GPU Strategies for Unstructured Mesh Physics

Charles R. Ferenbaugh

Scientific Software Engineering (HPC-1), LANL

April 10, 2012

LA-UR-12-20351

Background: What led to this study

- Historically, unstructured mesh apps are not well represented in advanced architecture studies
- Example: Roadrunner demonstration apps, 2006 – 2008
  - Five typical LANL physics apps were re-implemented for optimal performance on RR
  - All apps used either structured meshes or particles (or both)
  - None used unstructured meshes
- I work on FLAG, an unstructured mesh code, so I found this a bit disturbing
  - FLAG is the ASC/LANL ALE rad-hydro code

Background: My unstructured mesh studies

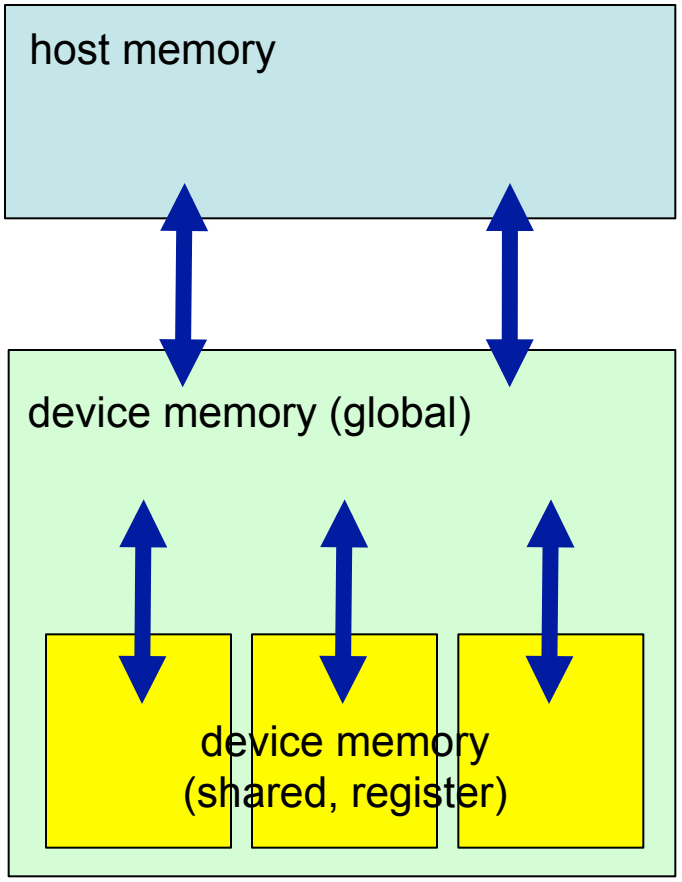
- 2009 – 2010: I did a small unstructured mesh study on the Cell
  - Extracted a small kernel from FLAG, developed a Cell-optimized version
  - Tested Cell-optimized code against CPU code; Cell code was about 5x faster
  - Presented results at NECDC 2010
- 2010 – 2011: I continued the study, moving from the Cell to GPUs
  - Presenting results today

Strategy for GPU study

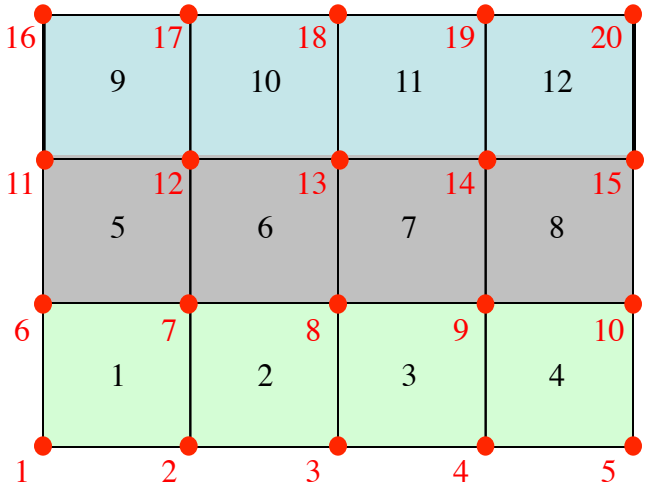
- Consider only a subset of FLAG
  - Basic hydro only (no radiation, ALE, HE burn, …)
  - Serial, 2D cylindrical geometry only
  - Just enough to support a few standard test problems (Sedov, Noh, TP37C shock tube)
- Isolate the artificial viscosity (AV) routines in a kernel
  - Remove all calls from AV to the rest of FLAG
  - FLAG still runs the rest of the physics
  - For the class of problems I'm looking at, AV is the performance hotspot
- Re-implement AV kernel on the GPU for improved performance

