Solution to PDEs using Radial Basis Function Finite-Differences (RBF-FD) on Multiple GPUs

Evan F. Bollig
bollig@scs.fsu.edu
Florida State University

Natasha Flyer
flyer@ucar.edu
NCAR

Gordon Erlebacher
gerlebacher@fsu.edu
Florida State University

July 3, 2012

Abstract

This paper presents parallelization strategies for the Radial Basis Function-Finite Difference (RBF-FD) method. As a generalized finite differencing scheme, the RBF-FD method functions without the need for underlying meshes to structure nodes. It offers high-order accuracy approximation but scales as $O(N)$ per time step, with N being with the total number of nodes. To our knowledge, this is the first implementation of the RBF-FD method to leverage GPU accelerators for the solution of PDEs. Additionally, this implementation is the first to span both multiple CPUs and multiple GPUs. OpenCL kernels target the GPUs and inter-processor communication and synchronization is managed by the Message Passing Interface (MPI). We verify our implementation of the RBF-FD method with two hyperbolic PDEs on the sphere, and demonstrate up to 9x speedup on a commodity GPU with unoptimized kernel implementations. On a high performance cluster, the method achieves up to 7x speedup for the maximum problem size of $27,556$ nodes.

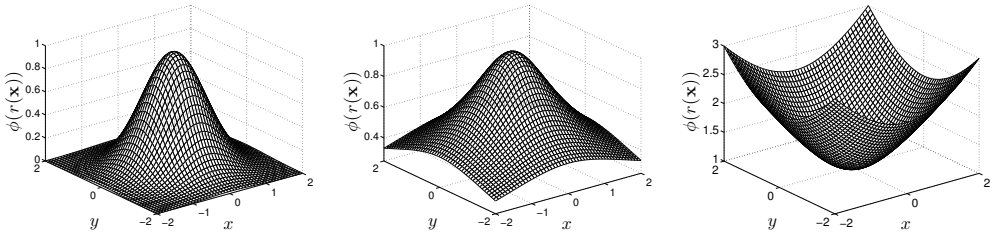
1 Introduction

Numerical methods based on radial basis functions (RBFs) are rapidly gaining popularity for the solution of partial differential equations (PDEs). With a history extending back four decades for RBF interpolation schemes [30], and two decades for RBFs applied to solving PDEs [33], many avenues of research remain untouched within their realm. Being a meshless method, RBF methods excel at solving problems that require geometric flexibility with scattered node layouts in n-dimensional space. They naturally extend into higher dimensions without significant increase in programming complexity [18, 57]. In addition to competitive accuracy and convergence compared with other state-of-the-art methods [15, 16, 18, 19, 57], they also boast stability for large time steps.

Examples of infinitely smooth RBFs in 2D space are shown in Figure 1 (good references on non-smooth and compactly supported RBFs, which are not considered in this paper due to lower order of convergence, are [6, 55]). RBF methods are based on a superposition of translates of these radially symmetric functions, providing a linearly independent but non-orthogonal basis used to interpolate between nodes in n-dimensional space. An example of RBF interpolation in 2D using 15 Gaussians is shown in Figure 2, where $\phi_j(r(\mathbf{x}))$ is an RBF centered at $\{\mathbf{x}_j\}_{j=1}^n$. The radial coordinate is $r = \sqrt{(x-x_j)^2 + (y-y_j)^2}$.

Infinitely smooth RBFs depend on a shape or support parameter ϵ that controls the width of the function. The functional form of the shape function becomes $\phi(\epsilon r)$. Decreasing ϵ increases the support of the RBF and in most cases, the accuracy of the interpolation, but worsens the conditioning of the RBF interpolation problem [44]. Fortunately, recent algorithms such as Contour-Padé [26] and RBF-QR [22, 24] allow for numerically stable computation of interpolants in the nearly flat RBF regime (i.e., $\epsilon \to 0$) where high accuracy has been observed [25, 37].

Historically, the most common way to implement RBFs is in a global sense. That is, the value of a function value or any of its derivatives at a node location is a linear combination of all the function values over the *entire* domain, just as in a pseudospectral method. If using infinitely smooth RBFs, this leads to exponential convergence of the RBF interpolant for smooth data [21]. As discussed in [19], global RBF



(a) Gaussian (b) Inverse Multiquadric (c) Multiquadric

Figure 1: Commonly used RBFs.

differentiation matrices (DM) are full, requiring $O(N^3)$ floating point operations (FLOPs) to assemble for a given node layout and $O(N^2)$ to time-step.

Alternatively, one can use RBF-generated finite differences (RBF-FD) to introduce sparse DMs (Note: for pure interpolation, compactly supported RBFs can also introduce sparse matrices [54]). RBF-FD was first introduced by Tolstykh in 2000 [49], but it was the simultaneous, yet independent, efforts in [46], [50], [56] and [8] that gave the method its real start. The RBF-FD method is similar in concept to classical finite-differences (FD) but differs in that the underlying differentiation weights are exact for RBFs rather than polynomials. RBF-FD share advantages with global RBF methods, like the ability to function without an underlying mesh, easily extend to higher dimensions and afford large time steps; however spectral accuracy is lost. Some of the advantages of RBF-FD include high computational speed together with high-order accuracy (6th to 10th order accuracy is common) and the opportunity for parallelization. As in FD, increasing the stencil size n increases the accuracy of the approximation. Given N total nodes in the domain (such as on the surface of a sphere), N linear systems, each of size $n \times n$, are solved to calculate the differentiation weights. Since $n \ll N$, the RBF-FD preprocessing complexity is dominated by $O(N)$, much lower than for the global RBF method of $O(N^3)$, with the cost per time step also being $O(N)$. RBF-FD have been successfully employed for a variety of problems including Hamilton-Jacobi equations [8], convection-diffusion problems [9, 48], incompressible Navier-Stokes equations [10, 46], transport on the sphere [23], and the shallow water equations [17].

As N grows larger, it behooves us to work on parallel architectures, be it CPUs or GPUs. With regard to the latter, there is some research on leveraging RBFs on GPUs in the fields of visualization [13, 53], surface reconstruction [7, 12], and neural networks [5]. However, research on the parallelization of RBF algorithms to solve PDEs on multiple CPU/GPU architectures is essentially non-existent. We have found three studies that have addressed this topic, none of which implement RBF-FD but rather take the avenue of domain decomposition for global RBFs (similar to a spectral element approach). In [14], Divo and Kassab introduce subdomains with artificial boundaries that are processed independently. Their implementation was designed for a 36 node cluster, but benchmarks and scalability tests are not provided. Kosec and Šarler [36] parallelize coupled heat transfer and fluid flow models using OpenMP on a single workstation with one dual-core processor. They achieved a speedup factor of 1.85x over serial execution, although there were no results from scaling tests. Yokota, Barba and Knepley [59] apply a restrictive additive Schwarz domain decomposition to parallelize global RBF interpolation of more then 50 million nodes on 1024 CPU processors. Only Schmidt et al. [45] have accelerated a global RBF method for PDEs on the GPU. Their MATLAB implementation applies global RBFs to solve the linearized shallow water equations utilizing the AccelerEyes Jacket [2] library to target a single GPU.

To our knowledge, this paper presents the first implementation of RBF-FD to span multiple CPUs. Each CPU has a corresponding GPU attached to it in a one-to-one correspondence. We thus also present the first known implementation of accelerated RBF-FD on the GPU. Within the scope of this paper we detail our method for spanning RBF-FD across multiple CPU/GPU processors and emphasize numerical validation of the implementation rather than optimization strategies. We will consider optimization in future work. The calculations are performed on Keeneland, a high performance computing installation supported by the National Science Foundation and located at Oak Ridge National Lab. Keeneland currently has 240 CPUs accompanied by 360 NVidia Fermi class GPUs with at least double that number expected by the end of 2012 [51].

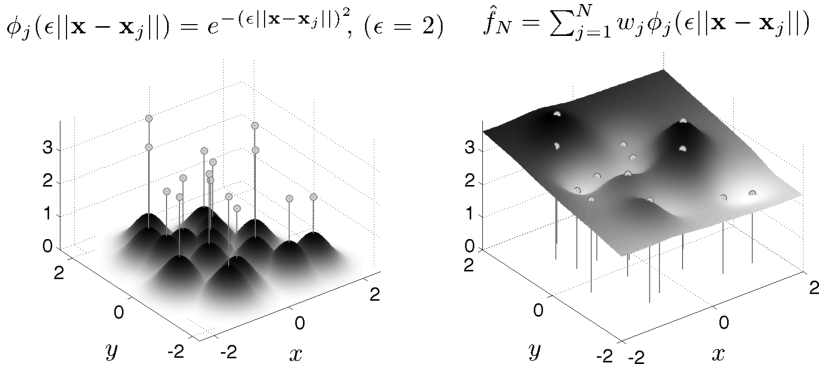


Figure 2: RBF interpolation using 15 translates of the Gaussian RBF with $\epsilon = 2$. One RBF is centered at each node in the domain. Linear combinations of these produce an interpolant over the domain passing through known function values.

The remainder of this paper is organized as follows: Section 2 introduces RBF-FD via interpolation. Section 3 details our parallelization strategies for the Keeneland system, including data partitions that are handled concurrently by different CPU processes and the data-parallel explicit time stepping scheme for the GPU. In Section 4, our implementation is verified against well-known hyperbolic PDE test cases on the sphere, advection of a cosine bell and vortex wrap-up. Finally, performance benchmarks and results are presented in Section 5, followed by conclusions and proposals for future optimization strategies in Section 6.

2 Calculating RBF-FD weights

Given a set of function values, $\{u(\mathbf{x}_j)\}_{j=1}^N$, on a set of N nodes $\{\mathbf{x}_j\}_{j=1}^N$, the operator $\mathcal{L}$ acting on $u(\mathbf{x})$ evaluated at $\mathbf{x}_j$, is approximated by a weighted combination of function values, $\{u(\mathbf{x}_i)\}_{i=1}^n$, in a small neighborhood of $\mathbf{x}_j$, where $n \ll N$ defines the size of the stencil.

$$\mathcal{L}u(\mathbf{x}) \mid_{\mathbf{x}=\mathbf{x}_j} \approx \sum_{i=1}^{n} w_i u(\mathbf{x}_i) + w_{n+1} p_0 \tag{1}$$

The RBF-FD weights, w_i, are found by enforcing that they are exact within the space spanned by the RBFs $\phi_i(\epsilon r) = \phi(\epsilon\,||\mathbf{x}-\mathbf{x}_i||)$, centered at the nodes $\{\mathbf{x}_i\}_{i=1}^n$, with $r = ||\mathbf{x}-\mathbf{x}_i||$ being the distance between where the RBF is centered and where it is evaluated as measured in the standard Euclidean 2-norm. Various studies show [17, 20, 23, 58] that better accuracy is achieved when the interpolant can exactly reproduce a constant, p_0. Assuming $p_0 = 1$, the constraint $\sum_{i=1}^n w_i = \mathcal{L}1|_{\mathbf{x}=\mathbf{x}_j} = 0$ completes the system:

$$\begin{pmatrix} \phi(\epsilon\,||\mathbf{x}_1-\mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_1-\mathbf{x}_2|| & \cdots & \phi(\epsilon\,||\mathbf{x}_1-\mathbf{x}_n||) & 1 \\ \phi(\epsilon\,||\mathbf{x}_2-\mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_2-\mathbf{x}_2|| & \cdots & \phi(\epsilon\,||\mathbf{x}_2-\mathbf{x}_n||) & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ \phi(\epsilon\,||\mathbf{x}_n-\mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_n-\mathbf{x}_2|| & \cdots & \phi(\epsilon\,||\mathbf{x}_n-\mathbf{x}_n||) & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix} = \begin{bmatrix} \mathcal{L}\phi(\epsilon\,||\mathbf{x}-\mathbf{x}_1||)|_{\mathbf{x}=\mathbf{x}_j} \\ \mathcal{L}\phi(\epsilon\,||\mathbf{x}-\mathbf{x}_2||)|_{\mathbf{x}=\mathbf{x}_j} \\ \vdots \\ \mathcal{L}\phi(\epsilon\,||\mathbf{x}-\mathbf{x}_n||)|_{\mathbf{x}=\mathbf{x}_j} \\ 0 \end{bmatrix}, \tag{2}$$

where w_{n+1} is ignored after the matrix in (2) is inverted. This $n \times n$ system solve is repeated for each stencil center $\mathbf{x}_j$, $j = 1...N$, to form the N rows of the DM with n non-zeros ($n \ll N$) per row. As an example, if $\mathcal{L}$ is the identity operator, then the above procedure leads to RBF-FD interpolation. If $\mathcal{L} = \frac{\partial}{\partial x}$, one obtains the DM that approximates the first derivative in x. In the context of time-dependent PDEs, the stencil weights remain constant for all time-steps when the nodes are stationary. Therefore, the calculation of the differentiation weights is performed once in a single preprocessing step of $O(n^3 N)$ FLOPs. Improved efficiency is achieved by processing multiple right hand sides in one pass, calculating the weights corresponding to all required derivative quantities (i.e., $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, ∇^2, etc.).

