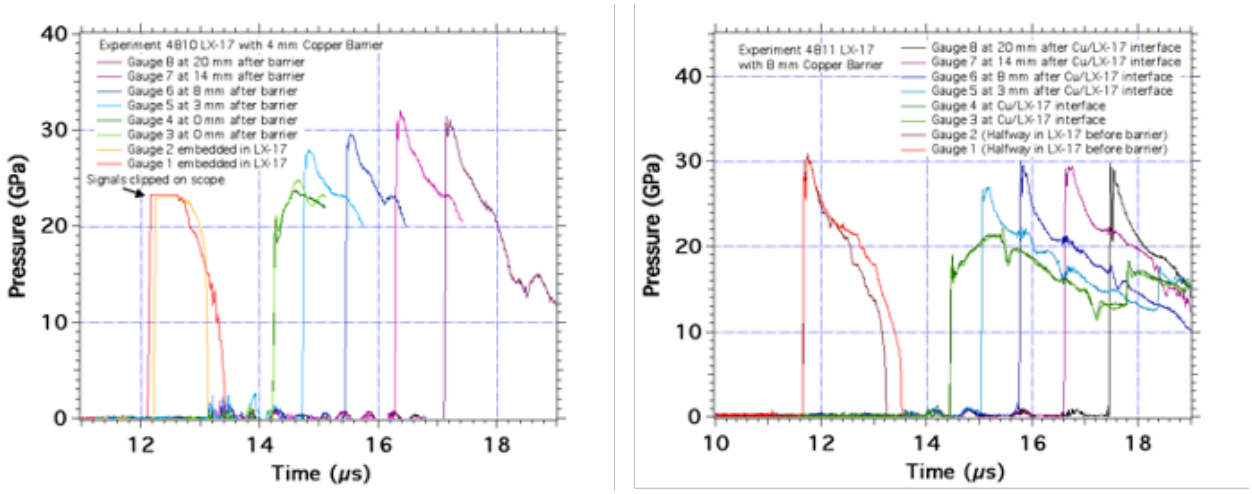


Figure 11: Pressure gauge records of experiment 4810 (left) with a 4 mm barrier and 4811 (right) with an 8 mm barrier. The gauges placed between the 10 mm slices on the donor side was clipped due to the scope setting not set adequately.

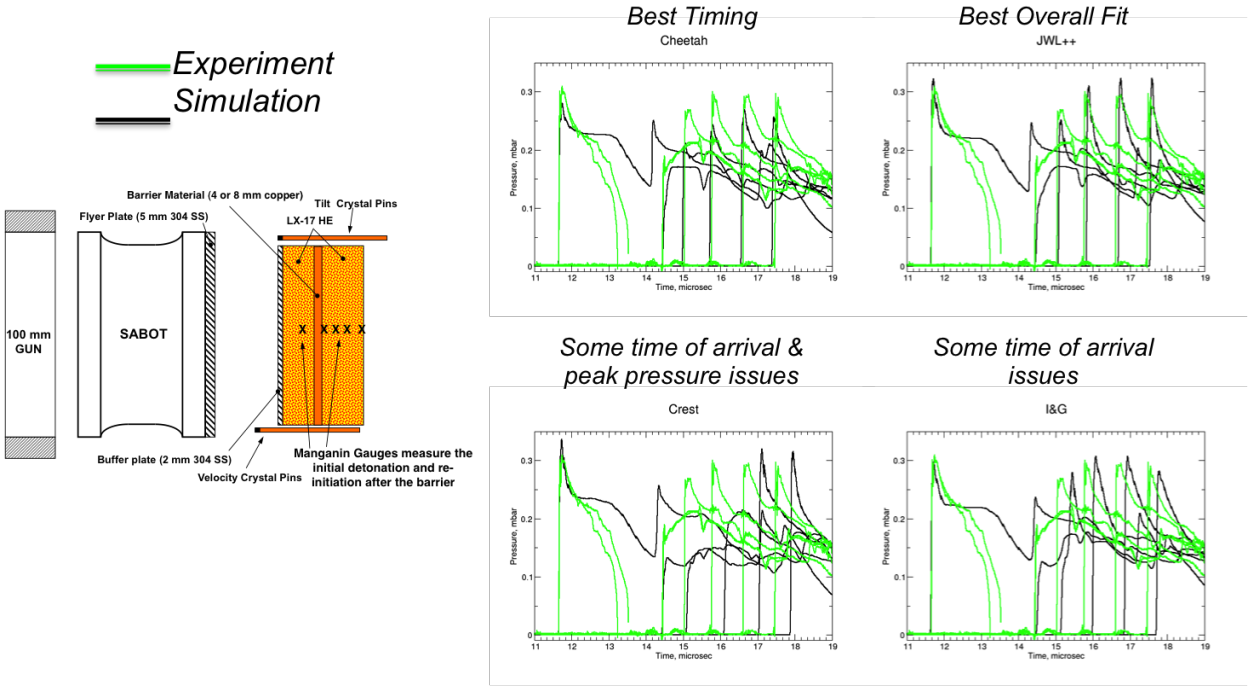


Figure 12: Barrier simulation compared to experiment

can simulate detonation of the high explosive after the shock passes through a 4 or 8 mm copper barrier. This barrier problem demonstrated that Cheetah had the best timing in comparison to the other reactive flow models. In general, ALE3D is used to study the detonation, deflagration, and convective burn processes associated with the energetic response to thermal and mechanical stimuli of both high explosives and propellants. If ALE3D can simulate this type of problem, we have good confidence we can use this for larger programmatic applications, including 3D simulations. Furthermore, we have ported Ignition and Growth to the GPUs, but users also need the ability to use Cheetah on the GPUs, which provides the most accurate solution for various scenarios of interest.

- Sierra and El Capitan are not needed for this 2D axisymmetric problem, but Sierra/El Capitan are needed for large 3D simulations where one needs to use this particular Cheetah reactive flow model. The 3D programmatic problem of interest currently uses 88 nodes on Sierra and 641,000,000 zones but we need to go to much higher resolutions which would put us around 5 billion zones for the next resolution up.
- Cheetah is used when the highest fidelity results are needed and when the user can afford to run at the required zone counts. So, for certain classes of problems, such as determining if and when a high explosive may detonate or when one needs to simulate desensitization of the high explosive, the current state of practice for ALE3D users is to use Cheetah. The state of the art will be to utilize Cheetah on the GPUs for large 3D simulations.

3.14 Barrier results

A scaling study varying the number of zones for the Barrier Problem was run on RZAnsel (ATS-2) and RZTopaz (CTS-1) using 1 node on either resource. A total of 4 GPUs and 4 CPU MPI tasks were used on the RZAnsel node. The RZTopaz node used 36 CPUs and 36 MPI tasks. The results, shown in Figure 13, include a 'rztopaz_alpha' and 'rztopaz_cheetahdev' curve showing the scaling for cases run using intel19 builds of the latest version of ALE3D (v4.33.705) which uses Cheetah v9.0, and the latest develop version of Cheetah (r7336). The 'rzansel_cheetahdev' results were run using a clang build of ALE3D with the develop version of Cheetah (r7336). All cases were run out to 4 microseconds, which is just long enough to detonate the LX17 charge in front of the barrier plate. The y-axis shows the grind time (μs/element/cycle) and the x-axis shows the total number of elements in the problem. All cases are using newly (2020–2021) developed Cheetah vector API, except for the 'rztopaz_alpha' simulation, which did not use the new Cheetah vector API.

Problems were run once with no pre-filled equation of state cache, and a second time with the cache from the first run, which represents nearly perfect filling of the cache file. The first case indicates computational resources required when a problem is first generated from input.

Typically, however, analyses require multiple similar simulations, in which case equation of state cache files can be reused to increase efficiency. The Cheetah code has the capability to pre-fill an equation of state cache at the beginning of the simulation using the GPU, but the barrier test case uses a 4 dimensional cache file that requires up to 4 hours on a single node to fill. Therefore this option would not be attractive on a single node, but the wall clock time of pre-filling would be reduced by N if N nodes were used. Future work will focus on further algorithmic development for cache filling.

The grind time is the average time required to process a single zone during a hydrodynamic cycle. In this problem the number of elements is increased by reducing the mesh element size for a fixed physical problem size. This increases both the number of elements and the number of cycles, due to a reduction in the hydrodynamic time step.