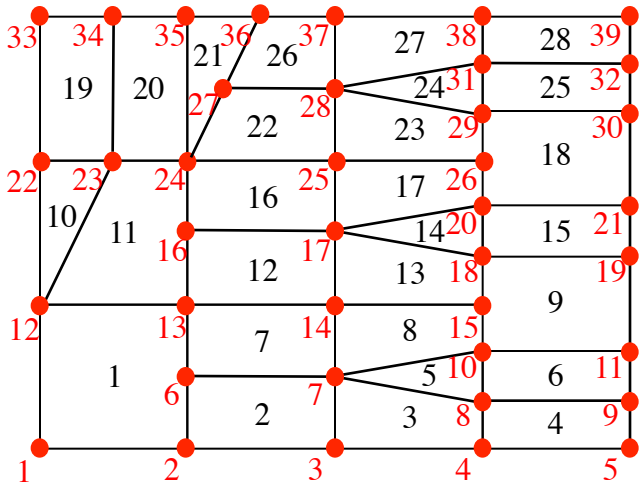
Why data chunking is essential for good performance...

- Data must move between host and device
  - Compute on one chunk while moving another
- Data must move between memory spaces on device
  - Use data from local memory as much as possible
- Data must be processed in parallel on 100's of threads
  - Make chunks independent



.... and why data chunking is hard on unstructured meshes



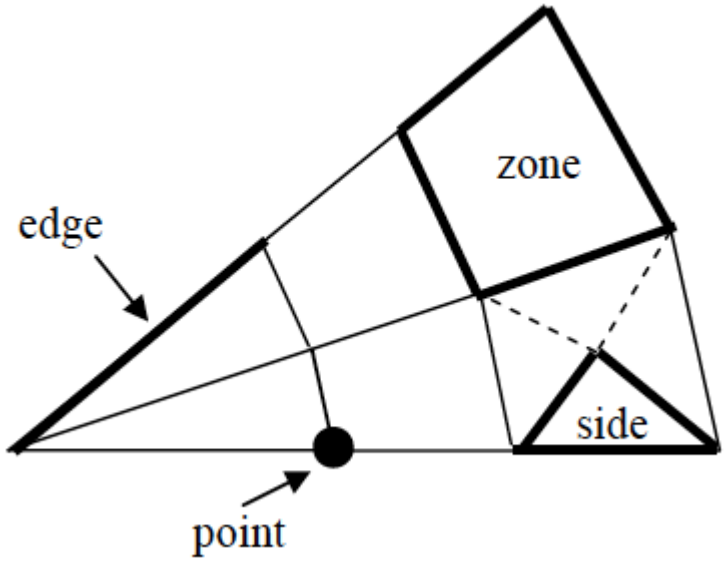


- Easy to split zones, points into contiguous chunks
- Simple arithmetic relation between points and zones

- No good way to split zones *and* points into chunks
- No pattern relating points and zones; must use explicit connectivity array

Overview of FLAG data structures

- Variables may be defined on edges, points, zones, or sides
- Connectivity arrays from sides to points/edges/zones are stored
- Computations are often done by loops over sides
  - Connectivity arrays are used to access variables over points/edges/zones



