Porting DENOVO to the Titan System

West Coast Titan Users and Developers Workshop
Jan. 29-31 2013

Wayne Joubert

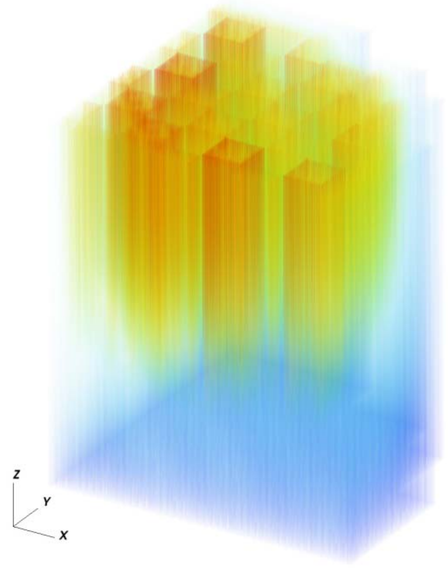
Scientific Computing Group

Oak Ridge Leadership Computing Facility

Oak Ridge National Laboratory

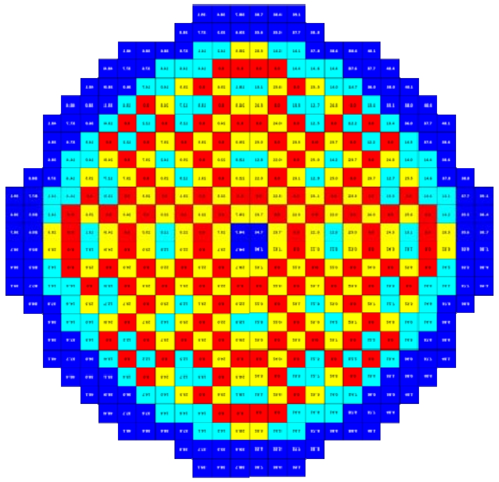
Science Motivation

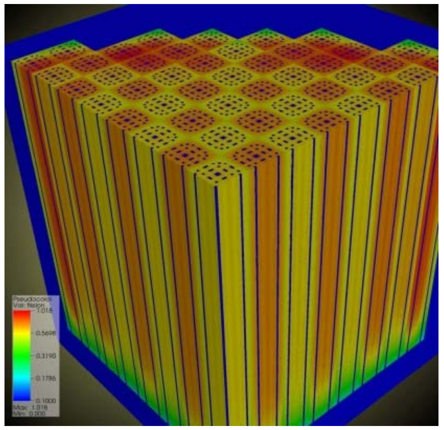
- Denovo solves the radiation transport equation, of importance to nuclear reactor core analysis (neutronics), radiation shielding, nuclear forensics and radiation detection
- Modeling of next-generation nuclear reactors will be a significant technical challenge
- Simulations will require a higher level of geometric fidelity, solution accuracy and physical model realism that is far beyond current computing capabilities
- Realistic models will require a first-principles predictive capability to model nuclear reactor behavior
- However, at present, full-core, pin-resolved transport simulations are beyond the scope of existing computer architectures and will require exascale-class systems and beyond



What is Denovo

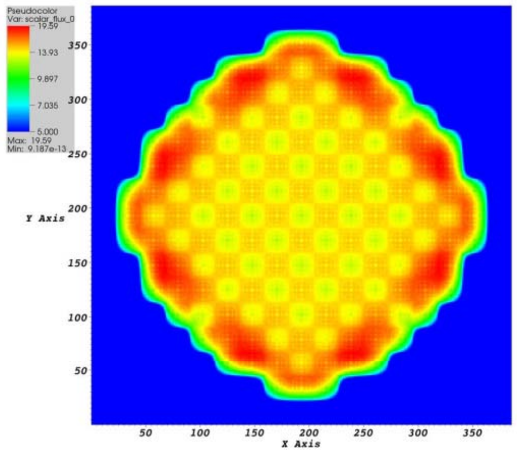
- Denovo is a radiation transport code used in advanced nuclear reactor design
- It solves for the density of particle flux in a 3-D spatial volume such as a reactor
- In particular, it solves the six-dimensional linear Boltzmann transport equation (3-space, 2-angle, 1-energy) to model the flow of neutrons in a reactor
- Denovo scales up to 275K+ cores of Jaguar
- Has been a participant in the Joule code effort, is part of the SCALE reactor code system and is a key part of the ORNL-based CASL project (Consortium for Advanced Simulation of Light Water Reactors)
- Was selected as an early readiness code for Titan