For each of the N small system solves of Equation (2), the n nearest neighbors to $\mathbf{x}_j$ need to be located. This can be done efficiently using neighbor query algorithms or spatial partitioning data-structures such as

Locality Sensitive Hashing (LSH) and kD-Tree. Different query algorithms often have a profound impact on the DM structure and memory access patterns. We choose a Raster (ijk) ordering LSH algorithm [4] leading to the matrix structure in Figures 6 and 7. While querying neighbors for each stencil is an embarrassingly parallel operation, the node sets used here are stationary and require stencil generation only once. Efficiency and parallelism for this task has little impact on the overall run-time of tests, which is dominated by the time-stepping. We preprocess node sets and generate stencils serially, then load stencils and nodes from disk at run-time. In contrast to the RBF-FD view of a static grid, Lagrangian/particle based PDE algorithms promote efficient parallel variants of LSH in order to accelerate querying neighbors at each time-step [28, 41].

2.1 Hyperviscosity

For RBF-FD, differentiation matrices encode convective operators of the form

$$D = \alpha \frac{\partial}{\partial \lambda} + \beta \frac{\partial}{\partial \theta} \tag{3}$$

where α and β are a function of the fluid velocity. The convective operator, discretized through RBF-FD, has eigenvalues in the right half-plane causing the method to be unstable [17, 23]. Stabilization of the RBF-FD method is achieved through the application of a hyperviscosity filter to Equation (3) [23]. By using Gaussian RBFs, $\phi(r) = e^{-(\epsilon r)^2}$, the hyperviscosity (a high order Laplacian operator) simplifies to

$$\Delta^k \phi(r) = \epsilon^{2k} p_k(r) \phi(r) \tag{4}$$

where k is the order of the Laplacian and $p_k(r)$ are multiples of generalized Laguerre polynomials that are generated recursively (see Section 3.2 [23]). We assume a 2D Laplacian operator when working on the surface of the sphere since a local stencil can be viewed as lying on a plane. Although it is clear that when the diameter of the stencil is not sufficiently small compared to the radius of the sphere, that the hyperviscosity might not work as advertised, numerical evidence suggests that if the RBFs adequately resolve the convective operator, the 2D hyperviscosity operator does its job. We base our work on the work of Fornberg and Lehto [23], which was performed on a sphere, and tune our parameters following their guidelines.

In the case of parabolic and hyperbolic PDEs, hyperviscosity is added as a filter to the right hand side of the evaluation. For example, at the continuous level, the equation solved takes the form

$$\frac{\partial u}{\partial t} = -Du + Hu, \tag{5}$$

where D is the PDE operator, and H is the hyperviscosity filter operator. Applying hyperviscosity shifts all the eigenvalues of D to the left half of the complex plane. This shift is controlled by k, the order of the Laplacian, and a scaling parameter γ_c, defined by

$$H = \gamma \Delta^k = \gamma_c N^{-k} \Delta^k.$$

Given a choice of ϵ (see Section 4.1), it was found experimentally that $\gamma = \gamma_c N^{-k}$ provides stability and good accuracy for all values of N considered here. It also ensures that the viscosity vanishes as $N \to \infty$ [17]. In general, the larger the stencil size, the higher the order of the Laplacian. This is attributed to the fact that, for convective operators, larger stencils treat a wider range of modes accurately. As a result, the hyperviscosity operator should preserve as much of that range as possible. The parameter γ_c must also be chosen with care and its sign depends on k (for k even, γ_c will be negative and for k odd, it will be positive). If γ_c is too large, the eigenvalues move outside the stability domain of our time-stepping scheme and/or eigenvalues corresponding to lower physical modes are not left intact, reducing the accuracy of our approximation. If γ_c is too small, eigenvalues remain in the right half-plane [17, 23].

3 Targeting Multi-CPU/GPU

Parallelization of the RBF-FD method is achieved at two levels. First, the physical domain of the problem—in this case, the unit sphere—is partitioned into overlapping subdomains, each handled by a different CPU

process. All CPUs operate independently to compute/load RBF-FD stencil weights, run diagnostic tests and perform other initialization tasks. A CPU computes only weights corresponding to stencils centered in the interior of its partition. After initialization, CPUs continue concurrently to solve the problem with explicit time-stepping. Communication barriers ensure that the CPUs execute in lockstep to maintain consistent solution values in regions where partitions overlap. The second level of parallelization offloads time-stepping of the PDE to the GPU. Evaluation of the right hand side of Equation (5) is data-parallel: the solution derivative at each stencil center is evaluated independently of the other stencils. This maps well to the GPU, offering decent speedup even in unoptimized kernels. Although the stencil weight calculation is also data-parallel, we assume that in this context that the weights are precomputed and loaded once from disk during the initialization phase.

Our current implementation assumes that we are computing on a cluster of CPUs, with one GPU attached to each CPU. The CPU maintains control of execution and launches kernels on the GPU that execute in parallel. Under the OpenCL standard [35], a tiered memory hierarchy is available on the GPU with *global device memory* as the primary and most abundant memory space. The memory space for GPU kernels is separate from the memory available to a CPU, so data must be explicitly copied to/from global device memory on the GPU.

3.1 Memory Layout

After partitioning, each CPU/GPU is responsible for its own subset of nodes. To simplify accounting, we track nodes in two ways. Each node is assigned a global index, that uniquely identifies it. This index follows the node and its associated data as it is shuffled between processors. In addition, it is important to treat the nodes on each CPU/GPU in an identical manner. Implementations on the GPU are more efficient when node indices are sequential. Therefore, we also assign a local index for the nodes on a given CPU, which run from 1 to the maximum number of nodes on that CPU.

It is convenient to break up the nodes on a given CPU into various sets according to whether they are sent to other processors, are retrieved from other processors, are permanently on the processor, etc. Note as well, that each node has a home processor since the RBF nodes are partitioned into multiple domains without overlap. Table 1, defines the collection of index lists that each CPU must maintain for both multi-CPU and multi-GPU implementations.

Figure 3 illustrates a configuration with two CPUs and two GPUs, and 9 stencils, four on CPU1, and five on CPU2, separated by a vertical line in the figure. Each stencil has size $n = 5$. In the top part of the figures, the stencils are laid out with blue arrows pointing to stencil neighbors and creating the edges of a directed adjacency graph. Note that the connection between two nodes is not always bidirectional. For example, node 6 is in the stencil of node 3, but node 3 *is not* a member of the stencil of node 6. Gray arrows point to stencil neighbors outside the small window and are not relevant to the following discussion, which focuses only on data flow between CPU1 and CPU2. Since each CPU is responsible for the derivative evaluation and solution updates for any stencil center, it is clear that some nodes have a stencil with nodes that are on a different CPU. For example, node 8 on CPU1 has a stencil comprised of nodes 4,5,6,9, and itself. The data associated with node 6 must be retrieved from CPU2. Similarly, the data from node 5 must be sent to CPU2 to complete calculations at the center of node 6.

The set of all nodes that a CPU interacts with is denoted by $\mathcal{G}$, which includes not only the nodes stored on the CPU, but the nodes required from other CPUs to complete the calculations. The set $\mathcal{Q} \in \mathcal{G}$ contains the nodes at which the CPU will compute derivatives and apply solution updates. The set $\mathcal{R} = \mathcal{G} \backslash \mathcal{Q}$ is formed from the set of nodes whose values must be retrieved from another CPU. For each CPU, the set $\mathcal{O} \in \mathcal{Q}$ is sent to other CPUs. The set $\mathcal{B} \in \mathcal{Q}$ consists of nodes that depend on values from $\mathcal{R}$ in order to evaluate derivatives. Note that $\mathcal{O}$ and $\mathcal{B}$ can overlap, but differ in size, since the directed adjacency graph produced by stencil edges is not necessarily symmetric. The set $\mathcal{B} \backslash \mathcal{O}$ represents nodes that depend on $\mathcal{R}$ but are not sent to other CPUs, while $\mathcal{Q} \backslash \mathcal{B}$ are nodes that have no dependency on information from other CPUs. The middle section Figure 3 lists global node indices contained in $\mathcal{G}$ for each CPU. Global indices are paired with local indices to indicate the node ordering internal to each CPU. The structure of set $\mathcal{G}$,

$$\mathcal{G} = \{\mathcal{Q} \backslash \mathcal{B} \quad \mathcal{B} \backslash \mathcal{O} \quad \mathcal{O} \quad \mathcal{R}\}, \tag{6}$$

is designed to simplify both CPU-CPU and CPU-GPU memory transfers by grouping nodes of similar type.

$\mathcal{G}$	: all nodes received and contained on the CPU/GPU g
$\mathcal{Q}$	: stencil centers managed by g (equivalently, stencils computed by g)
$\mathcal{B}$	: stencil centers managed by g that require nodes from another CPU/GPU
$\mathcal{O}$	: nodes managed by g that are sent to other CPUs/GPUs
$\mathcal{R}$	: nodes required by g that are managed by another CPU/GPU

Table 1: Sets defined for stencil distribution to multiple CPUs

The color of the global and local indices in the figure indicate the sets to which they belong. They are as follows: white represents $\mathcal{Q}\backslash\mathcal{B}$, yellow represents $\mathcal{B}\backslash\mathcal{O}$, green indices represent $\mathcal{O}$, and red represent $\mathcal{R}$.

