ABSTRACT

Many numerical methods based on Radial Basis Functions (RBFs) are gaining popularity in the geosciences due to their competitive accuracy, functionality on unstructured meshes, and natural extension into higher dimensions. One method in particular, the Radial Basis Function-generated Finite Differences (RBF-FD), has drawn significant attention due to its comparatively low computational complexity versus other RBF methods, high-order accuracy (6th to 10th order is common), and embarrassingly parallel nature.

Similar to classical Finite Differences, RBF-FD computes weighted differences of stencil node values to approximate derivatives at stencil centers. The method differs from classical FD in that the test functions used to calculate the differentiation weights are n-dimensional RBFs rather than one-dimensional polynomials. This allows for generalization to n-dimensional space on completely scattered node layouts.

We present our effort to capitalize on the parallelism within RBF-FD for geophysical flow. Many HPC systems around the world are transitioning toward significantly more Graphics Processing Unit (GPU) accelerators than CPUs. In addition to spanning multiple compute nodes, modern code design should include a second level of parallelism targeting the many-core GPU architectures. We will discuss our parallelization strategies to span computation across GPU clusters, and demonstrate the efficiency and accuracy of our parallel explicit and implicit solutions.

CHAPTER 1

INTRODUCTION

1.1 Introduction

The goal of this dissertation is to present a unified approach to parallel solutions of Partial Differential Equations (PDEs) with a method called Radial Basis Function-generated Finite Differences (RBF-FD).

fix: Many scientific problems of importance can be expressed as a collection of PDEs. Solutions to these problems provide answers to many simple questions such as the current temperature of a material, or perhaps the current position of a moving object. Complex and coupled PDEs can simulate the growth of zebra stripes or cheetah spots [**?**] or even model the flow of fluids. Need refs for examples

In order to solve these problems, computational numerical methods are employed on a discretized version of the domain. Traditionally, three (and how would I classify FV? PartOfUnity?) major categories exist for PDE solutions: finite difference (FD), finite element (FEM) and spectral element (SEM) [**?**]. Interestingly, all three of these methods rely on an underlying mesh, making them *meshed methods*. While each has had a turn in the spotlight, more recently a new category, or rather a generalization on all three previous categories, has emerged: *meshfree methods*.

The first task in traditional meshed methods is to generate an underlying grid/mesh. Node placement can be done in a variety of ways including uniform, non-uniform and random (monte carlo) sampling, or through iterative methods like Lloyd's algorithm that generate a regularized sampling of the domain (see e.g., [**?**]). meshed methods have constraints on edge length and angle. Delaunay answers this, but is costly to compute Mesh2d, Triangle, DIstmesh In addition to choosing nodes, meshed methods require connectivity/adjacency lists to form stencils (FD) or elements (FEM, SEM)—this implies an added challenge to cover the domain closure with a chosen element type. While these tasks may be straightforward in one- or two-dimensions, the extension into higher dimensions becomes increasingly more cumbersome [**?**].

Complex geometries, irregular boundaries and mesh refinement also pose a problem for meshed methods. As the complexity of the geometry/boundaries increases, so too should the resolution of the approximating mesh in order to accurately reconstruct the detail present. A naïve approach to refinement increases the density of nodes uniformly across the domain, adding much more computation and memory storage than necessary for activity that is

localized to sub-regions of the domain. Multiresolution methods attempt to compromise between accurate approximation of the domain and reduced resolution by one of two approaches: a) *multilevel methods* that decompose the model into a hierarchy with several levels of mesh detail, then only use a level when it is required to capture phenomena; and b) *adaptive irregular sampling* which has one level of detail, but non-uniform nodal density concentrated in areas of high activity [**?**]. Such techniques require robust methods and complex code capable of either coarsening/smoothing the approximate solutions to new level, or handling non-uniform node placement, element size etc.

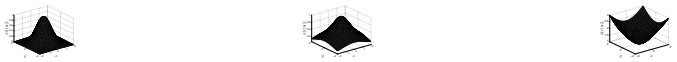
Ideally, we seek a method defined on arbitrary geometries, that behaves regularly in any dimension, and avoids the cost of mesh generation. The ability to locally refine areas of interest in a practical fashion is also desirable. Fortunately, meshfree methods provide all of these properties: based wholly on a set of independent points in n-dimensional space, there is minimal cost for mesh generation, and refinement is as simple as adding new points where they are needed.

Since their adoption by the mathematics community in the 1980s ([**?**]), a plethora of meshfree methods have arisen for the solution of PDEs. For example, smoothed particle hydrodynamics, partition of unity method, element-free Galerkin method and others have been considered for fluid flow problems [**?**]. For a recent survey of methods see [**?**].

A subset of meshfree methods of particular interest to the community today revolves around Radial Basis Functions (RBFs). RBFs are a class of radially symmetric functions (i.e., symmetric about a point, x_j, called the *center*) of the form:

$$\phi_j(\mathbf{x}) = \phi(r(\mathbf{x})) \tag{1.1}$$

where the value of the univariate function ϕ is a function of the Euclidean distance from the center point $\mathbf{x}_j$ given by $r(\mathbf{x}) = ||\mathbf{x} - \mathbf{x}_j||_2 = \sqrt{(x - x_j)^2 + (y - y_j)^2 + (z - z_j)^2}$. Examples of commonly used RBFs are available in Table **??** with their corresponding plots in Figure 1.1. 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 1.2, where $\phi_j(r(\mathbf{x}))$ is an RBF centered at $\{\mathbf{x}_j\}_{j=1}^n$.