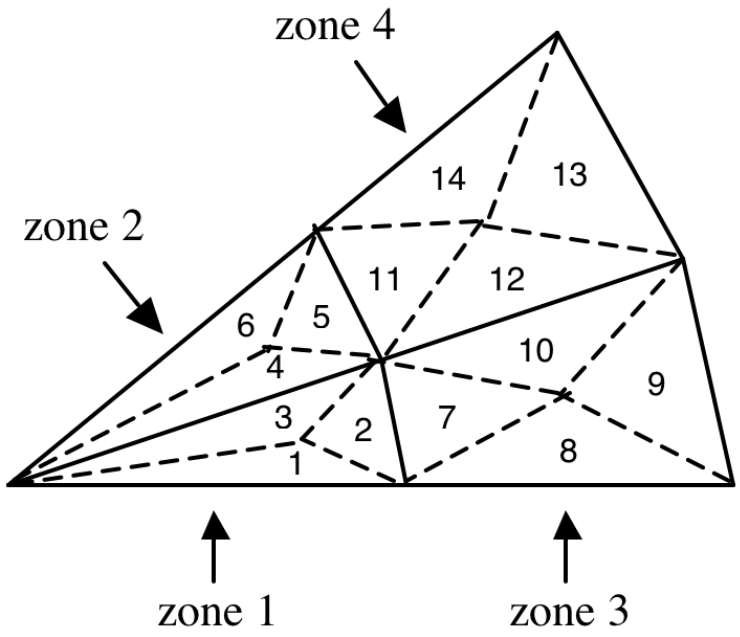
The $64K question: How to do data chunking?

When computing on a chunk of sides, how to make sure the corresponding variables on zones, points, ... , are available?

For zones, it's easy:

- Enumerate zones
- Enumerate sides of each zone
- This way, zone and side chunks always correspond

Must do something different for points and edges...

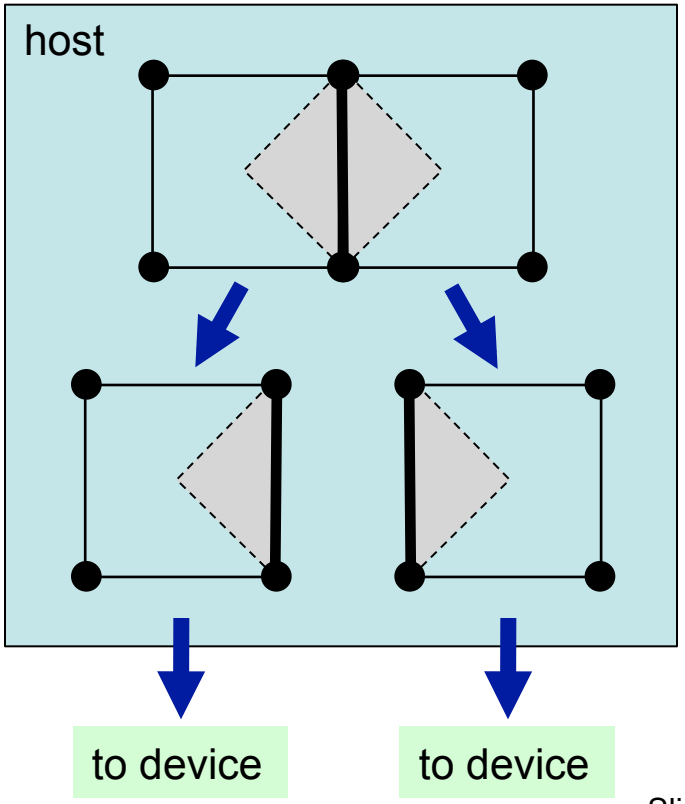


Strategy 1: Coalesce data on host

One possible data chunking strategy for points and edges: coalesce data on host before shipping to device

(This was the strategy used on the Cell...)

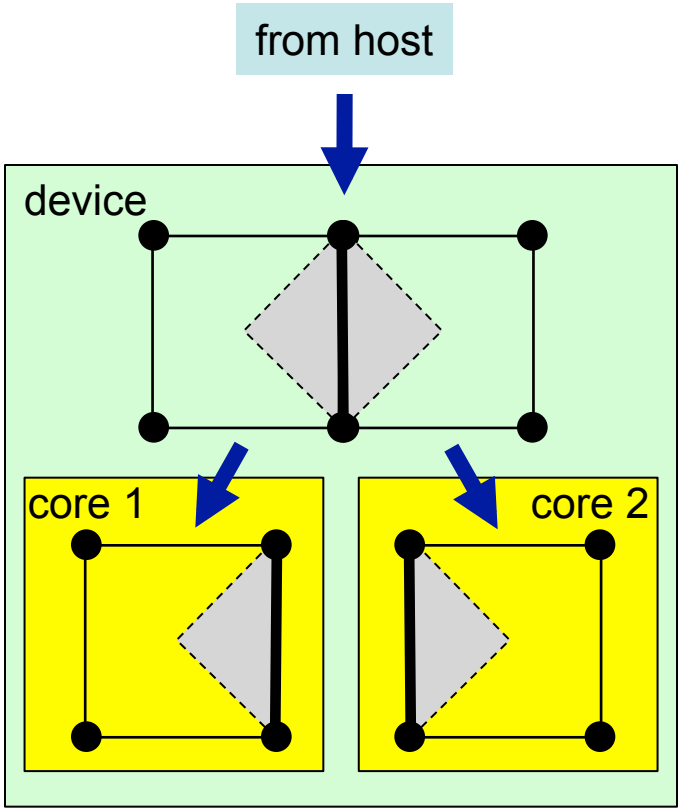
+ Every core has its own local, contiguous copy of the data
− Host does all gather/scatter work, serially
− Multiple copies of data => more data traffic to/from device



