Cell-Based Adaptive Mesh Refinement (AMR) Implemented on the Graphics Processing Unit (GPU)

Presented at:

University of New Mexico

Jan 19th, 2012

Presenters: David Nicholaeff, Bob Robey

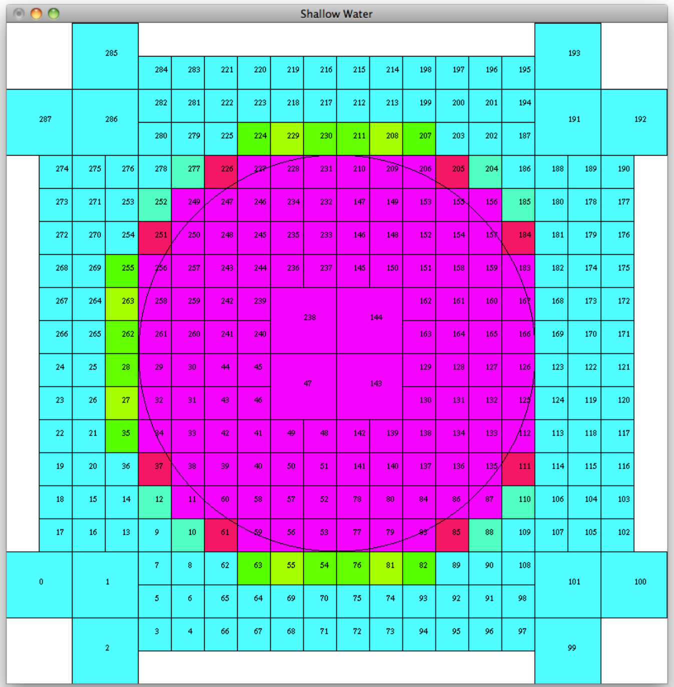
Co-Authors: Neal Davis, Dennis Trujillo

Los Alamos National Laboratory

LA-UR-11-05345

AMR GPU Summer Challenge

- Develop a Cell-Based AMR shallow-water code for the GPU using OpenCL
- Mini-App to demo technique
  - Finite difference on AMR
  - Data Order
    - Partitioning
    - Locality
  - Neighbor Search
  - Consistency Improvements



Background -- Cell-Based AMR

- Los Alamos has been a leading developer of cell-based AMR
  - Other AMR types are structured (i.e. Berger, Collela, Oliphant method)
- Advantages of Cell-Based AMR
  - Most flexible in terms of adapting the grid to the features of interest in the computational domain
  - Fewer Cells —> less memory
  - Better symmetry, especially spherical
  - Arrays become *1*-D —> better load balancing

Cell-Based AMR Rules

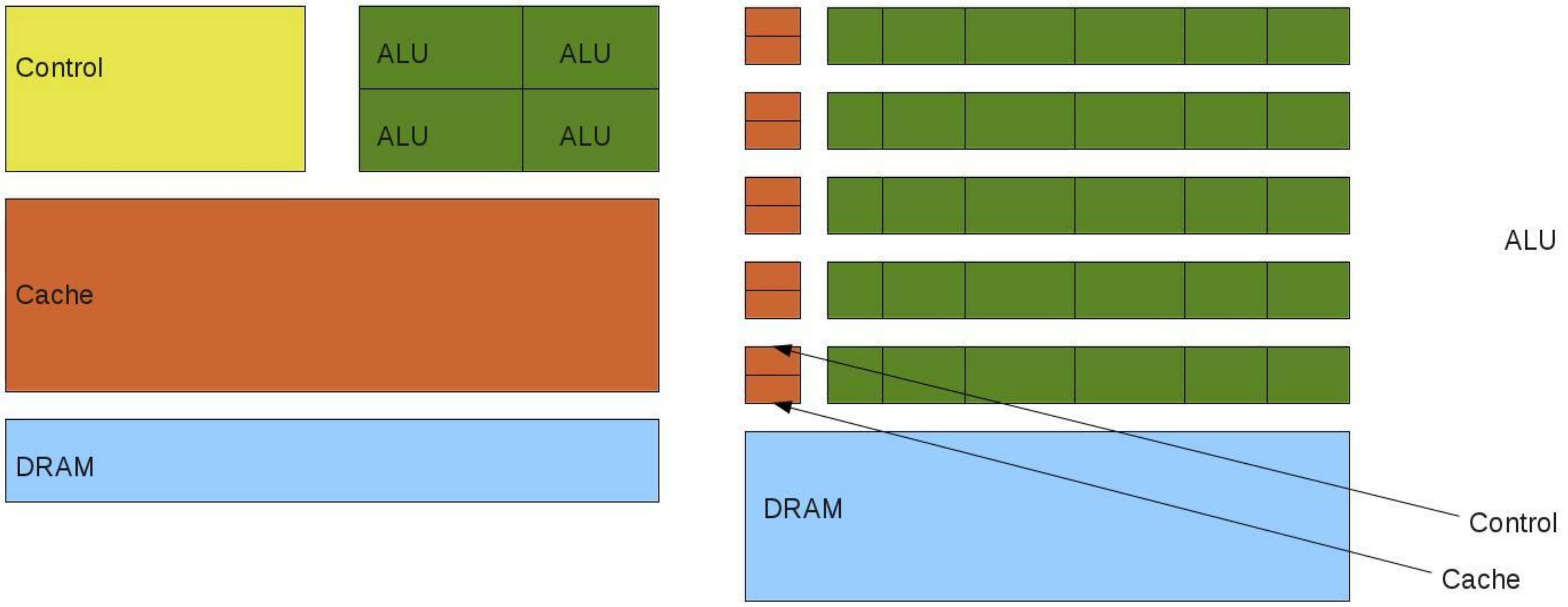
- Refinement factor limited to increase of two between adjacent cells
  - For physics and programming reasons
- Coarsening is limited to same group of four cells
- Need refinement spread-out one cell from shocks and material discontinuities
- Neighbor index arrays used in place of +*1*/-*1* or +*n*/-*n*
- Need to sweep through all cells and avoid doing operations level-by-level

OpenCL on Heterogeneous Platforms

- OpenCL -- Open Computing Language
- Allows for execution of code over multiple hardware types
- Based on C99, allows for C type program structure
- Code developed in kernels executed as workgroup over number of threads

The OpenCL logo is a trademark of Apple

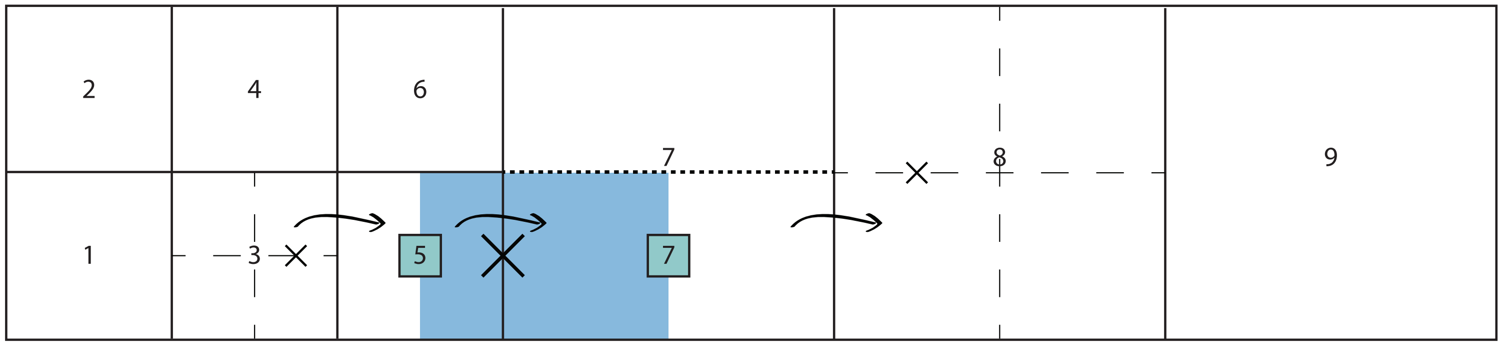
Hardware Overview



- CPU has more cache, fewer ALUs, and lower memory bandwidth
- GPU has more Arithmetic Logic Units (ALUs) and less cache

Finite Difference on AMR

- Originally not part of the effort, but realized that mass was not being conserved
- Must ensure that same flux is calculated from cells on both sides of interface



Finite Difference Equations

Adaptive Mesh Half-Timestep:

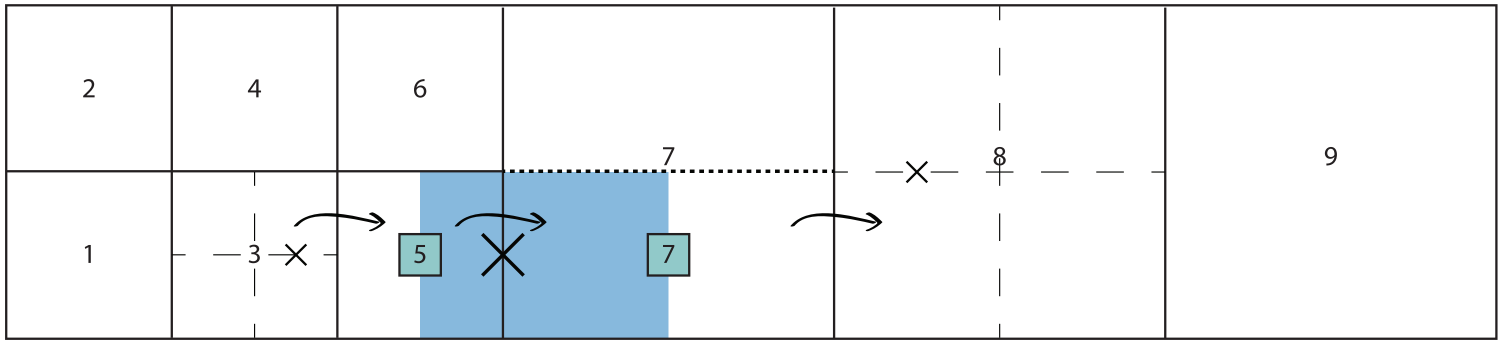
$$U^{n+1/2}_{i+1/2,\ j} = \frac{r_i U^n_{i+1,\ j} + r_{i+1} U^n_{i,\ j}}{r_{i+1} + r_i} - \Delta t \left(\frac{F^n_{i+1,\ j} A_{i+1} a_{i+1} - F^n_{i,\ j} A_i a_i}{V_{i+1} v_{i+1} + V_i v_i} \right)$$

$$U^{n+1/2}_{i,\ j+1/2} = \frac{r_j U^n_{i,\ j+1} + r_{j+1} U^n_{i,\ j}}{r_{j+1} + r_j} - \Delta t \left(\frac{G^n_{i,\ j+1} A_{j+1} a_{j+1} - G^n_{i,\ j} A_j a_j}{V_{j+1} v_{j+1} + V_j v_j} \right)$$

Regular Grid Half-Timestep:

$$U^{n+1/2}_{i+1/2,\ j} = \frac{U^n_{i+1,\ j} + U^n_{i,\ j}}{2} - \frac{\Delta t}{2\Delta x} \left(F^n_{i+1,\ j} - F^n_{i,\ j} \right)$$

$$U^{n+1/2}_{i,\ j+1/2} = \frac{U^n_{i,\ j+1} + U^n_{i,\ j}}{2} - \frac{\Delta t}{2\Delta y} \left(G^n_{i,\ j+1} - G^n_{i,\ j} \right)$$



Finite Difference Equations (cont.)

Adaptive Mesh Full-Timestep:

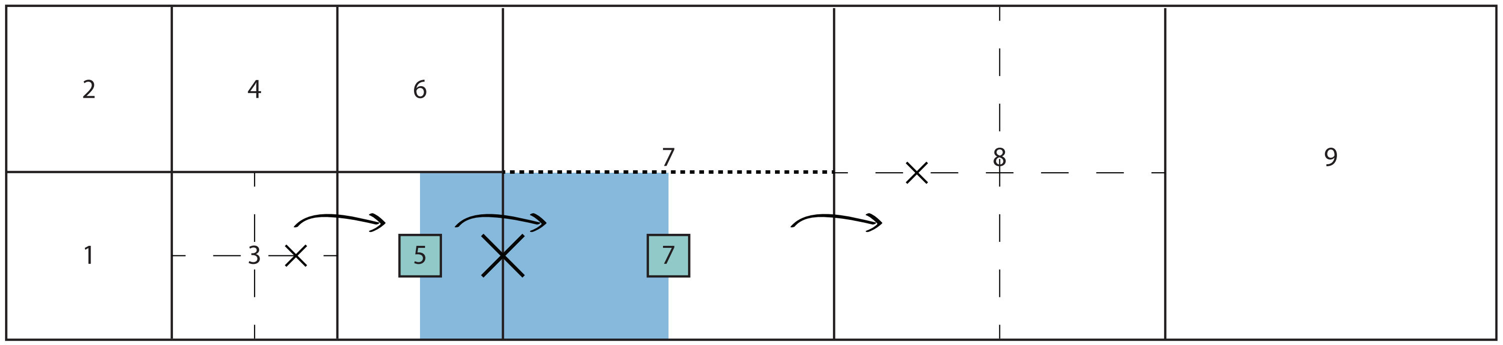
$$U_{i,\ j}^{n+1} = U_{i,\ j}^{n} - \Delta t \left(\frac{\overline{F_{i+1/2,\ j}^{n+1/2}} - \overline{F_{i-1/2,\ j}^{n+1/2}}}{\Delta x} + \frac{\overline{G_{i,\ j+1/2}^{n+1/2}} - \overline{G_{i,\ j-1/2}^{n+1/2}}}{\Delta y} \right)$$

Total Variation Diminishing (TVD) Correction:

$$U_{i,\ j}^{n+1} \mathrel{+}= \frac{\nu(1-\nu)}{2}\left[1-\phi(r^{+},\ r^{-})\right]\Delta U^{n}$$

★ Finite Difference vs. Finite Volume

$$\phi(r^{+},\ r^{-}) = \max\big(0,\ \min(1,\ r^{+},\ r^{-})\big)$$



Compact Numerical Methods

Current avenue of exploration…

- Principle of locality -- Physics and CS
    - *http://en.wikipedia.org/wiki/Principle_of_locality*
- Avoid large stencils
- Increase information content of the mesh
    - More information per cell vs. greater resolution
        - Superposition of exponentials (for shocks) and/or take several Fourier modes (for smoother regions)?
            - Better interpolation
            - Dictated by governing equations or modified equations?

Parallelism Implementation Issues

- Don't want data transfer to dominate runtime
- ***Locality*** at multiple levels (spatial locality in the mesh and/or in the cache?)
  - Locality (Partitioning) in MPI
  - Locality (Partitioning) in the GPU
  - Locality in Cache

