Isosurface Extraction and View-Dependent Filtering from Time-Varying Fields Using Persistent Time-Octree (PTOT)

Cong Wang and Yi-Jen Chiang
Polytechnic Institute of New York University, NY, USA

Isosurface Extraction and View-Dependent Filtering of Time-Varying Fields

- Problem instance:
  - Volume dataset with a time-varying field
- Want to do query:
  - Given an isovalue q and a time step t, extract and display all the points (a surface) whose scalar values at time step t are the isovalue q
- One of the most important and widely used techniques for volume visualization
- Large datasets pose a big challenge; want to do it efficiently
- One direction: View-dependent filtering:
  - Only extract the visible portions of an (opaque) isosurface

Motivation

Extraction + Filtering: 2 parts:

- Extraction: Search for active cells --- query in value domain
  - Interval tree [Cignoni et al 97.]
    - Achieves optimal search time, no space domain information
- Filtering: query in space domain
  - BONO [Wilhelms et al 92.]
    - Search on value domain is not optimal

We want to combine the best of the two worlds

- POT [Shi et al 06.]: optimal search & supports filtering, but only for steady-state data
- 4D POT [Shi et al 06.]: for time-varying data, but search is NOT output-sensitive, NO view-dependent filtering

Our New Algorithm

- Persistent Time-Octree (PTOT) : Combining the best of two worlds for time-varying data
  - Extraction: Optimal searching time (not known before)
  - Filtering: Supports view-dependent filtering
- Also, we develop filtering method in out-of-core setting: reduce both # of I/Os & disk seek time
  achieve huge speed up for large datasets

Previous Work

- For steady-state data
  - Many in-core methods, e.g.
    Marching Cubes [Lorensen et al 87], BONO [Wilhelms et al 92] , NOISE [Livnat et al 96] , Interval tree [Cignoni et al 97], QDV [Stockinger et al 05]
  - Out-of-core approaches
    - [Chiang et al. 97], [Chiang et al. 98], [Bajaj et al. 99], [Chiang et al. 01]
- For time-varying fields
  - In-core: THI tree [Shen 98]
  - Out-of-core: [Sutton et al. 00], [Chiang 03], [Gregorski et al. 04], [Waters et al. 06]
- View-dependent filtering techniques
  - [Livnat et al. 98], [Gao et al. 03], [Pesco et al. 04]

Previous Work (cont.)

- Combining value-domain searching and space-domain filtering
  - POT [Shi et al 06.]: optimal search & supports filtering, but only for steady-state data
  - 4D POT [Shi et al 06.]: for time-varying data, but search is NOT output-sensitive, NO view-dependent filtering

Our New Algorithm

- Persistent Time-Octree (PTOT) : Combining the best of two worlds for time-varying data
  - Extraction: Optimal searching time (not known before)
  - Filtering: Supports view-dependent filtering
- Also, we develop filtering method in out-of-core setting: reduce both # of I/Os & disk seek time
  achieve huge speed up for large datasets

Overview of Our Approach

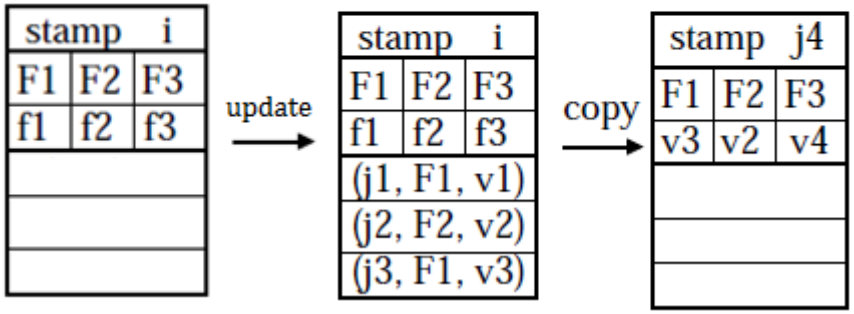
- Building Block: Persistent Data Structure [Driscoll et al 89.]
- Time-Octree as base data structure
- Build Persistent Time-Octree (PTOT) by Line Sweep Process
- Compact tree representation
- View-Dependent Filtering integrated with Implicit Occluders [Pesco et al 04.]
- Using CUDA to perform efficient hardware Marching Cubes

Building Block: Persistent Data Structure [Driscoll et al 89.]

