Porting the Denovo Radiation Transport Code to Titan: Lessons Learned

OLCF Titan Summit 2011

OLCF

Oak Ridge Leadership Computing Facility

Wayne Joubert

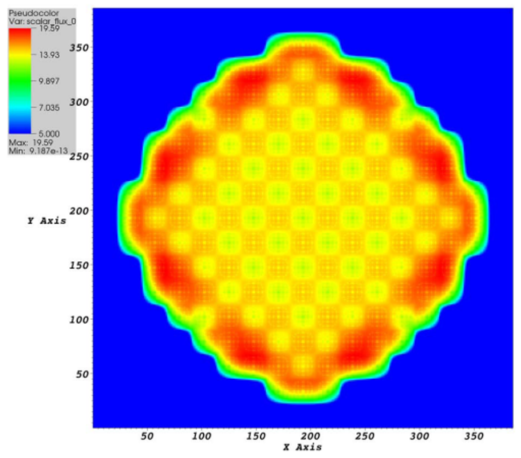
Scientific Computing Group

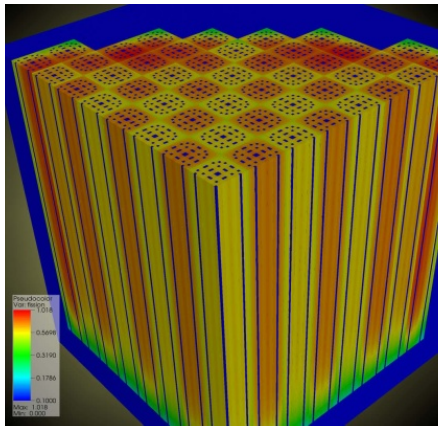
Oak Ridge Leadership Computing Facility

Oak Ridge National Laboratory

What is Denovo

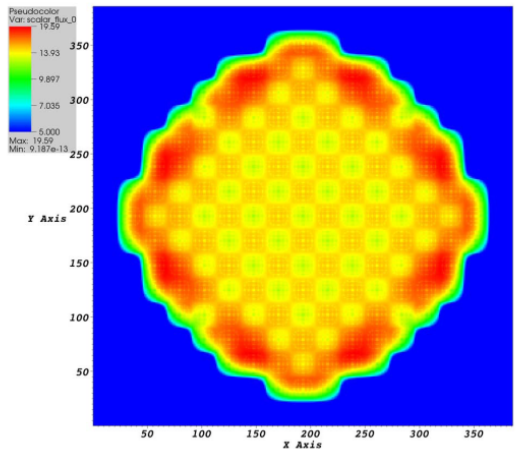
- Denovo is a radiation transport code used in advanced nuclear reactor design
- It solves for the density of particle flux in a 3-D spatial volume such as a reactor
- In particular, it solves the six-dimensional linear Boltzmann equation (3-space, 2-angle, 1-energy)
- Denovo scales up to 200K cores on ORNL's 2.3PF Jaguar system.
- It is part of the CASL project (Consortium for Advanced Simulation of Light Water Reactors) and the SCALE code system (Standardized Computer Analyses for Licensing Evaluation)
- It was selected as an early port code for Titan

