Porting the Denovo Radiation Transport Code to Titan: Lessons Learned

OLCF Titan Workshop 2012

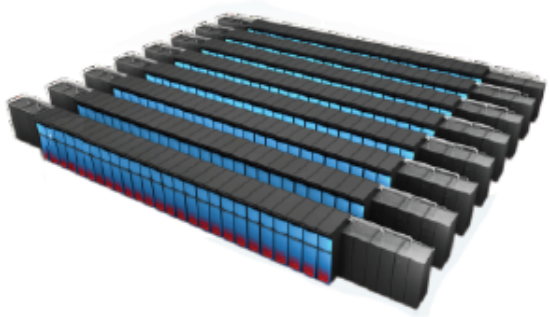
Wayne Joubert

Scientific Computing Group

Oak Ridge Leadership Computing Facility

Oak Ridge National Laboratory

Context: Titan Readiness Effort



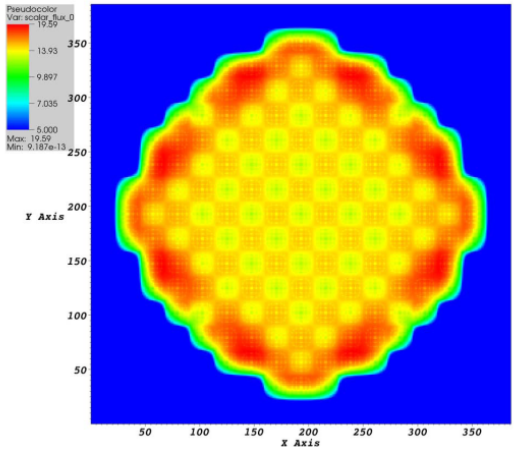
Denovo was selected as one of six early readiness applications for Titan

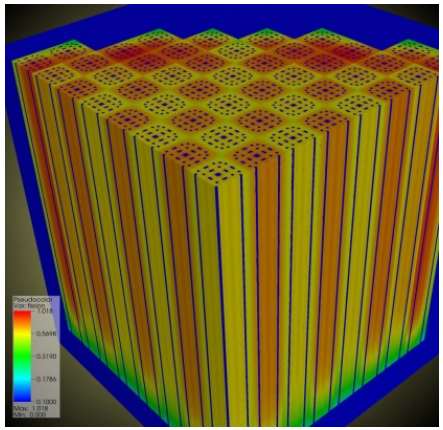
The intent was that the experience porting these early readiness codes would shed light on how to port other apps going forward to Titan

App	Science Area
LSMS	Materials
PFLOTRAN	Earth Sciences
CAM/SE	Climate
S3D	Combustion
LAMMPS	Biosciences
Denovo	Nuclear Energy

Denovo

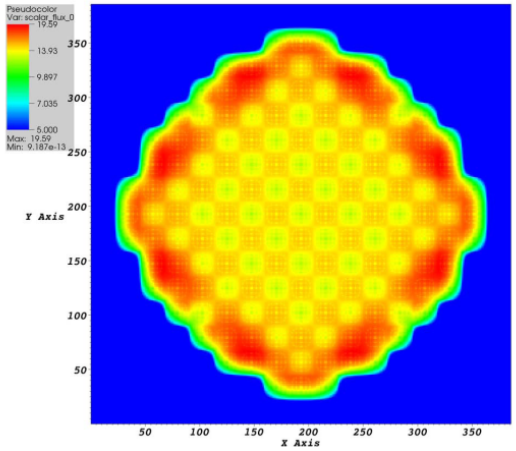
- Used to solve the radiation transport problem for advanced nuclear reactor design
- Solves the linear Boltzmann equation in six dimensions (3-space, 2-angle, 1-energy)
- Written primarily in C++ under an Agile software development process with rigorous SQE
- Scales up to 200K cores on ORNL's 2 PF Jaguar system.
- Denovo is targeted for porting to ORNL's next-generation HPC system