Strategy 2: Coalesce data on device

To avoid the bottleneck on the host while scattering/ gathering, what if we do the coalescing on the device?

+ Every core has its own local, contiguous copy of the data
+ Device does gather/scatter work in parallel
– Must move all data from host to device before chunking starts; can't overlap with compute



Strategy 2, modified: Coalesce data on device

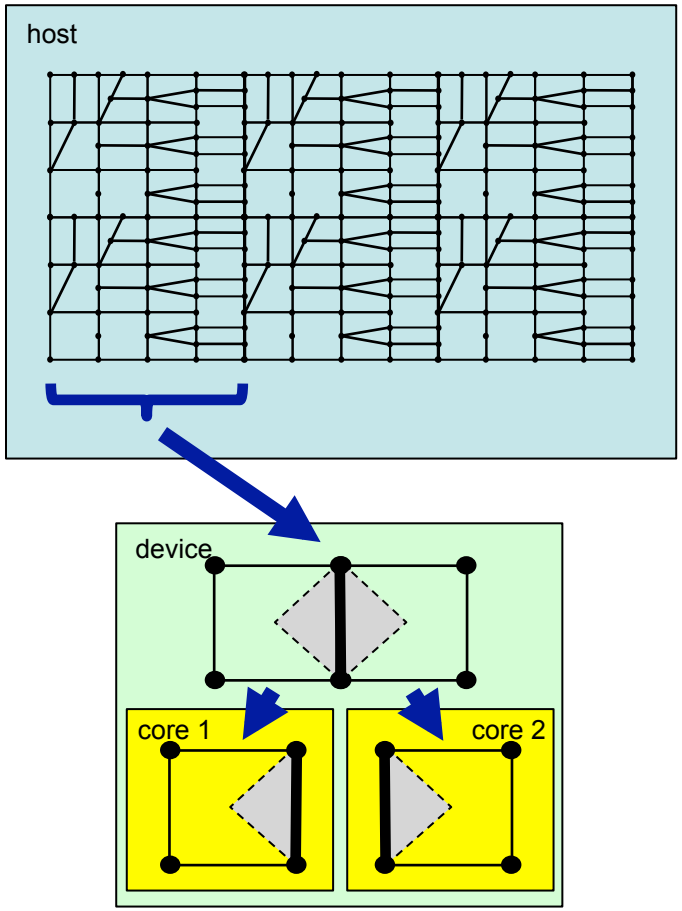
Use big chunks for host-device transfer, smaller chunks on device

- Big chunks have some overlap
- Overlap areas must stay on device for multiple compute cycles

+ Every core has its own local, contiguous copy of the data

+ Device does gather/scatter work in parallel

– Host/device data movement still isn't optimal (but it's better than before)