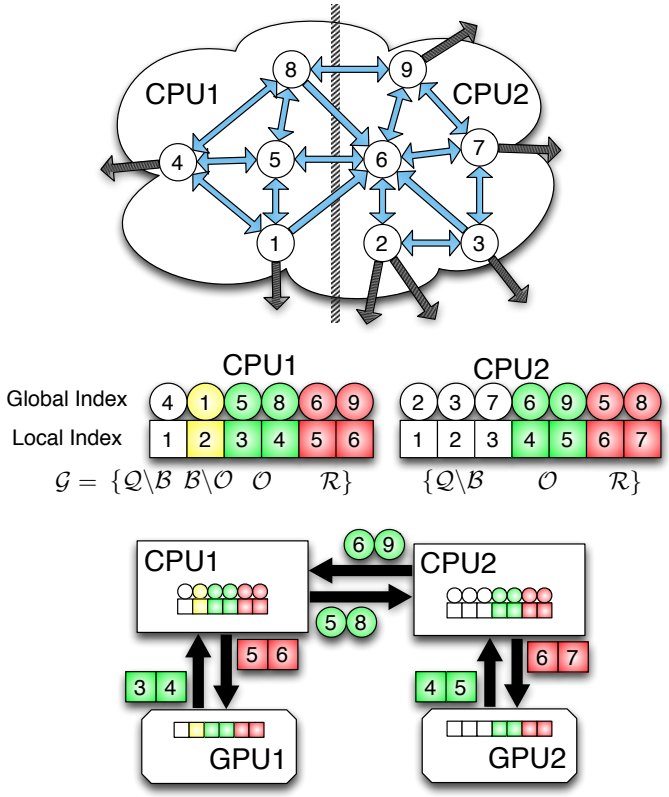


Figure 3: Partitioning, index mappings and memory transfers for nine stencils ($n = 5$) spanning two CPUs and two GPUs. Top: the directed graph created by stencil edges is partitioned for two CPUs. Middle: the partitioned stencil centers are reordered locally by each CPU to keep values sent to/received from other CPUs contiguous in memory. Bottom: to synchronize GPUs, CPUs must act as intermediaries for communication and global to local index translation. Middle and Bottom: color coding on indices indicates membership in sets from Table 1: $\mathcal{Q}\backslash\mathcal{B}$ is white, $\mathcal{B}\backslash\mathcal{O}$ is yellow, $\mathcal{O}$ is green and $\mathcal{R}$ is red.

The structure of $\mathcal{G}$ offers two benefits: first, solution values in $\mathcal{R}$ and $\mathcal{O}$ are contiguous in memory and can be copied to or from the GPU without the filtering and/or re-ordering normally required in preparation for efficient data transfers. Second, asynchronous communication allows for the overlap of communication and computation. This will be considered as part of future research on algorithm optimization. Distinguishing the set $\mathcal{B}\backslash\mathcal{O}$ allows the computation of $\mathcal{Q}\backslash\mathcal{B}$ while waiting on $\mathcal{R}$.

When targeting the GPU, communication of solution or intermediate values is a four step process:

1. Transfer $\mathcal{O}$ from GPU to CPU
2. Distribute $\mathcal{O}$ to other CPUs, receive R from other CPUs

3. Transfer $\mathcal{R}$ to the GPU

4. Launch a GPU kernel to operate on $\mathcal{Q}$

The data transfers involved in this process are illustrated at the bottom of Figure 3. Each GPU operates on the local indices ordered according to Equation (6). The set $\mathcal{O}$ is copied off the GPU and into CPU memory as one contiguous memory block. The CPU then maps local to global indices and transfers $\mathcal{O}$ to other CPUs. CPUs send only the subset of node values from $\mathcal{O}$ that is required by the destination processors, but it is important to note that node information might be sent to several destinations. As the set $\mathcal{R}$ is received, the CPU converts back from global to local indices before copying a contiguous block of memory to the GPU. This approach is scalable to a very large number of processors, since the individual processors do not require the full mapping between RBF nodes and CPUs.

Figure 4 illustrates a partitioning of $N = 10,201$ nodes on the unit sphere onto four CPUs. Each partition, illustrated as a unique color, represents set $\mathcal{G}$ for a single CPU. Alternating representations between node points and interpolated surfaces illustrates the overlap regions where nodes in sets $\mathcal{O}$ and $\mathcal{R}$ (i.e., nodes requiring MPI communication) reside. As stencil size increases, the width of the overlap regions relative to total number of nodes on the sphere also increases.

The linear partitioning in Figure 4 was chosen for ease of implementation. Communication is limited for each processor to left and right neighbors only, which simplifies parallel debugging. This partitioning, however, does not guarantee properly balanced computational work-loads. Other partitionings of the sphere exist but are not studied here because this paper's focus is neither on efficiency nor on selecting a partitioning strategy for maximum accuracy. Examples of alternative approaches include a cubed-sphere [31] or icosahedral geodesic grid [42], which can evenly balance the computational load across partitions. Other interesting partitionings can be generated with software libraries such as the METIS [34] family of algorithms, capable of partitioning and reordering directed graphs produced by RBF-FD stencils.

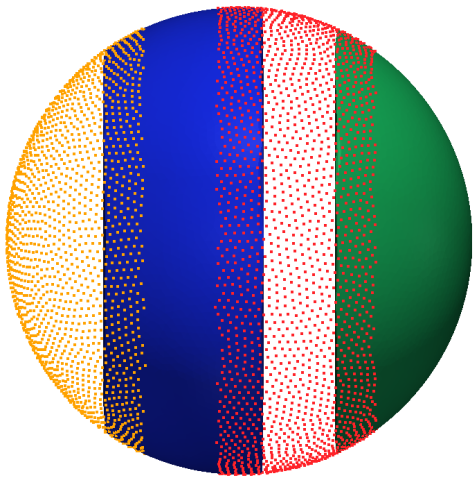


Figure 4: Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

3.2 Targeting the GPU

Our implementation leverages the GPU for acceleration of the standard fourth order Runge-Kutta (RK4) scheme. Nodes are stationary, so stencil weights are calculated once at the beginning of the simulation, and reused in every iteration. To avoid the cost of calculating stencil weights each time a test case is run, they are written to disk and loaded from file on subsequent runs. There is one set of weights computed for each new grid. Ignoring code initialization, the cost of the algorithm is simply the explicit time advancement of the solution.

Figure 5 summarizes the time advancement steps for the multi-CPU/GPU implementation. The RK4

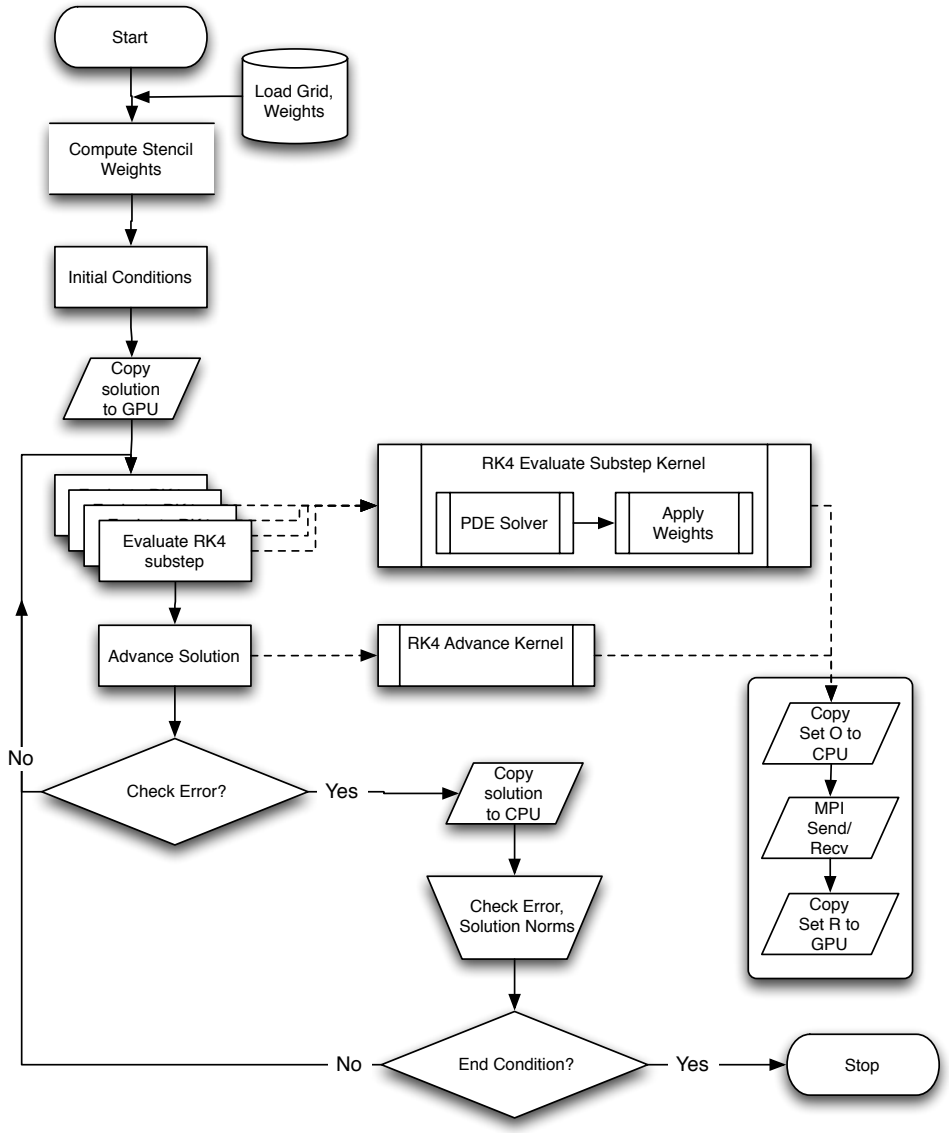


Figure 5: Workflow for RK4 on multiple GPUs.

steps are:

$$\begin{aligned}
\mathbf{k}_1 &= \Delta t f(t_n, \mathbf{u}_n) \\
\mathbf{k}_2 &= \Delta t f(t_n + \frac{1}{2}\Delta t, \mathbf{u}_n + \frac{1}{2}\mathbf{k}_1) \\
\mathbf{k}_3 &= \Delta t f(t_n + \frac{1}{2}\Delta t, \mathbf{u}_n + \frac{1}{2}\mathbf{k}_2) \\
\mathbf{k}_4 &= \Delta t f(t_n + \Delta t, \mathbf{u}_n + \mathbf{k}_3) \\
\mathbf{u}_{n+1} &= \mathbf{u}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4),
\end{aligned}$$

where each equation has a corresponding kernel launch. To handle a variety of Runga-Kutta implementations, steps $\mathbf{k}_{1\to 4}$ correspond to calls to the same kernel with different arguments. The evaluation kernel returns two output vectors:

1. $\mathbf{k}_i = \Delta t f(t_n + \alpha_i \Delta t, \mathbf{u}_n + \alpha_i \mathbf{k}_{i-1})$, for steps $i = 1, 2, 3, 4$, and

2. $\mathbf{u}_n + \alpha_{i+1}\mathbf{k}_i$

We choose $\alpha_i = 0, \frac{1}{2}, \frac{1}{2}, 1, 0$ and $\mathbf{k}_0 = \mathbf{u}_n$. The second output for each $\mathbf{k}_{i=1,2,3}$ serves as input to the next evaluation, $\mathbf{k}_{i+1}$. In an effort to avoid an extra kernel launch—and corresponding memory loads—the SAXPY that produces the second output uses the same evaluation kernel. Both outputs are stored in global device memory. When the computation spans multiple GPUs, steps $\mathbf{k}_{1\to3}$ are each followed by a communication barrier to synchronize the subsets $\mathcal{O}$ and $\mathcal{R}$ of the second output (this includes copying the subsets between GPU and CPU). An additional synchronization occurs on the updated solution, $\mathbf{u}_{n+1}$, to ensure that all GPUs share a consistent view of the solution going into the next time-step.

To evaluate $\mathbf{k}_{1\to4}$, the discretized operators from Equation (5) are applied using sparse matrix-vector multiplication. If the operator D is composed of multiple derivatives, a differentiation matrix for each derivative is applied independently, including an additional multiplication for the discretized H operator. On the GPU, the kernel parallelizes across rows of the DMs, so all derivatives for stencils are computed in one kernel call.

