* Introduction
  + RBF Methods are great for
    - Scattered Data
    - High dimensions
    - Higher order accuracy
    - Etc.
  + RBF methods have a 30+ year history
    - Interpolation: 1970
    - Franke comparison 1982
    - PDE History is only 1995-present
    - The method is still young with limited application
      * Little attention in HPC
  + Targeting HPC
    - Need to focus on hardware that will get us to Petascale
      * Keep method current
    - GPUs/Accelerators are expected to be (default|common|…) on petascale architectures
      * Target GPUs/Accelerators first
      * GPU for RBF is limited; only one related work
    - Focus on MPI to scale across HPC clusters
      * MPI for RBF is limited; only a few related works
  + We bring together the combination of RBF-FD, MPI, GPU
    - And we demonstrate combinations through applicaton to various problems
      * Implicit and Explicit PDEs
    - Goal is to construct building blocks for a large scale Geophysical simulation
* Preliminaries
  + RBF method history
    - Related methods in history leading to RBF-FD
  + Equations for Related RBF methods and how they compare
    - Show commonalities (RBF interpolation)
    - Differences
* RBF-FD
  + Related work
  + Define: Stencils
  + Equations to get stencil weights
  + Define: Differentiation matrix
  + Multiple Operators
  + Weight Operators
    - First and second derivatives
    - Cartesian Gradient
    - Cartesian Laplacian
    - Laplace-Beltrami
    - Constrained Gradient on Sphere
    - Hyperviscosity
  + Implementation of RBF-FD
    - Algorithm showing flow of RBF-FD
      * Two phases
        + Preprocessing
        + Application
      * Complexity of phases depends on choice of algorithm for each task
      * Phase 1 is strictly preprocessing and data can be loaded from disk to bypass on subsequent runs.
      * Generally, PDEs will have time-steps making phase 2 the more computationally intense.
      * Application phase would be similar regardless of method choice.
      * Here we discuss various design decisions within the preprocessing phase and consider potential impacts on performance (if any).
    - Preprocessing tasks
      * Grid Generation
        + We can load or generate grids
        + Regular grid is simplest and used for testing/debugging
        + For most tests on sphere we load MD node sets to confirm results with other RBF literature
        + For large node sets we also consider CVTs since they keep nodes from overlapping.

Have I mentioned nodes should not coincide with RBF-FD?

CVT simple algorithm (Lloyd’s)

* + - * Stencil generation
        + Brute force is obvious starting point, but this is O(N^2).
        + K-DTree is used by many in the RBF world to reduce complexity to O(N log N) search time for all stencils.

Bengt and Fasshauer refer to kDTree

Complexity

* + - * + Hashing can further reduce complexity.

Borrowed from SPH, this method builds the equivalent of an axis aligned bounding box. Internal representation though, since we have node indices, we can hash the coordinates directly to a cell in AABB.

Downside: approximate nearest neighbors

Still appropriate for RBF-FD though. No requirement to have strict Nearest Neighbors.

Complexity.

Impact of ordering on Sparsity

Alternative orderings to consider

* + - * + Performance comparison of 3 methods (incl. Figure)

What is the minimum N and n to justify LSH?

What is the number of divisions we need in the AABB? SPH uses ~2-10 nodes per cell

* + - * On Choosing Epsilon for weights
        + Ill conditioning is an issue

Lots of references allude to this

Bayona, others seek to find the optimal for general node placement

Stable methods bypass this struggle

* + - * + Choose epsilon proportional to h

Choose epsilon as function of N

Choose epsilon curve as function of k(N)

We follow approach in Lehto et al. to choose based on k(N)

Example Figures of contours generated following Lehto approach.