The rztopaz_alpha curve shows the results of the current public release of Cheetah on the TOSS3

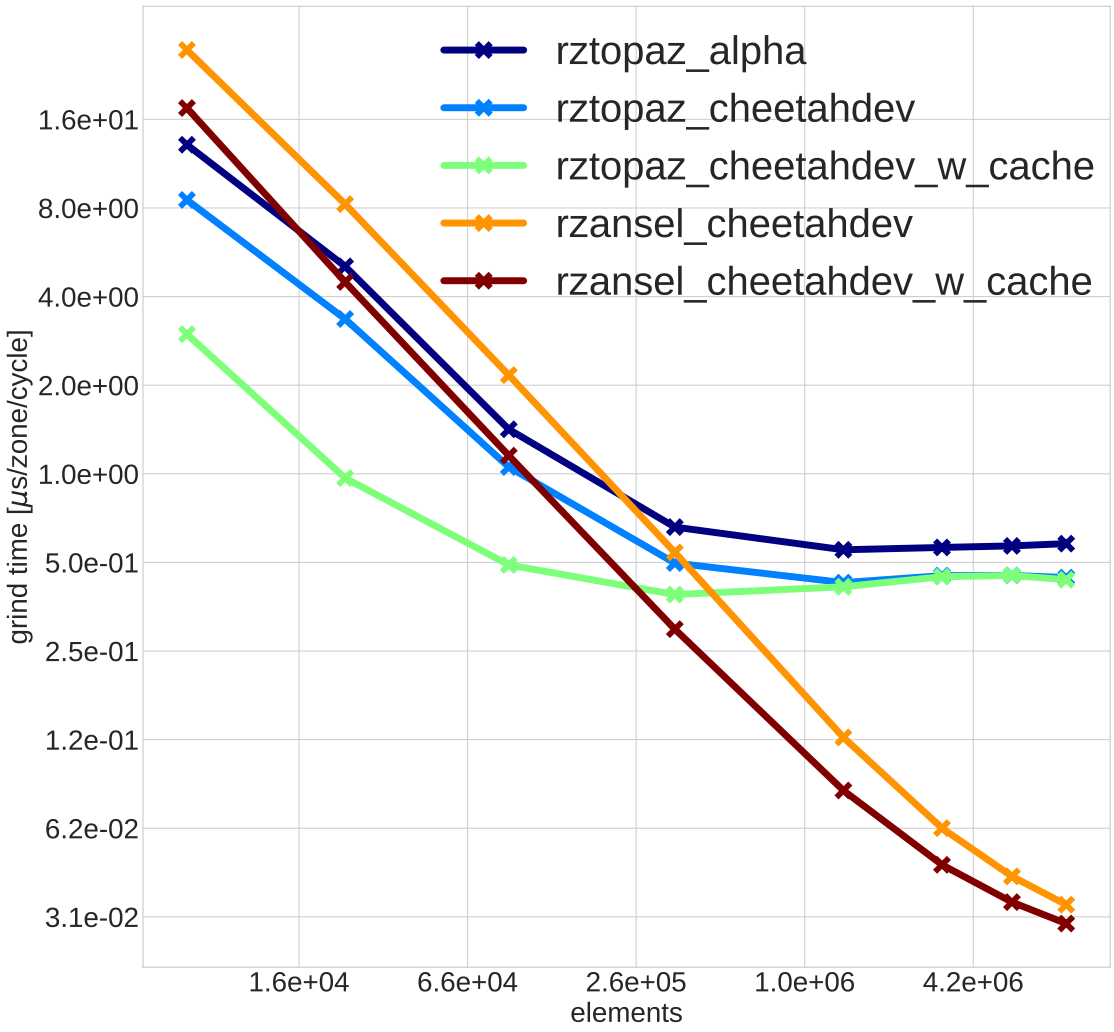


Figure 13: ALE3D grind time as a function of problem size for CPU and GPU platforms. The 'w_cache' simulations use a pre-filled cache saved from a previous run. At the largest problem size, grind time on the GPU is 16x faster than on CPU.

platform using the scalar API. As the problem size is scaled up, the grind time reaches a plateau as the time to process zones dominates the time required to populate the equation of state cache.

rztopaz_cheetahdev shows the grind time for the Cheetah vector API. The use of the vector API allows for more efficient CPU processing of long loops, which leads to a speedup of roughly 40-60%, depending on the number of elements. This shows how code modifications designed for GPU architectures can also make CPU based code more efficient.

The grind time on the GPU-based RZAnsel system shows a steady reduction as the number of elements is increased. At the largest scale studied (roughly 8 million elements), the grind time is 16 times less for one node of GPU-based RZAnsel than for one node of RZTopaz. At the largest scales studied, there is relatively little dependence of the grind time on whether the cache is pre-built or not, showing that the cache algorithm can be highly efficient if the number of zones is large. This behavior can be partly explained by the strong spatial correlation between zones in a problem. Since the zones are highly correlated, adding more zones does not necessarily add more points to the cache file.

The Barrier Problem is a fairly realistic test case in that not all MPI tasks have zones using the Cheetah model. Since Cheetah is more expensive than most material models, better load balancing could further reduce the wall time of this problem. Attempts to improve load balancing by giving Cheetah zones more weight in the partitioning algorithm did not lead to a significant decrease in wall time. This is likely due to the small number of partitions (four) in this problem.

We also evaluated the percentage of wall time devoted to Cheetah by various MPI tasks. Figure 14 shows the results all four MPI ranks of the GPU runs. Cheetah can take a substantial percentage of the total run time, as shown in the figure. The percentage is less for a pre-built cache (rzansel_cheetahdev_w_cache) than for a problem where the cache is populated on the fly (rzansel_cheetahdev). The variation is time across ranks is due to load imbalance.

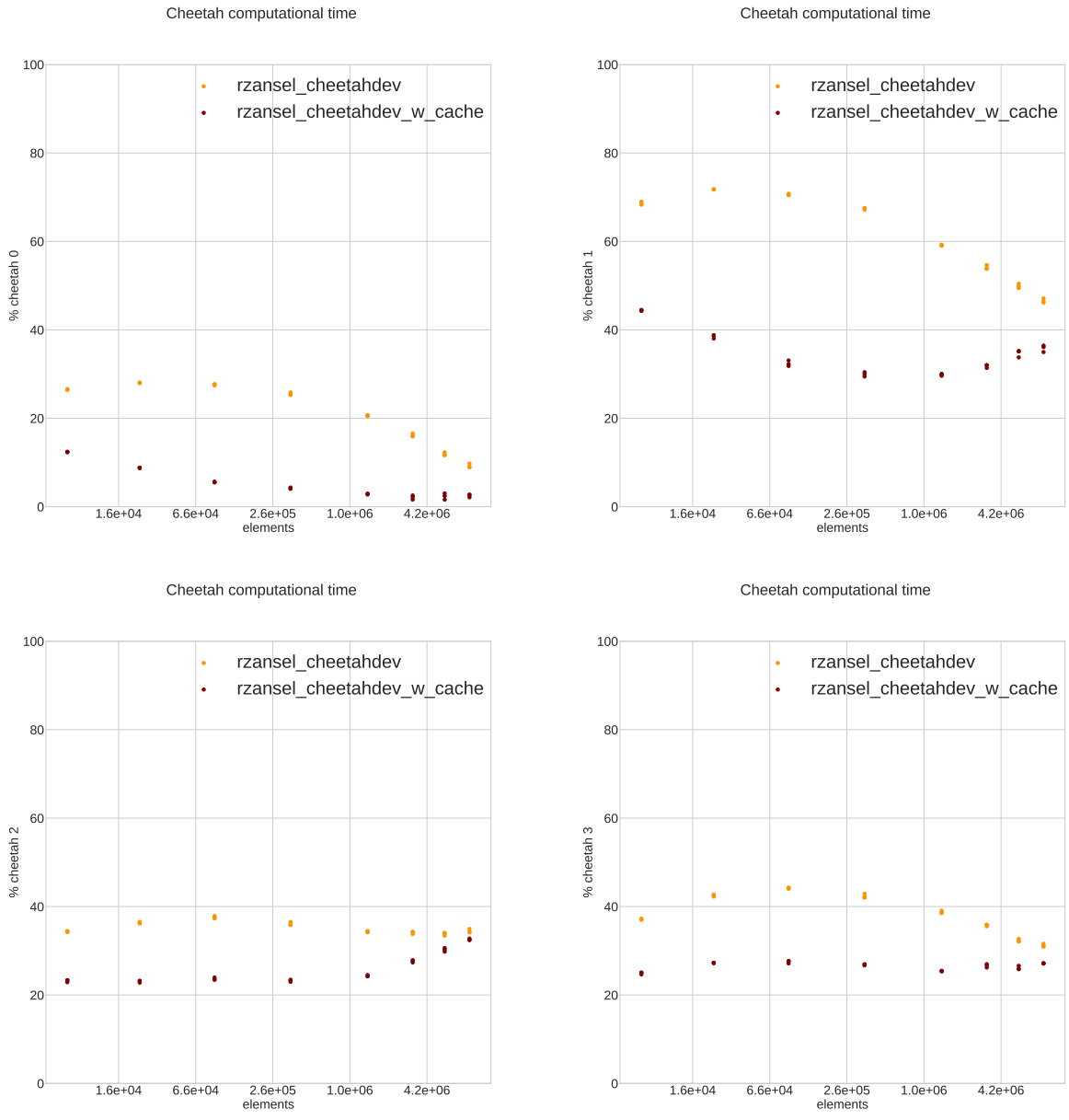


Figure 14: Percentage of total wall time devoted to Cheetah for the GPU runs shown in Figure 13. Variation in times across the ranks is due to load imbalance.