For the GPU, the OpenCL language [35] assumes a lowest common denominator of hardware capabilities to provide functional portability. For example, all target architectures are assumed to support some level of SIMD (Single Instruction Multiple Data) execution for kernels. Multiple *work-items* execute a kernel in parallel. A collection of work-items performing the same task is called a *work-group*. While a user might think of work-groups as executing all work-items simultaneously, the work-items are divided at the hardware level into one or more SIMD *warps*, which are executed by a single multiprocessor. On the family of Fermi GPUs, a warp is 32 work-items [40]. OpenCL assumes a tiered memory hierarchy that provides fast but small *local memory* space that is shared within a work-group [35]. Local memory on Fermi GPUs is 48 KB per multiprocessor [40]. The *global device memory* allows sharing between work-groups and is the slowest but most abundant memory. In the GPU computing literature, the terms *thread* and *shared memory* are synonymous to *work-item* and *local memory* respectively, and are preferred below.

Although the primary focus of this paper is the implementation and verification of the RBF-FD method across multiple CPUs and GPUs, we have nonetheless tested two approaches to the computation of derivatives on the GPU to assess the potential for further improvements in performance. In both cases, the stencil weights are stored in CSR format [3], a packed one-dimensional array in global memory with all the weights of a single stencil in consecutive memory addresses. Each operator is stored as an independent CSR matrix. The consecutive ordering on the weights implies that the solution vector, structured according to the ordering of set $\mathcal{G}$ is treated as random access.

All the computation on the GPU is performed in 8-byte double precision.

3.2.1 Naive Approach: One thread per stencil

In this first implementation, each thread computes the derivative at one stencil center (Figure 6). The advantage of this approach is trivial concurrency. Since each stencil has the same number of neighbors, each derivative has an identical number of computations. As long as the number of stencils is a multiple of the warp size, there are no idle threads. Should the total number of stencils be less than a multiple of the warp size, the final warp would contain idle threads, but the impact on efficiency would be minimal assuming the stencil size is sufficiently large.

Perfect concurrency from a logical point of view does not imply perfect efficiency in practice. Unfortunately, the naive approach is memory bound. When threads access weights in global memory, a full warp accesses a 128-byte segment in a single memory operation [40]. Since each thread handles a single stencil, the various threads in a warp access data in very disparate areas of global memory, rather than the same segment. This leads to very large slowdowns as extra memory operations are added for each 128-byte segment that the threads of a warp must access. However, with stencils sharing many common nodes, and the Fermi hardware providing caching, some weights in the unused portions of the segments might remain in cache long enough to hide the cost of so many additional memory loads.

3.2.2 Alternate Approach: One warp per stencil

An alternate approach, illustrated in Figure 7, dedicates a full warp of threads to a single stencil. Here, 32 threads load the weights of a stencil and the corresponding elements of the solution vector. As the 32

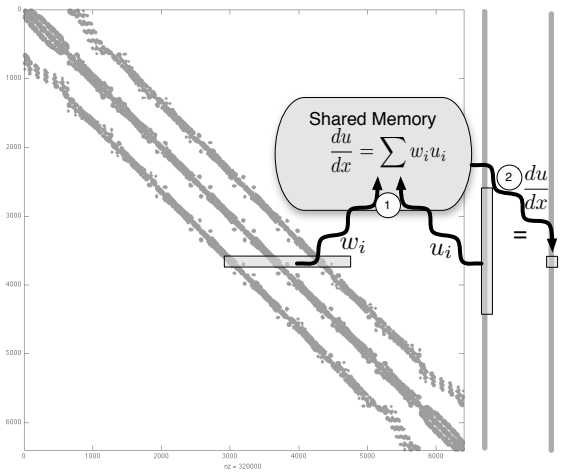


Figure 6: Naive approach to sparse matrix-vector multiply. Each thread is responsible for the sparse vector dot product of weights and solution values for derivatives at a single stencil.

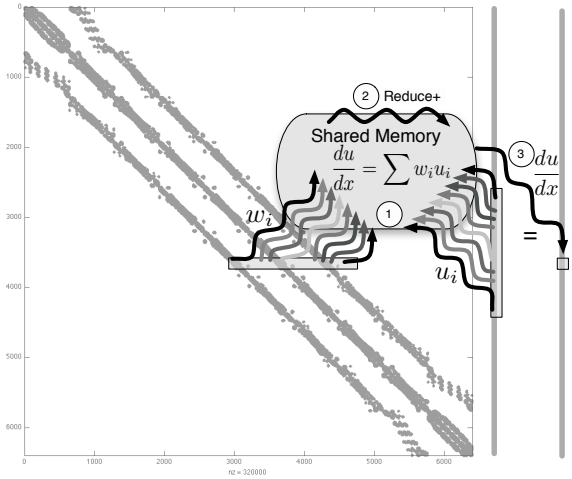


Figure 7: Alternative approach. A full warp (32 threads) collaborate to apply weights and compute the derivative at a stencil center.

threads each perform a subset of the dot product, their intermediate sums are accumulated in 32 elements of shared memory (one per thread). Should a stencil be larger than the warp size, the warp iterates over the stencil in increments of the warp size until the full dot product is complete. Finally, the first thread of the warp performs a sum reduction across the 32 (warp size) intermediate sums stored in shared memory and writes the derivative value to global memory.

By operating on a warp by warp basis, weights for a single stencil are loaded with a reduced number of memory accesses. Memory loads for the solution vector remain random access but see some benefit when solution values for a stencil are in a small neighborhood in the memory space. Proximity in memory can be controlled by node indexing (see e.g., [4] and [11]).

For stencil sizes smaller than 32, some threads in the warp always remain idle. Idle threads do not slow down the computation within a warp, but under-utilization of the GPU is not desirable. For small stencil sizes, caching on the Fermi can hide some of the cost of memory loads for the naive approach, with no idle threads, making it more efficient. The real strength of one warp per stencil is seen for large stencil sizes. As part of future work on optimization, we will consider a parallel reduction in shared memory, as well as assigning multiple stencils to a single warp for small n.

4 Numerical Validation

Here, we present the first results in the literature for parallelizing RBF-FDs on multi-CPU and multi-GPU architectures for solving PDEs. To verify our multi-CPU, single GPU and multi-GPU implementations, two hyperbolic PDEs on the surface of the sphere are tested: 1) vortex roll-up [38, 39] and 2) solid body rotation [32]. These tests were chosen since they are not only standard in the numerical literature, but also for the development of RBFs in solving PDEs on the sphere [16, 18, 22, 25]. Although any 'approximately evenly' distributed nodes on the sphere would suffice for our purposes, maximum determinant (MD) node distributions on the sphere are used (see [47] for details) in order to be consistent with previously published results (see e.g., [18] and [23]). Node sets from 1024 to 27,556 are considered with stencil sizes ranging from 17 to 101.

All results in this section are produced by the single-GPU implementation. Multi-CPU and multi-GPU implementations are verified to produce these same results. Synchronization of the solution at each time-step and the use of double precision on both the CPU and GPU ensure consistent results regardless of the number and/or choice of CPU vs GPU. Eigenvalues are computed on the CPU by the Armadillo library [43].

4.1 Vortex Rollup

The first test case demonstrates vortex roll-up of a fluid on the surface of a unit sphere. An angular velocity field causes the initial condition to spin into two diametrically opposed but stationary vortices.

The governing PDE in latitude-longitude coordinates, (θ, λ), is

$$\frac{\partial h}{\partial t} + \frac{u}{\cos\theta}\frac{\partial h}{\partial \lambda} = 0 \tag{7}$$

where the velocity field, u, only depends on latitude and is given by

$$u = \omega(\theta)\cos\theta.$$

Note that the $\cos\theta$ in u and $1/\cos\theta$ in (7) cancel in the analytic formulation, so the discrete operator approximates $\omega(\theta)\frac{\partial}{\partial\lambda}$.

Here, $\omega(\theta)$ is the angular velocity component given by

$$\omega(\theta) = \begin{cases} \frac{3\sqrt{3}}{2\rho(\theta)}\text{sech}^2(\rho(\theta))\tanh(\rho(\theta)) & \rho(\theta) \neq 0 \\ 0 & \rho(\theta) = 0 \end{cases}$$

where $\rho(\theta) = \rho_0\cos\theta$ is the radial distance of the vortex with $\rho_0 = 3$. The exact solution to (7) at non-dimensional time t is

$$h(\lambda, \theta, t) = 1 - \tanh\left(\frac{\rho(\theta)}{\gamma}\sin(\lambda - \omega(\theta)t)\right),$$

where γ defines the width of the frontal zone.

From a method of lines approach, the discretized version of (7) is

$$\frac{d\mathbf{h}}{dt} = -\text{diag}(\omega(\theta))D_\lambda\mathbf{h}. \tag{8}$$

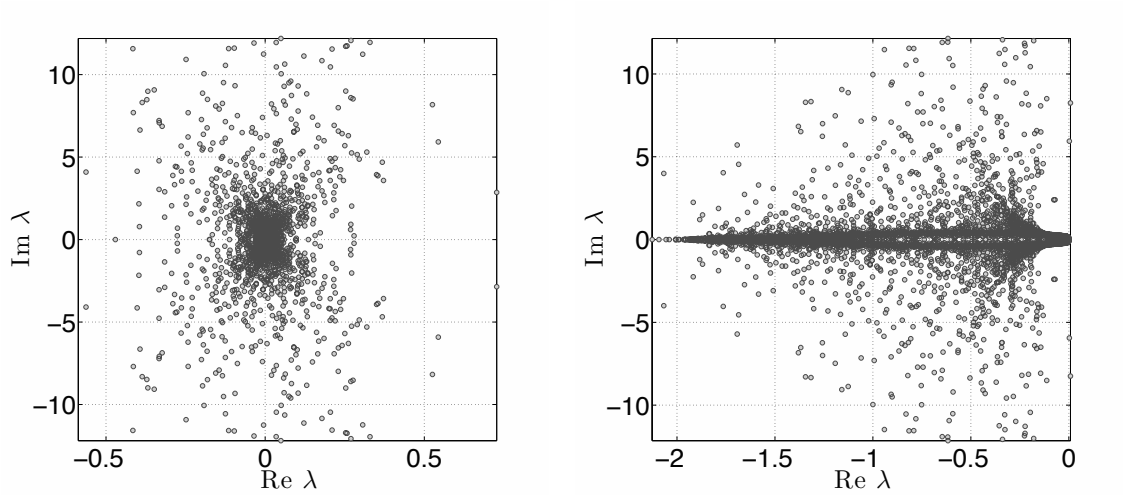
where D_λ is the DM containing the RBF-FD weights that approximate $\frac{\partial}{\partial\lambda}$ at each node on the sphere.

For stability, hyperviscosity is added to the right hand side of (8) in the form given in (5). The scaling parameter γ_c and the order of hyperviscosity k are given in Table 2. The goal when choosing k is to damp the higher spurious eigenmodes of $\text{diag}(\omega(\theta))D_\lambda$ while leaving the lower physical modes that can be resolved by the stencil intact. In this process, the eigenvalues will be pushed into the left half of the complex plane. Then, γ_c is used to condense the eigenvalues as near to the imaginary axis as possible. Figure 8(b) shows the effect of hyperviscosity on the eigenvalues of the DM, $-diag(\omega(\theta))D_\lambda$, in (8).

In order to scale to large node sets, the RBF shape parameter, ϵ, is chosen such that the mean condition number of the local RBF interpolation matrices $\bar{\kappa}_A = \frac{1}{N}\sum_{j=1}^{N}(\kappa_A)_j$ is kept constant as N increases ($(\kappa_A)_j$ is the condition number of the interpolation matrix in (2), representing the j^{th} stencil). For a constant mean

Table 2: Values for hyperviscosity and the RBF shape parameter ϵ for vortex roll-up test.