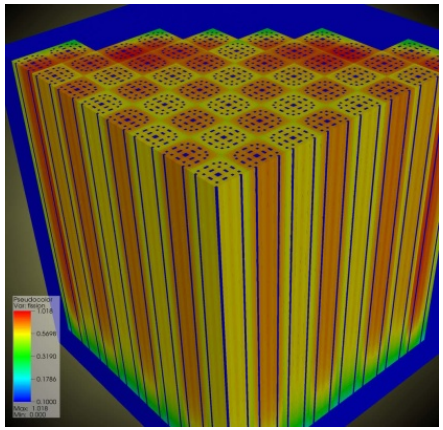




Denovo Algorithms

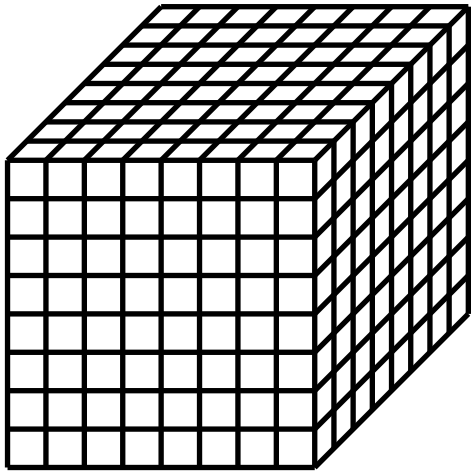
- Primary algorithms: the discrete ordinates method, 3-D sweep, GMRES linear solver and various eigensolvers, e.g., Arnoldi
- The execution time profile has a very prominent peak: nearly all the execution time (80-99%) is spent in a 3-D sweep algorithm.
- Because of this, the 3-D sweep is the central 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.

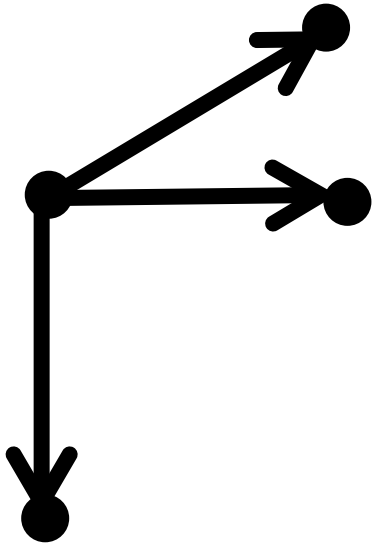




Denovo 3-D Sweep Algorithm

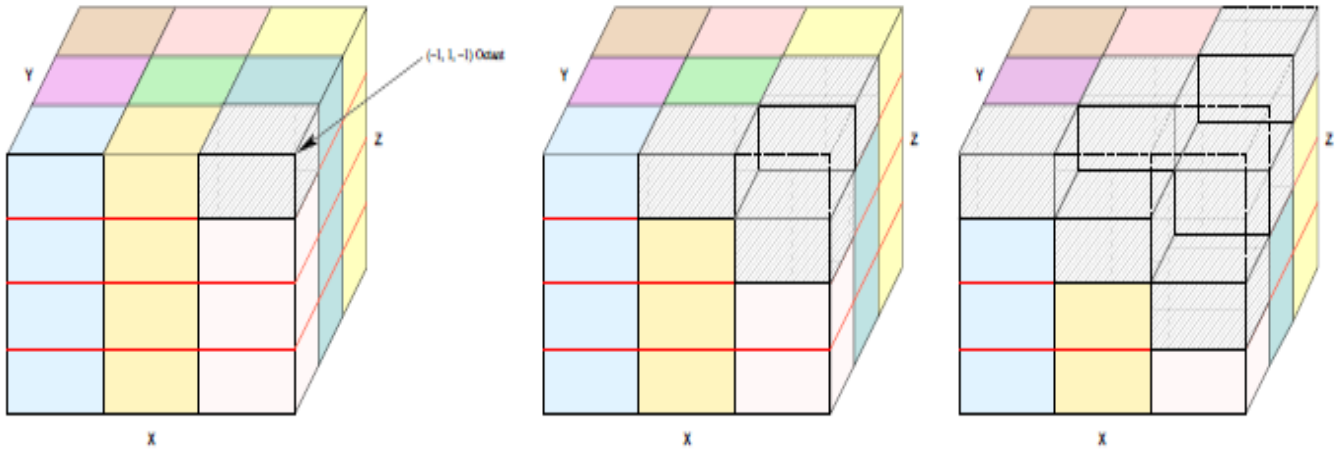
- Most of the Denovo runtime (80-99%) is spent in the KBA sweep algorithm
- This is a recursive wavefront algorithm that is difficult to parallelize
- Essentially a 4-point stencil, where each value depends on the previous values in x, y and z
- Induces a set of hyperplanes (wavefronts) that are processed in sequence to sweep through the grid from a corner