4 PEM Software Porting Highlights and Gaps

Completion criteria #4 and #5 of the milestone are:

4. Highlight porting efforts of PEM software integrated into IC software on Sierra.

5. Identify gaps in current effort to enable future prioritization for LLNL PEM-IC integrated development for Sierra and El Capitan.

This section satisfies these criteria by reviewing the test problem results in Section 3 to collect highlights and gaps for each PEM library.

4.1 LEOS Project

Currently the LEOS and LIP libraries are state-of-the-practice and are used in production on Sierra on a daily basis by all three ASC hydrocodes to run problems utilizing its GPUs. This is a clear highlight for the LEOS team and the PEM program. The performance improvement using GPUs for the Hotspot (Section 3.2) and Shape Charge (Section 3.4) problems is roughly 10x, which is consistent with the speedup seen for other parts of the hydrodynamic algorithm. This speedup ensures that the cost of LEOS evaluations remains a low fraction of the total runtime on GPU architectures.

One clear benefit from these test problems is that work on the Shape Charge problem helped the MARBL team improve strong scaling by a factor of 3 on ATS-2 by switching to pooled allocators for LEOS setup. They also found that adding the vector LEOS interface on improved performance on the CPU by about 2x.

The main gap for LEOS is the work still needed to enable LEOS callbacks from MSLib. Also, collaboration with the Opacity Server team to extend LIP to meet the needs of opacity interpolation would improve modularity.

Some signs of load imbalance are evident in Table 12. However, the total time spent in LEOS is still small so this is not a major concern, especially since the domain decomposition is controlled by the host code.

4.2 MSLib

MSLib has demonstrated an initial GPU capability with good speedups in both the Jetting Problem (8.6x, Section 3.6) and the Nonlocal Problem (26x, Section 3.8). However, providing the full range of features and models desired is still a work in progress. MSLib models that make use of LEOS, Cheetah, or the EOS callback feature do not yet work on GPUs. This is evident in the Jetting Problem where it was necessary to substitute a Mie-Grüneisen EOS within MSLib, rather than using LEOS.

The MSLib team also reports that overall performance is slower than desired on both CPU and GPU. There is a need for vectorization and associated modernization as well as opportunities for optimization for both types of platforms. While the GPU porting effort has mitigated dramatic slowdowns that would be seen if MSLib were required to run on the host, it has not addressed core implementation issues that reduce performance for both CPU and GPU runs.

The benefits of test problems are again evident in the Jetting Problem results in Table 17. The root cause of the unexpectedly large time spent in `map` calls has not been found, but identifying these sorts of issues is a key step in hardening this state-of-the-art code into a state-of-the-practice capability that can be used in routine production.

4.3 TDF Project

The capabilities provided by TDFlib are another highlight of the PEM GPU porting effort. TDFlib is a state-of-the-practice library on GPUs and it is routinely used in production on Sierra.

Before TDF was ported to GPUs host codes running on GPUs could and did call the unported TDFlib from the CPU. This required calling CPU functions from inner loops. When those inner loops were GPU kernels, the routine had to exit the kernel, call the TDFlib routine from the CPU, and then launch a new GPU kernel to continue the calculation on the GPU. Because of the accumulation of many kernel launches within a loop, the cost of accessing TDFlib could be double digit percentages of total runtime in some problems. With the calls accessible from the GPU as __device__ functions, the overhead became negligible.

Data from the Hotspot Problem 3.2 shows that the GPU-enabled version of TDF is taking less than 1% of total runtime on the GPU clearly indicating that there is no concern about needing to optimize TDF further, even looking forward to El Capitan.

The only significant gap for TDFlib is the need to prepare the library for El Capitan. However, as mentioned in Section 2.3 this should be a fairly modest effort. Taking the El Capitan port as an opportunity to simplify the copying of reaction data to the GPU will provide a needed improvement in code simplicity and maintainability.

4.4 Opacity Server

The Opacity Server is the only PEM library considered in this milestone that does not have a functional GPU port. This is a clear gap.

While there have been good rationales to minimize the priority of porting the client lookup functions to GPU, wide adoption of the opacity client to provide lookup functionality, and the resulting gains in software modularity will not occur until lookups achieve good GPU performance. Examples such as the Hotspot Problem (Section 3.2) clearly show the significant performance penalty to keeping opacity lookups on the CPU. Fortunately, there is a clear path to creating a GPU port of the client library and it appears this work can be easily completed before the arrival of El Capitan. The potential integration with LIP mentioned in Section 2.4 is another encouraging direction that is well aligned with the LLNL ASC strategy to modularize code and capture the benefits of re-use.

4.5 GIDIplus Project

The GIDI and MCGIDI components of the GIDIplus library have different use cases and different challenges. For clarity, we consider them separately.

GIDI

Although GIDI load times are not an issue for long simulations, they can be a significant fraction of the simulation time for some production use cases. Several factors contribute to long load times in GIDI:

- ASCII format: processed GNDS libraries are currently stored as large ASCII XML files, which do not support random access. Loading entire XML files requires significant disk I/O time.
- Converting string to doubles: GNDS stores floating-point numbers as ASCII strings using up to 15 significant digits. Converting to doubles (over 150 million floating-point numbers for the n + Cl37 file) requires non-trivial processing time.

- Volume of data: GNDS files store more data compared to the legacy NDF and MCF libraries.

GNDS offers another important advantage compared to NDF and MCF files: rather than having all targets combined in a single file, GNDS libraries are separated in multiple files, one for each combination of Projectile, Target and Evaluation (thus the name 'Protare'). These files are combined into a library using map files. Separating the data into individual Protare files significantly simplifies uncertainty quantification and sensitivity studies by allowing individual Protares to be swapped in and out of a library.

Several strides have recently been made towards improving GIDIplus load times. These include 1) introducing a 'hybrid' GNDS format with the hierarchy stored in XML and numbers stored in binary, random-access HDF5 files, 2) exploring the use of compression in HDF5 to further reduce disk space and I/O requirements, 3) precomputing sums to reduce how much data must be loaded for typical deterministic transport problems and 4) removing redundant data from GNDS files.

To facilitate the first and second efforts, a new compatibility layer called 'HAPI' was introduced into GIDIplus. HAPI (Hierarchical API) provides a transparent way to read data from either XML or from HDF5. This capability was recently introduced in a beta version of GIDIplus 3.20, and is currently being tested in transport codes. A hybrid XML/HDF5 version of the ENDL2009.4 library was also published on LC systems for testing. Because HDF5 supports random access, does not require a string-to-binary conversion, and can store compressed data, there are opportunities for improvement in the load times. Preliminary results indicate that loading uncompressed hybrid XML/HDF5 data through HAPI could speed up the load times by up to a factor of 6.6x. However, this depends on several factors, such as whether or not the data is cached locally. More research is needed to fully realize performance gains.

The third effort (precomputing sums of multi-group data) is driven by the fact that users almost always want full transfer matrices summed over all reactions. While GNDS files will continue to store transfer matrices by reaction to enable finer grained sensitivity studies, they can also store the summed matrices to reduce data load times. This effort is not yet complete, and will require changes both to GIDIplus and to the nuclear data processing code FUDGE (For Updating Data and Generating Evaluations).

The fourth effort to improve data load times revolves around identifying and removing redundant data from processed GNDS files. GNDS supports linking from one resource to another, so repeated data (such as multi-group bin boundaries) should be defined only once with all other instances replaced by links. A new GNDS format proposal (currently under review) would also support defining default units in a single location, further reducing redundant data.

One other gap was noted in the integration of GIDI into integrated codes. The burden of parallelizing file reading with GIDI is left to the integrated code. Because of this, significant scaling performance differences are seen between the different codes using GIDI (e.g., Mercury vs. Ardra for the loading of the NIF Chamber isotopes). Moreover, if an integrated code does not make an effort to parallelize the loading of data with GIDI, the performance will be significantly hampered. In the future, development of a helper library to parallelize the loading of GIDIplus data would make GIDI performance more uniform (and more efficient) across integrated codes.