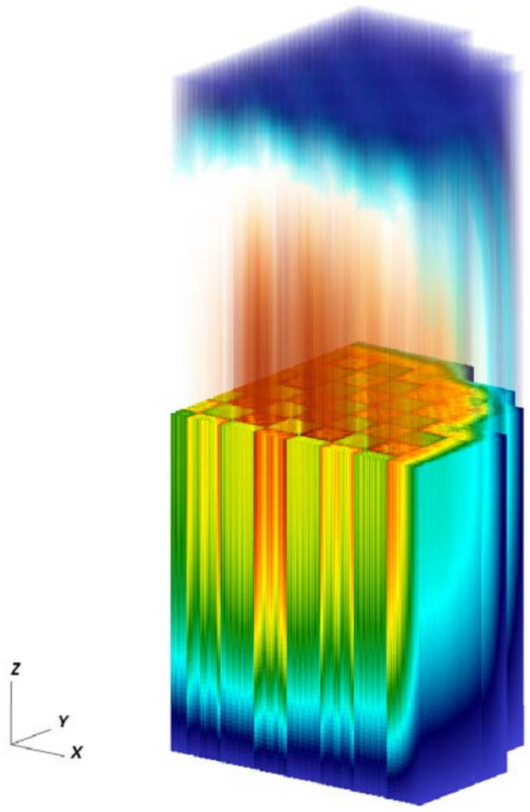




Denovo Algorithms

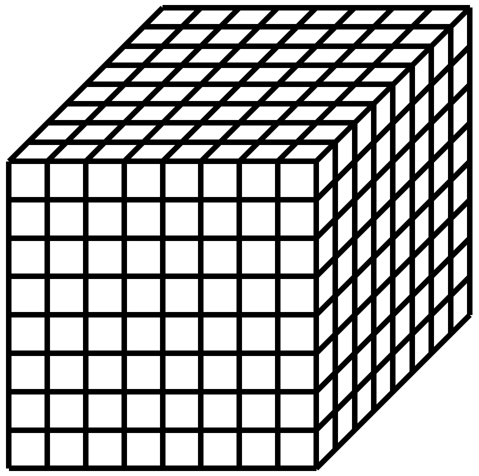
- Primary algorithms: the discrete ordinates method, 3-D sweep, GMRES linear solver and various eigensolvers, e.g., Arnoldi
- The execution time profile has a very prominent peak: nearly all the execution time (80-99%) is spent in a 3-D sweep algorithm. Second-highest is GMRES.
- Because of this, the 3-D sweep is the central focus of the effort to port Denovo to a accelerator-based system
- However, the sweep is a complex algorithm that is difficult to parallelize efficiently.

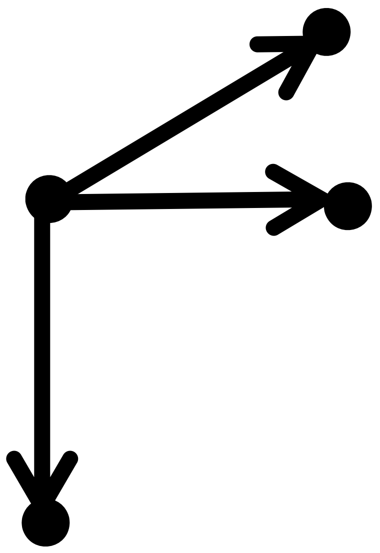




3-D Sweep Algorithm: Description

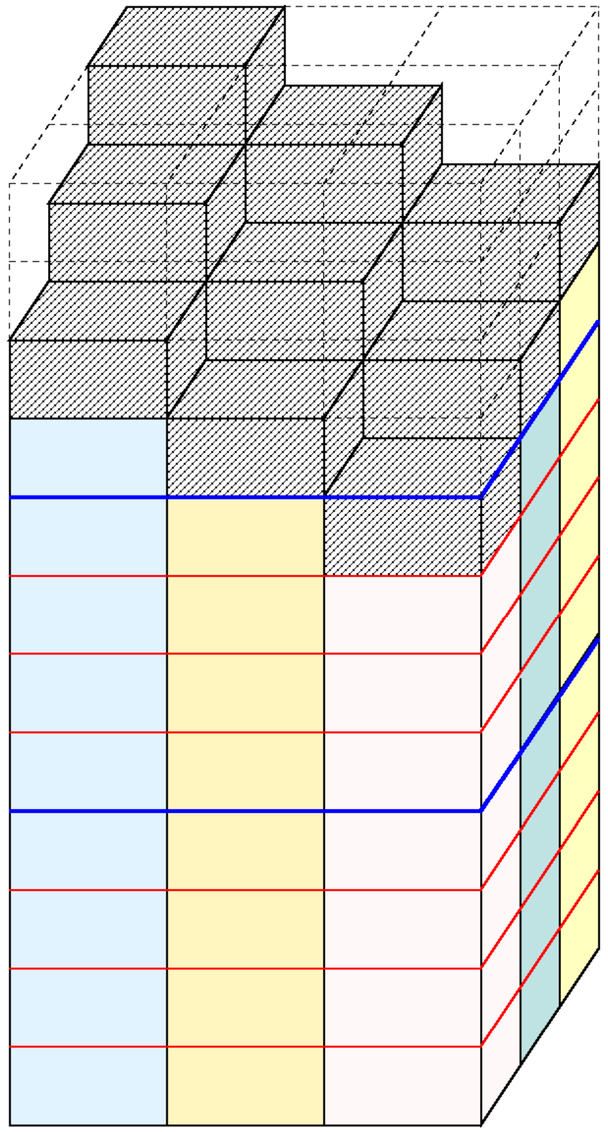
- Denovo is based on a 3-D structured grid
- The data dependency for the sweep operation is specified by a 4-point stencil
- The result at every gridcell is dependent on the result at the immediately lower gridcells in X, Y and Z, based on the direction of particle flow
- This induces a wavefront computation pattern – a sequence of diagonal planes sweeping inward from a corner that are recursively coupled.
- Thus, results at the far side of the grid cannot be computed until results at the near side are complete
- For standard parallel grid decompositions, most of the processors will be idle much of the time





Parallel Sweep: 1. High Level View

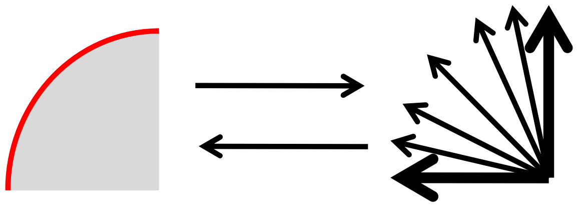
- The KBA algorithm solves this problem in parallel using a novel 2-D mapping of the problem to processors
- The calculation is started at one corner of the grid, other processors start work when their input data is available



Sweep Algorithm: 2. Per-Cell View

In addition to this “macro” view for the whole grid, at each gridcell there is also significant work to be done:

The input vector for the sweep is initially stored with a “moments” axis. (1) This moments axis must be transformed to an “angles” axis. (2) Then some element-level calculations are done, for the element unknowns. (3) Finally, the result must be transformed back to moments and the result stored in the output vector.