Strategy 3: Take advantage of GPU cache

Newer NVIDIA GPUs have L1, L2 hardware cache

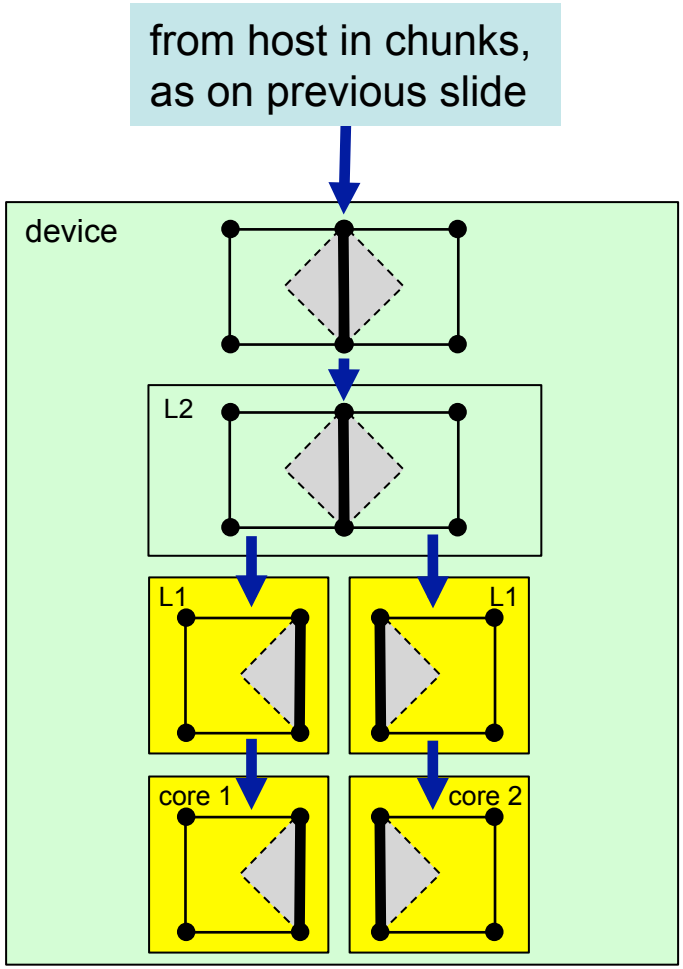
- Use cache instead of explicitly coalescing data