- A dynamic tree supports updates such as insert/delete. Each update creates a new version
- An ordinary dynamic tree keeps only the latest version. It is called an ephemeral tree
- A persistent tree keeps all versions from updates
  - m structural changes take O(m) additional space
- Any version i of a persistent tree can be queried
  - Asymptotically the same time as querying version i of the ephemeral tree

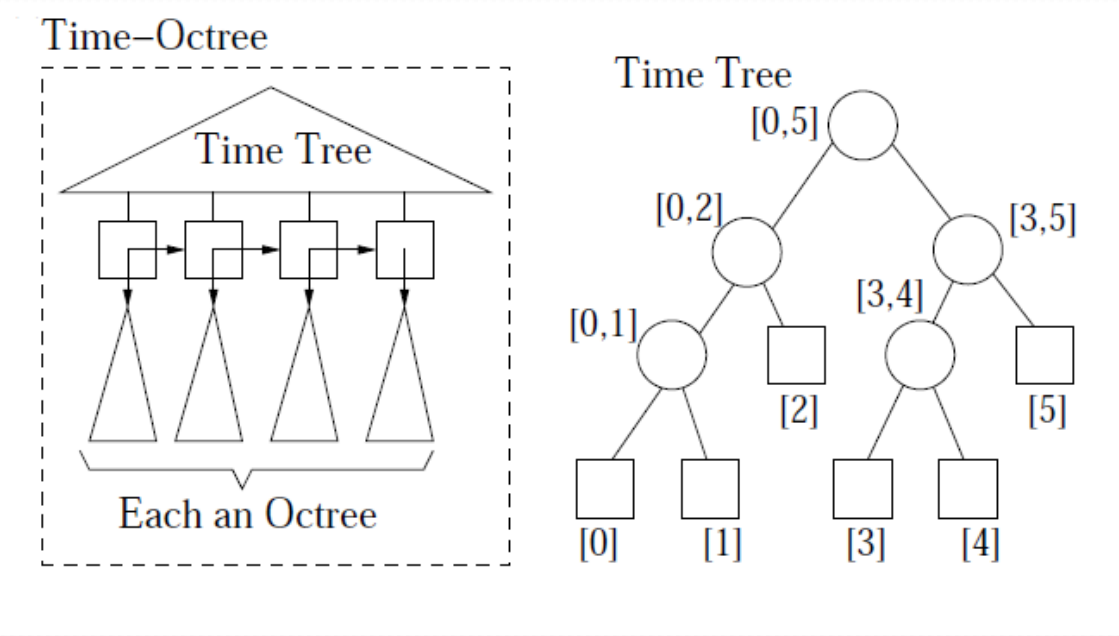
Persistent Tree: Node Copying

Stamp	Update
j1:	x.F1 := v1
j2:	x.F2 := v2
j3:	x.F1 := v3
j4:	x.F3 := v4



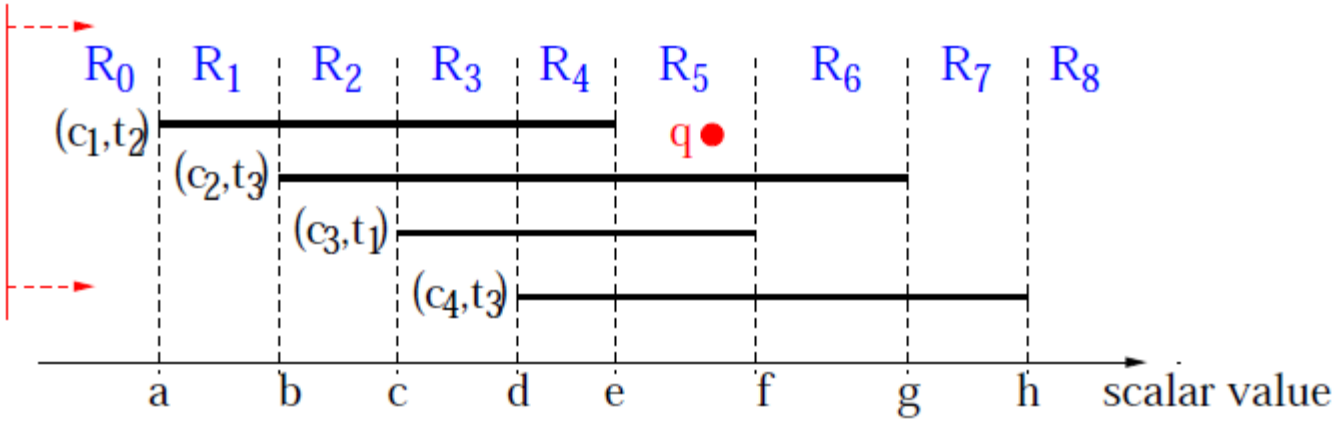
- Each node has constant number of extra fields to record some future updates
- When all extra fields in a node are used up, the next update incurs a node-copying: the node is copied to a new node with latest field values.

Base Data Structure: Time-Octree



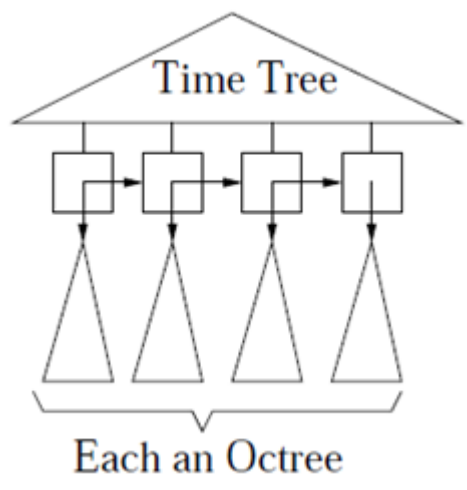