Thus we have these steps at each gridcell:

1. Load part of the input vector
2. Do small matrix-vector product to convert from moments to angles
3. Do discretization-related calculations on element unknowns
4. Do small matrix-vector product to convert from angles to moments
5. Store result in the output vector

Design Decisions: Programming Model

- Porting to GPU: options:
    1. CUDA
    2. OpenCL
    3. Compiler directives: PGI, CAPS, Cray, OpenACC.
- The 3-D sweep is a complex algorithm for which performance is highly sensitive to the implementation. To minimize project risk, a decision was made to take a more “close to the metal” approach by using CUDA. Additionally, directives were not production-ready at the start of the project.
- OpenCL more portable but can be less performant than CUDA on NVIDIA hardware. Also the OpenCL standard doesn’t support use of C++ for kernel code, though AMD does support it.
- We expect to be able to port to other parallel APIs going forward, e.g., OpenMP, OpenACC, compiler vectorization, etc. – the code structure is correct now, just need to change the details
- Usage of CUDA is abstracted into a facade class to minimize the lines of code with platform-specific dependencies.

Implementation: Refactor or Rewrite?

- Would prefer to refactor existing code, if possible.
- However, the original Denovo sweep had multiply-nested loop structure spanning multiple levels of the call tree. This would need to be permuted, which would require major code restructuring. Also, the memory access pattern was not properly localized for the GPU.
- Number of lines of code for the sweep is not huge (~ thousands).
- Thus, a rewrite approach was preferred over refactoring.

Mapping the Algorithm to the GPU

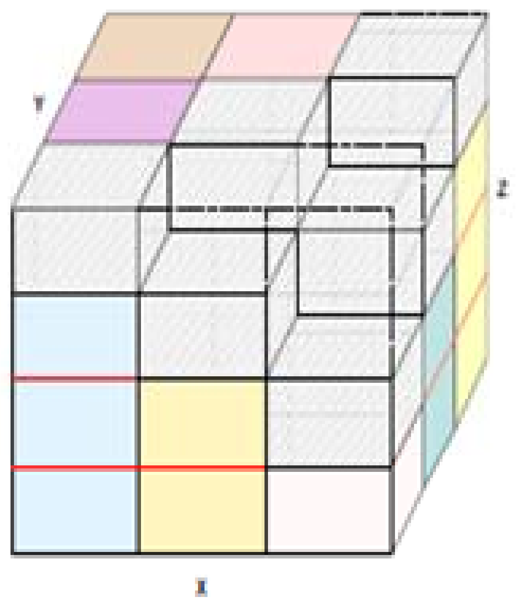
We have many candidate dimensions for parallelism: space (3), energy, moment/angle, octant, and also unknown (4 unknowns per gridcell for this discretization).

We need 4K-8K threads for the GPU to cover various latencies and keep the hardware busy.

Must be the right kind of parallelism – proper decoupling of data.

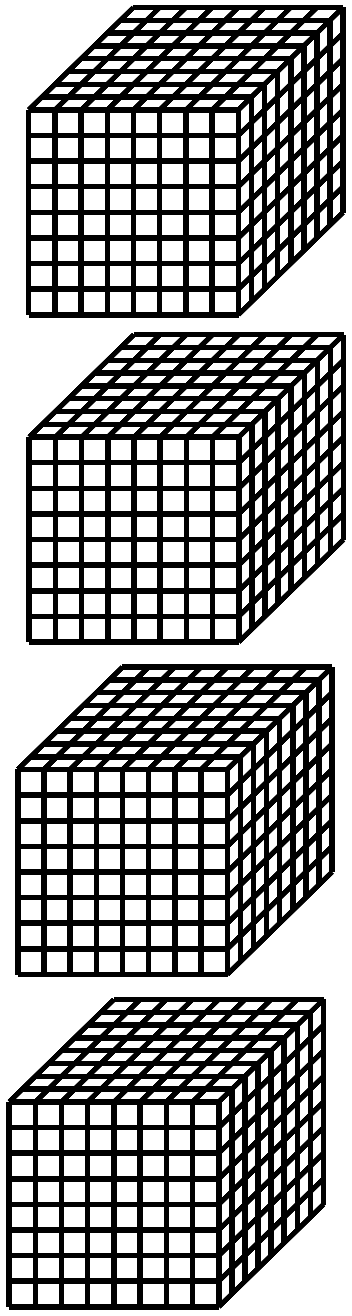
Also must have good memory access patterns (reuse of data loaded from global memory, coalesced stride-1 memory references, efficient use of registers, shared memory, caches on the GPU).

<u>Approach</u>: explore each problem dimension for potential thread parallelism.