Parallel Sweep: 1. High Level View

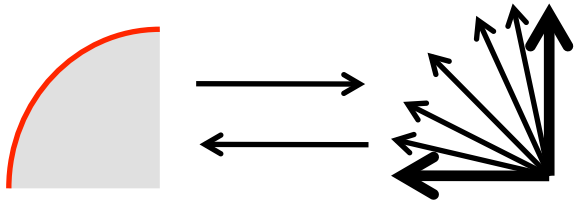
- The KBA algorithm (Koch, Baker, Alcouffe from LANL) solves this problem in parallel using a novel 2-D mapping of the problem to processors
- The calculation is started at one corner of the grid, other processors start work when their input data is available



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.

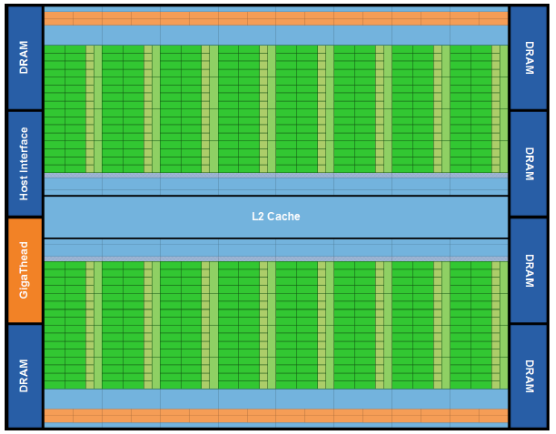


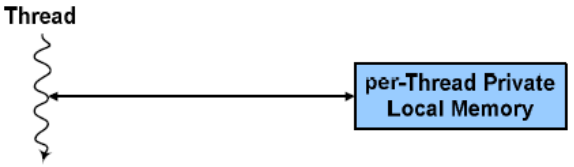
Thus we have these steps at each gridcell:

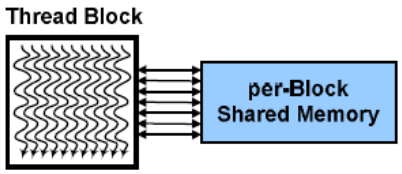
1. Load part of the input vector
2. Do small matrix-vector product to convert from moments to angles
3. Do discretization-related calculations on element unknowns
4. Do small matrix-vector product to convert from angles to moments
5. Store result in the output vector

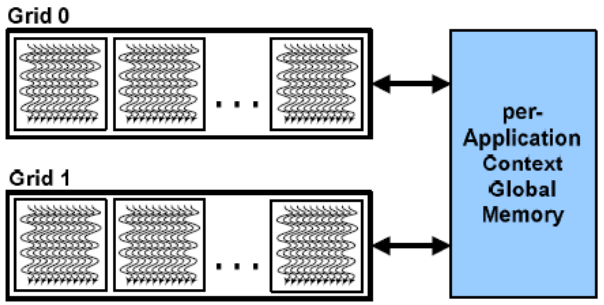
GPU Architecture

- The NVIDIA Fermi processor is a manycore architecture with 512 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.
- Connected to CPU by PCIe-2 bus









Images courtesy NVIDIA

How to Program the Sweep on the GPU?

- Decide what language / parallel API to use to program the GPU, through a careful analysis of the code and algorithms.
- Options:
  1. CUDA: a minor extension of C/C++ for GPU thread programming, also available for Fortran 90
  2. OpenCL: a multi-vendor standard similar to CUDA
  3. Compiler directives: PGI, CAPS, Cray, OpenACC ...
- Sweep is a complex algorithm, with many dimensions. Directives may not be flexible enough or expose enough hardware functionality to get the needed performance.
- NVIDIA supports OpenCL, but going forward CUDA will be better supported and more in-sync with new hardware features.
- Thus use CUDA. For portability use dual CPU/GPU coding style. Program defensively by using a coding style that isolates CUDA constructs in facade classes, well-positioned to port to future platforms.