Data ordering is a free parameter

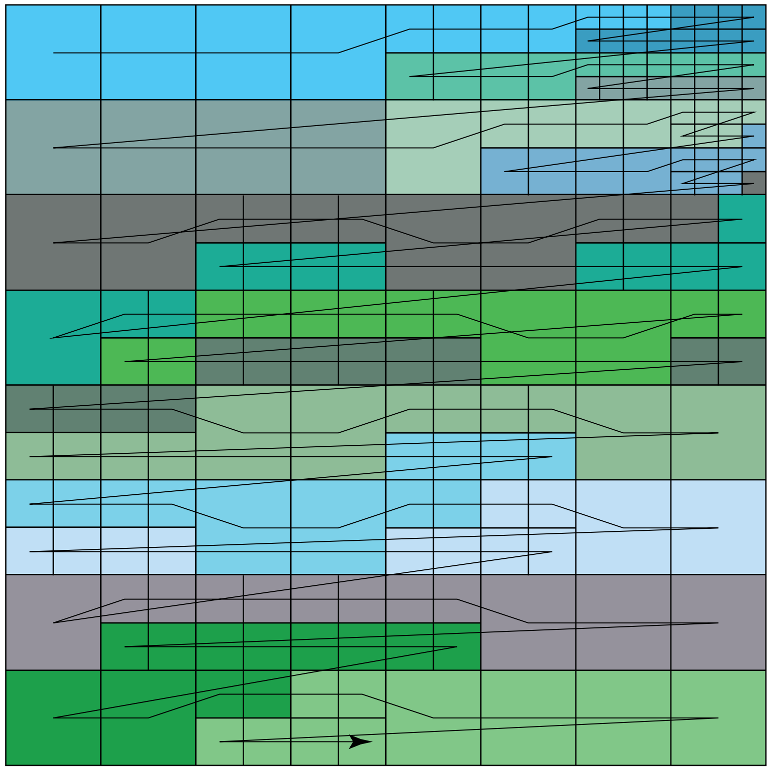
- Addresses all three locality issues

- Neighbor searching
- Enhanced precision sum for better consistency

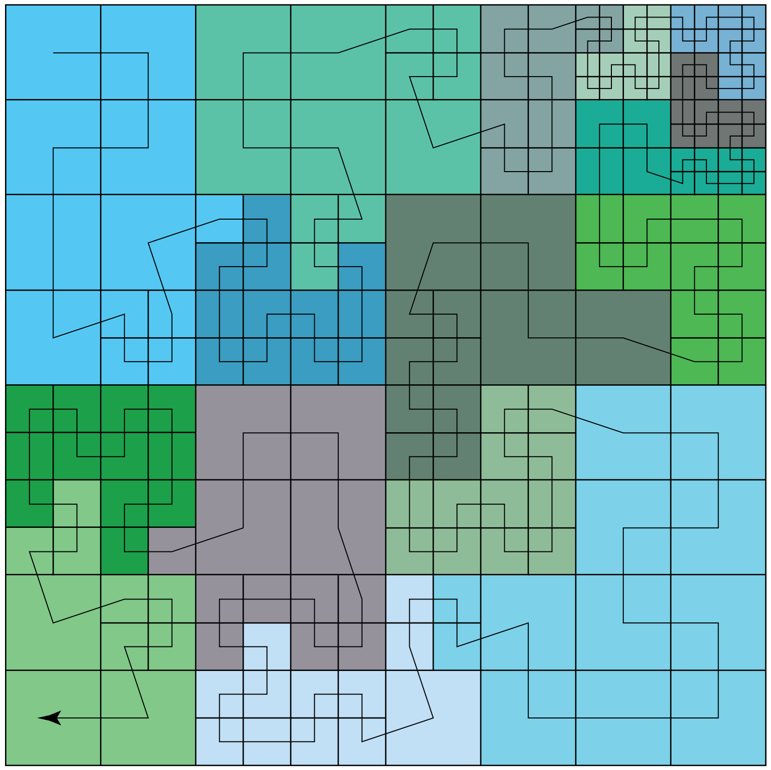
Partitioning in AMR

- A partition should group data such that each block of data has
  - Similar processing time, and
  - Minimal out-of-group data required for calculation,
- Or in other-words
  - Each block has an equal volume (# cells), and
  - Minimizes ***surface area to volume ratio***

Space-filling Curves:
A Tale of Two Partitions



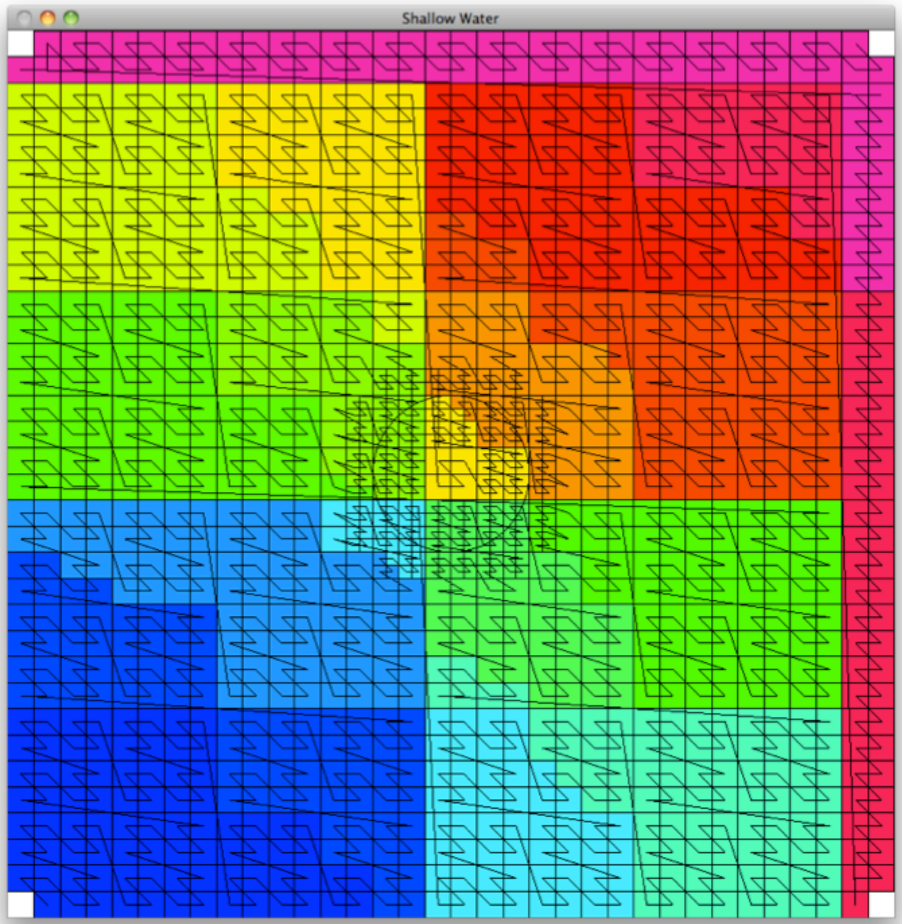
Naïve Partitioning



Hilbert Partitioning

Fragmentation of Block

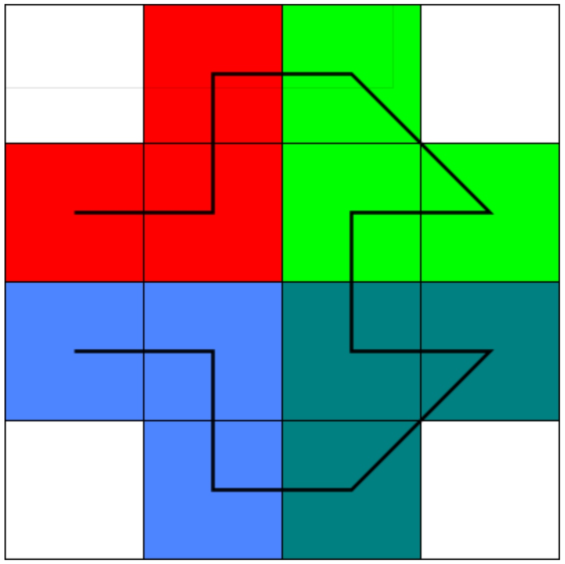
Using Z-order

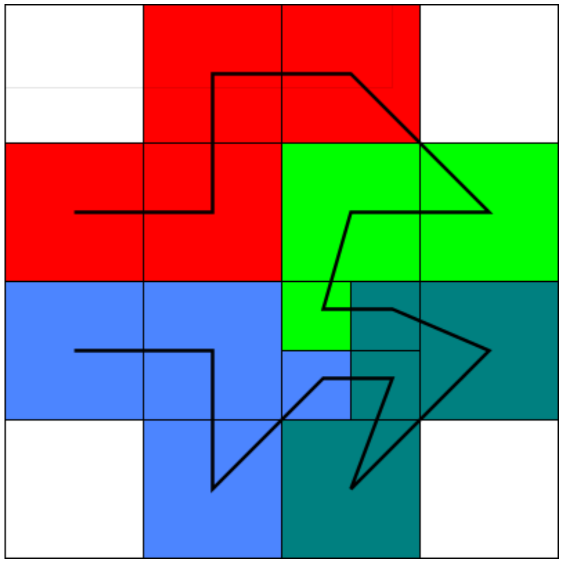


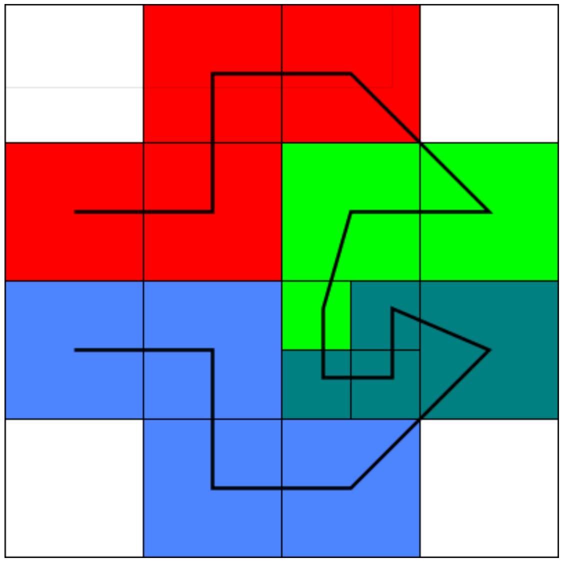
UNCLASSIFIED

Locality for Refinement

- 4 cells for refinement must be kept contiguous
  - Simplifies insertion and coarsening


Using Hilbert Curve


Non-contiguous Refinement Block

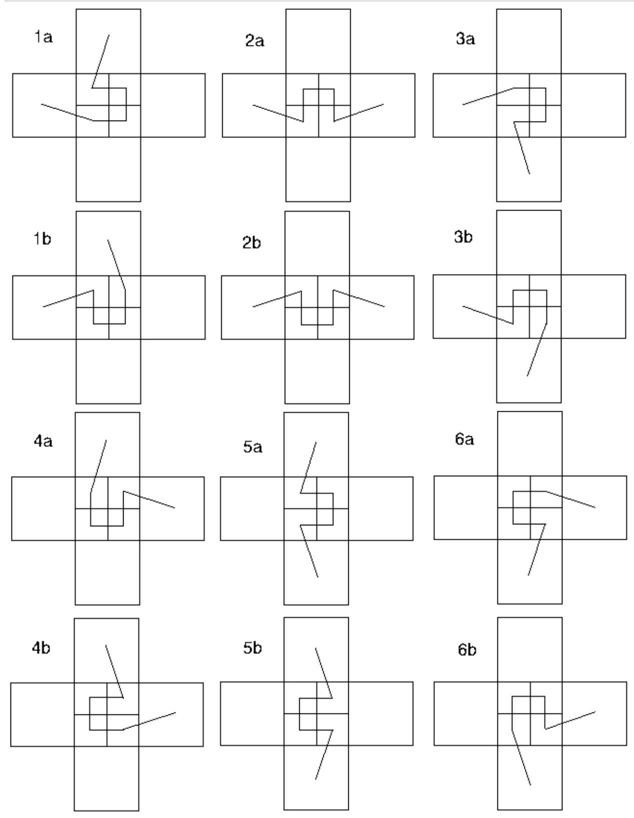

Contiguous Refinement Block

Locality for Refinement (cont.)