* Parallelizing RBF-FD solutions
  + As previously mentioned, the dominant cost in RBF-FD arises in the application phase when the DM is used to solve the PDE
    - Explicit solutions require SpMV, SAXPY for update
    - Implicit solutions require GMRES or another iterative solver; which in turns requires multiple SpMV, SAXPY internally.
    - Reduce as a means to calculate norms and monitor progress of application.
  + Parallelization is achieved at two levels
    - A domain decomposition allows us to distribute the SpMV and SAXPY operations
    - We target the GPU with OpenCL and ViennaCL
  + Multi-CPU/Multi-GPU Implementation is first in the RBF-FD community
    - Related work for RBF methods on GPU is limited to Schmidt
    - Distributed RBF methods limited to Knepley and a few others\*
  + Leveraging GPU
    - GPU features
      * Memory layout
      * Multi-Processors
      * Bandwidth on GPU (refer to Bell for significance)
      * Table comparing hardware of M2070, M2090, Phi
      * Trends in hardware since 2006
        + Cheap to purchase, superior performance
        + Trending technology that nearly all supercomputing centers are buying into; predominantly CUDA hardware, until 2012 when Intel released the Phi cards
        + Initially research focused on porting codes and determining the limits of the almost black-box hardware. Today it seems as though the buzz/hype over GPU computing is winding down as more and more research leverages existing code that was previously optimized for the GPU. That is understandable; focus on getting the science done, rather than the computer science. Newcomers are more interested in leveraging the GPUs rather than optimizing for them.
    - Choice to work with OpenCL and language summary
      * Functional portability vs performance portability
      * Promise for open standard in parallel languages
      * Work-Items, Work-Groups, NDRange, etc.
      * Local memory
      * Asynchronous kernel execution, Queues
      * Memory transfers
      * How does it compare to CUDA? OpenACC?
        + OpenCL attempts to harness the power of all hardware. Supports ATI, Nvidia, etc. Even has support for Cell BE and Phi.
        + No support for direct MPI communication (e.g., CUDA-MPI)
      * Language notation is heavy. Simplification is a must.
        + OpenCL C++ headers are one option, but this does not simplify the task our problem
        + Approach our problem from a higher level of abstraction with a sparse matrix library and primitives like SpMV. Answer question: why use libraries like ViennaCL?

Faster development

Public project with shared interests

Simple API

* + - * Why avoid the language?
        + Changes are occurring too fast

Hardware is great for performance (>1TFLOP) compared to price to purchase (<$1K)

But hardware is arriving at an incredible rate.

Not to mention adoption is successful in existing codes where optimization was possible on day 1. For codes developing from scratch it is still easier to debug without using the hardware. Most problems that people would put on GPU can be reduced to some BLAS or LAPACK primitive, so it makes sense to use a library like ViennaCL

* + - * + ViennaCL already adopted back-end switch for OpenMP, CUDA and OpenCL to allow more versatility.

Whatever vendor or language available libraries like ViennaCL will adapt and our code will change little.

* + - GPU hardware features
      * Custom kernels with one thread vs one warp
    - SpMV will be bandwidth limited
      * 2:1 operations; or 1:1 if a fused multiply and add is used
    - Custom kernels vs libraries like ViennaCL
      * Simple interface where transfer to/from GPU happens behind the scenes; incl. Kernels
      * Fast prototyping
      * Compatibility with BOOST, Eigen, MKL
        + BOOST in turn has compatibility with
      * ViennaCL formats
        + COO
        + CSR
        + ELL