CUDA/OpenCL vs. Directives

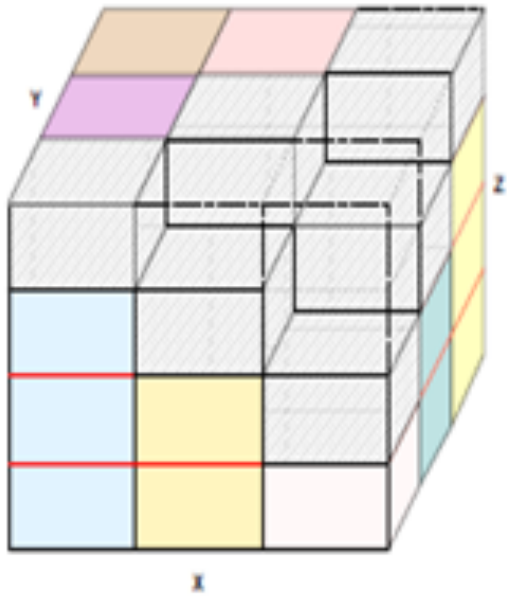
- Which is best?
- Will depend on the application. We have early readiness apps using both approaches: LAMMPS, Denovo (CUDA), S3D, CAM/SE (directives).
- Directives are easier for preexisting serial code by accelerating loops.
- CUDA allows more careful control of the mapping of the algorithm to the hardware.
- In some ways the issue is analogous to OpenMP vs. Pthreads. OpenMP potentially less invasive to serial code, Pthreads allows more flexibility. We have codes that use both, e.g., GTC (OpenMP), Madness (Pthreads).

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 not huge (~ thousands).
- Thus, a rewrite probably easier.

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 are told by NVIDIA that we need 4K-8K threads for the GPU to cover various latencies.

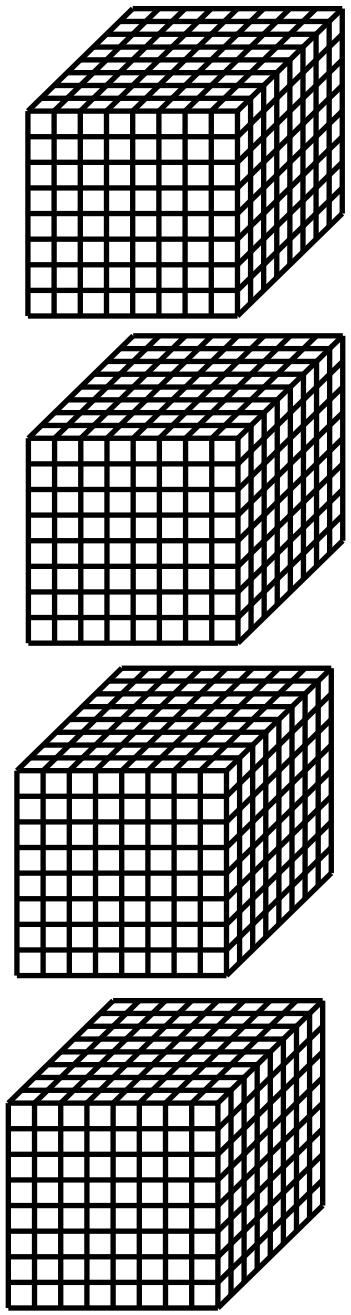
Also need the right kind of parallelism – proper decoupling of data.

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

Approach: explore each problem dimension for potential thread parallelism.