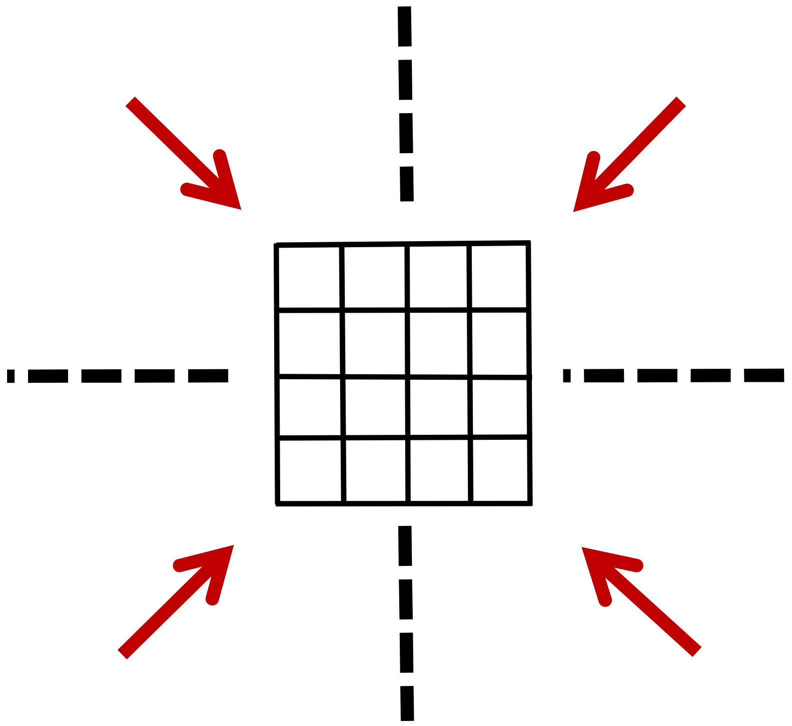
1. Parallelism in Energy

- Denovo exposes energy as a parallel dimension. These are fully independent, perfect axis for parallelism.
- Model problem has 256 energy groups – this helps, but we need enough for 4K-8K threads.
- Also need to use some of this 256 for node parallelism.



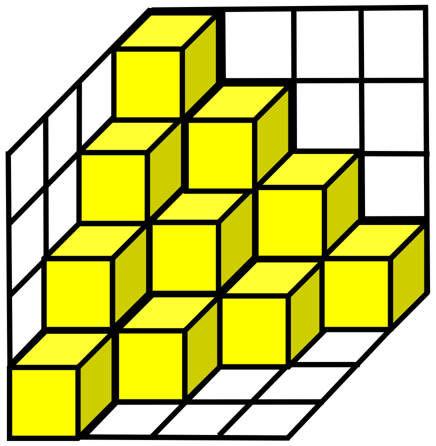
2. Parallelism in Octant

- Algorithm requires sweeps from 8 different directions.
- Sweep directions are independent, thus another 8X thread parallelism. Previously was an outer loop.
- Small issue: different octants update the same output vector, so we need to schedule properly to avoid write conflicts, slight loss of parallel efficiency



3. Parallelism in Space

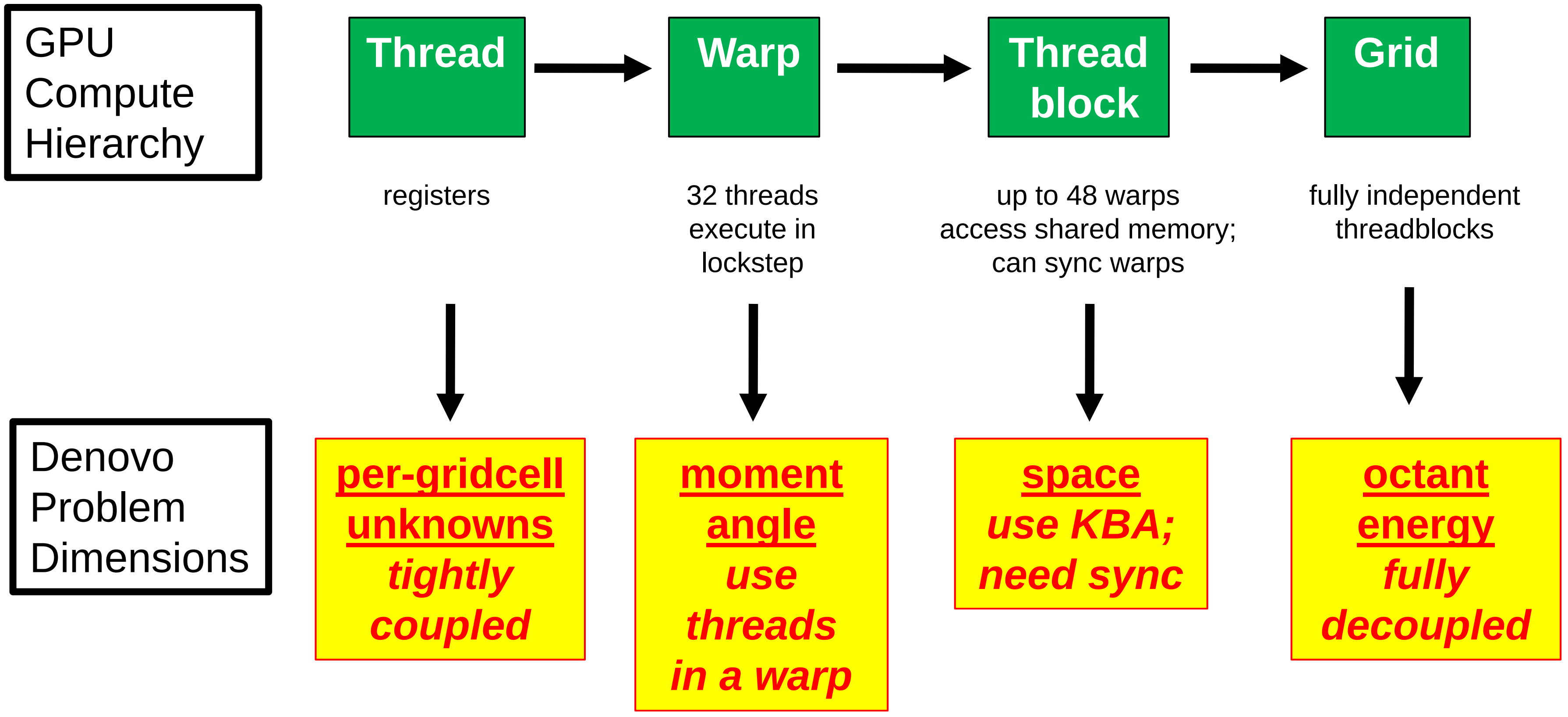
- We have this recursion, as mentioned before, that makes the computations non-independent
- However, the global KBA algorithm can be applied at the small scale of a single block in the GPU
- Set up block wavefronts, assign blocks to threads
- Sync between block wavefronts



4. Parallelism in Angle, Moment

- A strategy was formulated to parallelize the moment/angle axes at the gridcell level – map these axes to CUDA threads in-warp.
- Small dense matrix-vector products are perfect for thread parallelism – store vector in shared memory, relieve the register pressure.
- The two small matrices are the same across all gridcells, so they can be retained in L1 cache, to reduce a potentially high source of memory traffic.