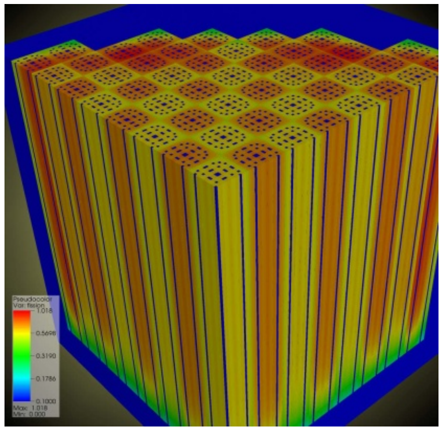




Denovo Algorithms

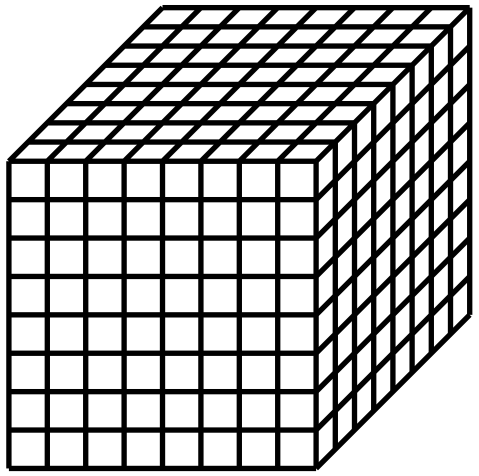
- Primary algorithms: the discrete ordinates method, 3-D sweep, GMRES linear solver and various eigensolvers, e.g., Arnoldi
- The execution time profile has a very prominent peak: nearly all the execution time (80-99%) is spent in a 3-D sweep algorithm.
- Because of this, the 3-D sweep must be the central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

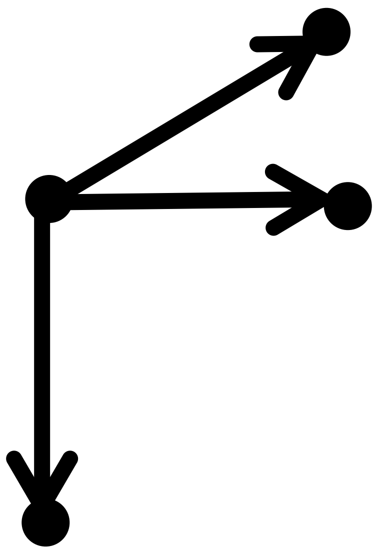




3-D Sweep Algorithm: Description

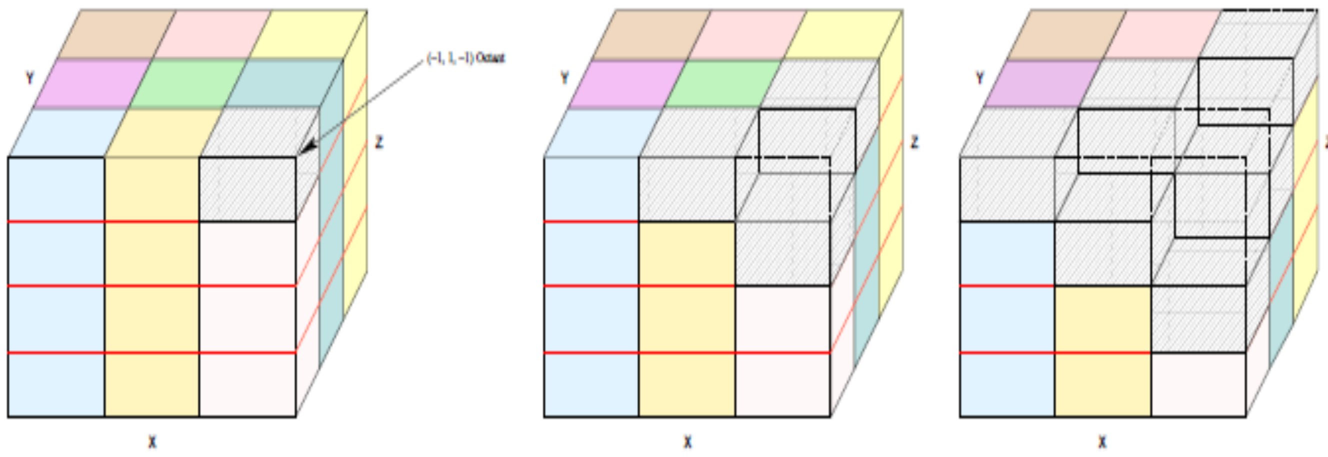
- Denovo is based on a 3-D structured grid
- The data dependency for the sweep operation is specified by a 4-point stencil
- The result at every gridcell is dependent on the result at the immediately lower gridcells in X, Y and Z.
- This induces a wavefront computation pattern – a sequence of diagonal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