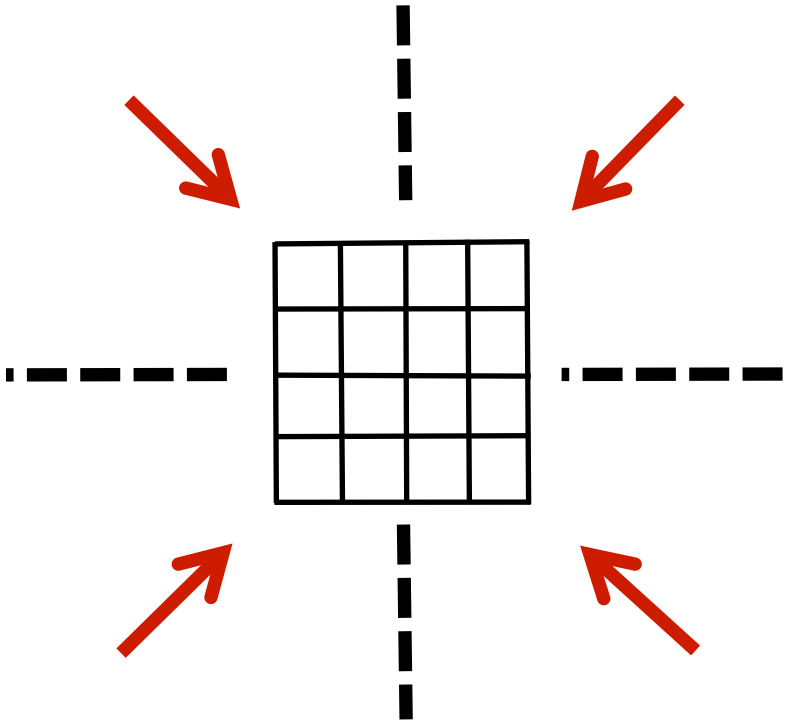
1. Parallelism in Energy

- 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.



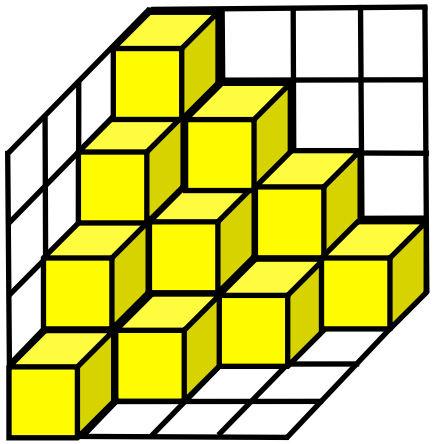
2. Parallelism in Octant

- 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 this small scale.
- Set up block wavefronts, assign blocks to threads.
- Sync between block wavefronts.



Intermezzo: GPU Memories

- With this paralleization scheme, code performed at only about 1% of peak flop rate, much lower than predicted by the performance model
- NVIDIA Fermi streaming multiprocessor (SM) has 64K of combined L1 cache + shared memory, 128K register file
- This sounds big, but it must be shared by hundreds of CUDA threads (!)
- To effectively use these fast memories, need to find problem axes for which data can be shared/reused between threads, put in shared memory instead of registers, thus reduce register spillage