- The top part is a fully balanced binary tree called time tree, which partitions the time domain
- The bottom part consists of a collection of octrees, one octree per time step, which partitions the space domain
- In time tree, we maintain a pointer from a leaf to the next leaf whose octree is also not empty

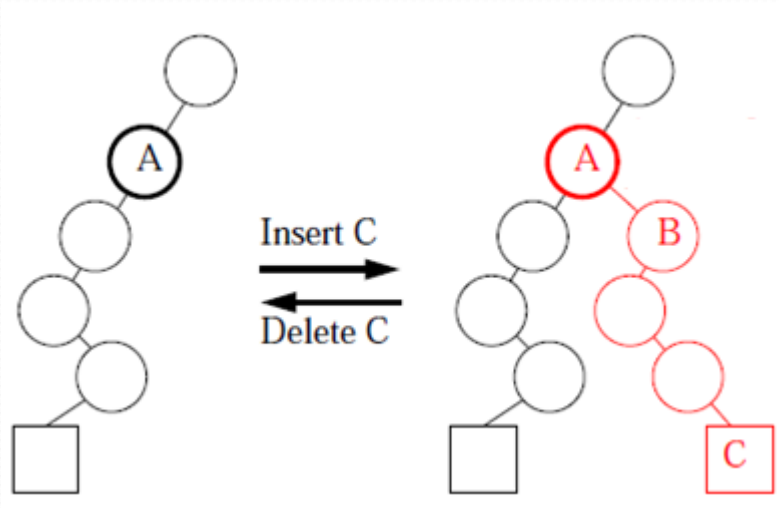
Line Sweep Process: PTOT Construction



- For each cell c at time step t, we produce an Interval Ic,t = [min,max] for its min, max scalar values.
- If an interval Ic,t contains isovalue q => the cell is an active cell
- Sort all interval endpoints and sweep from $-\infty$ to ∞
- Sweeping event: each creates a version
  - Encountering the left endpoint of an interval Ic,t: insert Ic,t to the time-octree
  - Encountering the right endpoint of an interval Ic,t: delete Ic,t from the time-octree
- Preprocessing phase; no more update to the PTOT from now on

