LA-UR-20-24473
Approved for public release; distribution is unlimited.

Title:	Jayenne GPU Strategy Update
Author(s):	Long, Alex Roberts
Intended for:	Share with outside collaborators and computer vendors
Issued:	2020-06-22

Jayenne GPU Strategy Update

ASC GPU Meeting

Alex Long

Jayenne Team: Kelly Thompson, Matt Cleveland, Ryan Wollaeger, Kendra Long, Ben Ryan, Tim Kelley, Alex Long

Special thanks to Tim Burke in XCP-3

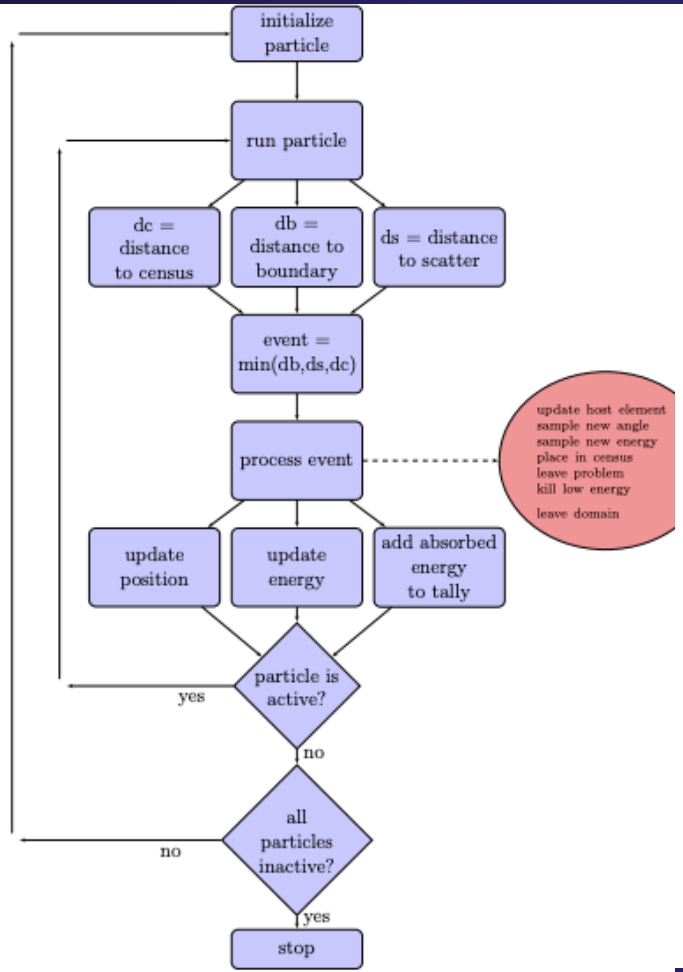
May 19. 2020

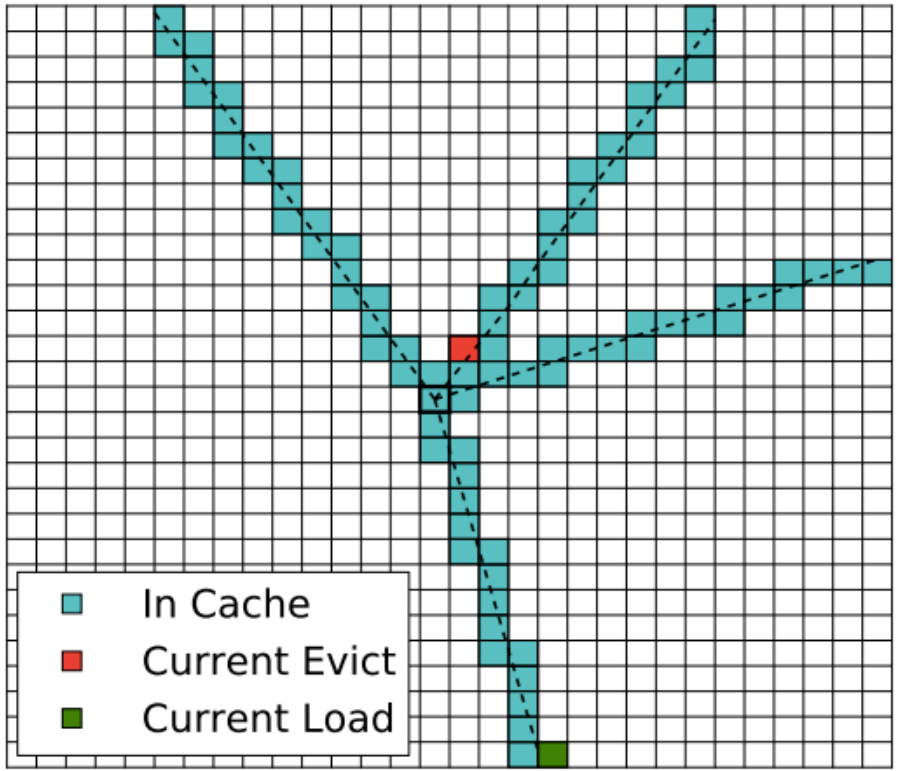
Several design questions drove our work on GPU porting

