SAND2016-5006C

Computational Approaches to DSMC on Next-Generation Processors

Matthew T. Bettencourt[1] Chris H. Moore

[1]Electromagnetic Theory,

Sandia National Laboratories,

Albuquerque, New Mexico, USA

DSMC 2015 Conference

Kapaa Hawaii

September 15th, 2015

Background

- US DoE Advanced Scientific Computing (ASC) is refreshing their major computing resources
  - FY16 Trinity – Self booted Xeon Phi cluster
  - FY18 Sierra – IBM Power with Nvidia chips -150 PF
  - FY21 ATS-3 – In bidding process
- Current software stack cannot take advantage of these platforms
- New program element to develop new applications
  - Advanced Technology Demonstration and Mitigation
- Sandia National Laboratories is developing a new code for plasma simulation EMPIRE
  - ElectroMagnetic Plasma In Radiation Environments

EMPIRE

- EMPIRE is a new open source application being developed for next generation platforms
- Physics and simulation goals –
  - Electromagnetic and electrostatic
  - Kinetic and fluid based plasma descriptions
  - Radiation transport and gas chemistry - DSMC
  - Beyond forward simulation – Sensitivities, UQ, optimization, …
- Computation capabilities –
  - Hybrid structured/unstructured mesh
  - Hybrid PIC-fluid description
  - In-situ meshing, mesh refinement
  - In-situ analysis, visualization and adjoint methods
  - Asynchronous Multi-Tasking (AMT), dynamic balancing

Mini-DSMC

- Goal – Develop a reduced physics application to act as a testbed for development of next generation DSMC models
  - Going beyond a single thread per element scaling
  - Efficient use of the hardware architecture
  - Repeatable results
  - Representative of the real application
  - Portable across architectures
- Solution
  - Develop threaded version using the Kokkos libraries
  - Test only the collision part of the code using
    - No time counter
    - Ionization, recombination and elastic scatter
    - Analytic and tabular cross sections

Kokkos: C++ Library / Programming Model for Manycore Performance Portability

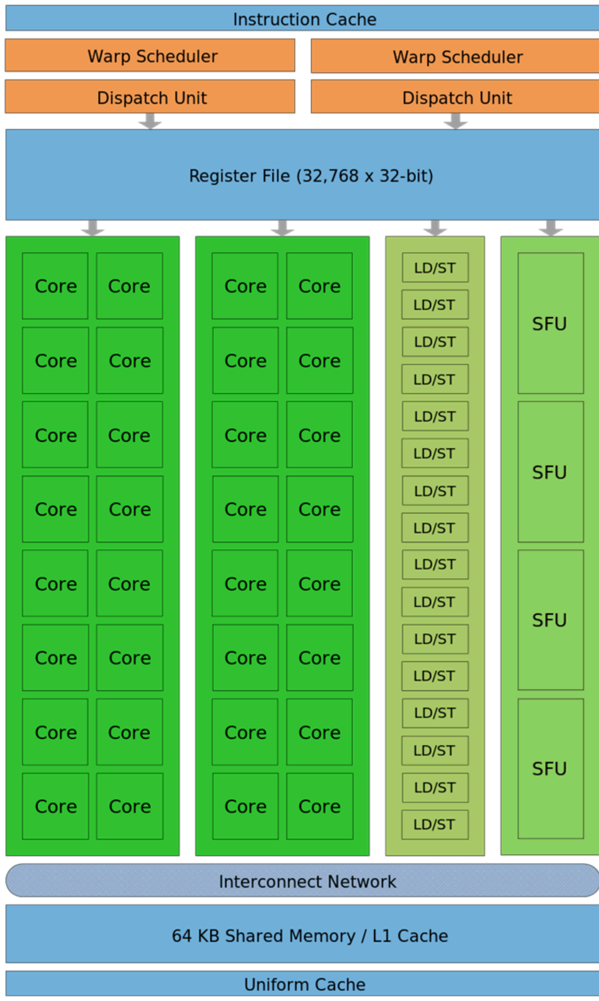
- Portable to Advanced Manycore Architectures
  - Multicore CPU, NVidia GPU, Intel Xeon Phi
    - Backends – Cuda, OpenMP, pthreads and serial
  - Maximize amount of user (application/library) code that can be compiled without modification and run on these architectures
  - Minimize amount of architecture-specific knowledge that a user is required to have
  - Allow architecture-specific tuning to easily co-exist
  - Requires a C++11 compiler
- Performant
  - Portable user code performs as well as architecture-specific code
  - Thread scalable – not just thread safety (no locking!)
- Usable
  - Small, straight-forward application programmer interface (API)
  - Constraint: don't compromise portability and performance