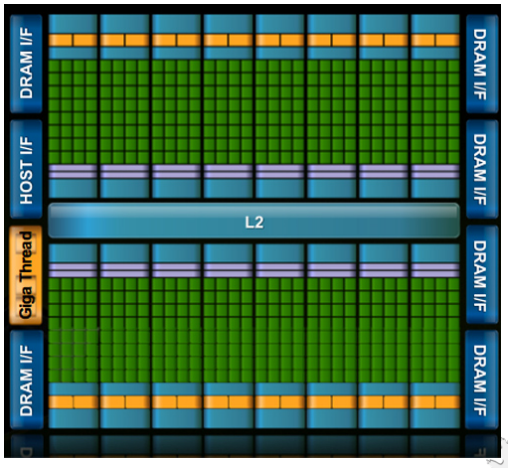
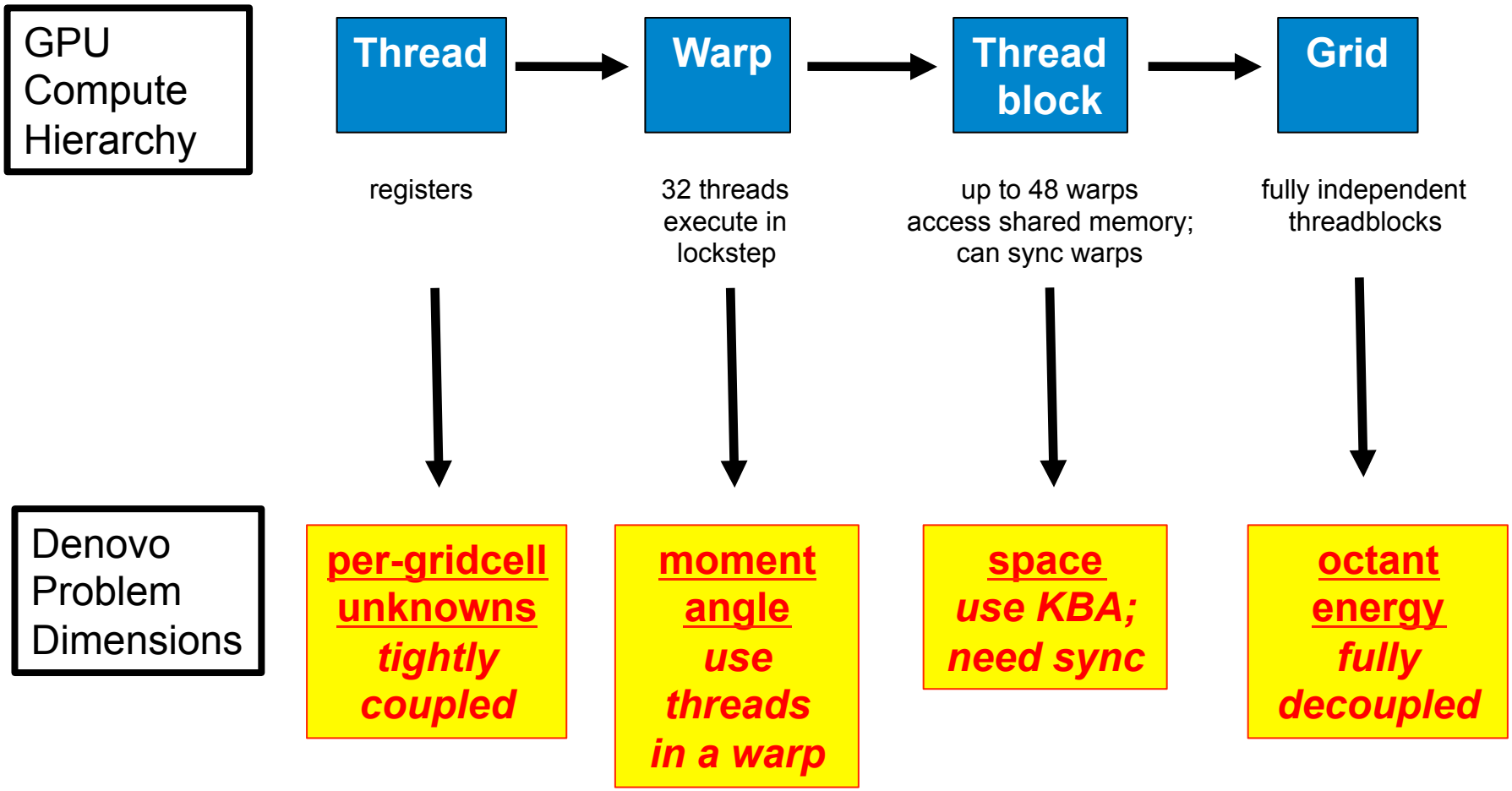