- Groundwork**: How should the code structure of Jayenne change for accelerator driven Monte Carlo transport?**
- Data: **Do we need to change our data layout?**
- Portability strategy: **Can we reuse the basic transport components between CPU and GPU?**

- Monte Carlo on GPUs, a brief literature review and some discussion with peers at other labs
- Early gains: Definitely do these things!
- Current performance numbers
- Event-based vs. history-based and how to optimize further

Brief overview of IMC transport—flowchart and memory indiriction





Large code design changes were made at the transport level

- My personal philosophy—use convenient code features and libraries for driver and data initialization, use more thread-safe and less state-owning design in critical parts of code
- Contractor model -> functional model
- Contractor model
  - All contractors owned mutable and non-mutable state (e.g. a shared pointer to tally, opacity)
  - Contractors also stored convenience values like the most recent calculated distance to event, to be used later in event processing
- Functional model
  - Use flat functions instead of classes with methods
  - pass particle and (const) mesh data, return distance, determine event, pass particle and event distance, return particle fate

Code structure changes were also driven by need for shared functionality with CPT

- Code called during transport was moved to its own directory
  - "imc_solver" contains "distance_to_collision", "apply_collision_event", "random_walk_elligible", etc.
  - "imc_solver" contains 7k lines of code, something we can reason about!
  - "low_mc" contains "get_distance_to_boundary", "next_cell", functions required for mesh based tracking (shared with CPT)
  - "low_mc" contains 10k lines of code, woohoo for sharing!
- Splitting directories has already served its purpose, team asks "will this code change effect GPU transport" when working in this directory
- On the topic of sharing, I'm very interested in using shacl surfaces—4x4 matrix to describe all surfaces used in Jayenne. Already used in GPU code!

Additions of “Worker model” and changes to Particle class made GPU port simpler

- An MPI rank can now asynchronously manage “workers” while still doing particle work
- Right now, workers are threads under MPI rank
- Worker model breaks the “one transporter, one worker” paradigm—one transporter can manage multiple workers of different types (e.g. CPU, GPU)
- Ready to take full advantage of mixed node architectures
- Particle now owns all RNG state (4 uint64_t), eliminating need to try to spawn and or save RNG state of threads (a reproducibility nightmare)
  - Two downsides: particle state is now tied to RNG state, which is conceptually strange, particle size bumped from two cache lines to three ☹

Moving from several SOAs to a single AOS for transport data simplifies movement

- Mesh data owns array of vertices, array of neighbors, Opacity class owns vector of vectors, mat state owns arrays
- Original motivations
  - We don’t access these data fields in a stride-one way during transport
  - Moving mesh data needed for transport requires many gathers instead of a single large send
  - It’s easy to copy transport data to device
  - CPU Prefetching is greatly simplified
- Move all of this data into a single contiguous block on a rank, store offsets for a given cell in an object
- AOS Complications: we want stride-one access across GPU threads and in CPU event-based vector lanes, harder to debug, unstructured complications
- Also, currently not looking at “mesh pulling” as parallel solution

GPU strategy is CUDA for transport

- Now transport code is split off from sourcing and cycle initialization
- What does it take to make a CUDA transporter?
  - NVCC flag “experimental-const-expr” absolutely indispensable, allows me to mark Draco (shared code between TRT packages) functions constexpr without having to introduce GPU specific decorators
  - Add CMake defined macros for “__device__” and “__host__” decorators to all functions and member functions
  - “std::vector” -> “std::array”, Tim Burke did the legwork of making a “constexpr” variant of std::array. Accessors are not marked as constexpr in C++ 14, C++17 fixes that issue, introduces a host of others
  - Now we can combine device functions into a single CUDA kernel