Kokkos: Collection of Libraries

- Core – lowest level portability layer
  - Portable data-parallel dispatch: parallel_for, parallel_reduce, parallel_scan
  - Multidimensional arrays with device-polymorphic layout for transparent and device-optimal memory access patterns
  - Access to hardware atomic operations
- Containers – built on core arrays
  - UnorderedMap – fast find and thread scalable insertion
  - Vector – subset of std::vector functionality to ease porting
  - Compress Row Storage (CRS) graph
- Linear Algebra
  - Sparse matrices and linear algebra operations
  - Wrappers to vendors' libraries
  - Portability layer for Trilinos manycore solvers
  - Trilinos/Tpetra built on top of Kokkos for MPI+X linear algebra

A Bit About Modern Architectures

- Available FLOPS are going up, memory bandwidth is staying low
  - Both data and instructions take up memory bandwidth, hardware solution SIMD/SIMT
    - Every thread does the same execution or pauses
- NVIDIA systems use Streaming Multiprocessors (SM) and your code is broken down on “warps”
  - Minimum 32 threads performing the same instructions
    - If statements can cause threads to stall
  - Limited SFUs (sin, erf, …)
  - Doubles take 2 threads
- Intel vector instructions similar



"Fermi" by NVIDIA

So, What Does This Mean?

- Take the following code snippet for a stacked cross section

```
for (ith_collision = 0; ith_collision < num_collisions; ith_collision++) {
  cumulative_sigma += get_sigma(energy);
  if (cumulative_sigma > random_fraction)
    collisions[ith_collision]->handle_collision(particle_1, particle_2,...)
}
```

- 16 threads enter this loop with 16 pairs of particles
  - Suppose 4 different types of collisions
- A subset pass the conditional
  - Rest of the threads stop
  - Assuming this is the same collision operator all the threads handle the collision, else they go in one by one
- Once all the first set of collisions are handled, the group goes on to the next set in the stack

Algorithmic Detail

- Three steps to perform collisions
  - Sort particles by element, then type, then location
    - Sort uses count sort algorithm, parallel_scan for counting and positioning by element
    - Small thread team does count sort by type
    - Location uses only 1 thread, n^2 sort
  - Estimate collision count
    - Grab list of random pairs, compute local v_sigma
    - parallel_reduce to determine v_sigma_max
    - Could track it inline with collision operator
  - Perform pair-wise collision detection
    - Pick pairs of particles, determine collision
    - If collision to occur, cache results
    - Collide collected particles.

Collision Algorithm Details

- Given a warp of threads (32->256), and some elements (1->256) do the following:
  - Every thread draw a **unique** pair of particles with the same potential collisions ($N^2+e^- \to$…)
    - Use “atomic compare swap” and exclusion lists
  - Compute stacked cross-section
  - Determine collision type (none, elastic, ionization,..) and add too shared memory list
    - indx=atomic_increment(&count_of_collisions[collision_type]);
    - Collision[collision_type][indx].add(particle1, particle2);
  - If no collision return particles to selection lists
  - Team thread barrier (not global barrier, trivial operation)
  - If size of collision list >= number of threads
    - Process collisions as a team of threads
    - Return particles to selection lists
  - Return to top