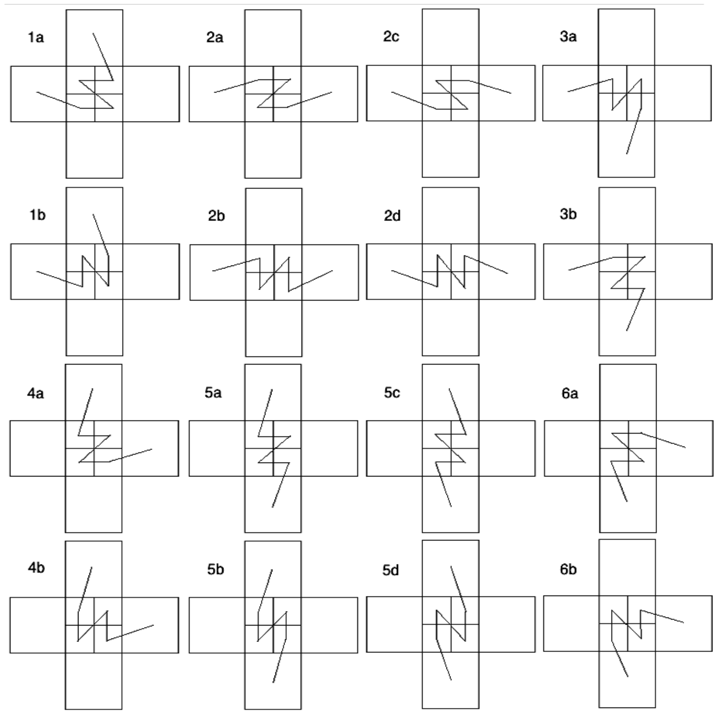
- Local insertion of space-filling curve
  - Eliminates need for recalc & “sort” after AMR step
- Possible local space-filling stencils
  - Z-order & Hilbert (2x2 stencil)
  - Match inward and outward directions of the coarse parent cell

Local stencil possibilities

Local Hilbert curve stencils



Local Z-order stencils



Partition Quality Measures Needed

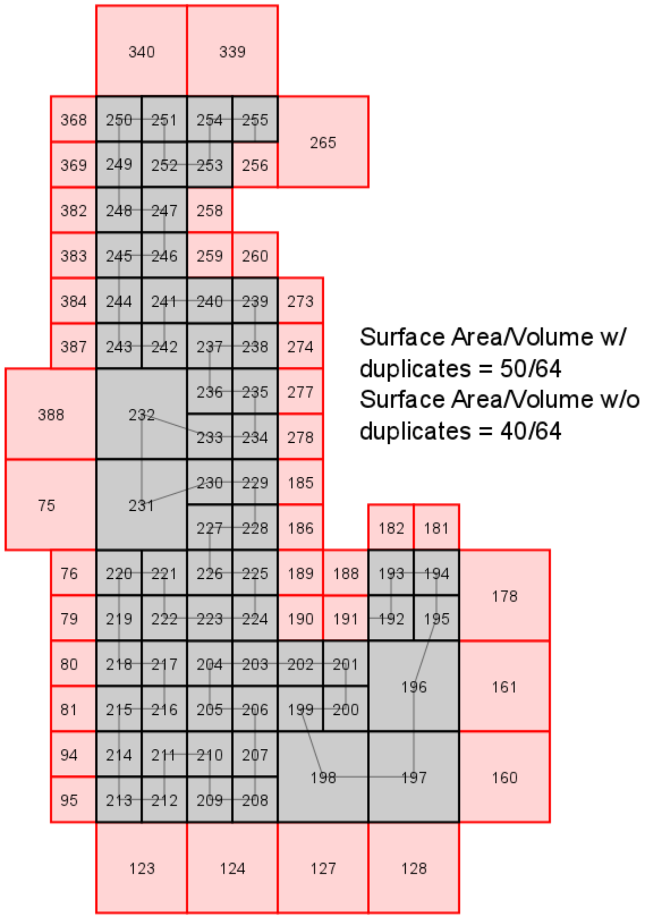
- Ratio of off-group data needed to group size
  - Measures across *n* groups in problem
    - Median, Average, Max
    - Which is best for MPI partitions? For GPU work-groups?
- Off-group characteristics
  - Median, Average, or Max of distance off block
  - Effect on cache access
  - Number of processors to communicate with