+ Every core has its own local copy of the data

+ No explicit scatter work needed

– Gather needs special handling, since caches are not coherent

– Host/device data movement has same problems as in strategy 2



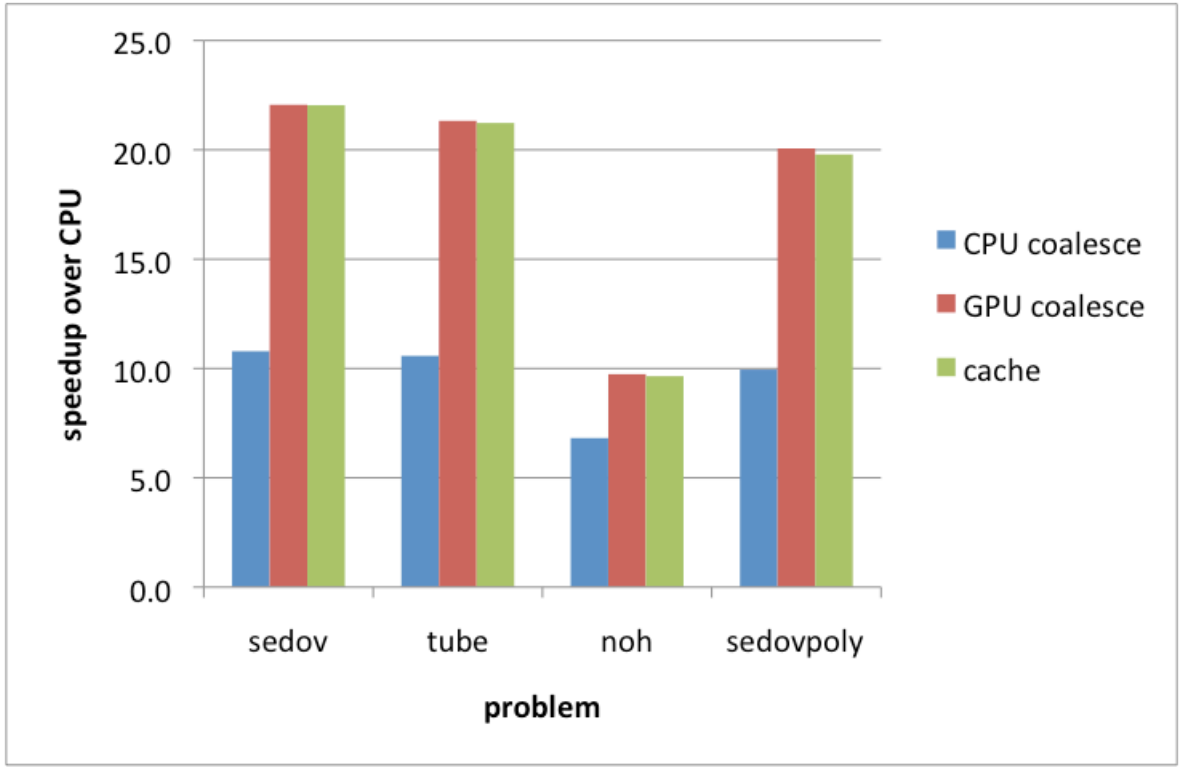
Testing details

- Tests were run on LANL Darwin cluster
  - CPU: AMD Opteron 6168, 1.90 GHz
  - GPU: NVIDIA Tesla C2050 (Fermi)
- All timings are for the AV kernel only
  - Includes data movement time between host and device
- Test cases are adapted from standard FLAG tests:

test name	# zones	# sides	# timesteps	mesh type
sedov	32400	129600	2065	square, all quad zones
tube	36864	147456	2123	rectangular, all quad zones
noh	6000	23940	27265	radial, tri and quad zones
sedovpoly	22801	136202	3127	square, mostly hexagonal zones

Timing results for chunking strategies

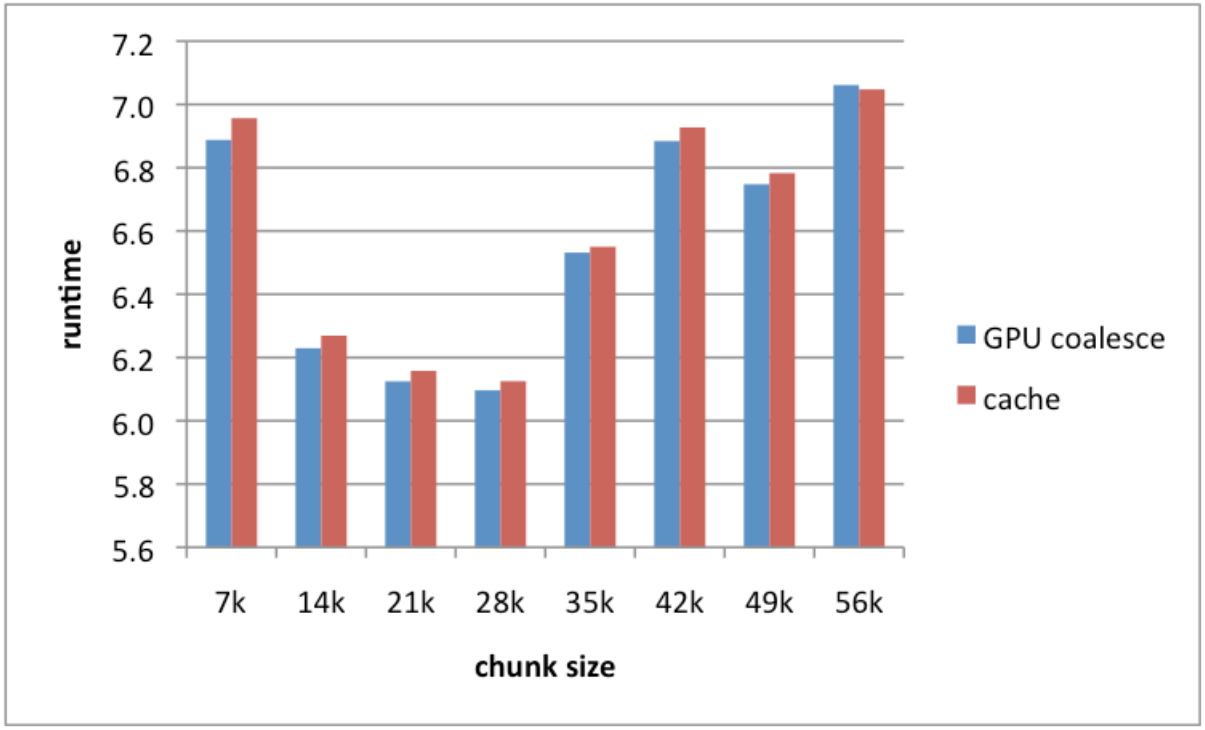
Runs with all three strategies, plus serial CPU as baseline