Image courtesy NVIDIA

4. Parallelism in Angle, Moment

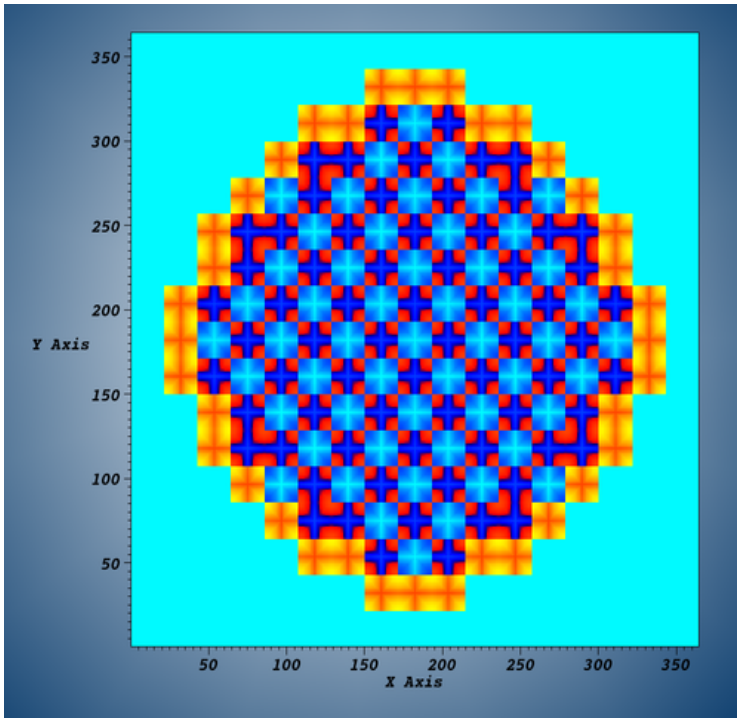
- A new strategy to parallelize the moment/angle axes at the gridcell level – map these axes to CUDA threads in-warp.
- Small dense matrix-vector products are perfect for thread parallelism – store vector in shared memory, relieve the register pressure.
- The two small matrices are the same across all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

Summary of Mapping of Dimensions



Results: Test Problem

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

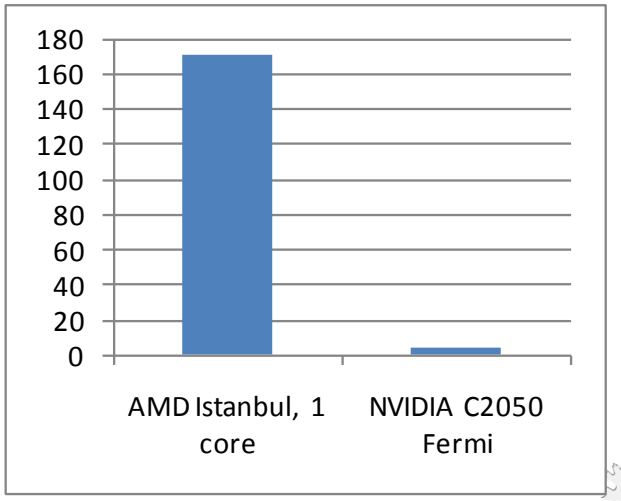


Results: Sweep GPU Performance

- Single core (AMD Istanbul) / single GPU (Fermi C2050) comparison

	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 Opteron core



Conclusions: Lessons Learned

1. Major code restructurings were required – this is required regardless of the parallel API used. Estimate >50% of development time spent in code restructuring irrelevant to GPU-specific features, rest of the time spent tuning code to GPU caches, etc.
2. CUDA was used to get good performance for this complex algorithm – directives add an abstraction layer, may not expose all needed performance. Other codes will be different – depends on the application (library calls, compiler directives, CUDA/OpenCL).
3. Isolating CUDA-specific constructs in one place in the code is good defensive programming to prepare for programming models that may change. C++ facade classes to hide details are useful.

Conclusions: Lessons Learned (2)

4. Programming in a dual CPU/GPU programming style helps reduce code redundancy, helps with debugging and improves portability.
5. It is challenging to negotiate conflict between deep code optimization and good SWE practice – often it is not easy to have both.

Conclusions: Lessons Learned (3)

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.
7. It is 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, even CUDA. Some details are proprietary.
9. It is difficult to know beforehand what will be the best strategy for parallelization or what will be the final outcome – a porting effort could easily fail if the GPU has inadequate register space for the planned algorithm mapping.

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.