Summary of Mapping of Dimensions

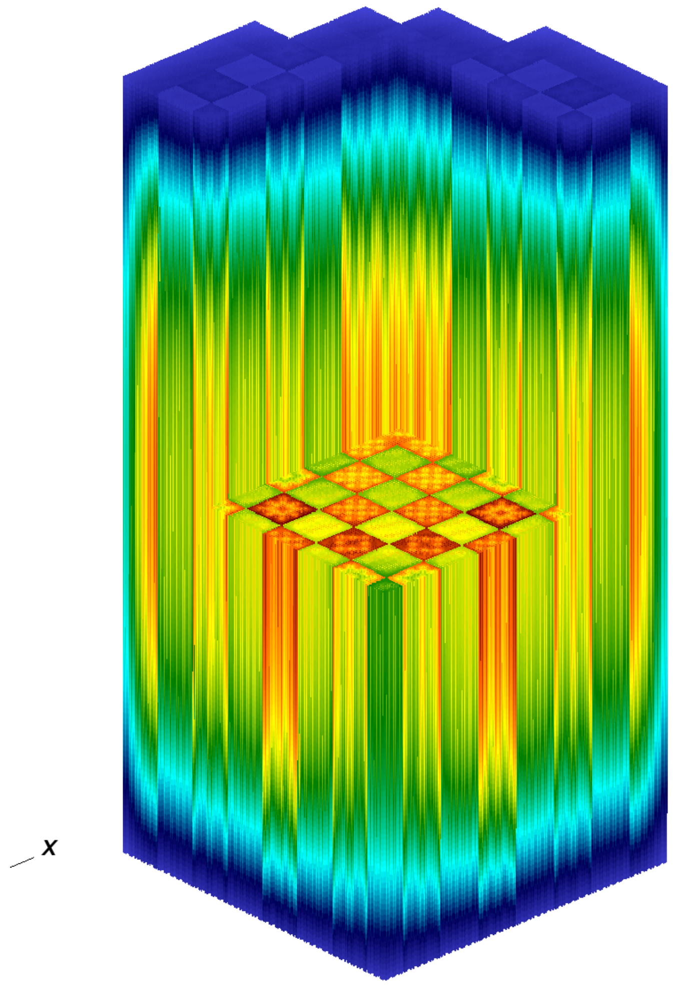

GPU Compute Hierarchy
Thread
Warp
Thread block
Grid
registers
32 threads execute in lockstep
up to 48 warps access shared memory; can sync warps
fully independent threadblocks
Denovo Problem Dimensions
per-gridcell unknowns tightly coupled
moment angle use threads in a warp
space use KBA; need sync
octant energy fully decoupled

GPU Kernel Management

- Use asynchronous data transfers / kernel launches, triple buffering, asynchronous MPI to overlap work.
- Use 16 MPI tasks on node, each task sends data independently to GPU, single MPI task on node manages kernel launches.

First Results: Test Problem

- 32x32x128 gridcells
- 16 energy groups
- 16 moments
- 256 angles
- Linear discontinuous elements – 4 unknowns per gridcell

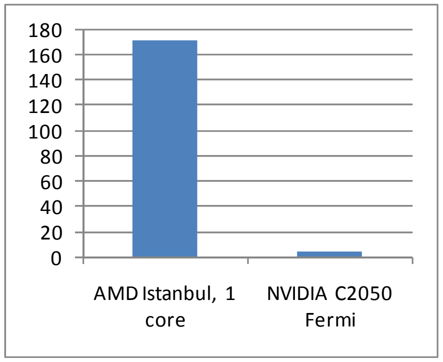


Results: Sweep GPU Performance

- Single core (AMD Istanbul) / single GPU (Fermi C2050) comparison
- For both processors, code attains about 10% of peak flop rate – this is considered good for this algorithm

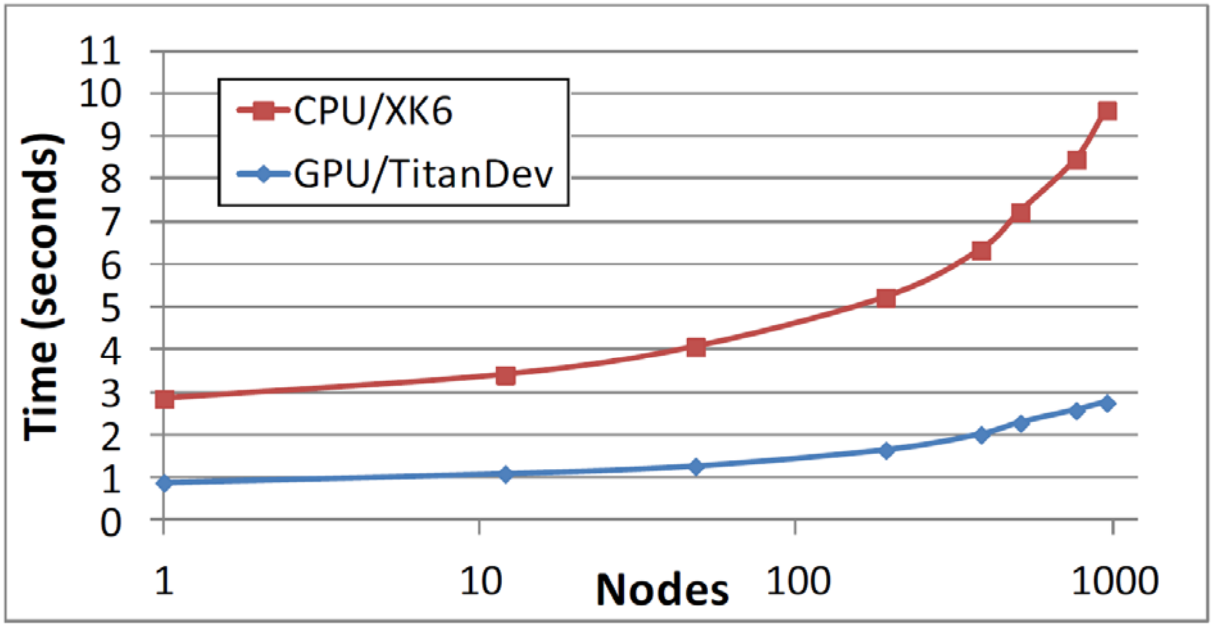
	AMD Istanbul 1 core	NVIDIA C2050 Fermi	Ratio
Kernel compute time	171 sec	3.2 sec	**54X**
PCIe-2 time (faces)	--	1.1 sec	
TOTAL	**171 sec**	**4.2 sec**	**40X**