MCGIDI

Based on the performance of the Godiva Problems in Section 3.10 our best assessment of MCGIDI performance is that the GPU port is performing well enough not to be a bottleneck in Monte Carlo transport. Unfortunately, our confidence in this assessment is low. There are at least three reasons for this lack of confidence:

- As explained in Section 3.10 and Appendix A we were unable to directly measure the performance of MCGIDI on GPUs. Our assessment is based only on the fact that problems dominated by collisions (which are calculated by MCGIDI) exhibit higher GPU speedups than problems dominated by mesh facet crossings. This implies that MCGIDI has better GPU speedup than other parts of the calculation and therefore is not a primary bottleneck for performance.
- Performance of Monte Carlo can vary significantly from problem to problem. Obtaining performance data from a larger variety of problems will be necessary to determine whether MCGIDI will satisfy performance requirements across production workloads.
- The GPU port of Mercury is under active development and performance continues to evolve and improve. As various parts of the Monte Carlo calculation are optimized the relative contribution of MCGIDI will necessarily increase unless MCGIDI is also further optimized.

Clearly it will be necessary to continue to monitor MCGIDI performance as the GPU port of Mercury is further optimized. Working with vendor partners to develop tools and techniques to quantitatively assess the performance of MCGIDI on the GPU will also be an ongoing activity.

4.6 Cheetah

The results of the Barrier Problem (Section 3.14) clearly show that Cheetah can obtain good speedups on GPUs. For simulations with 1–2 million elements per GPU, Cheetah demonstrates a speedup of approximately 8–16x. This simulation size is consistent with current recommendations for multiphysics simulations on GPUs so it is likely that similar speedups will be observed in production problems.

While this is a significant achievement, it is important to remember that this result was obtained using state-of-the-art code that is still in development branches. Considerable effort will be needed to test this code and harden it for production. For example, one significant problem observed while running test problems was that Cheetah's memory usage pattern led to runaway memory usage by Umpire. This was not a typical memory leak. Instead, Cheetah's pattern of storing and freeing memory when the number of zones sent to the GPU increased throughout the simulation led to fragmentation of the Umpire memory pool. The fragmented Umpire pool required new allocations that continually increased pool size until memory was exhausted. The Cheetah team is working to change its usage of Umpire to avoid this problem, while the Umpire team is developing new memory management heuristics to reduce the possibility of severe memory pool fragmentation.

The Cheetah team is developing additional techniques to improve performance that were not ready to be employed for this milestone due to limited code stability and maturity.

Improving performance of cache filling

Performance of the current version of Cheetah is limited by the fact that all of the non-linear solves that are needed for on-the-fly population of the EOS cache take place on the CPU. Moreover, some Cheetah inputs cover a larger, more complex region of the cache phase space. Therefore populating the cache on the CPU may not be an acceptable algorithm for all problems. Two possible strategies to mitigate this problem are in development.

The first strategy is on-the-fly speculative database filling (OSDF). The OSDF approach uses the real-time hydrocode EOS demands to generate a large set of EOS points that are likely to be needed in the next few simulation time steps to further advance the calculation. These speculative thermodynamic points are first sorted, and then their calculation is sent to the GPUs using asynchronous threading. GPU calculations are performed with a thread-safe, streamlined version

of the Cheetah solver, which has a slightly lower success rate than the full Cheetah solver. This "fast" solver aims to converge directly to solution and employs stricter stopping criteria, thereby avoiding excessive branching and GPU load imbalance due to exceedingly long times to solution for certain thermodynamic points. The approach preserves overall GPU performance while avoiding solution bottlenecks. OSDF is in principle effective for all types of Cheetah thermochemical and kinetic models, but may require further refinement of speculative points selection, as well careful computational and memory resource management jointly with the hydrocode to achieve optimal speeds. One possible disadvantage is that for large scale problems OSDF may compete with the hydrocode for GPU resources, and thus limit the maximum computational speeds achievable.

To alleviate this problem, a second database filling strategy, up-front dense database filling (UDDF) has been implemented. UDDF is targeted at lower complexity Cheetah models, with reduced number of kinetic variables, from 1 to 3, where the EOS database dimensionality is only 3 to 5; such models are in wide use due to their robustness and efficacy. In such cases it may be advantageous, depending on the size of the hydrodynamic problem, to densely fill the entire cache database before the start of the hydrodynamic simulations. UDDF employs the full computational resources allocated to the problem and distributes all the database points to the GPUs for calculation using the fast solver. Due to GPU memory limitations, it is especially important in this case to select appropriate bounds for the database variables to avoid generating too many EOS points that are unlikely to be needed by the hydrocode. Cheetah employs energy bounds based on density and temperature, which are amenable to intuitive physical limits. For kinetic variables, the concentration bounds are set either based on their initial values for species that decompose irreversibly, or on stoichiometric and physical constraints. The generation of databases containing 10s of millions of points is very efficient and should take a small fraction of simulation times for problems targeting 10's of nodes. Once the database has been generated, a wide range of hydrodynamic problems exploring its phase space can be run with maximal computational speeds.

Improving performance of chemical kinetics

The batched vector kinetics employed in Cheetah to solve chemical kinetics has the limitation that the time step is limited by the stiffest zone in the problem. It is often the case that a small number of zones in a problem will have unphysical densities and/or energies due to spatially localized numerical errors. Therefore it is possible that the solution of the entire problem could be limited by a few bad zones. A new kinetic cache algorithm which interpolates kinetic solutions on the GPU but solves them on the CPU does not have this limitation. This method offers the highest throughput when a full database of kinetic solutions is known or when the solution of the kinetic equations is computationally expensive.

Currently the kinetic cache is populated on the CPU. The input grid dimension of the kinetic cache is one greater than that of the EOS cache; the extra dimension is time. Due to the higher dimensionality, the kinetic cache can take more evaluations to fill than the EOS cache. However, less data per entry is stored, so its memory usage is typically similar to that of the EOS cache. When the kinetic cache is employed, it makes use of the EOS cache, leading to a "double-cache" algorithm.

The Cheetah team is studying the relative efficiency of the kinetic cache and vectorized kinetics. The kinetic cache offers roughly 50% speed-up over the more standard vectorized kinetics for simple chemical kinetic rates once the kinetic cache is populated, but it could offer greater speedups for more complicated chemical kinetic problems.

4.7 Summary

With the exception of the Opacity Server, all of the PEM libraries have demonstrated GPU capabilities with encouraging speedups. In the test problems considered here, the fraction of runtime used by the PEM libraries on GPUs is as good as on CPUs, indicating that the libraries will not be a performance bottleneck. Still, there is considerable work to be done to increase the range of features that are ported and optimized for GPUs, as well as testing and hardening code for production.

5 Conclusions and Recommendations

The information presented in this report fully satisfies all of the milestone completion criteria. We have summarized the required information on PEM libraries and IC codes, discussed the porting progress and challenges, defined test problems, and used the results of those test problems to help identify highlights and gaps of the integration between PEM and IC software.

To conclude the report, significant conclusions, observations, and recommendations from across the report are collected here for ease of reference and review.

1. The most important highlight of this report is that LEOS, MSLib, TDF, GIDIplus, and Cheetah have all ported at least some features to the GPU and have demonstrated speedups similar to other components of multiphysics codes. This significantly reduces concerns that PEM libraries will be a performance bottleneck on GPUs. While these successes are very encouraging, continued work is still needed. In particular:
   - LEOS and Cheetah should work with MSLib to complete work on callbacks.
   - The Cheetah team should continue work on the the GPU solver, the kinetics cache, and improved cache-filling approaches. These features will improve performance and extend the range of problems that can be accelerated.
   - The MSLib team should expand the number of ported models and pursue modernization and optimization of their code.
   - For GIDIplus, although GIDI load times are not a significant factor for large simulations, there are use cases where the load time is a bottleneck. Work should be completed to resolve this problem once and for all. Work to better characterize MCGIDI performance should also be a priority.
   - To improve performance and obtain benefits of modularity, the Opacity Client should be ported to GPU. As part of this effort, the LEOS team should collaborate with the Opacity Server team to explore extensions to LIP to support the Opacity Client port.
2. Work will be needed to harden codes for production and improve overall robustness. Expert help from both IC and PEM teams was frequently needed to complete test problems and obtain the best performance. Lessons learned will need to be documented for users and incorporated into codes to provide production-ready capabilities.
3. Coordinating the use of memory and managing the size of data will require continued attention. This is a multi-faceted problem:
   - The memory pool fragmentation problem described in this report is not unique to Cheetah. Other codes are experiencing similar issues. Since memory pools are essential for performance, it will be necessary to identify and implement best practices for their use. This is likely to include elements such as using different pools for temporary vs. permanent data or segregating data by size into different pools. Past coding patterns that treated all memory allocations equally are likely to lead to problems.
   - Using Umpire to coordinate pools between host and libraries is a best practice that should be encouraged.
   - Vector interfaces can consume considerable memory for temporaries. Libraries should be aware of their needs for temporary memory and seek to minimize it insofar as possible.
   - This size of data tables has traditionally been a primary concern for PEM libraries. However, this may be less of an issue as GPU memory capacity increases. Even large tables could consume only a small fraction of GPU memory. If it becomes desirable to

run multiple MPI ranks per GPU, then facilities to share tables across ranks in GPU memory would become important, just as they are on CPU platforms.