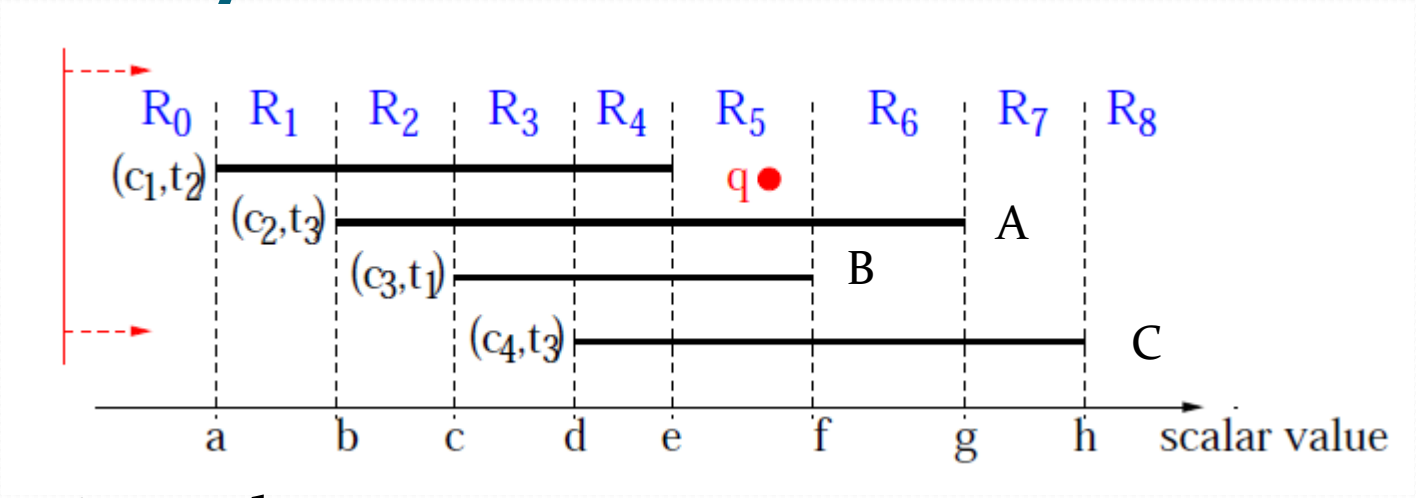
Insert/delete on Time-Octree





- To insert Ic,t , first use time step t to locate the leaf t in the time tree.
- Secondly, use cell c to locate the leaf in the octree.
- Create a leaf node, and grow missing nodes on the fly
- Deleting Ic,t is similar (a reverse op)

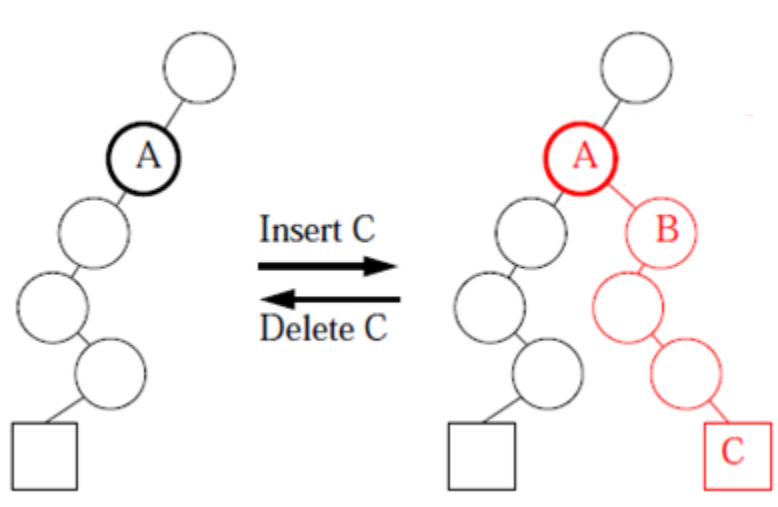
Query on PTOT



Run-time phase:

- Each range Ri corresponds to a version i of the time-octree
- If we ignore time step, version i of the time-octree contains exactly those active cells---just report everything
- These cells are further grouped into octrees according to their time steps--- for time step t, just report everything in the octree of t.
  - Optimal search time * Octree: contains space info for filtering

Compact Tree Representation

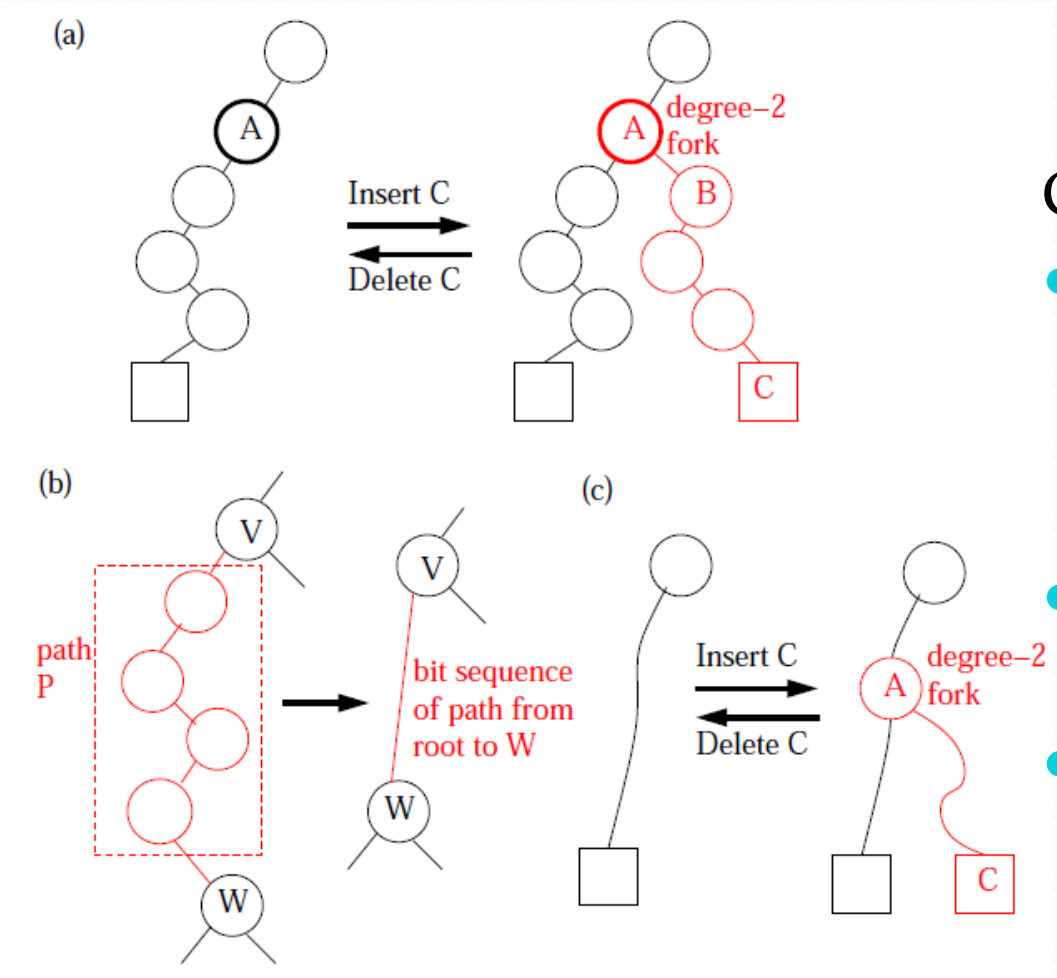


Problem in normal tree representation:

- Insert/delete an interval may introduce O(logT +logN)=O(log(NT)) structural changes

- Recall : for a persistent tree , m structural changes need O(m) additional space [Driscoll et al. 89]
- In our line sweep, there are O(NT) insert/delete ops, each with O(log (NT)) structural changes
  → O((NT) log(NT)) structural changes, and hence O((NT)log (NT) space for our PTOT, not linear

Compact Tree Representation (cont.)