NVIDIA Fermi is 40X faster than single Operon core



Sweep GPU Performance: TitanDev

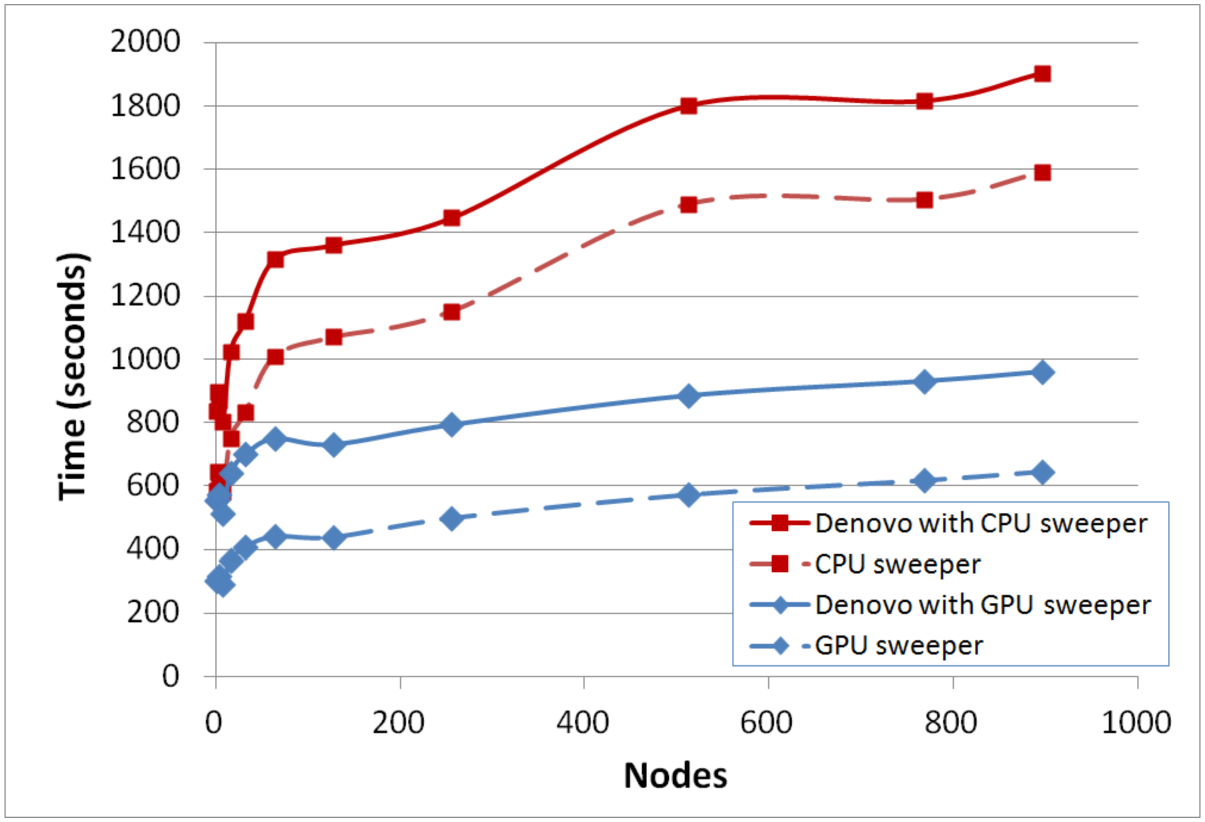
- Overall **3.5X** performance gain using NVIDIA Fermi X2090 nodes compared to AMD Interlagos CPU nodes
- CPU timings represent an additional **2X** improvement over the original sweep code



Performance Improvement Factors		GPU
		XK6 Fermi
CPU	XK6 / Interlagos	**3.5**
	XE6 / dual Interlagos	**3.3**