Parallel Sweep: 1. High Level View

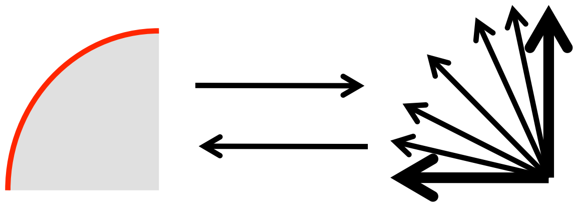
- The KBA algorithm solves this problem in parallel using a novel 2-D mapping of the problem to processors
- The calculation is started at one corner of the grid, other processors start work when their input data is available



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.

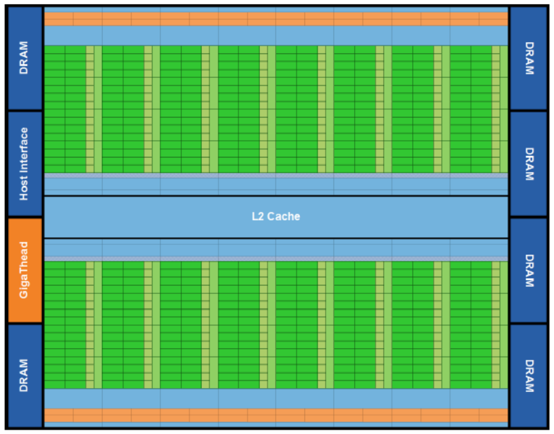


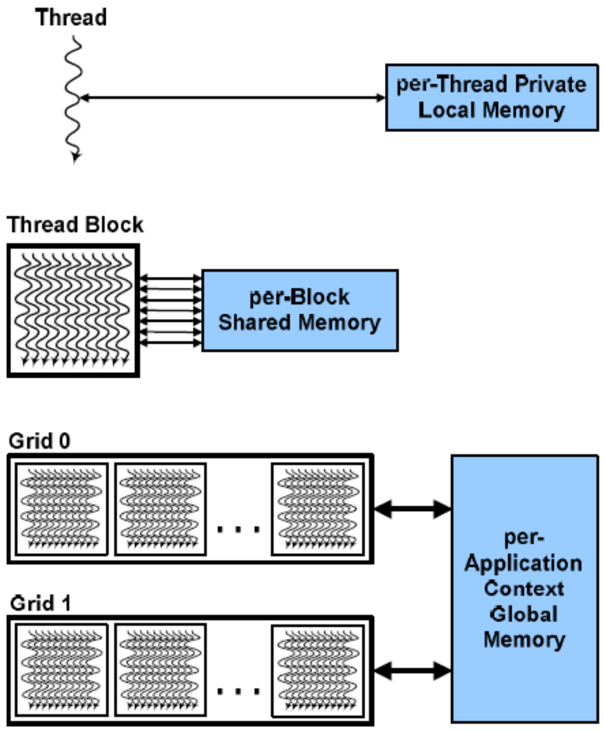
Thus we have these steps at each gridcell:

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

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




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

Images courtesy NVIDIA

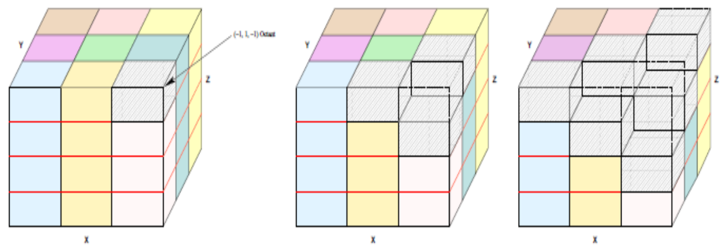
How to Program the Sweep on the GPU?

- Decide what language / parallel API to use to program the GPU.
- 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: similar to OpenMP (PGI, CAPS, Cray, ...)
- 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 support OpenCL, but going forward CUDA will be better supported and more in-sync with new hardware features.
- Thus use CUDA. Use C++ for consistency with Denovo base language.