Partition Measures

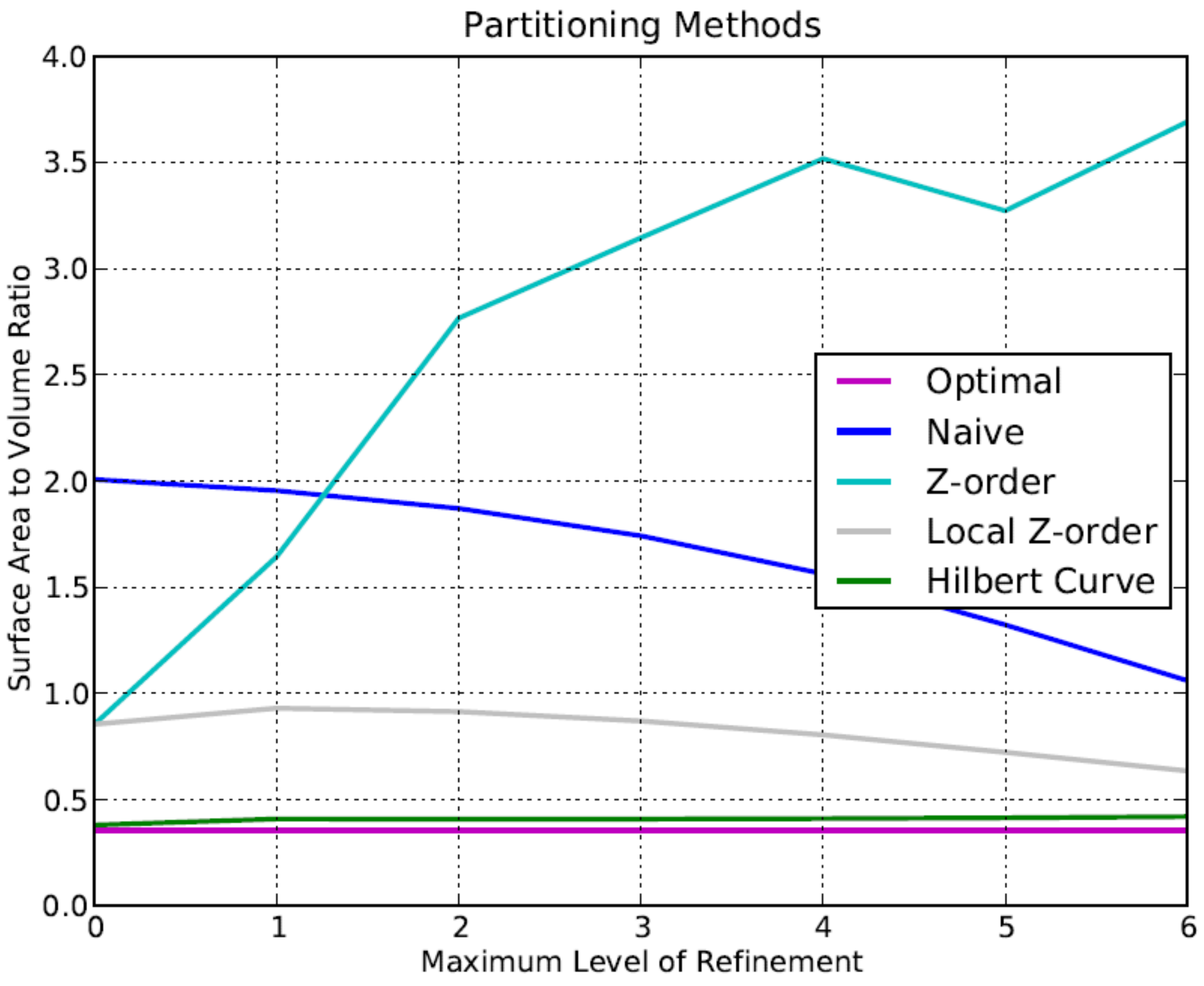
Surface area/Volume Measures

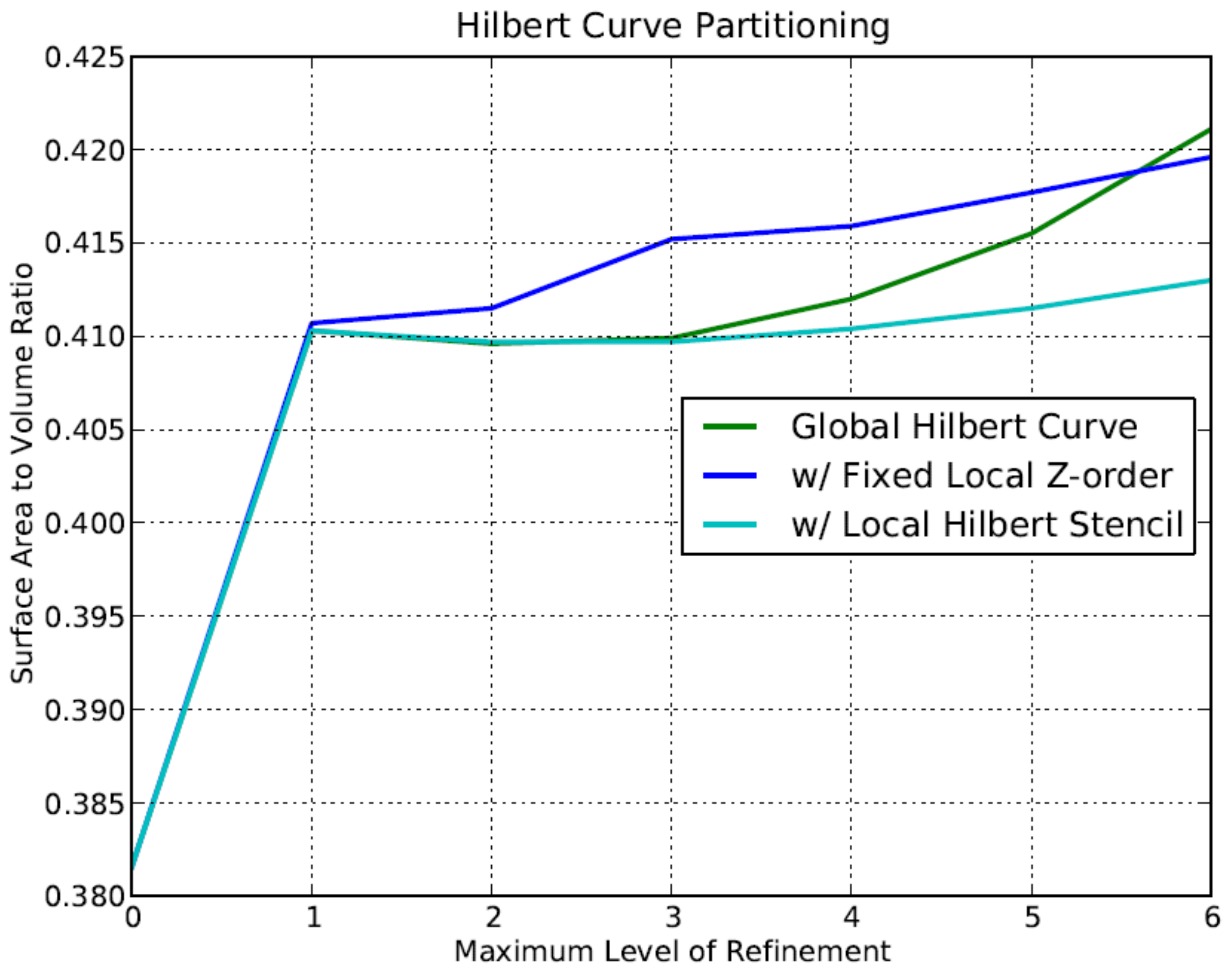
$$\frac{\text{Off-tile Accesses}}{\text{Tile Size}} = \frac{\text{\# red cells}}{\text{\# black cells}}$$

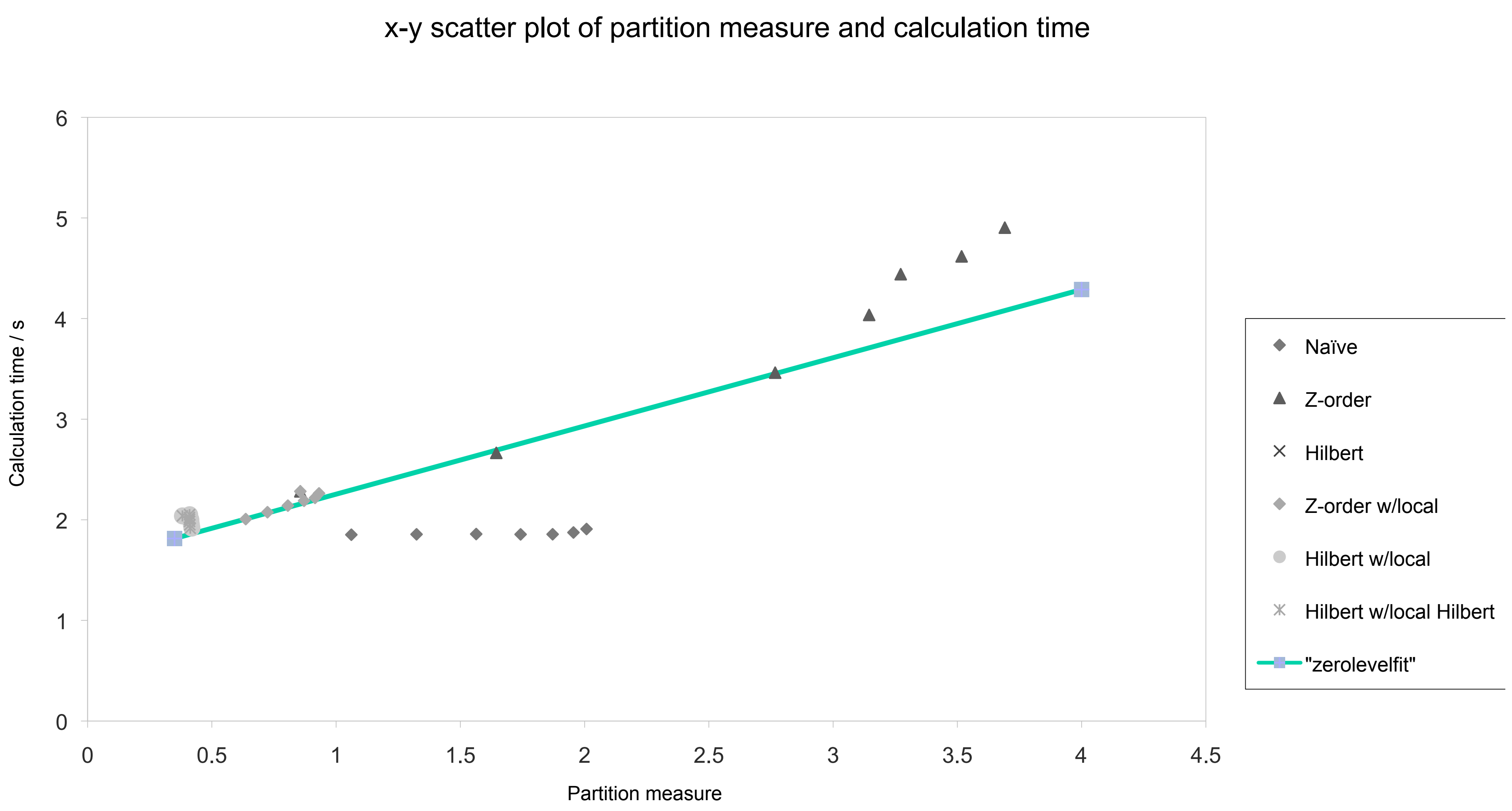
Can be with duplicates or w/o duplicates



Sample Computational Tile


Partitioning Methods
Surface Area to Volume Ratio
Maximum Level of Refinement
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
1
2
3
4
5
6
Optimal
Naive
Z-order
Local Z-order
Hilbert Curve


Hilbert Curve Partitioning
Surface Area to Volume Ratio
0.425
0.420
0.415
0.410
0.405
0.400
0.395
0.390
0.385
0.380
0
1
2
3
4
5
6
Maximum Level of Refinement
Global Hilbert Curve
w/ Fixed Local Z-order
w/ Local Hilbert Stencil


x-y scatter plot of partition measure and calculation time
Calculation time / s
Partition measure
0
1
2
3
4
5
6
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Naïve
Z-order
Hilbert
Z-order w/local
Hilbert w/local
Hilbert w/local Hilbert
"zerolevelfit"