Timing results for different “big chunk” sizes

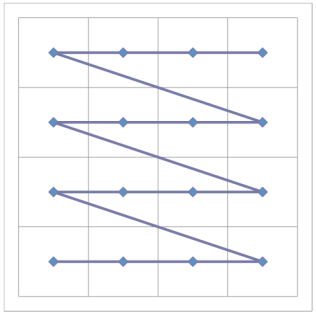
Runs for Sedov problem only

7K = max concurrent threads (14 SMs x 512 threads/SM)

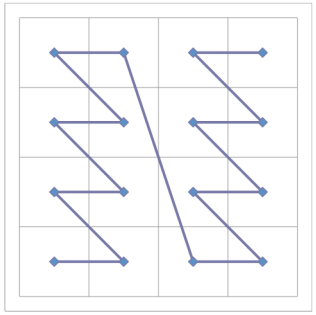


Does numbering of points make a difference?

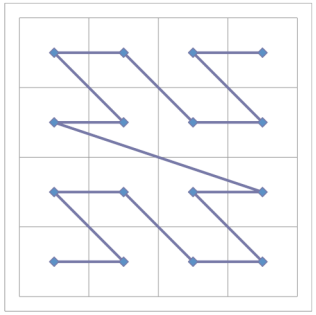
- I experimented with different orderings of points and zones, to try to optimize memory access



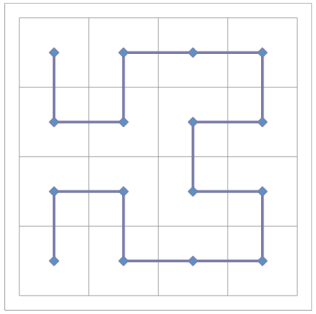
simple row/ column



column blocks (size 2)



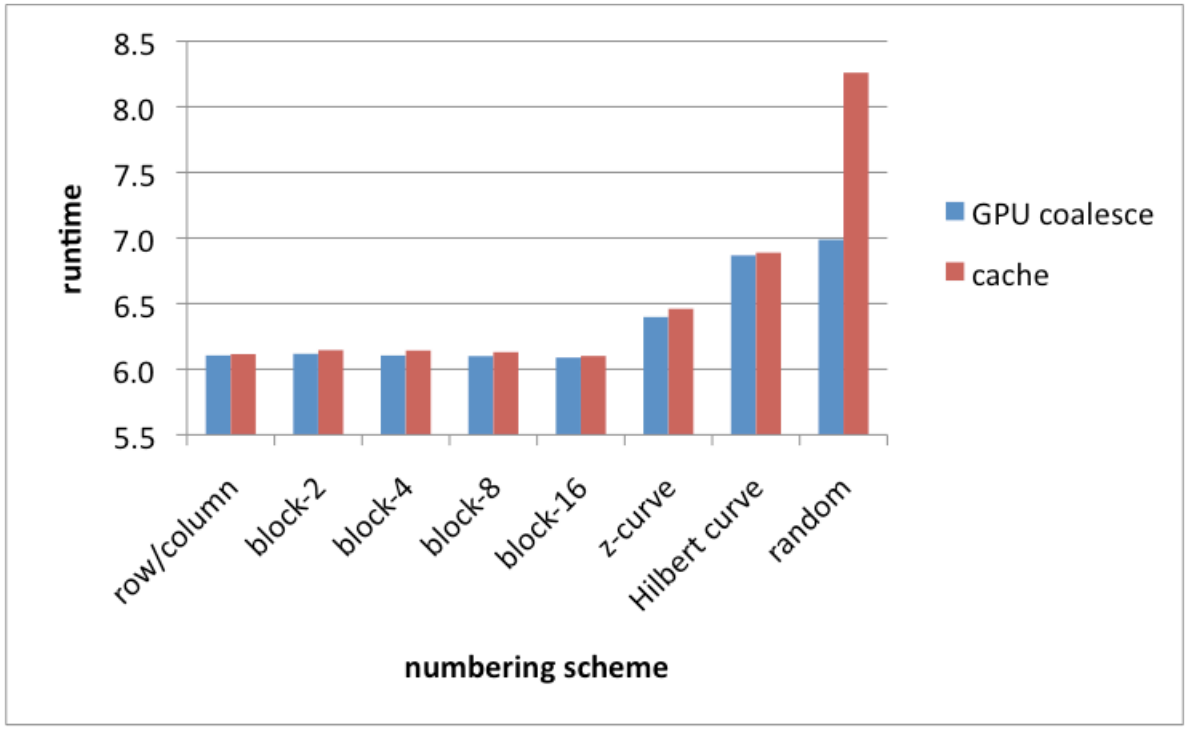
Z-curve



Hilbert curve

Timing results for different numbering schemes

Runs for Sedov problem only (square mesh)