	$\epsilon = c_1\sqrt{N} - c_2$		$H = -\gamma_c N^{-k}\Delta^k$	
Stencil Size (n)	c_1	c_2	k	γ_c
17	0.026	0.08	2	8
31	0.035	0.1	4	800
50	0.044	0.14	4	145
101	0.058	0.16	4	40



(a) No Hyperviscosity

(b) With Hyperviscosity

Figure 8: Eigenvalues of $\text{diag}(\omega(\theta))D_\lambda$ for the vortex roll-up test case for $N = 4096$ nodes, stencil size $n = 101$ and $\epsilon = 3.5$. Left: no hyperviscosity. Right: hyperviscosity enabled with $k = 4$ and $\gamma_c = 40$.

condition number, ϵ varies linearly with $\sqrt{N}$ (see [17] Figure 4a and b). This is not surprising since the condition number strongly depends on the quantity ϵr, where $r \sim 1/\sqrt{N}$ on the sphere. Thus, to obtain a constant condition number, we let $\epsilon(N) = c_1\sqrt{N} - c_2$, where c_1 and c_2 are constants based on [17].

Figure 9 shows the solution to Equation (7) at $t = 10$, on $N = 10201$ nodes, with stencil size $n = 50$. This resolution is sufficient to properly capture the vortices at $t = 10$, but lower resolutions would suffer approximation errors associated with insufficient grid resolution. For this reason, the solution at $t = 3$ is considered in the normalized ℓ_2 error convergence study presented in Figure 10. The time step $\Delta t = 0.05$ for all resolutions.

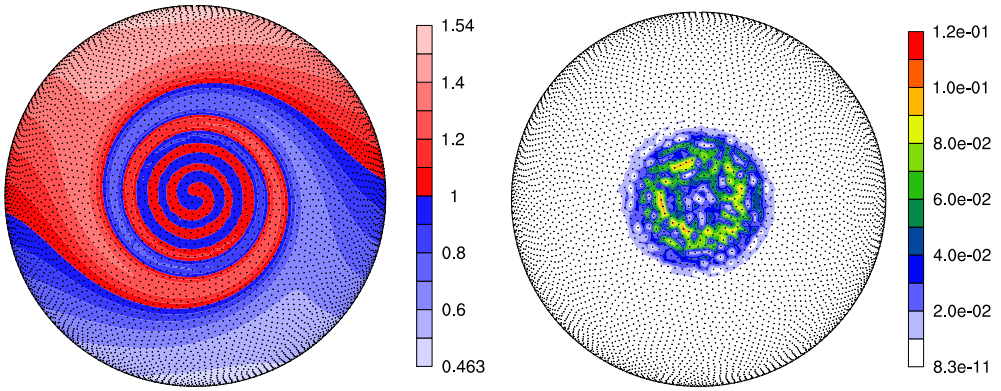
4.2 Solid body rotation

The second test case simulates the advection of a cosine bell over the surface of a unit sphere at an angle α relative to the pole of a standard latitude-longitude grid. The governing PDE is

$$\frac{\partial h}{\partial t} + \frac{u}{\cos\theta}\frac{\partial h}{\partial \lambda} + v\frac{\partial h}{\partial \theta} = 0, \tag{9}$$

with velocity field,

$$\begin{cases} u = u_0(\cos\theta\cos\alpha + \sin\theta\cos\lambda\sin\alpha), \\ v = -u_0(\sin\lambda\sin\alpha) \end{cases} .$$



(a) Computed Solution (b) Relative Absolute Error

Figure 9: Vortex roll-up solution at time $t = 10$ using RBF-FD with $N = 10,201$ and $n = 50$ point stencil. Normalized ℓ_2 error of solution at $t = 10$ is $1.25(10^{-2})$

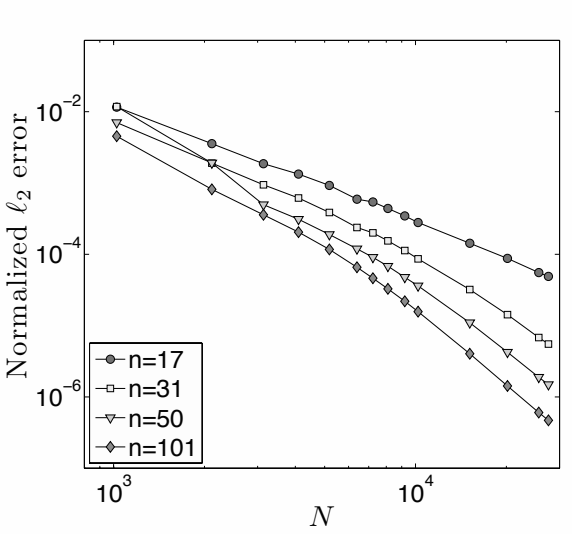


Figure 10: Convergence plot for vortex roll-up at $t = 3$.

inclined at an angle α relative to the polar axis and velocity $u_0 = 2\pi/(1036800 \text{ seconds})$ to require 12 days per revolution of the bell as in [18, 38].

The discretized form of (9) is

$$\frac{d\mathbf{h}}{dt} = -\text{diag}\left(\frac{u}{\cos\theta}\right) D_\lambda \mathbf{h} - \text{diag}(v) D_\theta \mathbf{h} \tag{10}$$

where DMs D_λ and D_θ contain RBF-FD weights corresponding to all N stencils that approximate $\frac{\partial}{\partial\lambda}$ and $\frac{\partial}{\partial\theta}$ respectively. Rather than merge the differentiation matrices in (10) into one operator, our implementation evaluates them as two sparse matrix-vector multiplies. The separate matrix-vector multiplies are motivated by an effort to provide general and reusable GPU kernels. Additionally, they artificially increase the amount of computation compared to the vortex roll-up test case to simulate cases when operators cannot be merged into one DM (e.g., a non-linear PDE).

By splitting the DM, the singularities at the poles ($1/\cos\theta \to \infty$ as $\theta \to \pm\frac{\pi}{2}$) in (9) remain. However, in this case, the approach functions without amplification of errors because the MD node sets have nodes near, but not on, the poles. As noted in [18, 23], applying the entire spatial operator to the right hand side

of Equation 2 generates a single DM that analytically removes the singularities at poles.

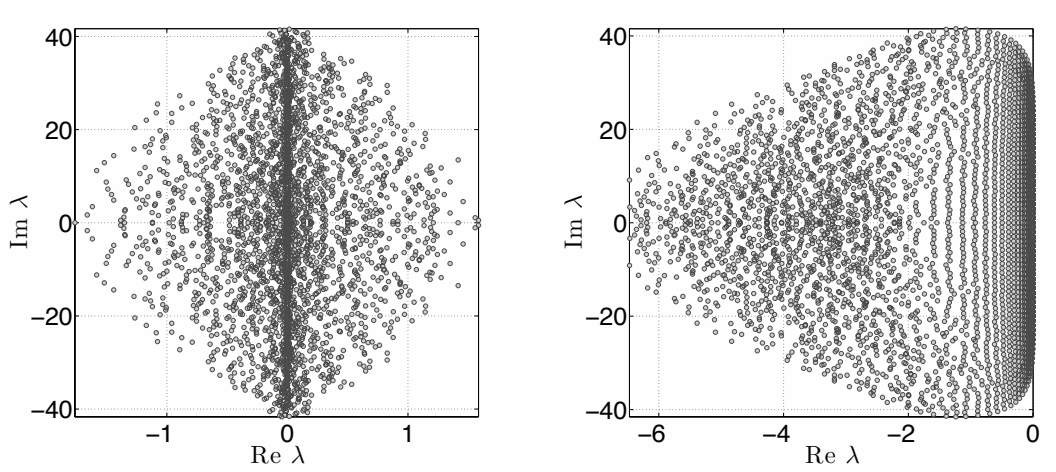
We will advect a C^1 cosine bell height-field given by

$$h = \begin{cases} \frac{h_0}{2}(1 + \cos(\frac{\pi\rho}{R})) & \rho \leq R \\ 0 & \rho \geq R \end{cases}$$

having a maximum height of $h_0 = 1$, a radius $R = \frac{1}{3}$ and centered at $(\lambda_c, \theta_c) = ({}^{3\pi}/_2, 0)$, with $\rho = \arccos(\sin\theta_c \sin\theta + \cos\theta_c \cos\theta \cos(\lambda - \lambda_c))$. The angle of rotation, $\alpha = {}^{\pi}/_2$, is chosen to transport the bell over the poles of the coordinate system.

Table 3: Values for hyperviscosity and RBF shape parameter for the cosine bell test.

	$\epsilon = c_1\sqrt{N} - c_2$		$H = -\gamma_c N^{-k}\Delta^k$	
Stencil Size (n)	c_1	c_2	k	γ_c
17	0.026	0.08	2	$8 * 10^{-4}$
31	0.035	0.1	4	$5 * 10^{-2}$
50	0.044	0.14	6	$5 * 10^{-1}$
101	0.058	0.16	8	$5 * 10^{-2}$



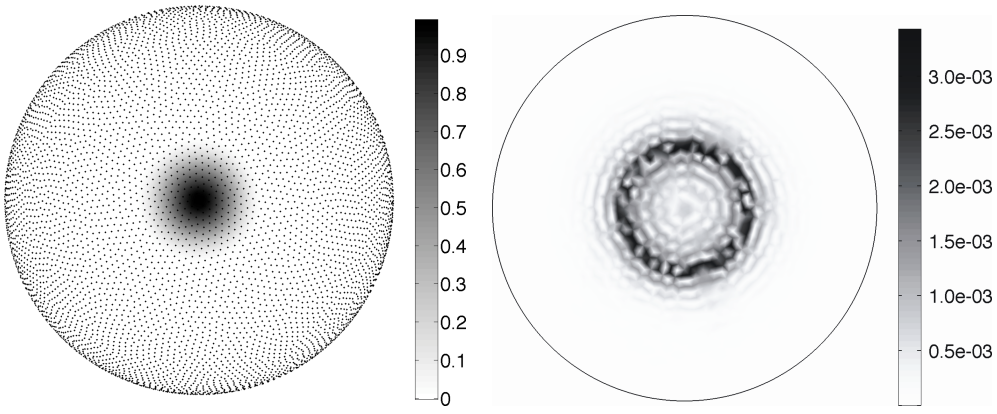
(a) No Hyperviscosity

(b) With Hyperviscosity

Figure 11: Eigenvalues of (10) for the cosine bell test case with $N = 4096$ nodes, stencil size $n = 101$, and $\epsilon = 3.5$. Left: no hyperviscosity. Right: hyperviscosity enabled with $k = 8$ and $\gamma_c = 5 * 10^{-2}$. Eigenvalues are divided by u_0 to remove scaling effects of velocity.

Figure 11 compares eigenvalues of the DM for $N = 4096$ nodes and stencil size $n = 101$ before and after hyperviscosity is applied. To avoid scaling effects of velocity on the eigenvalues, they have been scaled by $1/u_0$. The same approach as in the vortex roll-up case is used to determine the parameters for hyperviscosity and ϵ. Our tuned parameters are presented in Table 3.

Figure 12 shows the cosine bell transported ten full revolutions around the sphere. Without hyperviscosity, RBF-FD cannot complete a single revolution of the bell before instability takes over. However, adding hyperviscosity allows computation to extend to dozens or even thousands of revolutions and maintain stability (e.g., see [23]). After ten revolutions, the cosine bell is still intact. The majority of the absolute error (Figure 12(b)) appears at the base of the C^1 bell where the discontinuity appears in the derivative. At ten revolutions, Figure 13 illustrates the convergence of the RBF-FD method. All tests in Figure 13 assume 1000 time-steps per revolution (i.e., $\Delta t = 1036.8$ seconds).



(a) 10 Revolutions (b) Absolute Error at 10 revolutions

Figure 12: Cosine bell solution after 10 revolutions with $N = 10201$ nodes and stencil size $n = 101$. Hyperviscosity parameters are $k = 8$, $\gamma_c = 5(10^{-2})$.