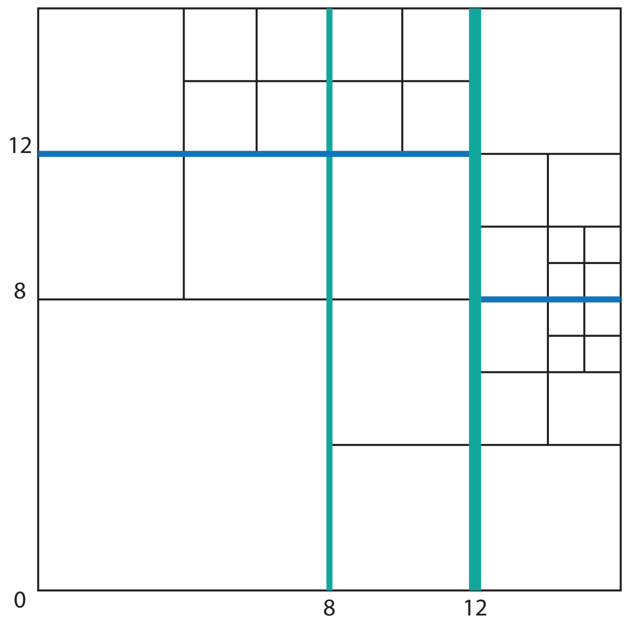
Neighbor Search Techniques

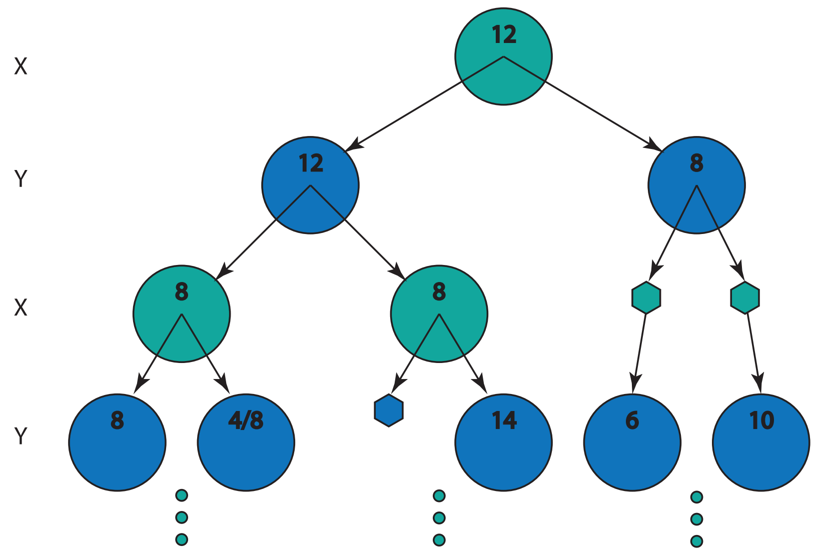
- Naïve is $O(n^2)$
- *k*-D Tree is a binary search algorithm rotating at each step among the *k* dimensions – $O(n \log n)$ look-up time
- A better solution is a hash table – $O(n)$ or $O(1)$ look-up time
  - Is there a good key (i.e. unique, small data table size and quick to calculate)?

k-D tree

- A *k*-dimensional binary spatial search
  - Tree Construction is O($n \log n$)
  - Querying an axis-parallel range in a balanced *k*-D tree takes O($n^{1-1/k}+m$) time, where m is the number of reported points
- Simplifies neighbor searches
  - Nevertheless, profiling shows 50% of the run time in neighbor calculation
  - So still better to track neighbors requiring many lines of code dealing with special cases and incur search costs only where needed

K-D Tree Example





The mesh is recursively bisected based on the number of cells, alternating between axes. A linked list is then created representing the tree structure. [$O(n \log n)$]

Current *k*-D tree code

- *2*-D only, expandable to *3*-D tree…
- Still needs insert and remove operators
  - Both $O(n \log n)$
- Has box and circle query only
- Needs optimization
- *K*-D tree not distributed (in the MPI sense)
- Can it be put on the GPU?
  - Yes – but not easily

Godiyal, Apeksha et al. “Rapid Multipole Graph Drawing on the GPU.”

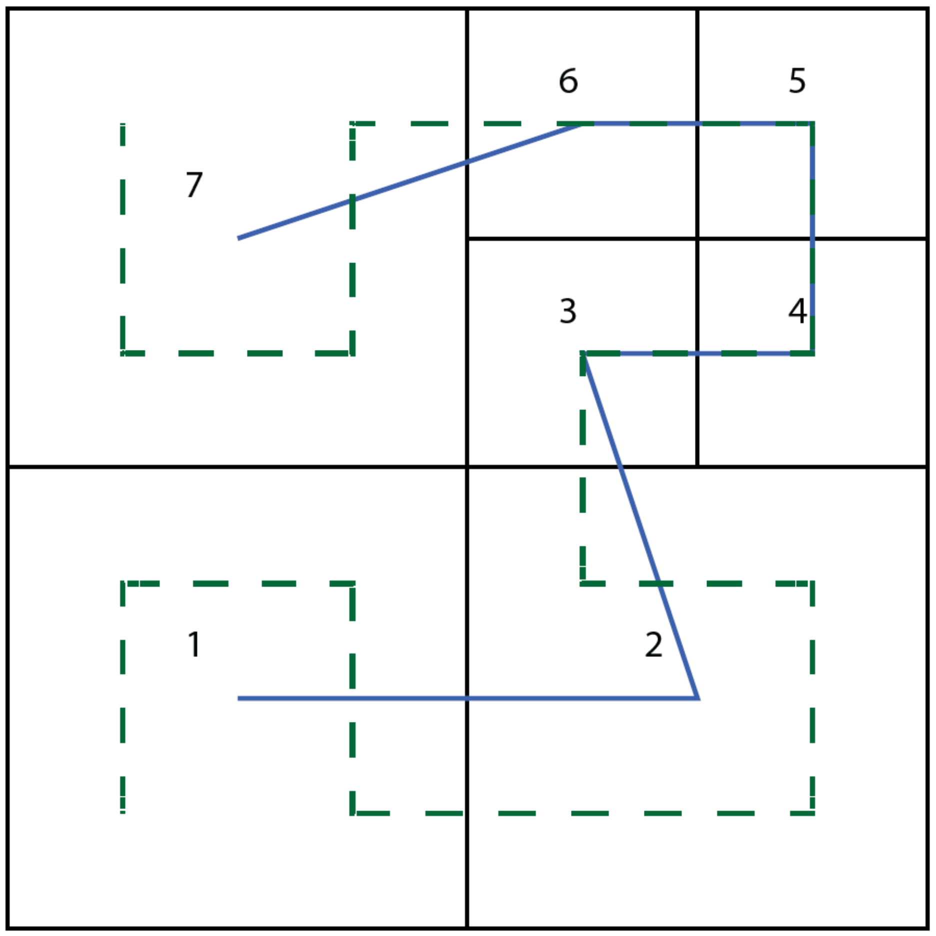
Hash Design Criteria

- Perfect Hash (no collisions) for simplicity and performance
- Minimizes memory requirements
- Exploits underlying structure of problem
- Simple key function
- Can be parallelized

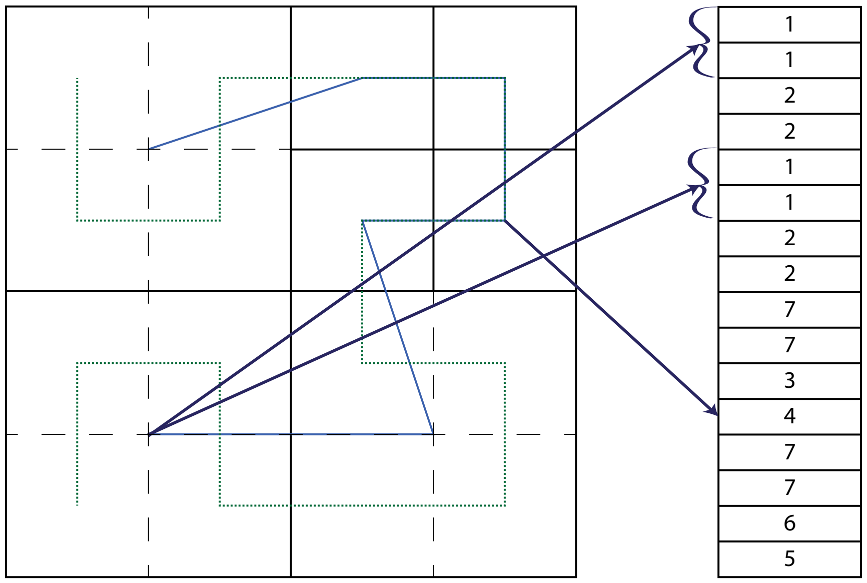
Two possible keys developed, one implemented