Pros/Cons

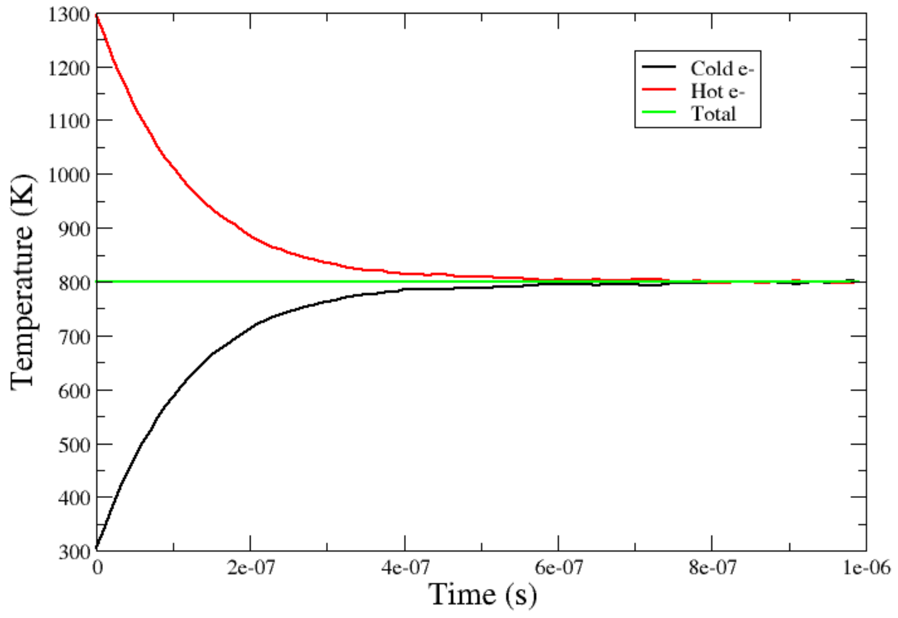
- Pros:
  - Good thread uniformity, SIMT execution
  - Single code base should perform well on multiple platforms
  - If limited to one thread/element, parallel repeatability
- Cons:
  - Added complexity
  - Heavy on register and shared memory usage
    - Lists and code for all collision types need to be in limited shared memory pool
  - Scattering and VHS cross sections uses transcendentals – contention for SMU

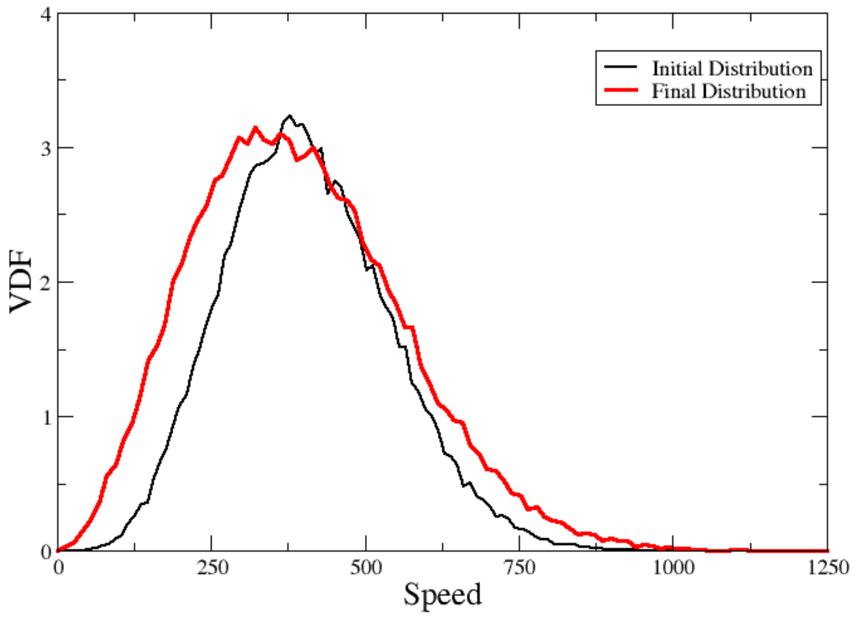
Is It All Worth It?