Refactor or Rewrite?

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

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. 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.

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.

Mapping the Algorithm to the GPU

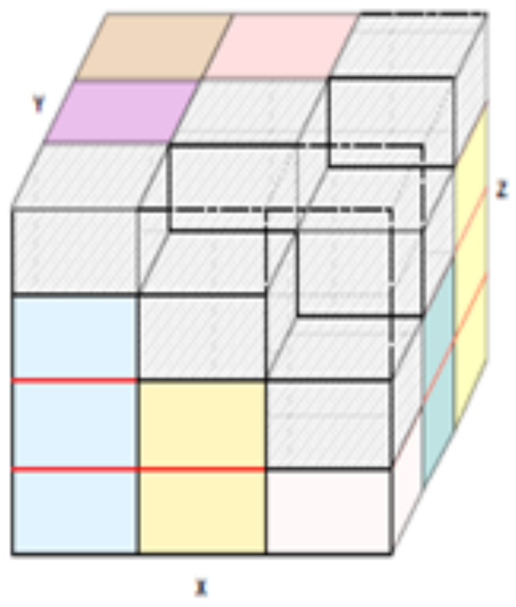
We have many candidate dimensions for parallelism: space (3), energy, moment/angle, octant, and also unknown (4 unknowns per gridcell for this discretization).

We are told by NVIDIA that we need 4K-8K threads for the GPU to keep all GPU streaming multiprocessors busy and cover various latencies.

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

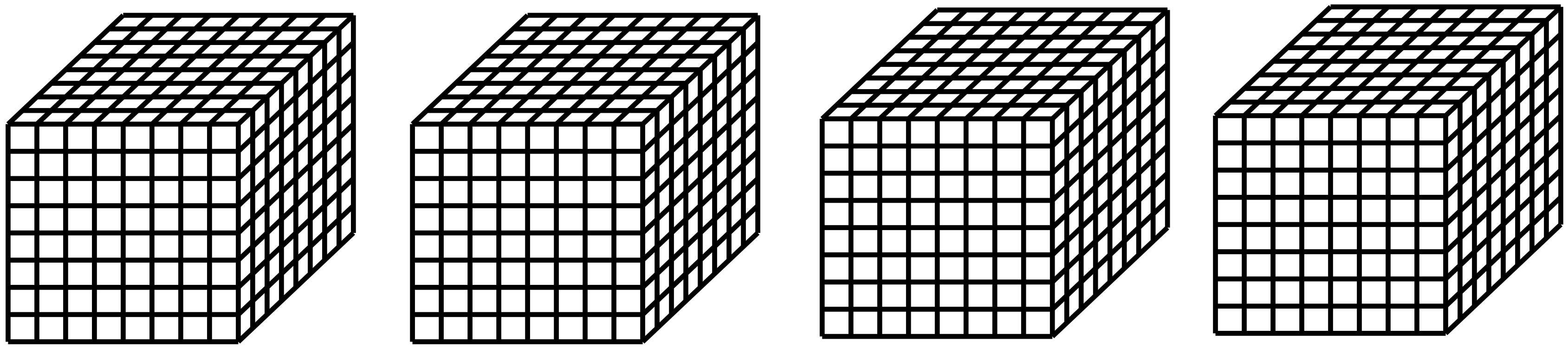
Also must have good memory access patterns (reuse of data loaded from global memory, coalesced stride-1 memory references, good use of registers, shared memory, caches on the GPU).

Approach: explore each problem dimension for potential thread parallelism.