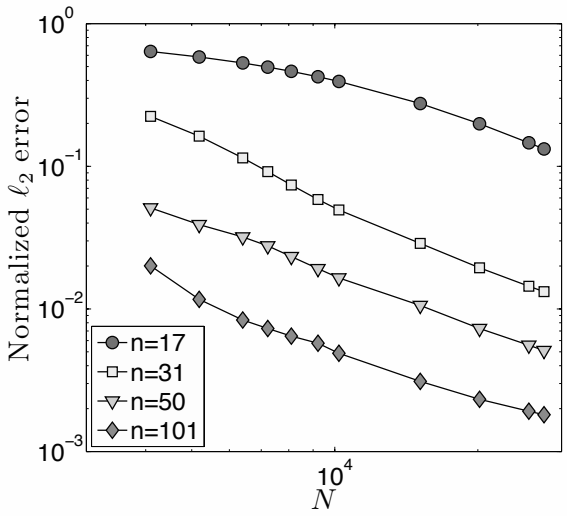


Figure 13: Convergence plot for cosine bell advection. Normalized ℓ_2 error at 10 revolutions with hyperviscosity enabled.

5 Performance Benchmarks

In this section, we present performance results under a variety of conditions: two different architectures, varying stencil sizes, and different grid resolutions. We also present some timings that break down the contributions from computation and communication between GPU and CPU, as well as between CPUs.

All timings correspond to the average run-time of a full RK4 time-step. Most of our results use speedup as a measure of comparison. Here, speedup is the ratio of serial execution time on a single CPU to parallel execution time. Benchmarks for single- and multi-GPU implementations are measured on the CPU to include all costs involved in the time-step, including the cost of updating GPU memory (if necessary), setting kernel parameters, and the latency in launching kernels. While we made some effort to improve the performance on the GPU, several optimization ideas were left unimplemented because they are outside the scope of this paper. Improved results that derive from further optimization will be presented elsewhere.

We ran the benchmarks on two architectures. The first, `troi`, represents a typical workstation that most

of the RBF community would develop and test on. `troi` has two quad-core Intel Xeon 5570 (2.93GHz) processors, 8MB cache shared between four cores, 12GB of system memory (RAM) and a NVidia GTX480 GPU with 1.5GB global memory and 15 multiprocessors. The second system is the Keeneland Initial Delivery (KID) system, a supercomputing cluster with 240 twelve-core Intel Xeon 5660 (2.80GHz) CPUs, 12MB cache per core shared between twelve cores, 24GB system memory, and 360 NVidia M2070 GPUs with 6GB of device memory and 14 multiprocessors each [51]. A high level layout of the Keeneland system is presented in Figure 14. The system was designed with its 240 CPUs partitioned into sets of two CPUs per compute node, four nodes per chassis, and six chassis per rack. An InfiniBand QDR Network handles inter-node communication. In addition to the dual CPUs, each node has three NVidia M2070 Fermi class GPUs.

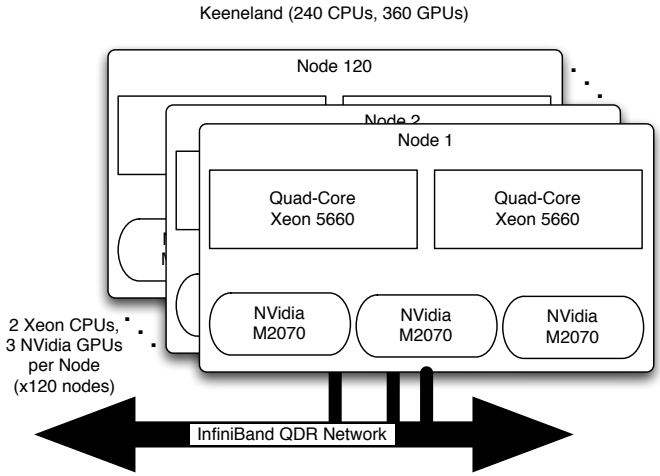


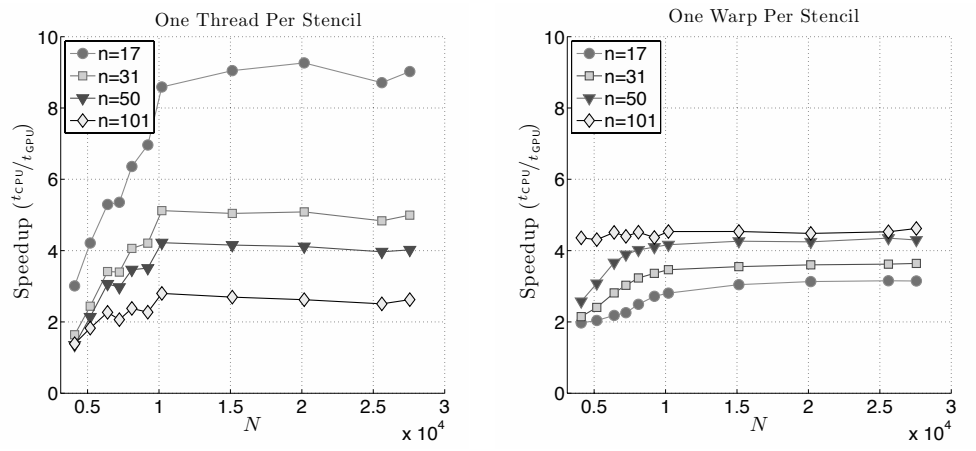
Figure 14: The Keeneland Initial Delivery (KID) hardware layout.

The Message Passing Interface (MPI) manages process creation and communication; all tests run under the OpenMPI implementation of the MPI standard [27]. The open source Armadillo library [43] is employed to solve the small linear systems in Equation (1) for stencil weights and compute eigenvalues for hyperviscosity tuning. Armadillo is a linear algebra package written in C++ that provides a templated front-end to optimized routines from BLAS, LAPACK and/or high performance replacements such as the Intel Math Kernel Library (MKL). Currently, Armadillo has no support sparse matrices, but the library enables high-level coding of linear algebra reminiscent of MATLAB. In addition, optimized back-end routines run in parallel on multiple CPU cores, which allows full utilization of available hardware.

Binaries on `troi` were built using the GNU gcc v4.4.3 compiler with the "-O2" optimization flag. Armadillo on `troi` is a front-end to LAPACK and BLAS routines. Binaries on Keeneland were built with the Intel icpc v12.0.4 compiler together with the "-O2" optimization flag. Armadillo wraps the Intel Math Kernel Library (MKL), which includes highly optimized/threaded replacements for the BLAS and LAPACK routines.

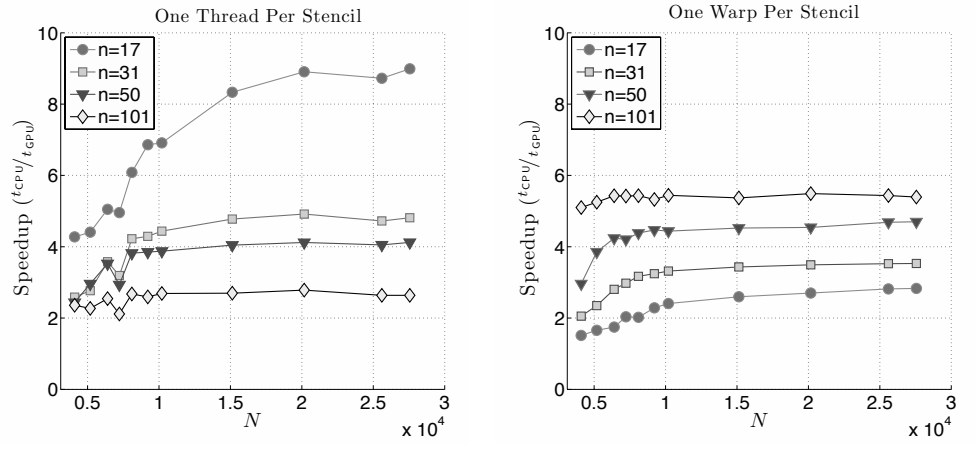
A nested loop performs the sparse matrix vector multiplication in our CPU implementation. In the following tests, compiler auto-optimizations such as loop unrolling, auto-vectorization, etc., improve the efficiency of the loops. However, alternatives will also be investigated to improve the performance of the CPU evaluation as we continue to improve the GPU performance. For example, the OSKI project [52] provides efficient, auto-tuned, drop-in replacements for sparse matrix-vector multiply in codes that already have matrices in CSR format. The Eigen [29] and BOOST::uBLAS [1] projects also provide drop-in replacements but lack auto-tuning capabilities.

All calculations on the GPU execute in double precision, which is crucial since the stencil weights must be in double precision. To verify this assertion, we computed the weights in double precision on the CPU and cast them to single precision prior to transfer to the GPU. In single precision, weights for many of the stencils summed to $O(10^{-4})$, a poor approximation to zero, which translates into a poor approximation of the derivatives. Additionally, casting the weights to single precision perturbs the eigenvalues of the differentiation matrix, which requires tuning new parameters for hyperviscosity to maintain stability.



(a) Vortex Roll-up By Thread

(b) Vortex Roll-up By Warp



(c) Cosine Bell Advection By Thread

(d) Cosine Bell Advection By Warp

Figure 15: Speedup of the vortex roll-up test case achieved on a single GPU with respect to a single CPU on `troi` (GTX480, GNU gcc compiler).

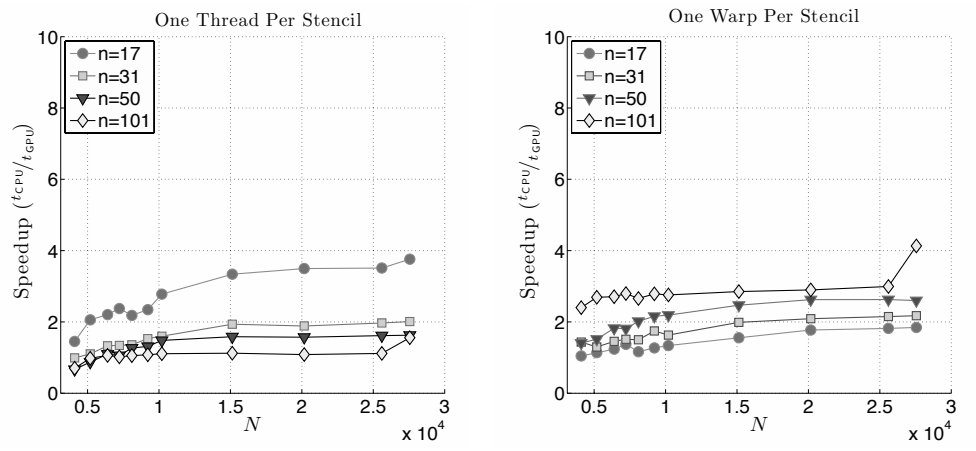
In the following test cases, processor affinity is enforced via the MPI command line arguments, "--mca mpi_paffinity_alone 1" . This ensures that execution proceeds without migration between CPUs. Our current implementation associates one GPU with each CPU. Tests are distributed such that each MPI process runs on an independent compute node on Keeneland, and therefore an independent CPU. This illustrates worst case communication costs by forcing MPI communication between CPUs/GPUs to route over the slowest interconnect.

5.1 One GPU vs One CPU

Figure 15 considers the speedup achieved by a single GPU with respect to a single CPU core on `troi`. In an un-optimized state, it is apparent that the choice of operating by thread or warp should depend on the stencil size.