- Short answer, it depends
  - For cases with small collision groups, and a high acceptance rates – No
    - For a two species problem with Maxwell cross section, elastic scattering and 100% acceptance, simple code is roughly 17% faster on GPU, 21% faster on CPU
  - For cases with large collisions groups – Yes
    - For a two species problem with three different cross section models and 6 different collision operators, simple code is <u>4.2x slower on GPU</u>, 25% slower on GPU
  - So, there is a small penalty for running the more complicated algorithm on simple problems
    - But a large benefit for running the complicated algorithm on complicated collision stacks

Results

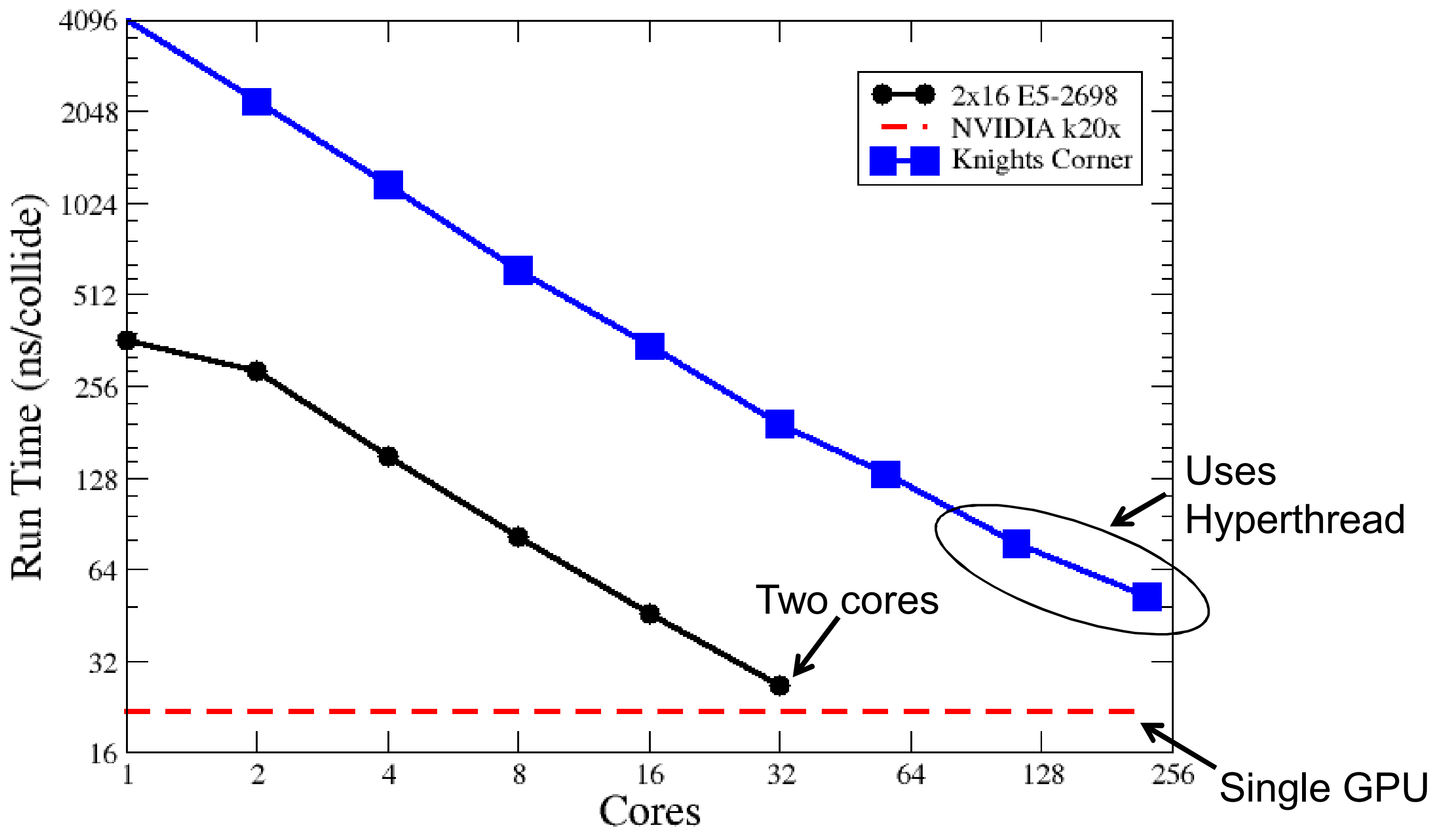
- Two simple test problems.
  - Thermal equilibrium - left
  - Relaxation of bimodal distribution - right