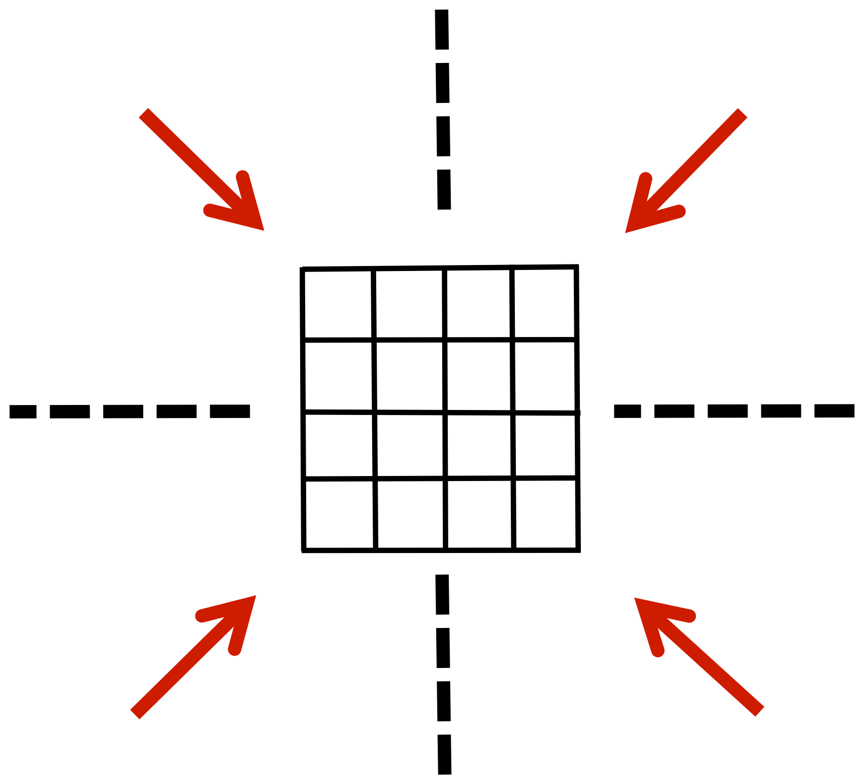
1. Parallelism in Energy

- Denovo exposes energy as a parallel dimension.
- Values for different energies are entirely independent in the 3-D sweep, thus the algorithm is embarrassingly parallel along this axis (!).
- Model problem has 256 energy groups – this helps, but we need to look further in order to get to 4K-8K threads.
- Also need to use some of this 256 for node parallelism.



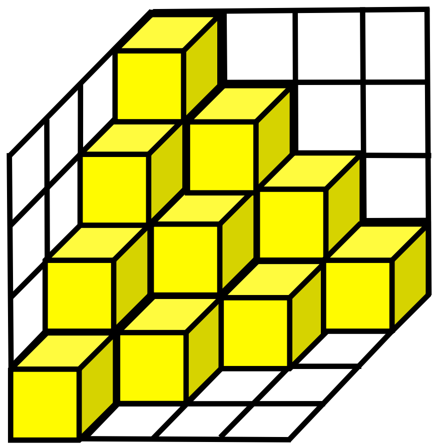
2. Parallelism in Octant

- Algorithm requires sweeps from 8 different directions.
- Sweep directions are independent, thus another 8X thread parallelism (!).
- Small issue: different octants update the same output vector, so we need to schedule properly to avoid write conflicts.



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.