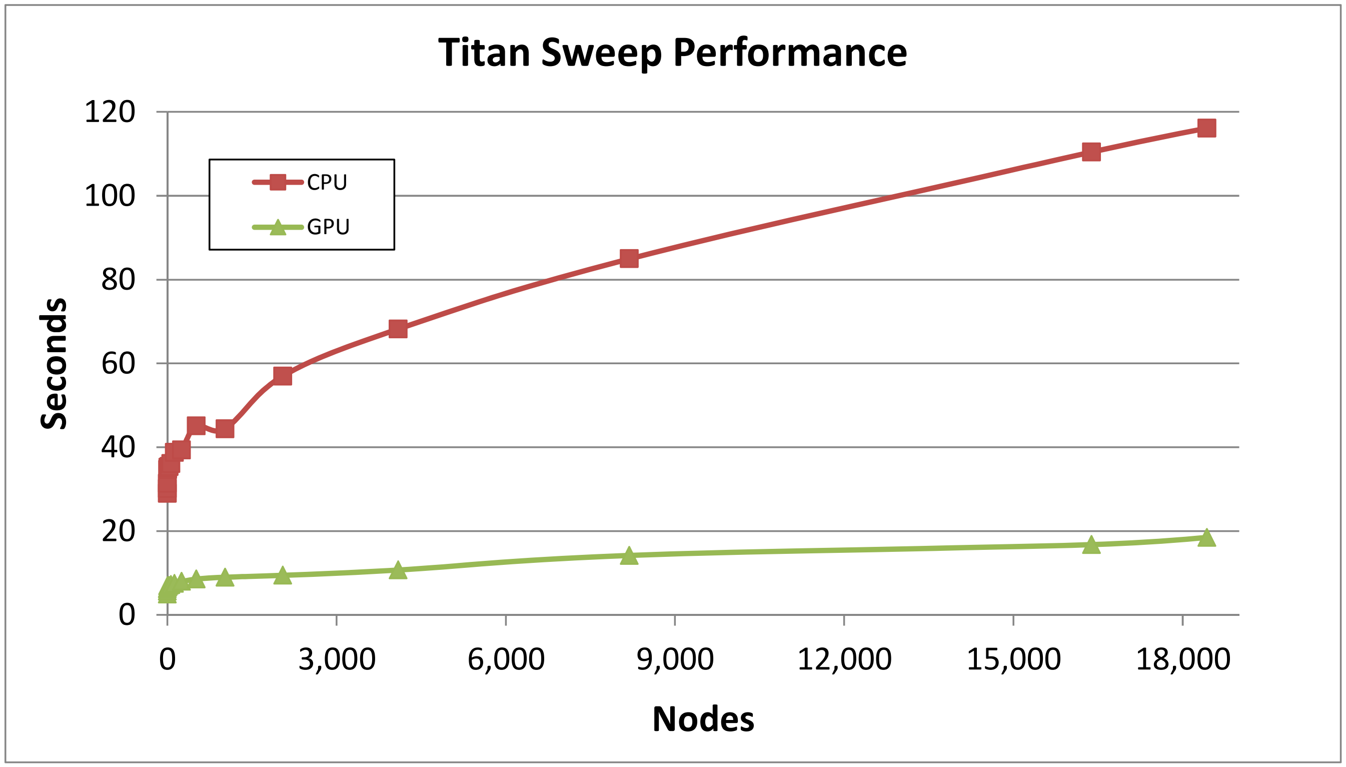
Denovo GPU Performance: TitanDev

- Full Denovo run, CPU vs. GPU sweeper, TitanDev, Fermi C2090
- We expect additional improvement from Kepler processors
- Will get further improvement using GPU version of GMRES via Trilinos



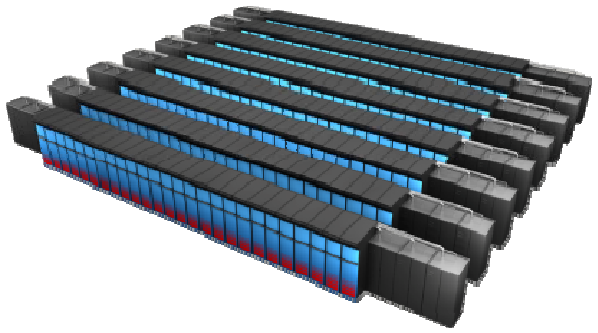
Titan Sweep Performance

- Weak scaling, increasing number of total gridcells
- Per node problem size (NX,NY,NZ) = (16,32,160), NE=56, NM=16, NA=256, LD elements
- Single sweep runtime on GPU is up to <u>6.6X</u> faster than CPU



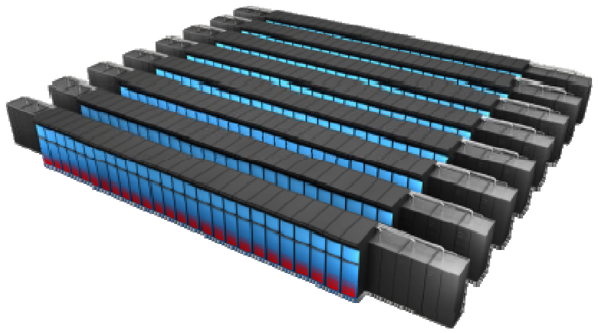
Conclusions: Lessons Learned

1. Major code restructurings were required – this consumed the majority of the work. This restructuring is required regardless of the parallel API used. The restructuring that was done will enable porting to other parallel APIs as needed.
2. CUDA was needed to get good performance for this complex algorithm – directives were new and not mature at the beginning of the project, they are improving now and will continue to do so.
3. Isolating CUDA-specific constructs in one place in the code is good defensive programming to help lessen the burden of porting to new programming models.



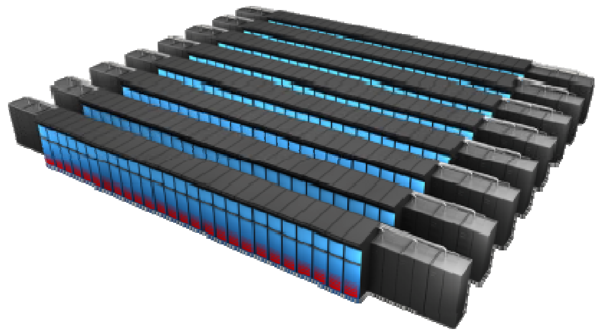
Conclusions: Lessons Learned (2)

4. Programming in a dual CPU/GPU programming style helps reduce code redundancy and helps with debugging.
5. It is challenging to negotiate conflict between heavy code optimization and good SWE practice – it's not always easy to have both, in general and specifically using CUDA.
6. It is helpful to develop a performance model based on flop rate, memory bandwidth and algorithm tuning knobs, to guide mapping of the algorithm to the GPU and evaluate tradeoffs.