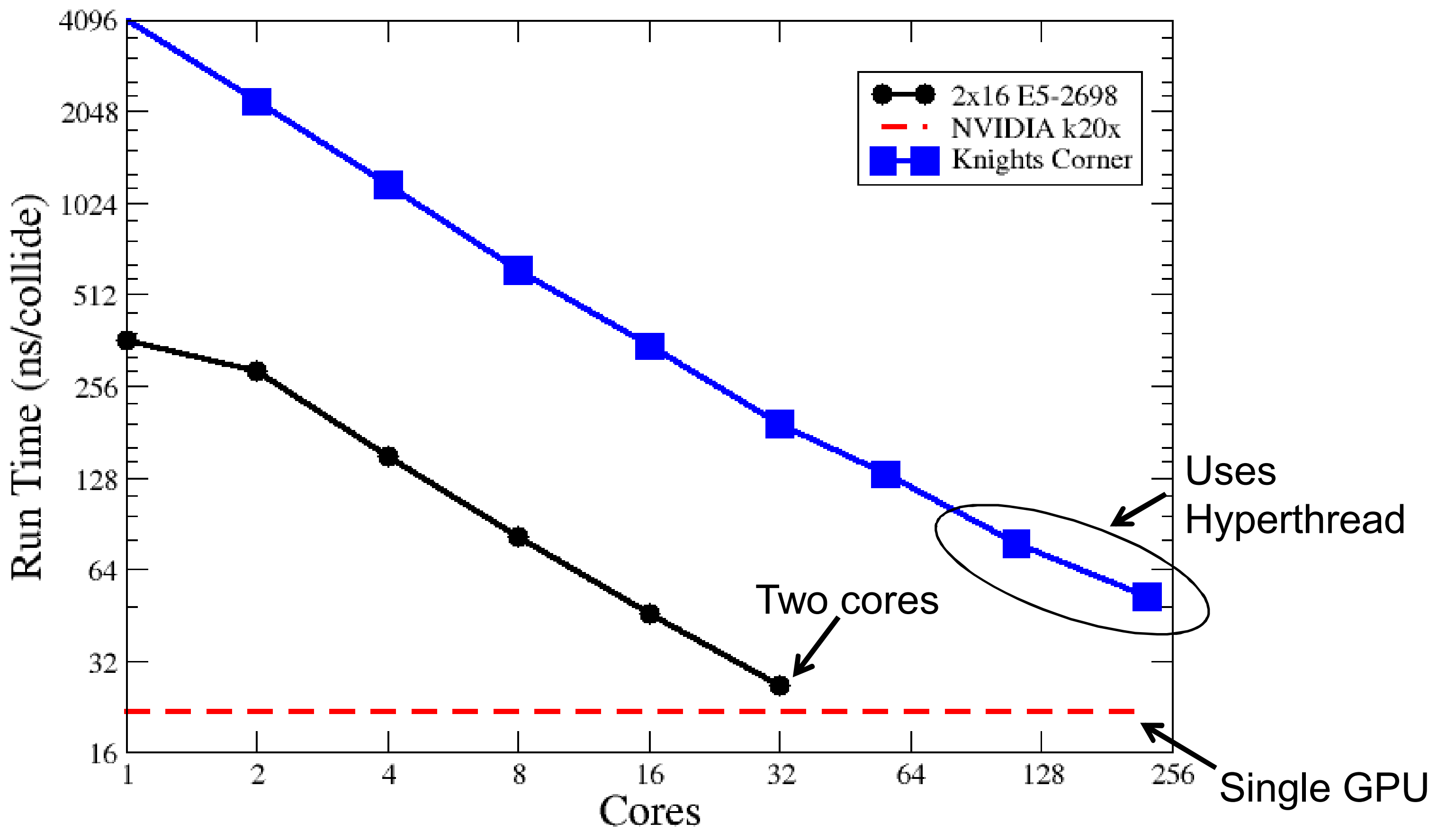
Scaling Results

Simple elastic collisions using VHS



Scaling Results

Three species (Ar, Ar+, e-) reaction and elastic



Summary

- An algorithmic approach to DSMC on GPUs has been presented
  - Collisions need to be grouped and handled as a team
  - Scales beyond one thread per collision cell
    - Repeatable results require only one thread per cell
  - Shows up to 4x performance improvement over naive approach with large collision group
- Incorporation into EMPIRE code base to follow