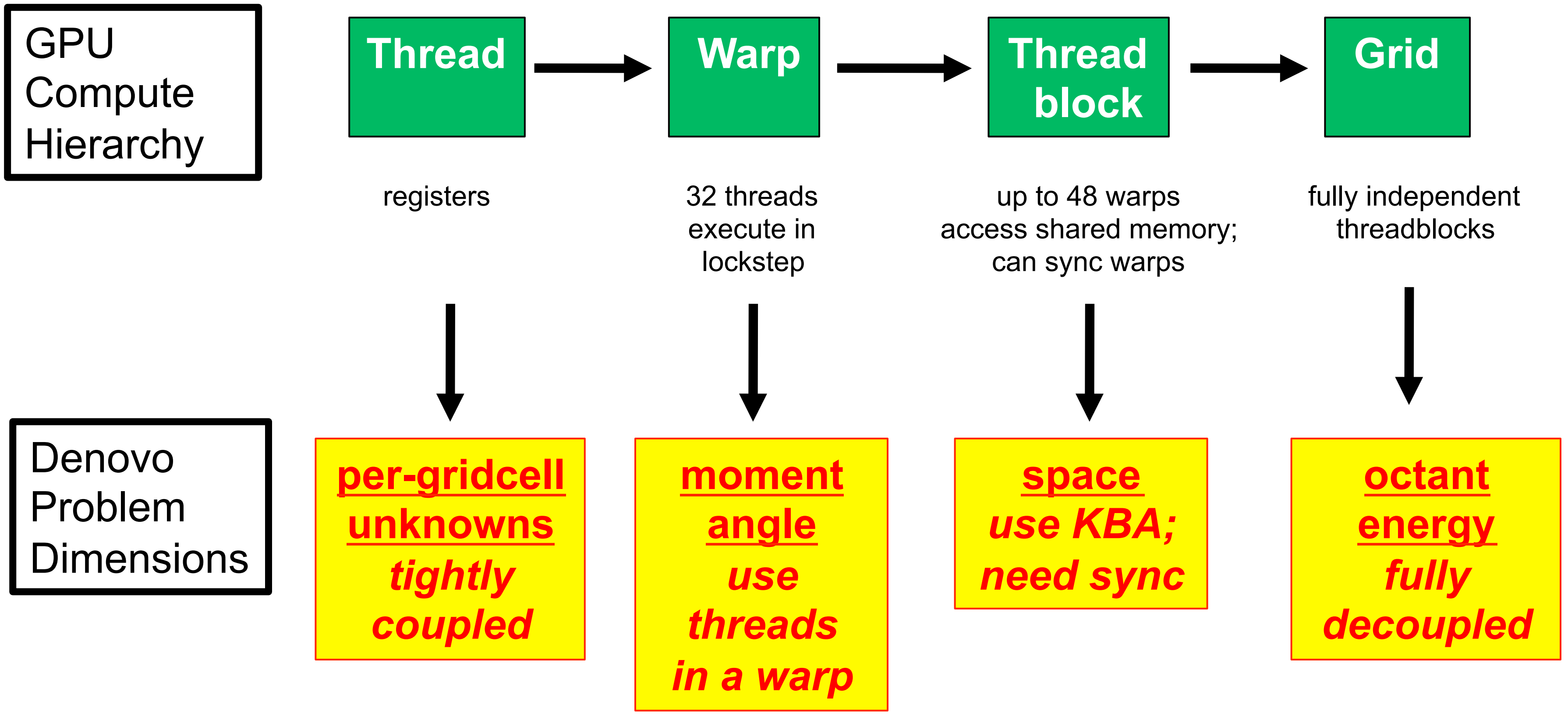
Performance

- With this paralleization scheme, code performed at only about 1% of peak flop rate, much lower than predicted by the performance model
- Reason: excessive use of registers caused spillage to main memory, thus poor performance
- Needed to find more/better axes of parallelism

4. Parallelism in Angle, Moment

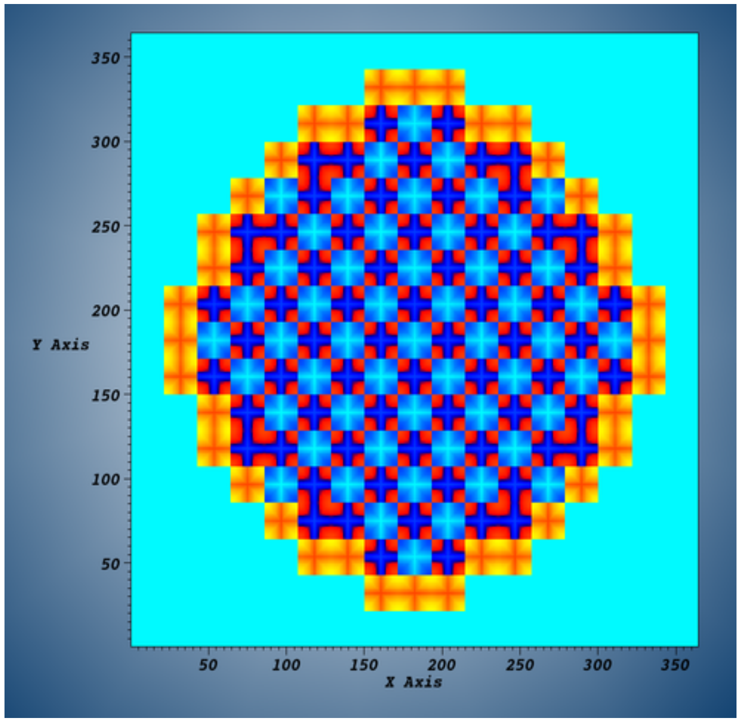
- A new strategy to parallelize the moment/angle axes at the gridcell level – map these axes to CUDA threads in-warp.
- Small dense matrix-vector products are perfect for thread parallelism – store vector in shared memory, relieve the register pressure.
- The two small matrices are the same across all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