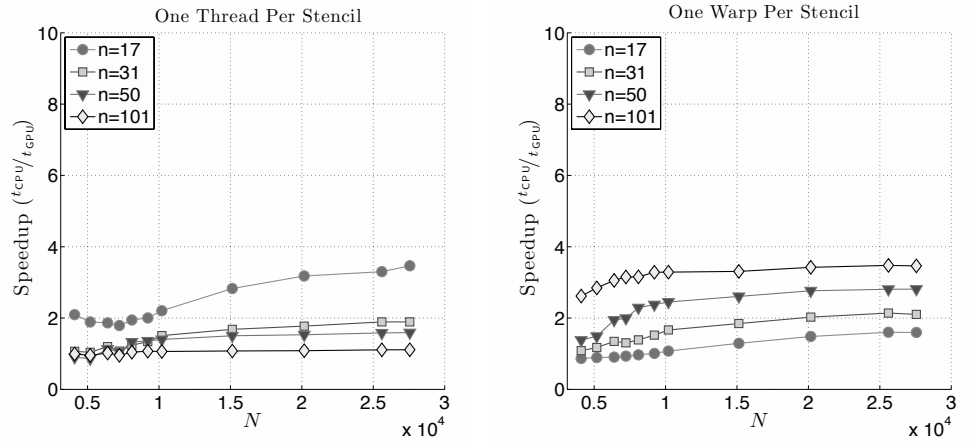
From Figures 15(a) and 15(c), we see promising results for the smallest stencil size, $n = 17$, and one thread per stencil with roughly 9x speedup in both the vortex roll-up and cosine bell advection. However, as anticipated, the naive implementation offers diminishing returns as stencil sizes increase and the kernel is increasingly memory bound.

Figures 15(b) and 15(d) show that a warp of threads dedicated to a large stencil size (e.g., $n = 101$) can



(a) Vortex Roll-up By Thread

(b) Vortex Roll-up By Warp



(c) Cosine Bell Advection By Thread

(d) Cosine Bell Advection By Warp

Figure 16: Speedup achieved on a single GPU with respect to a single CPU on the Keeneland Initial Delivery (KID) system (`keeneland.nics.utk.edu`). (NVidia M2070, Intel icc compiler)

achieve 2x or higher speedup over the one thread per stencil approach. We also note a higher speedup for large stencil sizes when comparing Figure 15(b) and Figure 15(d). This is attributed to the added work of calculating a second derivative for cosine bell advection as compared to just a single derivative in the vortex roll-up test cases. While its true that an extra derivative requires more memory loads for the second set of weights, some of the cost is amortized as the warp of threads load weights in 128-byte segments. Also, the solution values that are random access for the first derivative calculation are cached for use in the second derivative calculation.

Figures 16(a) to 16(d) display the speedup achieved by a single GPU on the Keeneland system. Due to differences in GPU hardware and the efficiency of auto-optimizations in the CPU code, speedup factors decrease in comparison to the promising results from Figure 15, but 3x speedup is still achieved by the un-optimized kernels. Once again, additional speedup is gained by increasing the number of derivatives for the one warp per stencil implementation (Figures 16(b) and 16(d)).

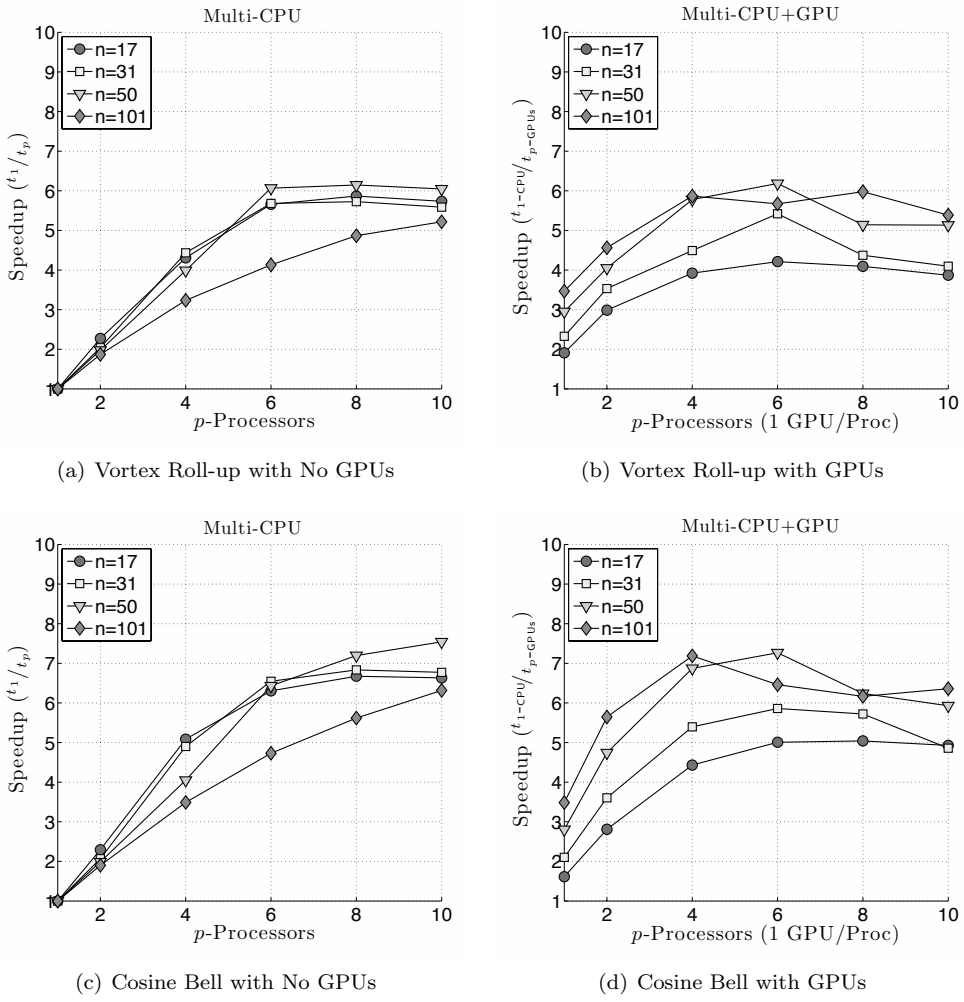
As a rule of thumb for $n > 32$ on both `troi` and Keeneland, our results demonstrate that one should operate with a full warp of threads per stencil to achieve the best gains. Future optimizations, such as merging multiple small stencils into one warp and performing reductions in parallel, will attempt to make one warp per stencil the default kernel for all stencil sizes.

5.2 Multi-CPU/GPU vs One CPU

Figure 17 demonstrates the strong scalability of our implementation across multiple CPUs and compares this to running on multiple GPUs. Strong scalability checks the change in solution time with respect to a fixed problem size and a variable number of processors. As the number of processors increases, the number of floating point operations decreases relative to the amount of inter-processor communication. In each case, a worst-case communication scenario is assumed where each process resides on an independent node. This implies that communication between processes must traverse the slowest interconnect in Keeneland: the InfiniBand QDR network. In each case, the $N = 27,556$ MD node set is partitioned into vertical slices as shown in Figure 4.

For a problem size of 27,556 nodes, the multi-CPU scales well up to 6 processors for all stencil sizes, achieving speedup factors between 4x and 7x. When the problem size reduces below four or five thousand nodes per processor, Figures 17(a) and 17(c) show diminishing returns. The break-even scaling point would be higher for a larger problem size.

Figures 17(b) and 17(d) test scaling for multiple GPUs and the one warp per stencil GPU kernel. We conclude that, for the chosen value of N, running on more than four GPUs produces no gain in efficiency.



(a) Vortex Roll-up with No GPUs

(b) Vortex Roll-up with GPUs

(c) Cosine Bell with No GPUs

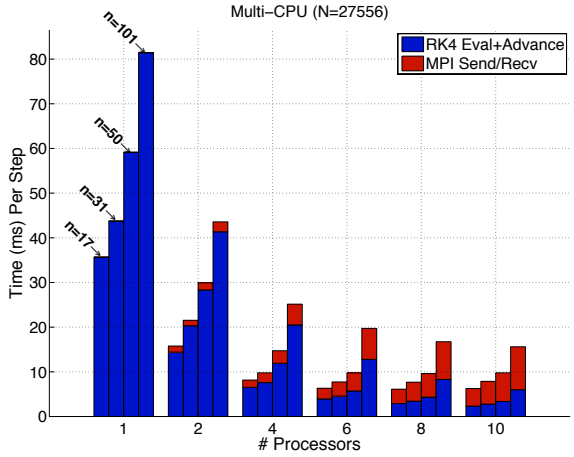
(d) Cosine Bell with GPUs

Figure 17: Speedup (relative to time on a single CPU) of the vortex roll-up and cosine bell test cases when spanning mutiple CPUs (a, c) and multiple GPUs (b, d). In all cases, $N = 27,556$.

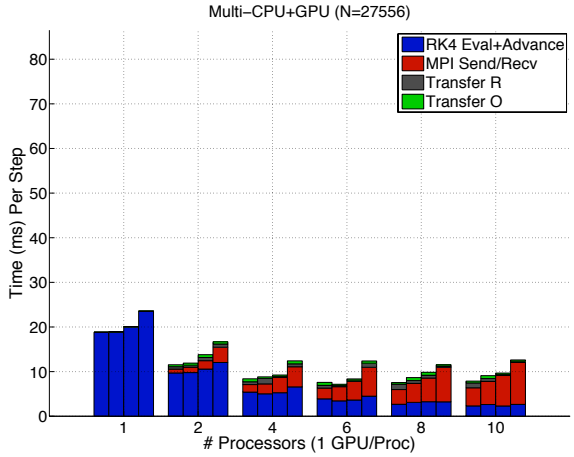
Figures 18(a) and 18(b) better illustrate the point at which the multi-CPU and multi-GPU implementa-

tions become dominated by inter-processor communication. The red bars in Figure 18(a) represent the wall clock time spent communicating (CPU to CPU) via MPI during a single time step. Computation time is shown in blue. We find that communication time gains dominance at six to eight CPUs.

The GPU accelerated compute times are shown in Figure 18(b). Here, the transfers of $\mathcal{O}$ and $\mathcal{R}$ (represented as green and gray bars respectively) occur every time there is MPI communication; so the true cost of synchronization is the sum of the three components. Given the extra synchronization overhead and faster compute times, we find that communication overtakes computation for more than four GPUs.



(a) Multiple CPUs



(b) Multiple GPUs (One Warp Per Stencil)

Figure 18: Wall clock time (in ms) for one time-step of vortex roll-up test case for various stencil sizes and with $N = 27,556$. The time-step is summation of time spent in computation (blue), MPI communication (red), solution transfer between CPU to GPU (gray) and back (green).

6 Conclusion

This paper presents the first known parallel implementation of the RBF-FD method for PDEs. Parallelization is achieved at two levels:

1. The problem geometry is partitioned and distributed across multiple CPUs, with common nodes of overlapping partitions synchronized by MPI communication.

2. Derivative approximations and explicit time-stepping are further parallelized on concurrent GPU hardware, with one GPU supplementing each CPU involved in computation.

Our MPI-based multi-CPU and multi-GPU implementations are tested for correctness by verifying stable and accurate convergence of two convective PDEs on the sphere.

Two GPU kernels, implemented in OpenCL, illustrate that the RBF-FD stencil size dictates the choice of parallelization strategy that achieves the most gain on the GPU. For small stencil sizes ($n < 32$), dedicating a single hardware thread to each stencil is sufficient; larger stencil sizes can benefit from additional parallelization of a full warp of 32 threads collaborating on each stencil.

Even in the un-optimized GPU kernels used here, speedup factors of 2-3x are realized on Keeneland, a production level cluster. For a typical workstation configuration, factors up to 9x are achieved for small stencil sizes.