Building woes

- At first, I tried to build two libraries—CUDA and C++ compiled imc_solver and then link the appropriate library if CUDA was enabled
  - This introduced an awkward configure step of making two libraries from the same source code
  - Also, the other libraries in Jayenne mostly didn't care since 80% of the "imc_solver" code was defined in headers
- Solution: just build CUDA library, put CUDA specific code in "*.cc" files, everyone else uses functions in "*.hh"
- CUDA libraries built with "separable compilation" (multiple compilation units) link differently than C++ libraries
  - CUDA runtime libraries are linked when executables are made
  - This created all kinds of strange linking issues, turning off separable compilation fixed all of them
- Using GCC 7+ on Darwin/RZ Ansel, waiting for NVCC 11 for XL

History-based and event-based GPU transport in the current literature

- Hamilton and SHIFT
  - *Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms*
    - Truncated history reduces runtime when physical properties lead to vastly different particle histories
    - Atomic add to count event types remove need for index sort
    - For multigroup neutronics, event-based transport is much slower than history-based
    - Coalesced memory access of particle data appears to give history-based an edge over event-based
- Bleile and Quicksilver
  - *Thin Threads: An Approach for History-Based Monte Carlo on GPUs*
    - Batch sizes of 100,000 or greater work best in GPU transport
  - *Investigation of Portable Event Based Monte Carlo Transport Using the NVIDIA Thrust Library*
    - Use prefix sum to count events without a sort,

History-based and event-based GPU transport in the current literature

- Brantley and Mercury
  - IMP (LLNL IMC code) shows 3-4x speedup over CTS nodes for crooked pipe problem
- Sweezey and MONTERAY
  - *A Monte Carlo volumetric-ray-casting estimator for global fluence tallies on GPUs*
    - Computing tallies via volumetric-ray-casting is much more effective on the GPU and can be done asynchronously during CPU transport phase

Transport on GPUs, initial port 15-20x slower, currently 3-4x faster (comparing to CTS-1 nodes)

- First cut of the GPU post was very slow, as expected
- First speedup from marking all functions called by transporter “inline”
  - Later, when I turned off separable compilation, this was required anyway
- Second major speedup—use shared memory for particles and global tallies
  - In the “big kernel” approach, particle and global tally data is reused enough to hide cost of copy to shared memory
  - Copying other fields to shared memory has not provided speedup—physics data is loaded each time a particle moves into a new cell
- Third and most dramatic speedup: Explicitly copying host to device, don’t rely on UVM (Note: I wasn’t using cudaMallocManaged before)

Additional improvements from CUDA features and the literature

- Use CUDA constant memory for the random walk data tables (about 2k) improved random walk by 2-2.5x
- Remove all global tally operations, these were largely diagnostic and can be gleaned from individual cell tallies
- Use Hamilton's "truncated history" to limit thread divergence (20% improvement in multigroup, thin-thick problems

Nuts and Bolts–The History Based method

- Follow a single particle until it dies or reaches census
- LLNL calls this the “big kernel” approach—all transport functions are reachable from the kernel launch
- This includes a “while” loop so some threads will finish earlier than others and be waiting at __syncthreads()
- Can it work? LLNL has shown some speedup with this approach (3-4x for IMC)

Nuts and Bolts–The History Based method

- The history based method roughly looks like this:

```
while (particle.alive())
  d_bound = get_distance_to_boundary(particle, cell_data)
  d_collision = get_distance_to_collision(particle, cell_data)
  d_census = particle.get_time_remaining()

  d_event = min(d_bound, d_collision, d_census)
  event_type = min_element([d_bound, d_collision, d_census])

  # this is a large routine in RZ geometry
  move_particle(particle, cell_data)

  # could update cell data!
  apply_event(event_type, particle, cell_data, tally_data)
```

Nuts and Bolts–The History Based method

- With UVM (bad) results for weak scaling a streaming problem to a full node of Darwin and RZ Ansel
  - Weak scaling in IMC scales the particle count only
  - 50^2 cells for 2D, 50^3 cells for 3D
  - 36 cores of CTS-1 and 4 GPUs both running 12 million particles

Problem	Runtime (s) Darwin –Power 9			Runtime (s) RZ Ansel			Runtime (s) CTS-1
	1 GPU	2 GPUs	4 GPUs	1 GPU	2 GPUs	4 GPUs	36 cores
XY (2D)	28.2	35.3	34.9	28.59	34.75	35.42	8.51
XYZ (3D)	35.3	49.4	49.2	32.53	45.39	45.97	37.33
RZ (2D)	33.1	37.3	37.6	32.87	35.30	36.04	9.93