Summary of Mapping of Dimensions


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

Results: Test Problem

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

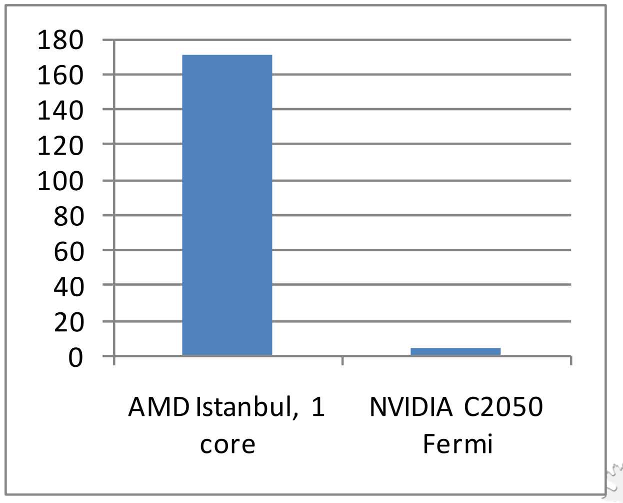


Results: Sweep GPU Performance

- 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



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.

Conclusions: Lessons Learned

1. Major code restructurings were required – this is required regardless of the parallel API used.
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 may be different.
3. Isolating CUDA-specific constructs in one place in the code is good defensive programming to prepare for programming models that may change

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.

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.
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

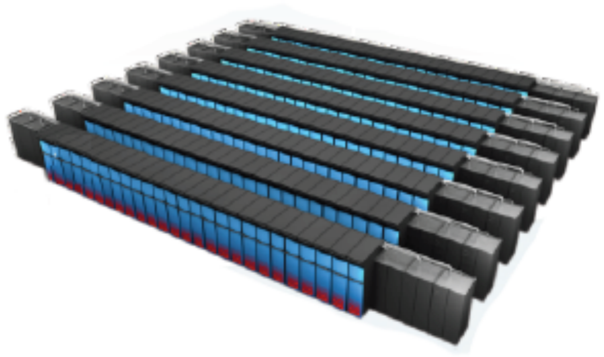
- Denovo development team: Tom Evans, Greg Davidson, Josh Jarrell, Chris Baker
- Cray: Kevin Thomas
- NVIDIA: John Roberts, Cyril Zeller, Paulius Micikevicius
- OLCF compute resources: JaguarPF, Yona, Lens

Supplementary Slides

Titan Tier-1 Readiness Apps

The OLCF has selected 6 strategic science applications for an early readiness port to Titan

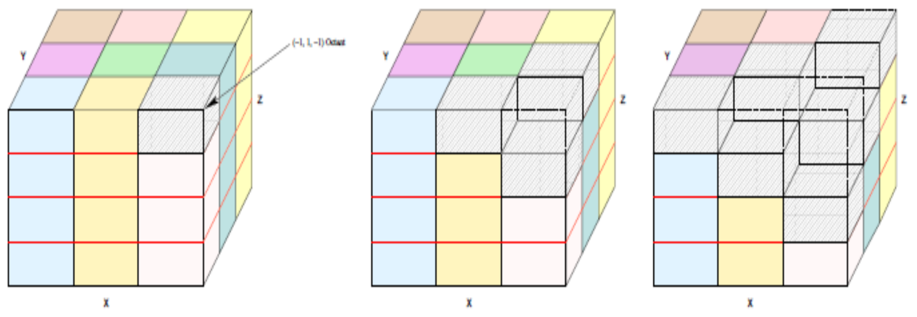
This talk concerns one of these applications: "Denovo"