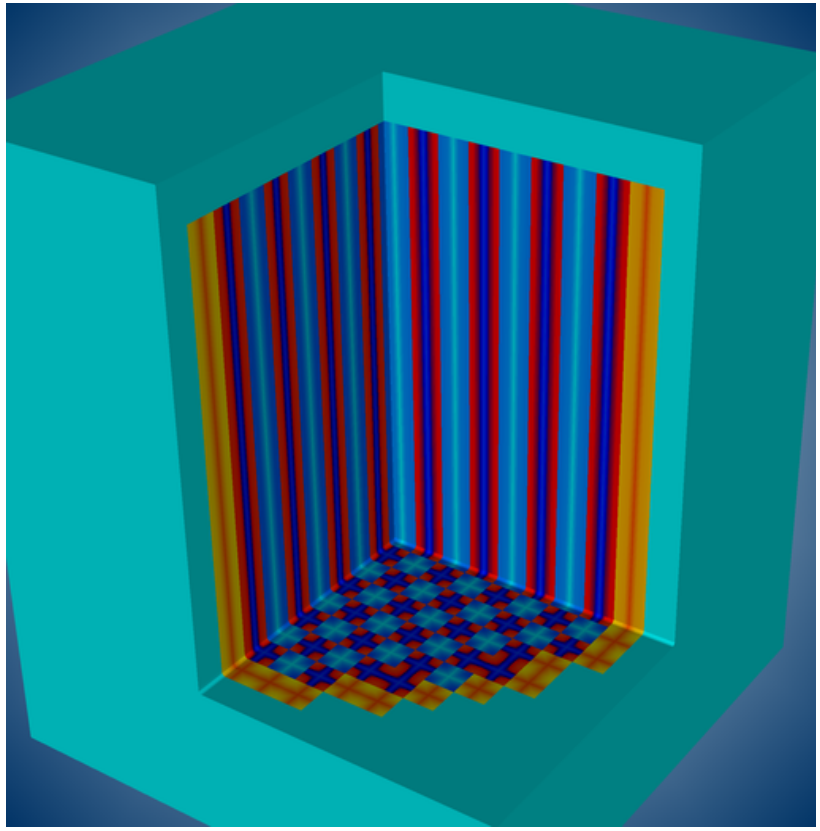
Acknowledgements

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

Supplementary slides

Denovo Science Problem

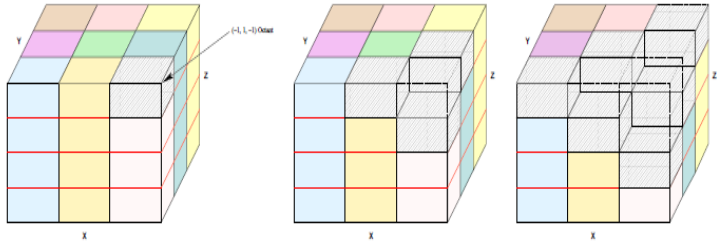
- For high-resolution nuclear reactor design
- Nuclear reactor analysis requires modeling the flux of moving neutrons in the reactor core
- At any spatial gridcell, there is a quantity of neutrons that is is binned by (1) direction of particle motion and (2) energy value
- This results in a 6-dimensional problem (3-space, 2-direction, 1-energy)
- The fine resolution required along each of these six dimension leads to problems of enormous size



Sweep Code Programming Model/Style

- Code is in C++.
- Decided to implement a single code that can run on both CPU and GPU. Makes sense for maintainability, also greatly helps debugging.
- Following older example of MPI, try to put CUDA-related code in one place, e.g., facade class. Want to be ready for unknown programming models coming in the future.

Sweep Performance Characteristics



- In order to port to GPU, need to understand the performance behavior of the sweep algorithm in detail
  - Data access pattern
  - How much time spent in flops, memory access, communication
  - Which problem dimensions can be thread-parallelized on the GPU
  - Is there enough space in the registers, caches to get the needed data reuse
- Rethink the algorithm from first principles, putting all algorithm design issues on the table.
- How do we restructure the algorithm to improve data reuse, expose thread parallelism?

A Sweep Code Performance Model

- It can be very useful to have a formula that expresses the runtime of a code in terms of:
  - Flop counts, memory access counts, message counts, ...
  - Hardware characteristics: clock speeds, bandwidths, ...
- Helps guide the parallelization / optimization process.
- Can understand performance tradeoffs for design decisions before writing any code.
- Understand what dominates (floating point operations, PCIe-2 transfer, memory bandwidth, etc.) – what is most in need of optimization.
- Also after writing the code helps diagnose whether performance of the code is where it should be.

Observations

- 40X faster than Istanbul core.
- Istanbul is 6-core, so Fermi about 7X faster than the entire Istanbul processor.
- For both CPU and GPU, code attains about 10% of peak flop rate – this is considered good for this algorithm.
- Expect more optimizations to be possible going forward.