4. This milestone again demonstrates the high value of test problems. In the course of obtaining data for this report several performance-limiting issues were found and fixed. Test problems served as focal point to drive collaboration between teams and resolve problems. Some issues, such as the pool fragmentation issue in Cheetah, are still to be resolved. However, exposing such problems is the first step to fixing them. These test problems as well as additional problems as needed should be used to focus attention on important use cases and identify needed optimizations.
5. Collecting performance data was not always as easy and straightforward as it could be. Work should be done to make it easier to collect and analyze performance data, especially for complex simulations that span hundreds of ranks or more.
    - The most obvious example is the collection of fine-grained GPU performance data for MCGIDI. Finding solutions to that problem will likely require ongoing effort with vendors.
    - We also encountered issues collecting data from the internal counters in the codes. It is a common practice for at least some codes to look at performance data only on rank 0. Too often this produces a distorted picture. Even when performance timers are collected across all ranks, methods to analyze the data are ad-hoc.
    - We see signs of load imbalance in some test problems. However, load imbalance can be particularly tricky to measure with in-situ timers. Care should be taken to ensure that timers are designed and implemented to segregate useful work from idle time due to load imbalance.
    - It would also be beneficial to deploy methods to capture performance data from production runs. This would help identify opportunities to improve performance by tuning run parameters or by identifying code paths for optimization that that are not evident in cases typically run by developers.
6. The path to El Capitan appears to be smooth. RAJA and HIP are expected to provide the necessary portability abstractions and the necessary code changes should be minimal. Of course, there is risk in any architectural transition and teams should be diligent and proactive to take advantage of early hardware to ensure performance meets expectations and resolve any issues as early as possible.

Acknowledgments

This work would not have been possible without the contributions of many dedicated and talented people. We gratefully acknowledge the contributions of our contributing authors as well as everyone who has played a role in developing or maintaining IC codes and PEM libraries.

Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344.

References

[1] URL: http://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0009r10.html.

[2] Adiak: Standard interface for collecting HPC run metadata. https://github.com/LLNL/Adiak. Accessed: May 9, 2021.

[3] N. R. Barton, A. Arsenlis, M. Rhee, J. Marian, J. V. Bernier, M. Tang, and L. Yang. A multi-scale strength model with phase transformation. *AIP Conference Proceedings*, 1426:1513–1516, 2012. URL: http://dx.doi.org/10.1063/1.3686570.

[4] Nathan Barton. Results from a new Cocks-Ashby style porosity model. *AIP Conference Proceedings*, 1793(1):100029, 2017. URL: http://dx.doi.org/10.1063/1.4971654, arXiv:http://aip.scitation.org/doi/pdf/10.1063/1.4971654, doi:10.1063/1.4971654.

[5] Nathan R. Barton, Robert A. Carson, Steven R Wopschall, and USDOE National Nuclear Security Administration. ECMech, 12 2018. URL: https://github.com/LLNL/ExaCMech, doi:10.11578/dc.20190809.2.

[6] Zdenêk P. Bažant and Milan Jirásek. Nonlocal integral formulations of plasticity and damage: Survey of progress. *Journal of Engineering Mechanics*, 128(11):1119–1149, 2002. URL: https://dx.doi.org/10.1061/(ASCE)0733-9399(2002)128:11(1119), doi:10.1061/(ASCE)0733-9399(2002)128:11(1119).

[7] B. Beck and C. M. Mattoon. Gidiplus, 2021. URL: https://github.com/LLNL/gidiplus.

[8] D. Böhme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Giménez, M. LeGendre, O. Pearce, and M. Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *Supercomputing 2016 (SC'16)*, Salt Lake City, UT, November 13-18 2016. LLNL-CONF-699263.

[9] P. Brantley, R. Bleile, S. Dawson, S. McKinley, M. O'Brien, M. Pozulp, R. Procassini, D. Richards, A. Robinson, S. Sepke, and D. Stevens. Mercury user guide: Version 5.26.2. Technical Report LLNL-SM-560687 (Modification #23), Lawrence Livermore National Laboratory, 2021.

[10] P. S. Brantley, C. A. Hagmann, and J. A. Rathkopf. MCAPM-C generator and collision routine (Gen2000/Bang2000) documentation (revision 1.2). Technical Report UCRL-MA-141957, 2003.

[11] J. B. Briggs, editor. *International Handbook of Evaluated Criticality Safety Benchmark Experiments*. Organization for Economic Co-operation and Development - Nuclear Energy Agency, NEA/NSC/DOC(95)03/I-IX, Paris, France, 2012.

[12] E. Cullen, C. J. Clouse, R. Procassini, and R. C. Little. Static and dynamic criticality: Are they different? Technical Report UCRL-TR-201506, Lawrence Livermore National Laboratory, November 2003.

[13] F. Di Natale. Maestro workflow conductor, 6 2017. URL: https://www.osti.gov/biblio/1372046.

[14] V. A. Dobrev, Tz. V. Kolev, and R. N. Rieben. High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM Journal on Scientific Computing*, 34(5):B606–B641, 2012.

[15] R. D. Hornung, A. Black, A. Capps, B. Corbett, N. Elliott, C. Harrison, R. Settgast, L. Taylor, K. Weiss, . White, C, and G. Zagaris. Axom, 10 2017. URL: https://www.osti.gov/biblio/1408513, doi:10.11578/dc.20201027.5.

[16] D.J. Luscher, Nathan Barton, Scott Crockett, Ann E. Wills, Carl Greeff, Leonid Burakovsky, and Sky Sjue. Working draft 1.1: A proposed common model of multi-phase strength and equation of state for a tri-laboratory collaboration. Technical Report LLNL-TR-814438 / LA-UR-20-26489, Lawrence Livermore National Laboratory / Los Alamos National Laboratory, 2020.

[17] T Pardoen, Y Marchal, and F Delannay. Thickness dependence of cracking resistance in thin aluminium plates. *Journal of the Mechanics and Physics of Solids*, 47(10):2093–2123, 1999. URL: http://www.sciencedirect.com/science/article/pii/S0022509699000113, doi:https://doi.org/10.1016/S0022-5096(99)00011-3.

[18] W. Walters. A brief history of shaped charges. Technical report, Army Research Lab, Aberdeen Proving Ground, MD. Weapons and Materials Research, 2008.

[19] Brett M. Wayne, Nathan R. Barton, and USDOE National Nuclear Security Administration. SNLS, 9 2018. URL: https://github.com/LLNL/SNLS, doi:10.11578/dc.20181217.9.

Appendix A Program Counter Sampling

In Section 3.10 Godiva Results we used program counter (PC) sampling to estimate the percent of Mercury runtime in the GIDIplus library. The maximum possible Mercury speedup that we can achieve by modifying GIDIplus is given by Amdahl's law

$$s(p) = \frac{1}{1-p} \tag{1}$$

where p is the fraction of Mercury runtime in GIDIplus. Table 32 has some example values of s for different p.

p	0.95	0.50	0.20	0.10	0.01
s	20	2	1.25	1.11	1.01

Table 32: Examples of the maximum possible Mercury speedup s that we can achieve by modifying GIDIplus when GIDIplus takes a fraction p of the total Mercury runtime.

If p is large then GIDIplus may provide the best opportunity for making Mercury run faster. Sections A.1 and A.2 describe how we use PC sampling to find p. Section A.3 discusses alternatives to PC sampling.

A.1 Google Performance Tools PC Sampling

The PC sampling results that we presented in Tables 24 and 25 come from Google Performance Tools (`gperftools`). Listing 1 demonstrates how to use `gperftools`. Lines 2, 4, and 6 are the code changes required to generate a PC sampling profile. Lines 9 and 10 show how to compile and link with `gperftools`. Lines 12 and 13 show how to use the `gperftools pprof` script to convert the binary profile to human-readable form.

Listing 2 shows some of the `gperftools` text report, and Figure 15 shows the `gperftools` PDF report. We estimate p by taking the sum of the PC samples in the GIDIplus subtrees and dividing by the total number of samples. Before we divide we subtract the PC samples in the callback subtree from the sum[5]. Table 34 shows how we computed our estimate of p for the continuous energy **Godiva Sphere** calculation on CTS-1.

A good PC sampler introduces minimal overhead. Table 24 shows that the `gperftools` PC sampler caused the 455 second problem to run for 466 seconds, a slowdown of only 2.6%. We can increase or decrease the overhead by setting environment variables which modify the PC sampler settings. For example, `gperftools` reads `CPUPROFILE_FREQUENCY=x` to set the sampling frequency. We used the default frequency of 100 Hz.

We are very satisfied with `gperftools`.