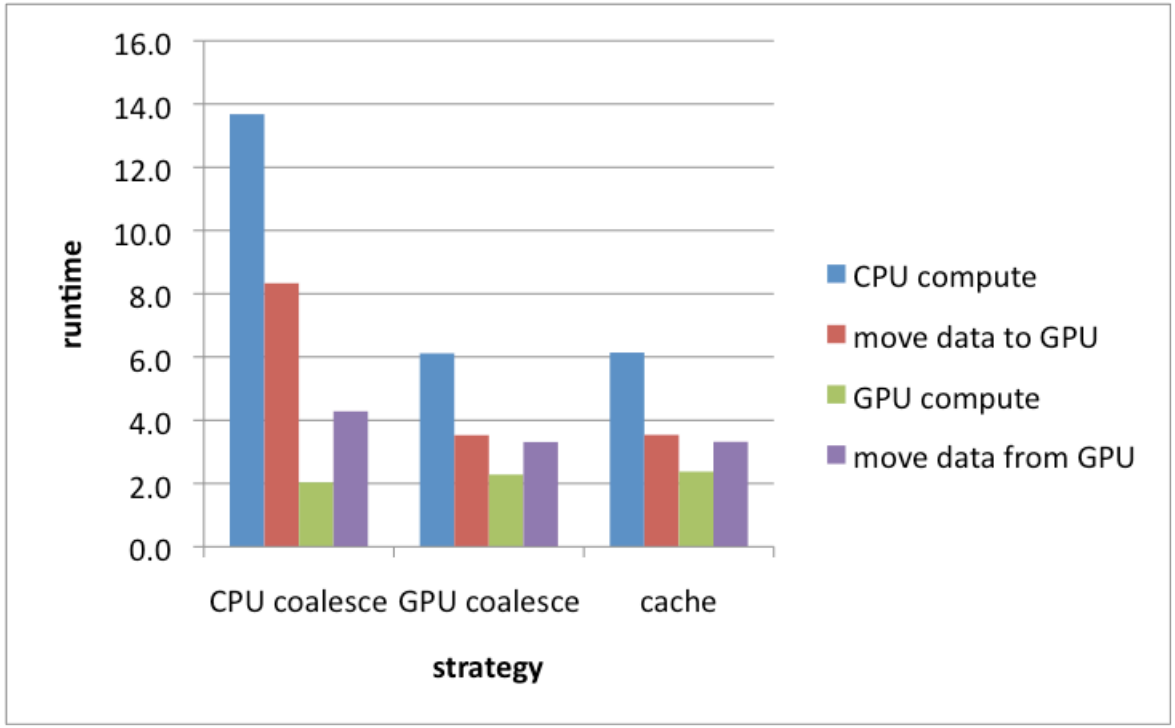
What is the limiting factor for run times?

- GPU implementations have four distinct steps:
  - host compute (mostly data reordering)
  - device compute
  - data movement from host to device
  - data movement from device to host

  On newer NVIDIA GPUs, all four of these overlap
- To find out which was taking the most time, I added synchronization points to remove all overlapping

Timing results with non-overlapped steps

Runs for Sedov problem only



Possible improvements

- Replace naive “big chunk” algorithm with something more sophisticated
- Change from GPU-offload approach to all-on-GPU
  - Would reduce data movement between host and device
  - Would allow for significant changes to data structures
  - Would be hard to do for current version of FLAG (~480K lines)
- Use future hardware/software improvements
  - Software cache for host-to-device transfers? (a la Roadrunner)
  - Integrated host-device memory space

The next step...

- I've created an unstructured mesh mini-app PENNANT
  - Contains a few basic algorithms from FLAG hydro
  - Restricted to serial runs, 2-D cylindrical geometry
  - Can run a few test problems, including a Noh problem on a polygonal mesh
- I'll use PENNANT for further GPU studies, this year
- It's available for use in other co-design efforts
  - Initial CPU version released at LANL
  - Will be used by CCS-7 mini-app project, possibly others...
  - Will be released open-source in the near future