Our multi-CPU/GPU implementations demonstrate strong scaling with up to 7x speedup over a single CPU for the largest problem size considered here. With access to 240 CPUs and 360 GPUs on Keeneland, scaling is important. For the small problem sizes tested here, near linear scaling of multiple CPUs is achieved on fewer than 6 processors. Multiple GPUs reduce this number further as the cost of communication increases and the time to compute decreases. Much larger problems (i.e., millions or billions of nodes) are necessary to leverage all available processors. Before optimizing GPU kernels, it is necessary to decrease/amortize the cost of MPI communication. In anticipation for this, future implementations will overlap communication and computation. Our local index mapping already provides the contiguous block of stencil indices that do not depend on MPI communication and can be evaluated without waiting on other processors. Additionally, future investigation into better node set partitioning could help balance the overlap between communication and computation.

Certainly, additional effort is necessary to realize fully optimized GPU kernels. Details such as memory alignment, idle threads, and additional use of shared memory will all be significant to improve efficiency within our memory bound kernels.

7 Acknowledgements

This work is supported by NSF awards DMS-#0934331 (FSU), DMS-#0934317 (NCAR) and ATM-#0602100 (NCAR).

Many thanks to Bengt Fornberg, Grady Wright, Kiran Katta, Ian Johnson, Steve Henke and Joseph Lohmeier for helpful discussion and insight.

References

[1] *BOOST C++ Libraries*. `http://www.boost.org`.

[2] AccelerEyes. *Jacket User Guide - The GPU Engine for MATLAB*, 1.2.1 ed., November 2009.

[3] Bell, N., and Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, 1 (2009), 1.

[4] Bollig, E. F. Fast neighbor queries and other optimization strategies for efficient radial basis function pde methods. *in preparation* (2011).

[5] Brandstetter, A., and Artusi, A. Radial Basis Function Networks GPU Based Implementation. *IEEE Transaction on Neural Network 19*, 12 (December 2008), 2150–2161.

[6] Buhmann, M. D. *Radial Basis Functions: Theory and Implementations*. Cambridge University Press, 2003.

[7] Carr, J. C., Beatson, R. K., McCallum, B. C., Fright, W. R., McLennan, T. J., and Mitchell, T. J. Smooth Surface Reconstruction from Noisy Range Data. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (New York, NY, USA, 2003), ACM, pp. 119–ff.

[8] Cecil, T., Qian, J., and Osher, S. Numerical Methods for High Dimensional Hamilton-Jacobi Equations Using Radial Basis Functions. *JOURNAL OF COMPUTATIONAL PHYSICS 196* (2004), 327–347.

[9] Chandhini, G., and Sanyasiraju, Y. Local RBF-FD Solutions for Steady Convection-Diffusion Problems. *International Journal for Numerical Methods in Engineering 72*, 3 (2007).

[10] Chinchapatnam, P. P., Djidjeli, K., Nair, P. B., and Tan, M. A compact RBF-FD based meshless method for the incompressible Navier–Stokes equations. *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment 223*, 3 (Mar. 2009), 275–290.

[11] Connor, M., and Kumar, P. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics 16* (2010), 599–608.

[12] Corrigan, A., and Dinh, H. Computing and Rendering Implicit Surfaces Composed of Radial Basis Functions on the GPU. *International Workshop on Volume Graphics* (2005).

[13] Cuntz, N., Leidl, M., Darmstadt, T., Kolb, G., Salama, C., Böttinger, M., Klimarechenzentrum, D., and Hamburg, G. GPU-based Dynamic Flow Visualization for Climate Research Applications. *Proc. SimVis* (2007), 371–384.

[14] Divo, E., and Kassab, A. An Efficient Localized Radial Basis Function Meshless Method for Fluid Flow and Conjugate Heat Transfer. *Journal of Heat Transfer 129* (2007), 124.

[15] Flyer, N., and Fornberg, B. Radial basis functions: Developments and applications to planetary scale flows. *Computers & Fluids 46*, 1 (July 2011), 23–32.

[16] Flyer, N., and Lehto, E. Rotational transport on a sphere: Local node refinement with radial basis functions. *Journal of Computational Physics 229*, 6 (Mar. 2010), 1954–1969.

[17] Flyer, N., Lehto, E., Blaise, S., Wright, G. B., and St-Cyr, A. Rbf-generated finite differences for nonlinear transport on a sphere: shallow water simulations. *Submitted to Elsevier* (2011), 1–29.

[18] Flyer, N., and Wright, G. B. Transport schemes on a sphere using radial basis functions. *Journal of Computational Physics 226*, 1 (2007), 1059 – 1084.

[19] Flyer, N., and Wright, G. B. A Radial Basis Function Method for the Shallow Water Equations on a Sphere. In *Proc. R. Soc. A* (December 2009), vol. 465, pp. 1949–1976.

[20] Fornberg, B., Driscoll, T., Wright, G., and Charles, R. Observations on the behavior of radial basis function approximations near boundaries. *Computers & Mathematics with Applications 43*, 3-5 (Feb. 2002), 473–490.

[21] Fornberg, B., and Flyer, N. Accuracy of Radial Basis Function Interpolation and Derivative Approximations on 1-D Infinite Grids. *Adv. Comput. Math 23* (2005), 5–20.

[22] Fornberg, B., Larsson, E., and Flyer, N. Stable Computations with Gaussian Radial Basis Functions. *SIAM J. on Scientific Computing 33*, 2 (2011), 869—-892.

[23] Fornberg, B., and Lehto, E. Stabilization of RBF-generated finite difference methods for convective PDEs. *Journal of Computational Physics 230*, 6 (Mar. 2011), 2270–2285.

[24] Fornberg, B., and Piret, C. A Stable Algorithm for Flat Radial Basis Functions on a Sphere. *SIAM Journal on Scientific Computing 30*, 1 (2007), 60–80.

[25] Fornberg, B., and Piret, C. On Choosing a Radial Basis Function and a Shape Parameter when Solving a Convective PDE on a Sphere. *Journal of Computational Physics 227*, 5 (2008), 2758 – 2780.

[26] Fornberg, B., and Wright, G. Stable computation of multiquadric interpolants for all values of the shape parameter. *Computers & Mathematics with Applications 48*, 5-6 (2004), 853 – 867.

[27] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.

[28] Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2010), SCA '10, Eurographics Association, pp. 55–64.

[29] Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[30] Hardy, R. Multiquadratic Equations of Topography and Other Irregular Surfaces. *J. Geophysical Research*, 76 (1971), 1–905.

[31] Ivan, L., Sterck, H. D., Northrup, S. A., and Groth, C. P. T. Three-Dimensional MHD on Cubed-Sphere Grids: Parallel Solution-Adaptive Simulation Framework. In *20th AIAA CFD Conference* (2011), no. 3382, pp. 1325–1342.

[32] Jakob-Chien, R., Hack, J., and Williamson, D. Spectral transform solutions to the shallow water test set. *Journal of Computational Physics 119*, 1 (1995), 164–187.

[33] Kansa, E. J. Multiquadrics–A scattered data approximation scheme with applications to computational fluid-dynamics. I. Surface approximations and partial derivative estimates. *Computers Math. Applic*, 19 (1990), 127–145.

[34] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1999), 359–392.

[35] Khronos OpenCL Working Group. *The OpenCL Specification (Version: 1.0.48)*, October 2009.

[36] Kosec, G., and Šarler, B. Solution of thermo-fluid problems by collocation with local pressure correction. *International Journal of Numerical Methods for Heat & Fluid Flow 18* (2008).

[37] Larsson, E., and Fornberg, B. A Numerical Study of some Radial Basis Function based Solution Methods for Elliptic PDEs. *Comput. Math. Appl 46* (2003), 891–902.

[38] Nair, R., Thomas, S., and Loft, R. A discontinuous Galerkin transport scheme on the cubed sphere. *Monthly Weather Review 133*, 4 (Apr. 2005), 814–828.

[39] Nair, R. D., and Jablonowski, C. Moving Vortices on the Sphere: A Test Case for Horizontal Advection Problems. *Monthly Weather Review 136*, 2 (Feb. 2008), 699–711.

[40] NVidia. *NVIDIA CUDA - NVIDIA CUDA C - Programming Guide version 4.0*, March 2011.

[41] Pan, J., and Manocha, D. Fast GPU-based Locality Sensitive Hashing for K-Nearest Neighbor Computation. *Proceedings of the 19th ACM SIGSPATIAL GIS '11* (2011).

[42] Randall, D., Ringler, T., and Heikes, R. Climate modeling with spherical geodesic grids. *Computing in Science & Engineering* (2002), 32–41.

[43] Sanderson, C. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Tech. rep., NICTA, 2010.

[44] Schaback, R. Multivariate Interpolation and Approximation by Translates of a Basis Function. In *Approximaton Theory VIII–Vol. 1: Approximation and Interpolation*, C. Chui and L. Schumaker, Eds. World Scientific Publishing Co., Inc, 1995, pp. 491–514.

[45] Schmidt, J., Piret, C., Zhang, N., Kadlec, B., Yuen, D., Liu, Y., Wright, G., and Sevre, E. Modeling of Tsunami Waves and Atmospheric Swirling Flows with Graphics Processing Unit (GPU) and Radial Basis Functions (RBF). *Concurrency and Computat.: Pract. Exper.* (2009).

[46] Shu, C., Ding, H., and Yeo, K. S. Local radial basis function-based differential quadrature method and its application to solve two-dimensional incompressible Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering 192*, 7-8 (2003), 941 – 954.

[47] Sloan, I. H., and Womersley, R. S. Extremal systems of points and numerical integration on the sphere. *Adv. Comput. Math 21* (2003), 107–125.

[48] Stevens, D., Power, H., Lees, M., and Morvan, H. The use of PDE centres in the local RBF Hermitian method for 3D convective-diffusion problems. *Journal of Computational Physics* (2009).

[49] Tolstykh, A. On using RBF-based differencing formulas for unstructured and mixed structured-unstructured grid calculations. In *Proceedings of the 16 IMACS World Congress, Lausanne* (2000), pp. 1–6.

[50] Tolstykh, A. I., and Shirobokov, D. A. On using radial basis functions in a "finite difference mode" with applications to elasticity problems. In *Computational Mechanics*, vol. 33. Springer, December 2003, pp. 68 – 79.

[51] Vetter, J., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., and Yalamanchili, S. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *IEEE Computing in Science and Engineering 13*, 5 (2011), 90–95.

[52] Vuduc, R., Demmel, J. W., and Yelick, K. A. Oski: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing* (2005).

[53] Weiler, M., Botchen, R., Stegmaier, S., Ertl, T., Huang, J., Jang, Y., Ebert, D., and Gaither, K. Hardware-Assisted Feature Analysis and Visualization of Procedurally Encoded Multifield Volumetric Data. *IEEE Computer Graphics and Applications 25*, 5 (2005), 72–81.

[54] Wendland, H. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics 4* (1995), 389–396. 10.1007/BF02123482.

[55] Wendland, H. *Scattered Data Approximation*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2005.

[56] Wright, G. B. *Radial Basis Function Interpolation: Numerical and Analytical Developments*. PhD thesis, University of Colorado, 2003.

[57] Wright, G. B., Flyer, N., and Yuen, D. A. A hybrid radial basis function–pseudospectral method for thermal convection in a 3-d spherical shell. *Geochem. Geophys. Geosyst. 11*, Q07003 (2010), 18 pp.

[58] Wright, G. B., and Fornberg, B. Scattered node compact finite difference-type formulas generated from radial basis functions. *J. Comput. Phys. 212*, 1 (2006), 99–123.

[59] Yokota, R., Barba, L., and Knepley, M. G. PetRBF — A parallel O(N) algorithm for radial basis function interpolation with Gaussians. *Computer Methods in Applied Mechanics and Engineering 199*, 25-28 (May 2010), 1793–1804.