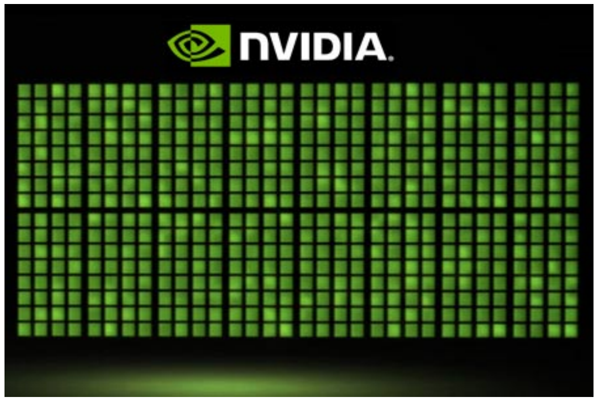


App	Science Area
LSMS	Materials
PFLOTRAN	Earth Sciences
CAM-HOMME	Climate
S3D	Combustion
LAMMPS	Biosciences
Denovo	Nuclear Energy

Algorithm Stress Points

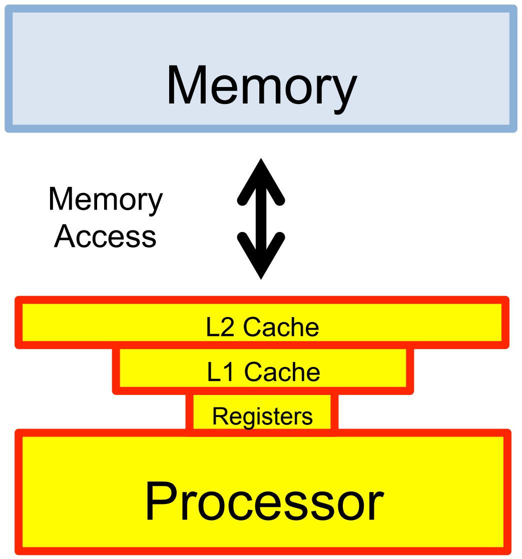
1. Wavefront startup time when not all processors are active yet.
2. Communication latency for many small face messages.
3. Structuring the algorithm to get good performance at the gridcell level





Optimizing for Memory Traffic

- Traditionally, algorithm designers optimize algorithms to reduce the (floating point) operation count.
- The more appropriate metric going forward is memory accesses (register or cache misses). Memory access is the bottleneck.
- Guiding principle: reduce memory traffic by doing as much work as possible with data values that are loaded into the CPU from memory.



Example: array copy:

BAD:

```
for ( i=0; i<N; ++i ) a[i] = b[i];
for ( i=0; i<N; ++i ) c[i] = b[i];
```

2N loads, 2N stores

GOOD:

```
for ( i=0; i<N; ++i ) {
    a[i] = b[i];
    c[i] = b[i];
}
```

N loads, 2N stores

Potential Axes of Parallelism

1. Space: X, Y, Z
2. Unknowns per gridcell, depending on discretization (4)
3. Energy group
4. Angle
5. Moment
6. Octant

Further Work

- Analysis of code performance with help from NVIDIA revealed that more thread parallelism is needed. To address this, a more efficient implementation of spatial thread parallelism is being implemented.
- Actual use cases of Denovo sometimes involve only one moment and one unknown/gridcell, leaving the GPU warp underpopulated.   To address this, a new variant is being developed which applies in-warp parallelism to the energy and spatial axes, dealing with the sync issues for spatial parallelism.