Both use the finest level of the mesh for the hash

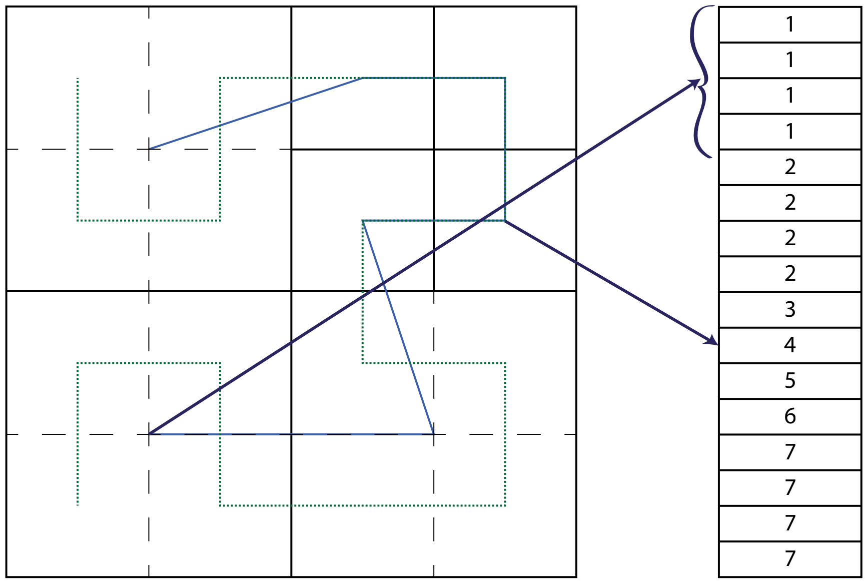
Hash Key Reference (for next slide)



Two Possible Keys



CARTESIAN 2D GRID KEY



HILBERT CURVE KEY

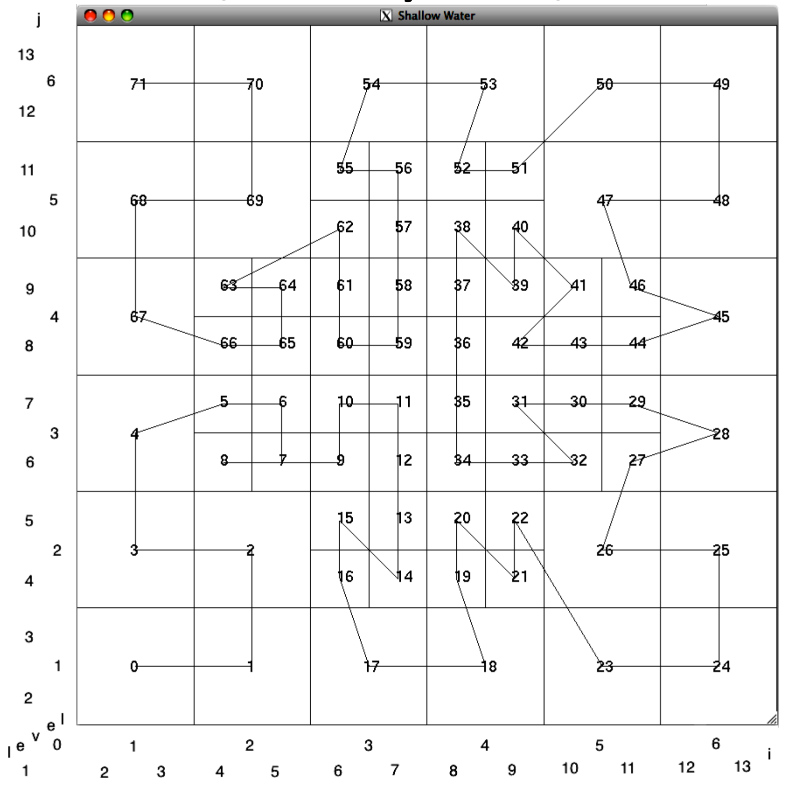
Based on a key function taking the Cartesian indices and refinement level as variables, a cell's neighbors are directly mapped into their respective arrays. [$O(n)$ creation, $O(1)$ lookup on the CPU]

1st Hash Key Candidate

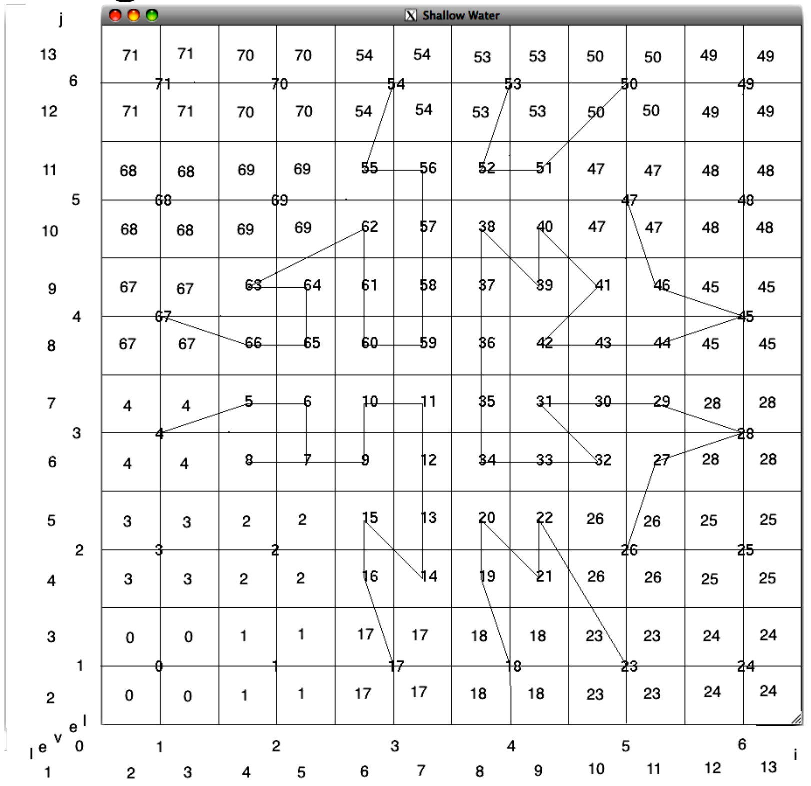
- Key is finest cell in a *2*-D (or *3*-D) grid
  - Store *1*-D cell index in *2*-D array at the finest level
  - If cell is coarser than the finest level, all finer cells are set with the cell index
  - Look-up calculates equivalent finest cell location of each neighbor and retrieves cell index
- Performance
  - Requires hash table size of finest level of the grid

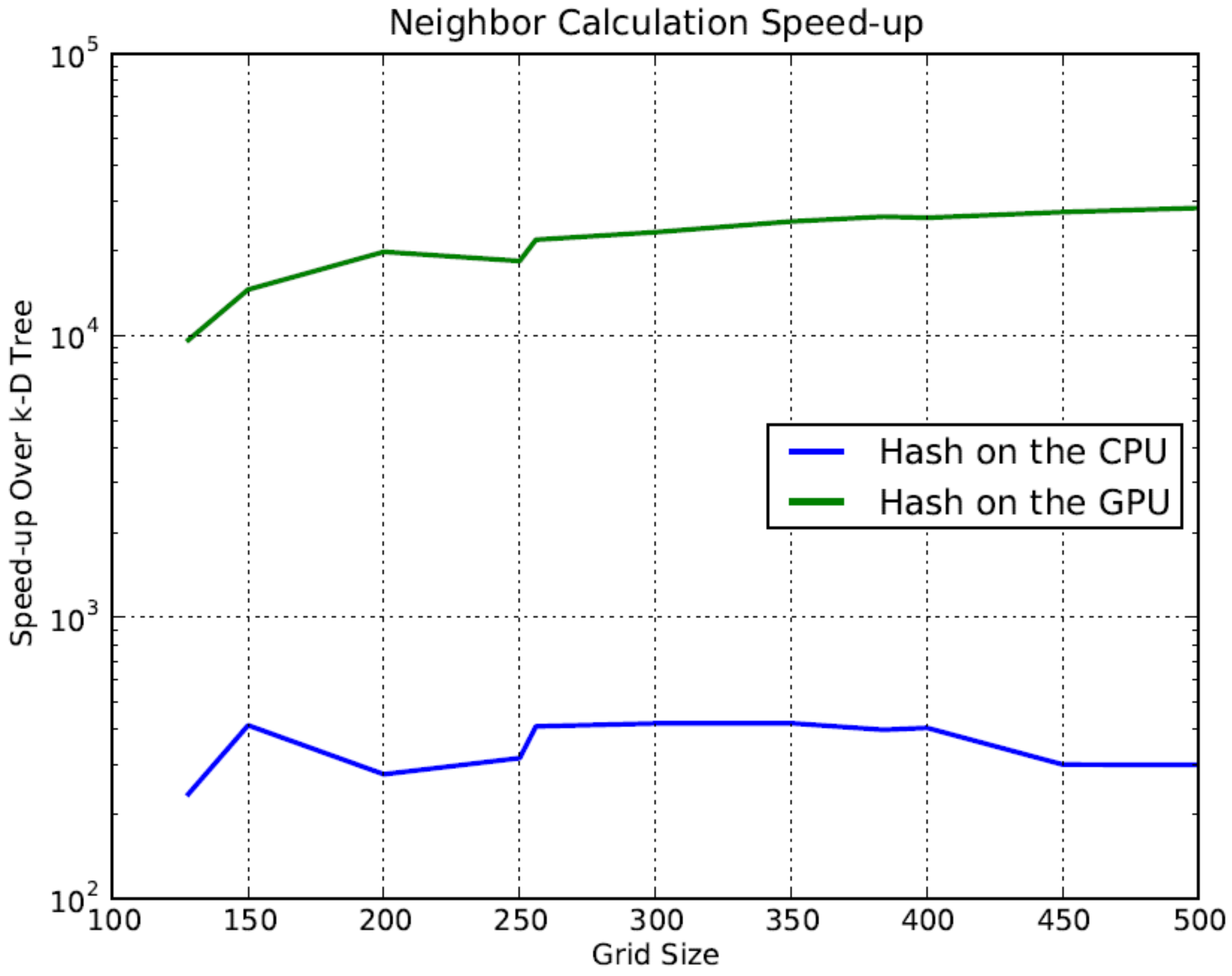
Determining neighbors by hash table look-up

Right neighbor of cell 22 (column 9, row 5) is 10,5.