Nuts and Bolts–History Based with “Truncated History”

- Limit the number of events a particle can undergo in the history-based method before the particle is “re-queued”, then launch with transport kernel again with a smaller number of particles
- Threads within a warp are less likely to be waiting at a __syncthreads() because one particle required 1000 events to finish while everything else was done at 250 events
- Hamilton shows a ~2x speedup for a small 2D reactor core problem (C5G7 benchmark) with this method
- For IMC we would expect a similar situation with any kind of temperature or material variations in a problem
- I implemented “truncated history” with Thrust to sort inactive particles from transport queue
  - Doing this work showed me a path towards event-based transport

Nuts and Bolts–The History Based method

- With UVM (bad) results for a highly scattering problem
  - 50^2 cells for 2D, 50^3 cells for 3D
  - Scattering opacity is 100 cm^{-1}

Darwin -- Power 9	
Problem	1 GPU
XY (2D)	192.9
XYZ (3D)	294.6
RZ (2D)	256.1

CTS-1	
Problem	36 cores
XY (2D)	30.28
XYZ (3D)	45.52
RZ (2D)	33.22

Nuts and Bolts–History Based with “Truncated History”

- This is what the algorithm looks like

```
# indices of particles that are active, all active at start
active_indices = {1, 2, ..., n_particles-1, n_particles}

while(active_indices)
  # launch  "big kernel" which now marks indices inactive (-1)  when complete
  transport<<<n_blocks, n_threads_per_block>>>(particles, active_indices)

  # sort and remove inactive indices
  remove_if(active_indices,-1)
```

Nuts and Bolts–History Based with "Truncated History"

- With UVM (bad) results for the Su Olson "picket fence" problem
  - Two opacities, about 0.01 cm^{-1} and 100.0 cm^{-1}, 64 groups, alternating thick and thin opacities
  - Allow 100 events before a re-sort and re-launch
  - One kernel launch becomes about twenty kernel launches with fewer and fewer particles

Darwin -- Power 9	
Method	1 GPU Runtime (s)
Standard	205.2
Truncated History	163.9

Nuts and Bolts–Event-based transport

- Instead of following a single particle until it finishes transport in a “while” loop, determine the next event for all particles, group particles by event and dispatch events together
- Fundamentally, transpose this:

```
for (particle in all_particles):
    while(particle.alive())
```

To this:

```
while(!all_particles.empty())
    for(particle in all_particles)
        determine_event(particle)
    for(particle in event_buffer)
        dispatch_event(particle)
```

Event-based transport has better occupancy, better branch efficiency, worse performance

- I used thrust to do event selection, event processing and queue sorting
  - Brief aside: if you can express your loop with a simple lambda, does it matter what you select for a portability solution? KOKKOS “for_each” could work here just as well
- Surprisingly easy to implement a first-cut—just reusing the same components of the transporter in a different order
- Why is event-based slower than history based?
  - Hamilton guessed continuous energy had more work than multigroup such that “processing event” was sped up more in event-based
  - Data reordering is likely required
  - I need to rerun this with explicit memory management

6/2/2020 Update: Using explicit memory management gives about 10x improvement

- UVM was a convenient way to port the code but it was mostly a temporary solution until I had time to try some other memory management
- Memory paging with UVM seems like it should hide some memory motion
- Copy 4 fields to the GPU: geometric/connectivity mesh data, physics mesh data, particles and tally data
- This added about 200 lines of code, still easy to port if a language requires a different way to manage memory needed by accelerator
- **Seeing about 4x improvement over CTS nodes for transport dominated problems** (consistent with LLNL and Sweezy's table)
- Haven't started optimizing the kernel yet…
- MPI and problem setup look like the next bottlenecks!