Ideal for RBF-FD given assumption all stencils are size n

* + - * + HYB

All indicators point to ideal for RBF-FD with variable stencil size

* + - * Performance comparison of ViennaCL formats reveals expected 27x speedup over CPU (boost SpMV, not as optimal as MKL; only optimized for one core).
        + What is the fastest SpMV for RBF-FD?
        + Need: MKL SpMV for comparison
        + Need: table of GFLOPs
      * Other known formats
        + CUSPARSE offers some (limited to CUDA)
        + MKL offers some
        + Block options in clSpMV; clSpMV (OSKI) is also best competitor
  + Distributed Multi-CPU/Multi-GPU
    - PETSc, Hypre, Trilinos are all libraries/frameworks that we could have developed in, but none of them had GPU support. We continued with custom built code, but that decision required addition decisions from us.
    - Domain decomposition allows one to span multiple processors
      * Partitions are handled by independent processors
      * Communication between partitions at each time-step to resolve missing node values
      * For initial development, Send/recv between domains in round-robin. For improved scaling an Alltoallv collective is used.
        + MPI\_iAlltoallv expected in MPI v3 (mid to late 2013).
      * Figure of Matrix decomposition
    - Nodes are partitioned into domains according to some plan
      * For development we initially choose partitions in X-direction.
        + Figure of X partiitoning
        + Results in unequal partitions
        + Show typical ratio (N min partition / N max partition)
      * METIS can be used to partition with better load balancing
        + Metis algorithm
        + Algorithm depends on symmetric adjacency graph
        + Symmetry will not impact stencils/weight calculation, and can be safely assumed since statistically if A contains B, then B will contain A (unless they are distant and a fringe of the stencil).
        + Ratio of METIS partitioning
        + Figure of METIS partitioning (requires VTK rendering)
      * Other libraries exist to help with this process: SCOTCH, ParMETIS, hMETIS, and others
    - Internally we assemble node sets to manage the partition and properly index nodes
      * Construction of Q, B, R etc
      * Figure showing decomposition across multiple processors with value message passing
      * Node sets allows us to control the flow of data to/from the GPU.
        + Memcpy is contiguous which greatly simplifies life
* Explicit and Implicit Solutions
  + Explicit
    - As test case we consider Cosine Bell and Vortex Rollup on the sphere.
    - Algorithms require u’ = Du + Du + Hu
    - Essentially, just repeat what the paper had
      * Hyperviscosity to stabilize
      * Convergence
      * Include convergence figure without stability to observe that it is possible for instability to not appear and cause misdirection.
    - Performance of CosineCL vs CosineVCL
  + Implicit
    - Don’t invert large sparse matrices. Use iterative solvers to get the solution easier
    - GMRES algorithm
      * Cite Saad
      * Cite H\*\*\*\* for multi-GPU CUDA GMRES
      * Givens vs Householder implementation
      * ViennaCL had Householder, Givens is easier to parallelize so implemented to include parallel options
      * Introduced ILU0 preconditioner
      * Need: convergence plots and tables for GMRES to demonstrate correctness
        + Regular grid, simple equation; what is VCL test problem for convergence?
    - Stokes problem
      * Equations
      * Discretization
        + Projection operators to get on sphere
      * DM form
      * Manufacturing solution
        + Spherical harmonics on Divergence-free field
      * Node interleaving condenses the weights for better memory load, easier partitioning
      * Need: convergence plots
      * Regular noise in the solution may point to need for Hyperviscosity or stable method for weight calculation
        + Need: RBF-GA test for convergence
  + Conclusion: we have RBF-FD implementations of building blocks necessary for a complex geophysics simulation.
    - Now all we need to do is make sure it is efficient.
* Performance and Optimization
  + Define throughput
    - Measure performance as GFLOPs
    - Compare to CPU (even if limited to 1 core) GFLOPs
    - Need: What are the peaks for each hardware type?
      * Given the achieved and the peak we can normalize the throughput as % peak and consider speedup comparison.
  + Optimizations for CL kernels
    - How custom kernels compare to ViennaCL
    - Recall that operation is limited by bandwidth
      * What is the minimum N and n to justify the GPU?
    - Some hope exists in using clSpMV approach to optimize
  + Distributed performance
    - Define two types of scaling
    - How does method scale?
    - How do we avoid loss due to mem xfer to GPU?
      * Need: overlap
    - Need: similar metrics as those in Tuminaro paper
    - Need: similar to Knepley paper
* Node Ordering and Preconditioning
  + Node ordering condenses bandwidth
    - Compared to SymRCM our bandwidth achieved may be wider, but it is FREE.
    - Conditioning of symrcm and node ordered matrices (U, X, Z, Raster).
    - Impact on conditioning? If any?
    - clSpMV heuristics applied to our modified sparsity patterns may prove enlightening.
  + Other preconditioners
    - Most preconditioners are based on node or stencil information
    - Preconditioners:
      * ILU0
      * ILUP
      * AMG
      * MG
    - What is the impact of the preconditioners with and without RBF-GA?
    - What is the best preconditioner for RBF-FD?
    - Does masking blocks when preconditioning help?
* Conclusion
  + First implementation of RBF-FD to span multiple CPUs. Went above and beyond to extend to multiple GPUs.
  + Considered both explicit and implicit PDEs to ensure we have all building blocks necessary to tackle large-scale Geophysical simulations
  + Looked at preconditioning and node orderings to ensure the fastest possible time to accurate solutions
  + Etc.
* Appendices
  + Direct computation of weights vs indirect to reassure ourselves that indirect do not lose too much accuracy
    - Data already exists; justify choice of functions and derivative types
* Missing Topics:
  + Evidence of overlapping comm and comp
  + Evidence of clSpMV algorithms for even more throughput