Looking up in "hash table" at 10,5 gives cell 26.




Neighbor Calculation Speed-up
Speed-up Over k-D Tree
Grid Size
Hash on the CPU
Hash on the GPU
10^5
10^4
10^3
10^2
100
150
200
250
300
350
400
450
500

Neighbor search summary

- With structured grid patterns, hash table look-up is better
- For large *3*-D meshes, there may be a better key that uses less memory, but has collisions
- For irregular searches, *k*-D tree may be better
- **Hash table look-up put on the GPU in 1 day**

Consistency Improvements

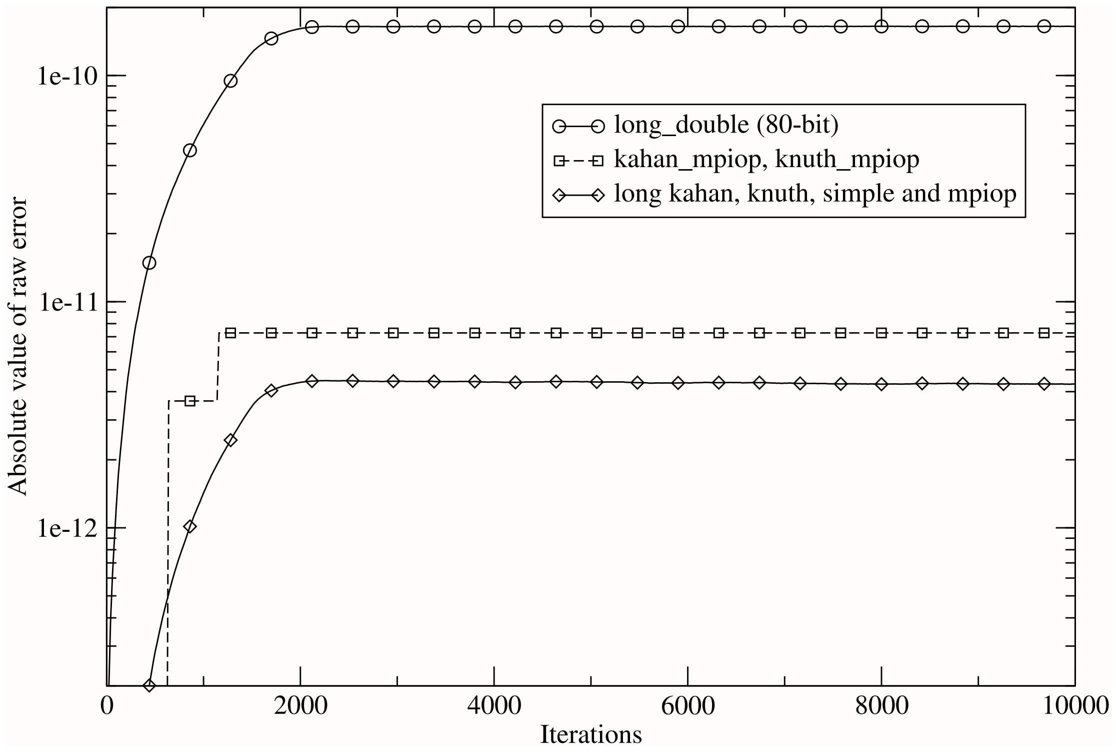
- Enhanced Precision Sum for global sums such as total mass
- Test problem:
  - 1 million cells on 4 mesh levels
  - 6 orders of magnitude difference in initial conditions
  - Every refinement for 100 cycles switch mesh order from Hilbert to Z-order and report maximum difference in total mass sum for each

Regular Sum: 448 x machine epsilon Kahan Sum: 0

- Reduce differences when running on varying numbers of processors, different order of data
- Avoids masking of coding errors

Enhanced Precision Sum

- Simple global mass sum raw error is 1.0e-4 (off the graph)
- Enhanced precision sums reduce the error 7 orders of magnitude



Comparison of error in various enhanced precision sums

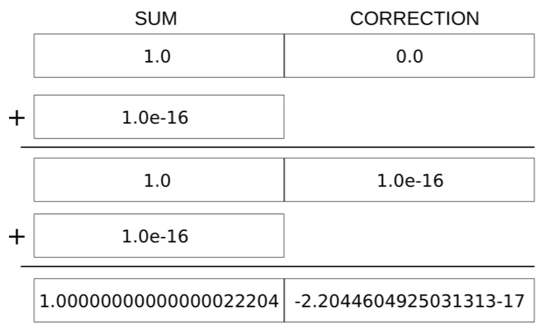
Enhanced Precision Sum

Parallel MPI_OP Kahan Sum

```
double parallel_mpiop_kahan_sum(double **var )
{
   double corrected_next_term, new_sum;
   struct esum_type local, global;
   local.sum=0.0;
   local.correction = 0.0;
   for (int j =0; j<jsize; j++){
      for (int i =0; i<isize; i++){
         corrected_next_term = var[j][i] + local.correction;
         new_sum = local.sum + corrected_next_term ;
         local.correction = corrected_next_term –
                            (new_sum-local.sum);
         local.sum = new_sum;
      }
   }
   MPI_Allreduce(&local, &global, 2, MPI_TWO_DOUBLES,
           KAHAN_SUM, MPI_COMM_WORLD);

   return(global.sum);
}
```

Using a “Carry Digit”



Robey, R., Robey, J., and Aulwes, R., “In Search of Numerical Consistency in Parallel Computing”, Vol. 37, Issues 4-5, Apr-May 2011, pp 217-229

Enhanced Precision Sum Setup and Wrapup

Setup

```
MPI_Type_contiguous ( 2 , MPI_DOUBLE, &MPI_TWO_DOUBLES) ;
MPI_Type_commit(&MPI_TWO_DOUBLES) ;
MPI_Op_create ( (MPI_User_function *)kahan_sum, commutative , &KAHAN_SUM) ;

void kahan_sum( struct esum_type * in , struct esum_type * inout , int *len , MPI_Datatype *MPI_TWO_DOUBLES){
  double corrected_next_term , new_sum;
  corrected_next_term = in->sum + ( in->c o r r e c t i o n+inout->c o r r e c t i o n) ;
  new_sum = inout->sum + corrected_next_term ;
  Inout->c o r r e c t i o n = corrected_next_term - (new_sum – inout->sum) ;
  Inout->sum = new_sum;
}
```

Wrapup

```
MPI_Op_free(&KAHAN_SUM) ;
MPI_Type_free(&MPI_TWO_DOUBLES)
```

Performance and Speedup Example

Coarse Grid: 450x450 ***GPU:*** Nvidia C2050

CPU: AMD Opteron 6168 ***Compilers:*** gcc, openmpi-1.5.3

Function	CPU (s)	GPU (s)	Speedup
Timestep Calc	10.576	0.143	73.96
Apply BCs	1.518	7.182	41.40
Finite Difference (TVD)	295.826		
Refinement Potential	11.630	0.397	29.30
Rezoning	7.115	0.684	10.40
Partitioning	2.258	0	---
Mass Sum (Kahan)	2.346	0.095	24.70
Write to Device	0	0.007	---
Read from Device	0	0.013	---
Total	331.27	8.52	38.9

Summary

- Cell-based AMR has the potential to work well on the GPU
  - *1*-D easier than *2*-D on the GPU
  - Performance looks attractive
- GPU port forces re-examination of techniques, resulting in innovative ideas
  - Local space-filling stencils
  - Hash-based neighbor search
  - Enhanced precision sums

Acknowledgements

- Carter Edwards, SNL
  - Hilbert Space Filling Curve
  - Available in Zoltan
- Richard Barrett, SNL
  - L7 library for MPI sparse communication
- Rachel Robey
  - Real-time OpenGL graphics (no bugs!)

Please Reference…

Nicholaeff, Davis, Trujillo, Robey. *Cell-based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units. In Review, Journal of Computational Physics, 2012.*