A.2 Nvidia Nsight Compute PC Sampling

We did not present PC sampling results from ATS-2 in this report because of difficulties we encountered trying to use Nvidia Nsight Compute (`ncu`). We can not use `gperftools` on ATS-2 because `gperftools` does not sample device code. On ATS-2 we use the tool that Nvidia recommends, which is `ncu`. Table 33 compares the `gperftools` and `ncu` PC samplers.

[5]The callback functions that we register with GIDIplus contain Mercury code, not GIDIplus code, so we do not include them in p. Since it is Mercury code which gets executed as a part of the GIDIplus programming model we consider it a third category - not GIDIplus code, but not exactly Mercury code either.

	gperftools	ncu
Supported on	CTS-1	ATS-2
Slowdown	1.02x to 1.09x	10,000x to ∞ (hangs)
Can sample code on	Host only	Device only
Designate which code to sample by	Calling Start/Stop	Kernel name regex
Recompilation required for	Executable	Libraries and executable
Captures stack traces	✓	✗
Usable without professional help	✓	✗

Table 33: Comparison of the `gperftools` and `ncu` PC samplers.

Listing 3 shows **step 1** to use `ncu` PC sampling: recompiling all libraries which contain device code and then relinking Mercury. To save space, library sources can be deleted after compilation finishes, which is the default behavior of `spack`. Using the `--keep-stage` argument tells `spack` not to delete them. We do this because `ncu` needs the sources; `ncu` will emit warnings or hang if it cannot find them.

Step 2 is determining a regular expression that matches the name of the kernel(s) that we want to sample. By default `ncu` does not sample anything and there is no option to sample everything. One way to find the name of kernels is to run Mercury with Nvidia's legacy `nvprof` and pipe the output through `c++filt` to demangle the symbols. Another way is to run Mercury with Nvidia Nsight Systems (`nsys`), click the bars in the timeline which correspond to kernels, copy the kernel name that appears and run the name through `c++filt`.

Listing 4 shows **step 3**: invoking `ncu` to generate the PC sampling profile. Table 35 explains the `ncu` arguments that we used.

Listing 5 shows **step 4**: converting the binary profile to human-readable form. This step requires opening the binary profile in the `ncu-ui` GUI, which has to be done on an x86 machine, and clicking to export the PC profile for each kernel invocation one-at-a-time. This is very tedious.

Listing 6 shows a few lines of the CSV file that we exported in **step 4**. The CSV contains 85 source files concatenated together. The comma-separated values include the line number in the source file (value 0), the line of source code (value 1), the number of times that PCs corresponding to the line were sampled (value 3), and 51 other values.

Listing 7 shows **step 5**, the final step, parsing the CSV file to find p. The script finds the boundaries of the source files in the CSV, then takes the sum of PC samples for all lines of source code. This is the total number of samples. The script then takes the sum of samples for all lines in GIDIplus files and divides by the total to get p.

Listing 8 shows the output from running the Python script in **step 5**. The p value in Listing 8 is for a single kernel invocation in a tiny Mercury problem which runs for 13 seconds. Sampling just 1 kernel invocation makes the problem run for 26 seconds. Since the kernel takes only a few milliseconds without sampling, it seems that the 2x increase in total runtime comes from a 10,000x slowdown in the sampled code. The `ncu` slowdown is 6 orders of magnitude greater than the `gperftools` slowdown.

We think different kernel invocations may give different p values. One part of the 5-step `ncu` workflow that makes looking at different kernel invocations difficult is the tedium of clicking in `ncu-ui` to export a CSV for each kernel invocation. Our Nvidia consultant gave us a new workflow which avoids `ncu-ui`. The new workflow picks up after **step 2** in the previous workflow. We will now use letters to denote the steps in the new workflow to distinguish them from steps in the original workflow.

Listing 9 shows **step A**: copying `sections` and `SourcePage.py` into place. The `sections` directory is part of the `ncu` installation. Listing 10 shows `SourcePage.py`, a file that was given to us by our Nvidia consultant. Lines 57, 63-65, and 67 are our additions. Along with our modifications to lines 72, 76, and 79, our additions cause `SourcePage.py` to write the CSV to a file instead of stdout. We write one CSV per kernel invocation.

Listing 11 shows **step B**: invoking `ncu`. We quote an informal email from our Nvidia consultant who explains the behavior of this new invocation and how `SourcePage.py` is used:

> The key here is that we make a copy of the "sections" directory (which ships with Nsight Compute) locally and then add our custom file SourcePage.py to that directory. Then, at the command line, we specify to use this sections directory rather than the default one in /usr/tce. From then, we can use the tool normally. The invocation I've shown above is the minimal invocation needed to collect just the source counter data, and right now the custom tooling just collects PC samples and nothing else. You could also use it with your existing `--set detailed` if you want additional metrics, as long as you remember to do the `--section-folder` specification.
>
> At present, we just loop through all files that contributed to each kernel launch and print out to stdout the line of source (column 1), the number of samples (column 2), and the source code (everything else). Please give this a whirl and see if this data would be useful to you. If so, we can work out the details of finishing it up for exactly what you need, like summing up over kernel invocations, writing to a file, etc. It's a pretty straightforward Python script and will be easy to modify for our needs.

Listing 12 shows a few lines of the CSV file that we exported in **step B**. The CSV contains 79 source files concatenated together. The colon-separated values are the line number in the source file (value 0), the number of times that PCs corresponding to the line were sampled (value 1), and the line of source code (value 2).

Listing 13 shows **step C**: parsing the new CSV file to find p. One great thing about the new workflow is that the CSV contains fully-qualified paths so we can be sure which of the 79 files belong to GIDIplus; it can be difficult to tell with just the basename.

Listing 14 shows the output from running the Python script in **step C**. Once again, the p value in Listing 14 is for a single kernel invocation in a tiny Mercury problem. Unfortunately, `ncu` hangs if we try to sample more than 1 kernel invocation. Our Nvidia consultant reproduced the problem and attributed it to a quadratic-time algorithm in `SourcePage.py` which causes Mercury to appear to hang because of the large amount of code in the tracking kernel. Once again we quote informal email correspondence with our Nvidia consultant:

> It looks like the issue here is that the algorithm currently written scales really poorly - the kernel has got O(100k) metric events and for each one we are searching through your O(100k) lines of source to find the match. So the code is not hanging it's just running very very slowly. I am seeing if I can rewrite it to be more efficient.

We enjoyed working with our Nvidia consultant and we will continue to pursue a solution for PC sampling on ATS-2 until the machine retires. We wish it was easier.

A.3 Alternatives to PC Sampling

Internal timers written in the source code are an alternative to PC sampling. We use internal timers in Mercury to understand where in the code we are spending our runtime. We do not time

GIDIplus because of the overhead that it would introduce. Mercury's particle processing rate is between 1 million and 1 billion Hz. Each segment that a particle traverses requires 1 or more GIDIplus calls. The clock rate of the CTS-1 and ATS-2 processors is order GHz, or 1 billion Hz. The overhead of starting and stopping a timer at the same frequency as the processor clock rate would disturb the calculation to the point where no useful information could be obtained.

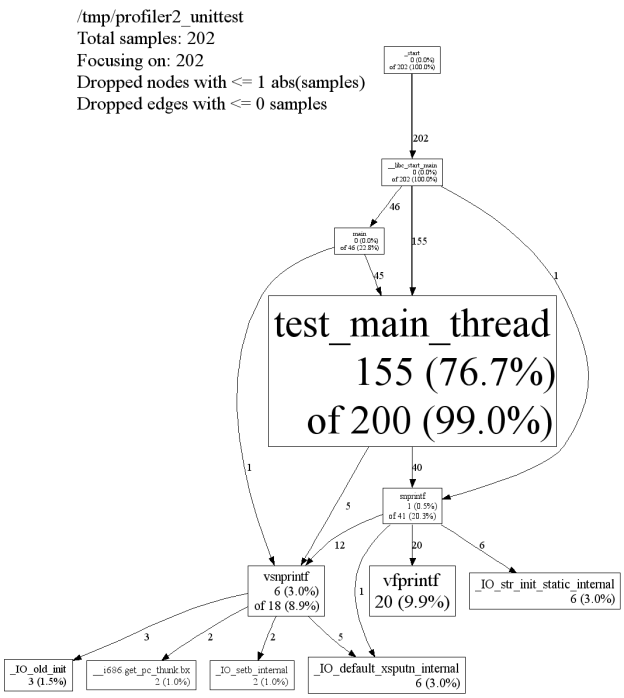


Figure 15: Example PDF report from `gperftools` PC sampling, taken from https://gperftools.github.io/gperftools/pprof-test-big.gif. The graph is a bit larger for Mercury but the structure is the same.