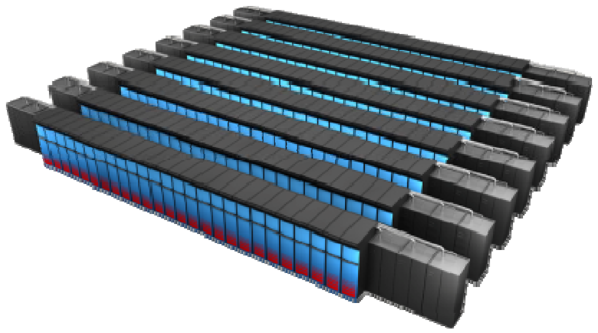
Conclusions: Lessons Learned (3)

7. It is sometimes worthwhile to write small codes to test performance for simple operations, incorporate this insight into algorithm design.
8. It is a challenge to understand what the processor is doing, under the abstractions. Performance optimization requires that performance behaviors be exposed, not hidden.
9. It is difficult to know beforehand what will be the best strategy for parallelization or what will be the final outcome – e.g., difficult to predict how many registers will be needed, and register space is limited.



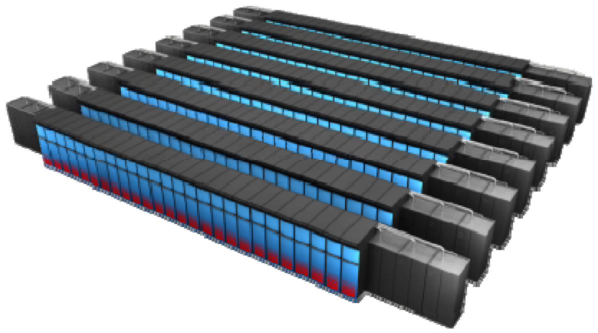
Conclusions: Lessons Learned (4)

10. Performance can be very sensitive to small tweaks in the code – must determine empirically the best way to write the code.
11. Often, the GPU porting effort for the algorithm also improves performance on the CPU (in this case, in fact, 2X).
12. Expert help is useful, e.g., NVIDIA forums, trainings, etc.



Acknowledgements

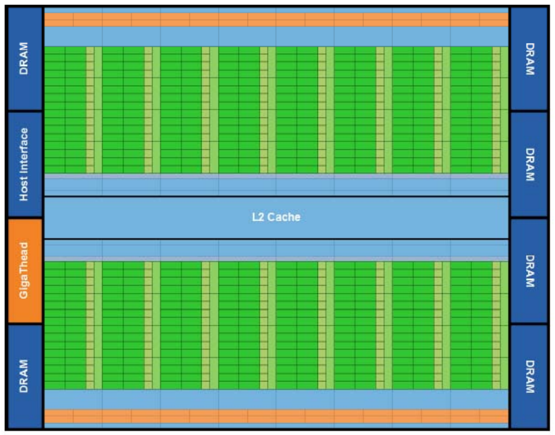
- Denovo development team: Tom Evans, Greg Davidson, Josh Jarrell, Chris Baker, Steve Hamilton
- Cray: Kevin Thomas
- NVIDIA: John Roberts, Cyril Zeller, Paulius Micikevicius
- OLCF compute resources: JaguarPF, Chester, Yona, Lens

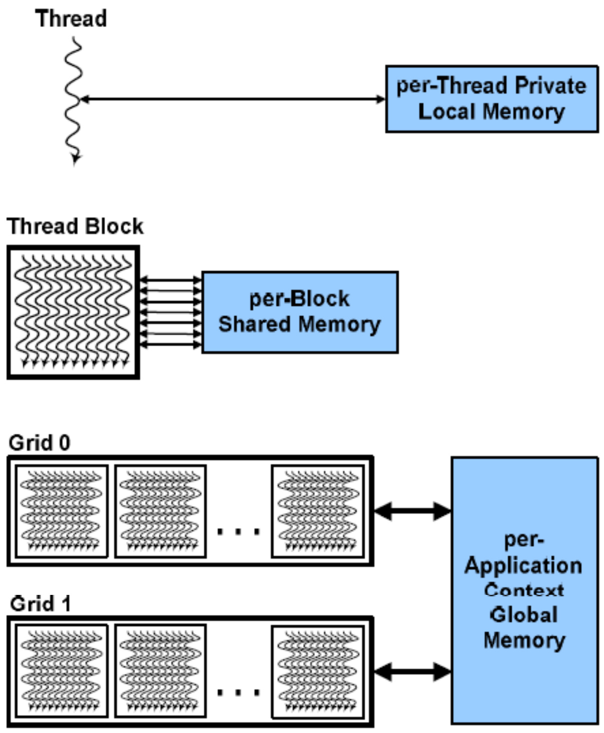


Supplementary slides

GPU Architecture

- The NVIDIA GPU processor is a manycore architecture with hundreds of compute cores.
- They are programmed via threads.
- Threads are arranged in groups of 32 (warps) that compute in lockstep.
- These are collected into threadblocks.
- Threadblocks are independent and form a grid.
- Programs access main ("global") memory.
- Programs can also use a faster, smaller "shared" memory – a programmable cache.
- Also L1 cache, L2 cache, registers.
- GPU connected to CPU by PCIe-2 bus




Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
. . .
Grid 1
. . .
per-Application Context Global Memory

Images courtesy NVIDIA