Compact Tree:

- Any internal node u other than the root is removed if u has only one child.
- A bit sequence is used to store path information
- Insert/delete an interval has O(1) structural changes → linear space

Out-of-Core Scheme

- Partition the space domain into meta-cells, each with k*k*k cells
- Use PTOT to index the meta-cells. Choose k so that # of meta-cells is not too big, and the PTOT can entirely fit in main memory
- During Query:
  * Keep PTOT in main memory for searching & filtering
  * Keep meta-cells on disk; read only the needed meta-dells to main memory (I/O + extraction)

View-Dependent Filtering

- Recall: for query (q, t), intervals in the version for q are active, and they are further grouped into octrees. The octree of t is called the active octree.
- Naïve approach
  - Traverse the active octree hierarchically in the front-to-back visibility order, use isosurface portions already rendered as occluders
    - Initial occluder is too small
    - Visibility order is often highly non-sequential in terms of the storage order on disk → large disk seek time
- Our approach
  - Integrate with Implicit Occluders [Pesco et al. 04]

View-Dependent Filtering (cont.)

- Build Implicit Occluders [Pesco et al 04.] using octree skeleton, one for each time step
  - De-couples rendering isosurface and constructing occluders
  - Much larger initial occluder
- Batched I/O reads
  - Traverse front to back in visibility order (in-core)
  - Perform occlusion filtering until we accumulate L meta-cell IDs that need to be read (all using the current occluder M)
  - Sort these L IDs, read the meta-cells in this sorted order one at a time → sequential I/Os, small disk seek time
- Grow the occluder by rendering isosurface to z-buffer: The new occluder M' is used for next batch of L meta-cell IDs.