Listing 1: Using `gperftools` PC sampling.

```
$ cat main.c
#include "gperftools/profiler.h"
int main(int argc, char **argv) {
    ProfilerStart("mypcsamples.pro");
    foo();
    ProfilerStop();
    return 0;
}
$ gcc -c -o main.o main.c -I/path/to/gperftools/include
$ gcc -o main main.o -L/path/to/gperftools/lib -lprofiler -lunwind
$ ./main
$ pprof --text main mypcsamples.pro > mypcsamples.pro.txt
$ pprof --pdf  main mypcsamples.pro > mypcsamples.pro.pdf
```

Subtree root	Sign	Samples	Percent of total
`MCGIDI::Product::sampleProducts`	+	193920	36.6
`MCGIDI::HeatedCrossSectionsContinuousEnergy::sampleReaction`	+	17229	3.3
`MCGIDI::DomainHash::index`	+	8536	1.6
`MCGIDI::HeatedCrossSectionContinuousEnergy::crossSection`	+	19074	3.6
`GIDI_Product_Handler::push_back`	-	109732	20.7
		129027	24.4

Table 34: Estimating p using the `gperftools` PC sampling profile.

Listing 2: The first 3 lines of the text report from `gperftools` PC sampling.

```
Total: 529895 samples
   29732   5.6%   5.6%    32315   6.1% MCGIDI::binarySearchVector (inline)
   18601   3.5%   9.1%    46941   8.9% MCGIDI::Probabilities::Xs_pdf_cdf1d::sample
```

Listing 3: Using `ncu` PC sampling **step 1**: recompile libraries which contain device code.

```
$ git clone ssh://git@rz-bitbucket.llnl.gov:7999/spack/llnl.wci.git
$ cd llnl.wci
$ git remote add poz ssh://git@rz-bitbucket.llnl.gov:7999/~pozulp1/llnl.wci.git
$ git fetch poz
$ git checkout feature/pozulp1/nvcc_lto_gidiplus
$ cd ..
$ git clone https://github.com/spack/spack.git
$ cd spack
$ git checkout v0.14.2
$ cd etc/spack/defaults
$ for f in ../../../../llnl.wci/*.yaml; do rm -f $(basename $f); ln -s $f; done
$ cp config.yaml config.yaml.bak
$ sed -i 's,  - $tempdir/$user/spack-stage,# - $tempdir/$user/spack-stage,g' config.yaml
$ sed -i 's,  - ~/.spack/stage,# - ~/.spack/stage,g' config.yaml
$ sed -i 's,# - $spack/var/spack/stage,  - $spack/var/spack/stage,g' config.yaml
$ cd ../../../
$ lalloc 1
$ ./bin/spack install --no-cache --keep-stage gidiplus@3.18.125+cuda%clang@blueos
$ ./bin/spack install --no-cache --keep-stage umpire@4.1.0+cuda%clang@blueos
$ exit
$ ./bin/spack find -p gidiplus@3.18.125+cuda%clang@blueos
$ ./bin/spack find -p umpire@4.1.0+cuda%clang@blueos

# recompile mercury and link it with the libraries we just compiled
$ ./mbuild --with-nvcc=11 --with-gidiplus=/path/to/gidiplus --with-umpire=/path/to/umpire --make opt
```

Listing 4: Using `ncu` PC sampling **step 3**: invoking `ncu`.

```
$ lrun -n1 --smpiargs="-disable_gpu_hooks" ncu \
  --set detailed --import-source on --kernel-regex-base demangled \
  -k .*PV_Cycle_Tracking.* -c 5 -s 5 -o mypcprofile -f mercury deck.inp
```

`ncu` argument	Explanation
`--set detailed`	turn on PC sampling
`--import-source on`	avoids "'File Mismatch" and "File Not Found" errors in the CSV
`--kernel-regex-base demangled`	match regular expression against demangled symbols
`-k .*PV_Cycle_Tracking.*`	regular expression to use for matching the kernel name(s)
`-c 5`	sample 5 kernel invocations
`-s 5`	skip 5 kernel invocations before sampling
`-o mypcprofile`	name of the profile binary file to output
`-f mercury`	name of the executable to run

Table 35: Explanation of `ncu` arguments that we used.

Listing 5: Using ncu PC sampling **step 4**: Converting the binary profile to human-readable form.

```
$ ncu-ui mypcprofile.ncu-rep

Click Page: Details (it's a dropdown in the upper left of the gui)
Select Source
Click View: Source and SASS (it's a dropdown also upper left)
Select Source
Repeat for every kernel invocation:
  Click Copy as Image V (the downarrow in the upper right)
  Select Export to CSV
  mypcprofile-0.csv
  Save
```

Listing 6: A few lines from the 68,552 line CSV file that we exported in **step 4**.

```
191,"HOST_DEVICE inline MCGIDI_VectorSizeType binarySearchVector( double a_x, Vector<double> const &a_Xs, bool a_boundIndex = false ) {",,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
192,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
193,"    MCGIDI_VectorSizeType lower = 0, middle, upper = (MCGIDI_VectorSizeType) a_Xs.size( ) - 1;",,32,22,47604,153062,9,,,,,,,,,,,,,0,0,0,0,0,0,0,0,0,0,0,10,0,0,0,22,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,22
194,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
195,    if( a_x < a_Xs[0] ) {,,4609,4571,285624,765310,45,0,Global(4),Load(4),64(4),106754,0,0,0,44974,106757,61783,0,0,0,0,0,0,4436,0,0,0,0,7,0,38,0,0,0,128,0,0,0,0,0,4436,0,0,0,0,7,0,0,0,0,0,128
196,        if( a_boundIndex ) return( 0 );,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
197,        return( -2 );,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
198,    },,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
199,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
200,    if( a_x > a_Xs[upper] ) {,,2642,2613,190416,459186,27,0,Global(4),Load(4),64(4),106757,0,0,0,44974,106757,61783,0,0,0,0,0,0,2452,0,0,0,0,0,0,29,0,0,0,161,0,0,0,0,0,2452,0,0,0,0,0,0,0,0,0,0,161
201,        if( a_boundIndex ) return( upper );,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
202,        return( -1 );,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
203,    },,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
204,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
205,    while( 1 ) {,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
206,        middle = ( lower + upper ) >> 1;,,385,292,633504,1974722,27,,,,,,,,,,,,,0,0,0,0,0,0,0,0,0,1,0,93,0,0,0,291,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,291
207,        if( middle == lower ) break;,,987,811,858024,2295333,49,26661,,,,,,,,,,,,0,0,0,0,0,0,0,0,0,4,0,176,0,0,0,797,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,797
208,        if( a_x < a_Xs[middle] ) {,,11840,11759,585900,1821660,18,,Global(4),Load(4),64(4),714172,0,0,0,356544,737667,381123,0,0,0,0,0,0,11355,0,0,0,0,2,0,81,0,0,0,402,0,0,0,0,0,11355,0,0,0,0,2,0,0,0,0,0,402
209,            upper = middle;},,486,412,585900,1821660,18,,,,,,,,,,,,,0,0,0,0,0,0,0,0,0,0,0,74,0,0,0,412,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,412
210,        else {,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
211,            lower = middle;,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
212,        },,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
213,    },,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
214,    return( lower );,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
215,},,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```

Listing 7: Using `ncu` PC sampling **step 5**: Parse the CSV file to find p.

```
#!/usr/bin/env python3

import numpy as np
import pandas as pd

csv = pd.read_csv('mypcprofile_1.csv')

df = pd.DataFrame(csv[csv['#'].isnull()].Source.rename('fname'))

colname2mnemonic = {
    'Sampling Data (All)':  'samples',
    # 'Sampling Data (Not Issued)': 'samples_not_issued',
    # 'Instructions Executed': 'instr_execd'
}

perfile = np.array_split(csv, df.index[1:])

for colname, mnemonic in colname2mnemonic.items():
    quantity = [f[colname].sum() for f in perfile]
    num_lines_with_quantity = [(f[colname].dropna() != 0).sum()
                               for f in perfile]
    df[mnemonic] = quantity
    df[f'num_lines_with_{mnemonic}'] = num_lines_with_quantity

df['percent'] = 100 * (df.samples / df.samples.sum())


dfmd = pd.DataFrame(index='GIDI Non-GIDI Total'.split())
df['gidifile'] = df.fname.str.endswith('hpp') | df.fname.str.endswith('cpp')
wheregidi = df.gidifile

num_gidi_files = wheregidi.sum()
total_files = df.shape[0]
dfmd['Files'] = [num_gidi_files, total_files - num_gidi_files, total_files]

file_edge_index = np.array(df.index.tolist() + [csv.shape[0]])
flengths = (file_edge_index - np.roll(file_edge_index, 1))[1:]
df['flength'] = flengths
num_gidi_lines = df[wheregidi].flength.sum()
total_lines = df.flength.sum()
dfmd['Lines'] = [num_gidi_lines, total_lines - num_gidi_lines, total_lines]

num_gidi_lines_sampled = df[wheregidi].num_lines_with_samples.sum()
total_lines_sampled = df.num_lines_with_samples.sum()
dfmd['Lines sampled'] = [num_gidi_lines_sampled,
                         total_lines_sampled - num_gidi_lines_sampled,
                         total_lines_sampled]

num_gidi_samples = df[wheregidi].samples.sum()
total_samples = df.samples.sum()
dfmd['Samples'] = np.array([num_gidi_samples,
                            total_samples - num_gidi_samples,
                            total_samples]).astype(int)

pct_gidi_samples = df[wheregidi].percent.sum()
dfmd['Percent'] = np.array([pct_gidi_samples,
                            100 - pct_gidi_samples,
                            100]).round(2)

ingidi = dfmd.Percent['GIDI']
print(dfmd)
print(f'\nTracking kernel spent {ingidi}% of runtime in GIDI')

# Uncomment to show gidi files sorted by percent of samples in file
# print(df[wheregidi].sort_values('percent'))
# print(df.sort_values('percent'))
```

Listing 8: Output of Python script in Listing 7.