6/2/2020 Update: Improved performance on scattering problem

- Results for a highly scattering problem
  - 50^2 cells for 2D, 50^3 cells for 3D
  - Scattering opacity is 100 cm^{-1}
- About 20% transport time

This is probably the case with the most data reuse and thus the best speedup over CTS-1

Darwin GPU w/ UVM	
Problem	1 GPU
XY (2D)	192.9
XYZ (3D)	294.6
RZ (2D)	256.1

Darwin, explicit memory	
Problem	1 GPU
XY (2D)	5.64
XYZ (3D)	8.6
RZ (2D)	5.35

CTS-1	
Problem	36 cores
XY (2D)	30.28
XYZ (3D)	45.52
RZ (2D)	33.22

~6x speedup over CTS

6/2/2020 Update: Improved performance on streaming problem

- Results for a highly scattering problem
  - 50^2 cells for 2D, 50^3 cells for 3D
  - Scattering opacity is 1 cm^{-1}
- About 12% transport time

No data reuse—streaming dominated means each thread always loads a new cell

Darwin GPU w/ UVM	
Problem	1 GPU
XY (2D)	28.2
XYZ (3D)	35.3
RZ (2D)	33.1

Darwin, explicit memory	
Problem	1 GPU
XY (2D)	5.08
XYZ (3D)	8.49
RZ (2D)	4.78

CTS-1	
Problem	36 cores
XY (2D)	8.51
XYZ (3D)	37.31
RZ (2D)	9.93

~2-3x speedup over CTS

6/2/2020 Update: Improved performance on a domain decomposed problem

- Results for the simplified hohlraum problem
  - Run weekly in our performance regression
  - Highly scattering media (particle can have ~100,000 scatters)
  - In the table "DD" means "domain decomposed"

Machine and setup	Parallel Mode	Runtime (s)
CTS-1, 4 ranks, 9 threads/rank	DD	697
CTS-1, 4 ranks, dynamic threads	DD	501
CTS-1, 36 ranks	Replicated	300
Darwin P9, 2 ranks, 2 GPUS	DD	157
Darwin P9, 4 ranks, 4 GPUS	DD	203
Darwin P9, 1 rank, 1 GPU	"Replicated"	71
Darwin P9, 2 ranks, 2 GPUs	Replicated	57
Darwin P9, 4 ranks, 4 GPUs	Replicated	48

With MPI, 4x speedup over CTS-1 run

No MPI, 6x speedup over no MPI CTS-1 run

6/2/2020 Update: Domain decomposed runs give confidence in speedup in more realistic runs

- Hohlraum wrap up:
  - Hohlraum problem runs 100 cycles, our driver acts as simple “host” and all transport data is built up every cycle
  - Problem is not saturated with particle work and is load-imbalanced, representing a pessimistic scenario
- It’d be great to leave particle data on the device—not sure how other packages feel about that

The Jayenne GPU port is running faster than CTS nodes with a lot of optimization work remaining

- Most of last two years spent getting to a place where we could port to CUDA (where PARTISN started from)
  - Starting writing CUDA in the past 9 months
  - Started tinkering with performance in the last 3 months
- Our timers show the transport phase is not a dominant cost in our current tests
  - We need better timers to find out where the new bottlenecks are with the GPU (we want to move to Caliper)
- MPI and load balance issues significantly inhibit performance
  - “straggling” problem
  - In pure replicated (almost no MPI communication) we see 6x speedup over CTS-1 node

Looking Forward: Steps to performance on Sierra?

- After this exercise, we should stop and evaluate
  - Gather more metrics—register use, occupancy, memory bandwidth
  - What are we doing wrong?
  - What can we learn from other codes at LANL? (PARTISN, VPIC)
- For running large simulations we'll need to optimize MPI communications
  - direct MPI writes to GPU particle buffer?
- Consider "load balancing" issues above the GPU transport level
  - Use our own decomposition, dynamically assign workers to ranks, give ranks an overlapping mesh domain to limit particle passing

Looking Forward: How do our plans change for El Capitan?

- Currently, the CUDA used in the history-based method is simple enough that the HIPIFY tool (CUDA ->HIP) should generate code for AMD devices
- As mentioned previously, the Thrust “for_each” + lambda could just as easily be replace with a KOKKOS “parallel_for” + lambda
- A larger warp size will increase thread divergence in the big kernel method, should have little effect on the event-based method
- “leave everything on the GPU” approach means we may have to do sourcing and initialization all on the GPU. This will require porting code that is even more reliant on the std library than the transporter

6/2/2020 Update: Domain decomposed runs give confidence in speedup in more realistic runs

- Hohlraum wrap up:
  - Hohlraum problem runs 100 cycles, our driver acts as simple “host” and all transport data is built up every cycle
  - Problem is not saturated with particle work and is load-imbalanced, representing a pessimistic scenario
- It’d be great to leave particle data on the device—not sure how other packages fee about that