Isosurface Extraction Using CUDA

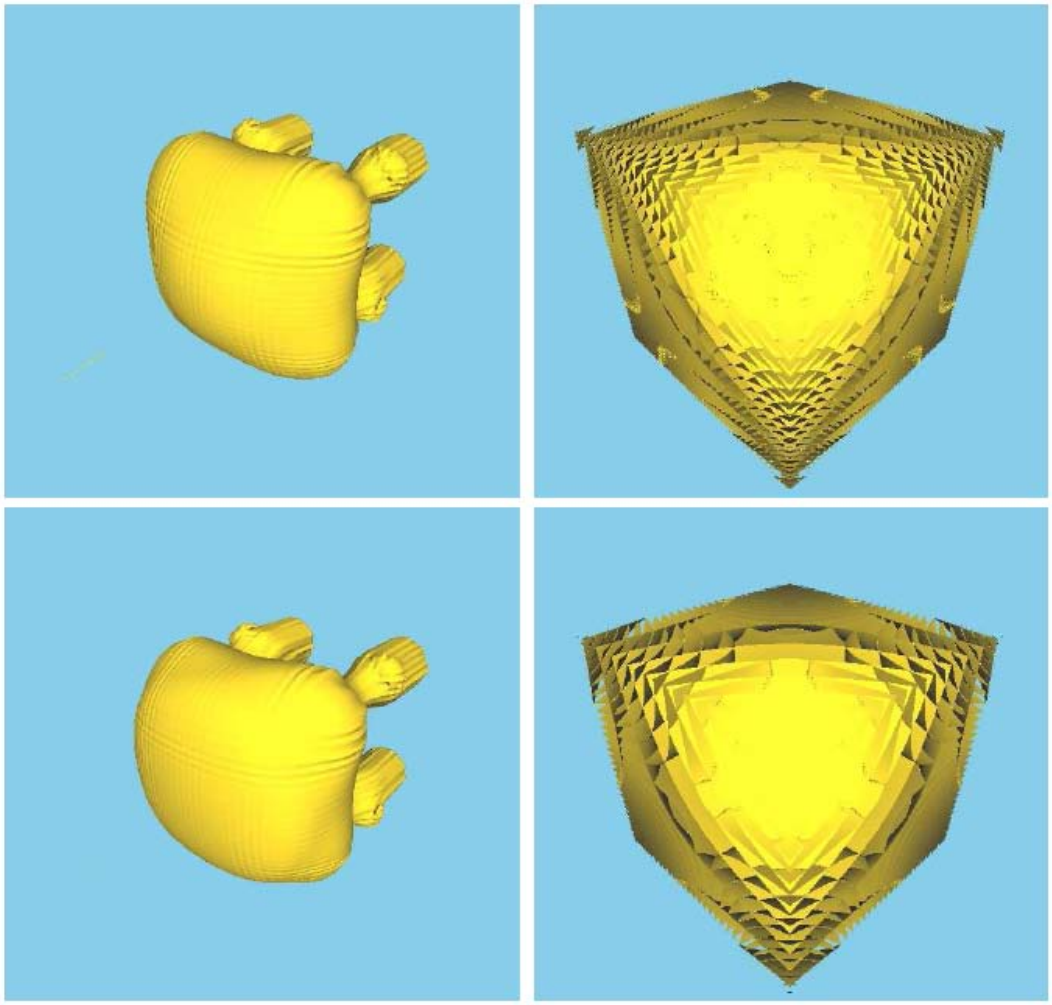
- The original CUDA code is good for sending the whole dataset at the beginning
  - Only works on small dataset
- Naively: send one meta-cell at a time
  - GPU concurrency is not fully used
- We use some technique so that we can send B meta-cells in a batch
  → increase concurrency in GPU (see paper for details)
  - 3.89s (B = 32) vs. 13.89s (B = 1) for one time step on Vort, 11M triangles on an average

Experiments

- Datasets
  - Resolution: 1024*1024*1024, 4GB each time step. Up to 48 time steps. 64GB – 192GB in size
  - RAM size: 1GB. Meta-cell size: 32*32*32
- Preprocessing:
  Meta-cell construction: 140MB footprint
  PTOT construction: up to 870MB footprint
  Overall data structure : size overhead: only 9.5%
- Runtime:
  Memory footprint: no more than 230MB