```
           Files  Lines  Lines sampled  Samples  Percent
GIDI          15  13460            311   111280    25.08
Non-GIDI      70  55092           1380   332495    74.92
Total         85  68552           1691   443775   100.00

Tracking kernel spent 25.08% of runtime in GIDI
```

Listing 9: Using `ncu` PC sampling **step A**: copying `sections` and `SourcePage.py` into place.

```
ml nsight-compute/2020.3.0
rsync -a $(dirname $(ap $(which ncu)))/sections .
pushd sections; ln -s ../SourcePage.py; popd
```

Listing 10: `SourcePage.py`, which tells `ncu` to emit the CSV (thus avoiding the `ncu-ui` GUI).

```
import NvRules
import sys

def get_identifier():
    return "SourcePage"

def get_name():
    return "Source Page"

def get_description():
    return "Source metrics"

def get_section_identifier():
    return "SourceCounters"

def read_file(name):
    lines = []
    with open(name, "r") as f:
        for line in f:
            lines.append(line.rstrip())

    return lines


files_cache = {}

def metric_per_line(action, metric_name):
    global files_cache

    metric = action.metric_by_name(metric_name)

    num_instances = metric.num_instances()
    pcs = metric.correlation_ids()

    values_per_line = {}

    for i in range(num_instances):
        pc = pcs.as_uint64(i)
        source_info = action.source_info(pc)
        if source_info != None:
            file_name = source_info.file_name()
            line = source_info.line()
            if not file_name in files_cache:
                file_content = read_file(file_name)
                files_cache[file_name] = file_content

            value = metric.as_uint64(i)
            if not file_name in values_per_line:
                values_per_line[file_name] = {}
            if not line in values_per_line[file_name]:
                values_per_line[file_name][line] = 0

            values_per_line[file_name][line] += value

    return values_per_line

counter = 0

def print_metric(metric):
    if not metric:
        return

    global counter
    fname = 'sampling_source_%d.csv' % counter
    counter += 1

    with open(fname, 'w') as outfile:
        for file_name in files_cache:
            num_lines = len(files_cache[file_name])
            source = files_cache[file_name]

            print("file: ", file_name, file=outfile)
            for line in range(num_lines):
                value_line = line + 1
                if not value_line in metric[file_name]:
                    print("{:>4d}: {:>16d}: {:s}".format(value_line, 0, source[line]), file=outfile)
                else:
                    value = metric[file_name][value_line]
                    print("{:>4d}: {:>16d}: {:s}".format(value_line, value, source[line]), file=outfile)


def apply(handle):
    ctx = NvRules.get_context(handle)
    action = ctx.range_by_idx(0).action_by_idx(0)

    samples_metric = action.metric_by_name("group:smsp__pcsamp_warp_stall_reasons").as_string()
    sample_metric_names = samples_metric.split(',')

    agg_samples = {}
    for metric_name in sample_metric_names:
        if metric_name:
            metric_values = metric_per_line(action, metric_name)
            for file_name in metric_values:
                if not file_name in agg_samples:
                    agg_samples[file_name] = {}

                for line in metric_values[file_name]:
                    value = metric_values[file_name][line]
                    if not line in agg_samples[file_name]:
                        agg_samples[file_name][line] = 0

                    agg_samples[file_name][line] += value

    print_metric(agg_samples)
```

Listing 11: Using `ncu` PC sampling **step B**: invoking `ncu`.

```
$ lrun -n1 --smpiargs="-disable_gpu_hooks" ncu \
  --section-folder ./sections --section SourceCounters \
  --kernel-regex-base demangled -k .*PV_Cycle_Tracking.* \
  -s 5 -c 1 -f mercury deck.inp
```

Listing 12: A few lines from the new 65,742 line CSV file that we output in **step B**.

```
192:                0: HOST_DEVICE inline MCGIDI_VectorSizeType binarySearchVector( double a_x, Vector<double> const &
    ↪ a_Xs, bool a_boundIndex = false ) {
193:                0:
194:                0:     MCGIDI_VectorSizeType lower = 0, middle, upper = (MCGIDI_VectorSizeType) a_Xs.size( ) - 1;
195:                0:
196:            22241:     if( a_x < a_Xs[0] ) {
197:                0:         if( a_boundIndex ) return( 0 );
198:                0:         return( -2 );
199:                0:     }
200:                0:
201:            14985:     if( a_x > a_Xs[upper] ) {
202:                0:         if( a_boundIndex ) return( upper );
203:                0:         return( -1 );
204:                0:     }
205:                0:
206:                0:     while( 1 ) {
207:             1493:         middle = ( lower + upper ) >> 1;
208:             3498:         if( middle == lower ) break;
209:            57001:         if( a_x < a_Xs[middle] ) {
210:             1472:             upper = middle; }
211:                0:         else {
212:                0:             lower = middle;
213:                0:         }
214:                0:     }
215:                0:     return( lower );
216:                0: }
```

Listing 13: Using `ncu` PC sampling **step C**: Parse the new CSV file to find p.

```
#!/usr/bin/env python3

import numpy as np
import os
import pandas as pd


def parse_ncu_profile(fname):

    csv = pd.read_csv(fname, sep=':', usecols=[0, 1], header=None)
    csv.columns = 'line samples'.split()

    df = pd.DataFrame(csv[csv['line'] == 'file'].samples.rename('fnamelong'))
    df['fname'] = df.fnamelong.apply(lambda x: os.path.basename(x))

    perfile = np.array_split(csv, df.index[1:])

    samples_in_file = [f.samples[1:].astype(int).sum() for f in perfile]
    num_lines_with_samples = [(f.samples[1:].astype(int) != 0).sum()
                              for f in perfile]
    df['samples'] = samples_in_file
    df['num_lines_with_samples'] = num_lines_with_samples

    df['percent'] = 100 * (df.samples / df.samples.sum())

    file_edge_index = np.array(df.index.tolist() + [csv.shape[0]])
    flengths = (file_edge_index - np.roll(file_edge_index, 1))[1:]
    df['flength'] = flengths

    dummy2substr = {
        'mercury': 'build_mercury',
        'gidi': 'gidiplus',
        'cuda': 'cuda-11.2.0-beta',
        'rng': 'rng',
    }

    index = dummy2substr.keys()
    columns = ['files', 'lines', 'lines sampled', 'samples', 'percent']

    dfmd = pd.DataFrame(columns=columns, index=index)
    for dummy, substr in dummy2substr.items():
        df[dummy] = df.fnamelong.str.contains(substr)

        files = df[dummy].sum()
        lines = df[df[dummy]].flength.sum()
        lines_sampled = df[df[dummy]].num_lines_with_samples.sum()
        samples = df[df[dummy]].samples.sum().astype(int)
        percent = df[df[dummy]].percent.sum().round(2)

        dfmd.loc[dummy] = [files, lines, lines_sampled, samples, percent]

    dfmd.index = dfmd.index.str.upper()
    dfmd.loc['Total'] = dfmd.sum()
    dfmd.columns = dfmd.columns.str.capitalize()

    return df, dfmd


if __name__ == '__main__':
    df, dfmd = parse_ncu_profile('sampling_source_0.csv')

    print(dfmd)
    ingidi = dfmd.loc['GIDI'].Percent
    print(f'\nTracking kernel spent {ingidi}% of runtime in GIDI.')

    # Uncomment to show gidi files sorted by percent of samples in file
    # print(df[df.gidi].sort_values('percent'))

    # Uncomment to show all files sorted by percent of samples in file
    # print(df.sort_values('percent'))
```

Listing 14: Output of Python script in Listing 13.

```
         Files  Lines Lines sampled Samples Percent
MERCURY     61  48374           1394  317526   71.71
GIDI        14  13076            325  110220   24.89
CUDA         3   4131              4    8153    1.84
RNG          1    366             20    6911    1.56
Total       79  65947           1743  442810     100

Tracking kernel spent 24.89% of runtime in GIDI.
```