High energy-density physics experiments are dominated by energy transfer between radiation and material

- Photons of different frequencies, $E_g$ diffuse through material at different rates, $\nabla \cdot D_g \nabla E_g$
- Each group of photons deposits energy in the material, $\sigma_g E_g$
- Material radiates photons into each group, $\sigma_g B_g(T(u))$

$$\frac{\partial E_g}{\partial t} - \nabla \cdot D_g \nabla E_g = \sigma_g [B_g(T) - E_g] + S_g$$

$$\frac{\partial u}{\partial t} = \sum_g \sigma_g [E_g - B_g(T)] + Q,$$

Radiation flows through a pipe, heating material
We linearize the coupling and discretize using a finite element method

- We have one implicit diffusion matrix solve for each group of photons
  - Can have one or hundreds of groups
- Once radiation is known, we can update the material
- We use unstructured meshes as a form of adaptivity, greatly reducing our zone counts
  - Code and discretization more complicated than regular mesh, but it has been worth it in the past

Material interfaces and gradients are resolved with unstructured mesh
This discretization is highly parallelizable

- Each radiation group has its own diffusion matrix (one box) that is solved separately.
- Each group coupled to material locally, making it convenient to construct each group at the same time.
- Each global matrix is a sum of sub-matrices which are computed as integrals over each mesh zone:

$$ A_g = \sum_{\text{zones}} \int_{z}^{\sigma_g w_i w_j} \text{d}V $$

- Each zone integral can be done in parallel.
We want to explore new technologies without disrupting production codes

- Production codes are very large and cumbersome
  - Code base in (heavily templated) C++, C, Python, and FORTRAN
  - 37 third party libraries
- Current methods scale well up to at least 32,000 processors with pure MPI

Goals for the new code: Mulard

- Explore on-node parallelism (threads, GPU, etc.)
- Explore matrix storage strategies for the multigroup system
- Flexible for quick refactoring
- Complications of production codes
Mulard built on top of the open source MFEM finite element discretization library

- MFEM\(^1\) is a self-contained with no dependencies
- Roughly ten different problems can be run that stress different parts of the solver
  - Some have analytic solutions for code verification
  - Vary number of groups, time dependence, and steady state
  - Can run arbitrary meshes in 2D and 3D
  - Meshes can be refined to increase zone count
- Mulard includes many production-like features, preventing “obvious” optimizations
  - Materials and problem specification behind opaque interface
  - Objects hide details of linear solvers, finite elements, and mesh
  - Run-time selection of finite element orders
  - Write visualization files to inspect the problem

\(^1\)http://code.google.com/p/mfem/
Mulard is written to be understandable, not fast
(But it’s not slow, either)

- Wanted to start with clean, well documented code
- It’s organized the way efficient scalar processor code has been in the past
- Lots of comments explaining:
  - the math, so it can be refactored
  - (unenforced) production constraints
  - variables that could be thread-private
  - Doxygen function descriptions

<table>
<thead>
<tr>
<th>Code portion</th>
<th>Files</th>
<th>Lines</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libraries</td>
<td>25,841</td>
<td>4,456,843</td>
<td>uncounted</td>
</tr>
<tr>
<td>Production</td>
<td>4,062</td>
<td>545,676</td>
<td>uncounted</td>
</tr>
<tr>
<td>MFEM</td>
<td>99</td>
<td>38,509</td>
<td>4,417</td>
</tr>
<tr>
<td>Mulard</td>
<td>15</td>
<td>3,208</td>
<td>1,181</td>
</tr>
</tbody>
</table>

Computed using http://cloc.sourceforge.net
Real work increases with mesh size

- As mesh size increases
  - Number of time steps also increases
  - Percentage of floating point instructions goes up
  - Saturate memory bandwidth (IPC goes down)
- In each cycle, setup time shouldn’t be higher than solve time
  - But there are some known optimizations in production code
As groups increase, everything grows roughly linearly.

- Instructions per cycle goes down as saturate bandwidth.
- Floating point instruction fraction is roughly constant.
We have many plans for future exploration

- Solve groups in parallel
- Compute one matrix graph, pre-allocate optimal sparse storage
- Always store material tables, not some analytic formulas, for more memory, but uniform access
- Some discretization changes are also possible, and be more efficient and just as good
Mulard is ready to start exploring on-node parallelization

- Mulard implements a relatively full-featured multigroup radiation diffusion code
- Code base is small and flexible enough to easily refactor
- Already exploring discretization issues with Mulard
- Future work:
  - Add automatic test suite
  - Compare performance metrics to production codes
  - Change to use table look-ups for material properties
  - Explore new programming models (OpenMP, Cuda, etc.)

Mulard should be released soon

- We welcome others using it and want your feedback
- Contact Tom Brunner brunner6@llnl.gov
Thanks to all the people that have helped me

- Brian Miller for giving this talk in my place!
- Tzanio Kolev on the MFEM team
- Jeff Keasler for helping me collect some performance data

The origin of “Mulard”

- The letters come from: MULtigroup RADiation Diffusion
- A mulard is a hybrid domesticated duck
  - They are also very tasty