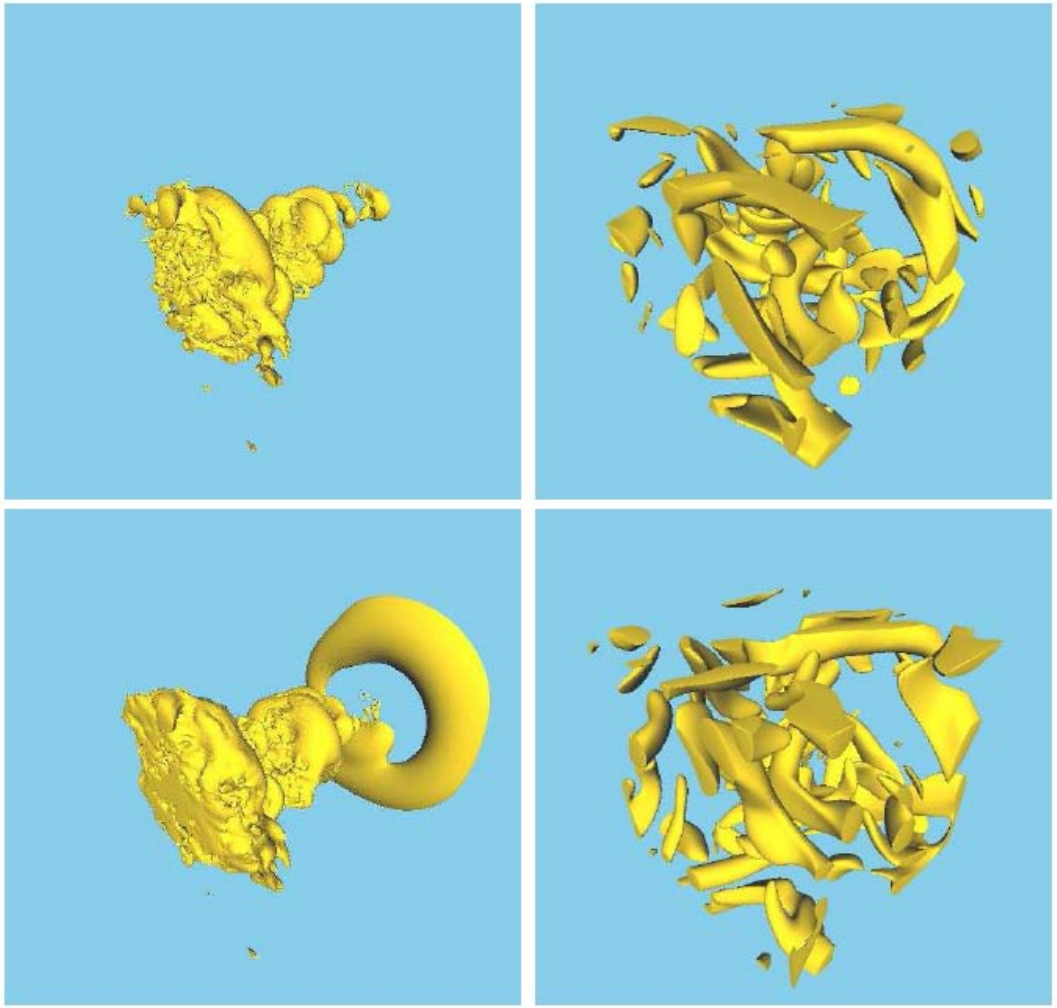
Representative Isosurfaces

Jets Syn



Representative Isosurfaces (cont.)

Turb Vort



Experiments

- Run-Time Query: View-Independent Isosurfacing
  - PTOT vs. 4D-POT [Shi et al. 06]
    - Smaller tree size
    - Faster index searching time
  - PTOT vs. THI tree [Shen 98]
    - Tree size and index searching time were worse
      - Difference was small. The dominating cost was the I/O time
    - THI reports a super-set of active meta-cells ---- large I/O penalty, much worse total time
      (e.g. Vort 48 time steps: 1839 sec vs. 1999 sec)

Experiments

- Run-Time Query: View-Dependent Filtering
  - Three methods (all using our PTOT):
    - Implicit (ours): use implicit occluders, batched I/O with L=128
    - Explicit: rendered isosurface as occluder
    - No-Occ: no occlusion filtering. All active meta-cells are sorted by IDs and read sequentially
  - Explicit:
    - Smallest # of meta-cells read and extracted
  - No-Occ:
    - Largest # of meta-cells read and extracted (since no filtering)
    - Running time could be better than Explicit!
      - sequential (sorted) disk reads are important!
  - Implicit:
    - Strikes a balance between reducing # of I/Os and reducing disk seek time
    - Always the fastest with large margin (e.g., Syn for 10 time steps: 342 sec vs. 467 sec (Explicit) vs. 865 sec (No-Occ))

Implicit vs. Explicit (Video)

Implicit Explicit

Conclusions

- PTOT data structure achieves optimal searching for active cells in time-varying fields
- Supports view-dependent filtering
- Integrates with implicit occluders, strikes a balance between reducing the number of I/Os and reducing the disk seek time; achieves great performance

- In Figs 1—4 all spaces betweens words are gone. Correct version: http://cis.poly.edu/chiang/PTOT-vis09.pdf

Acknowledgments

- We thank Nvidia for the CUDA code
- *NSF grant CCF-0541255, NSF CAREER Grant CCF-0093373*