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

Figure 1.1: Commonly used RBFs.

With a history extending back four decades for RBF interpolation schemes [**?**], and two decades for RBFs applied to solving PDEs [**?**], 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 [**?**, **?**]. In addition to competitive accuracy and convergence compared with other state-of-the-art methods [**?**, **?**, **?**, **?**, **?**], they also boast stability for large time steps.

Like most numerical methods, RBFs come with certain limitations. For example, RBF

interpolation is—in general—not a well-posed problem, so it requires careful choice of positive definite or conditionally positive definite basis functions [**?**, **?**]. The example 2D RBFs presented in Figure 1.1 are infinitely smooth and satisfy the (conditional) positive definite requirements.

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 [**?**]. The conditioning of the system also dramatically decreases as the number of nodes in the problem increases. Fortunately, recent algorithms such as Contour-Padé [**?**] and RBF-QR [**?**, **?**] allow for numerically stable computation of interpolants in the nearly flat RBF regime (i.e., $\epsilon \to 0$) where high accuracy has been observed [**?**, **?**].

Historically, the most common way to leverage RBFs for PDE solutions is in a global interpolation 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 spectral (exponential) convergence of the RBF interpolant for smooth data [**?**]. As discussed in [**?**], global RBF methods require $O(N^3)$ floating point operations (FLOPs) in pre-processing, where N is the total number of nodes, to assemble and solve a dense linear system for differentiation coefficients. The coefficients in turn are assembled into a dense Differentiation Matrix (DM) that is applied via matrix-vector multiply to compute derivatives at all N nodes for a cost of $O(N^2)$ operations. assumes explicit scheme

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 [**?**]). RBF-FD was first introduced by Tolstykh in 2000 [**?**], but it was the simultaneous, yet independent, efforts in [**?**], [**?**], [**?**] and [**?**] that gave the method its real start. 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.

The RBF-FD method is similar in concept to classical finite-differences (FD): both methods approximate derivatives as a weighted sum of values at nodes within a nearby neighborhood. The two methods differ in that the underlying differentiation weights are exact for RBFs rather than polynomials.

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 derivative evaluations on the order of $O(nN) \implies O(N)$ FLOPs.

RBF-FD have been successfully employed for a variety of problems including Hamilton-Jacobi equations [**?**], convection-diffusion problems [**?**, **?**], incompressible Navier-Stokes equations [**?**, **?**], transport on the sphere [**?**], and the shallow water equations [**?**].

As N grows larger, it behooves us to work on parallel architectures, be it CPUs or GPUs.

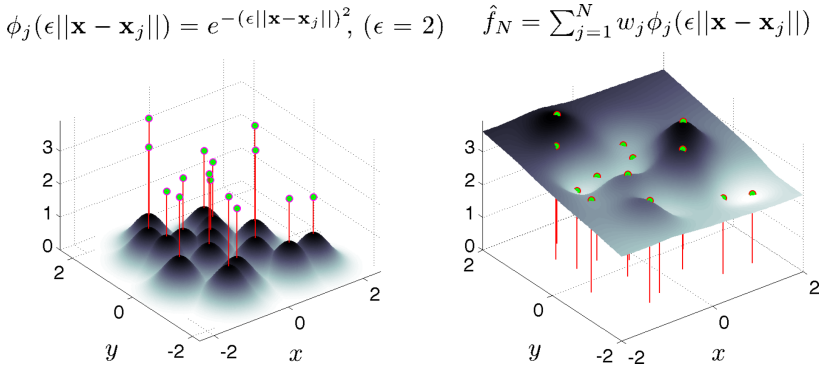


Figure 1.2: RBF interpolation using 15 translates of the Gaussian RBF with $\epsilon = 2$. One RBF is centered at each node in the domain. Linear combinations of these produce an interpolant over the domain passing through known function values.

With regard to the latter, there is some research on leveraging RBFs on GPUs in the fields of visualization [**?**, **?**], surface reconstruction [**?**, **?**], and neural networks [**?**]. 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 [**?**], 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 [**?**] 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 [**?**] 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. [**?**] 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 [**?**] library to target a single GPU.

Within this dissertation, we have developed 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 introduced the first known implementation of accelerated RBF-FD on the GPU. continue with outline of chapters We present our explicit and implicit solutions to PDEs with our multi-GPU RBF-FD implementation.

1.2 Cut from Paper1

Sufficient understanding of RBF interpolation led to the development of the first global RBF method for PDEs in [**?**]. Most popular among global methods is collocation, wherein RBF interpolation approximates the PDE solution by solving a large, dense linear system. In some cases RBF collocation demonstrates higher accuracy for the same number of nodes when compared to other state-of-the-art pseudospectral methods (e.g., [**?**] [**?**] [**?**]). In [**?**], spectral accuracy was demonstrated for hyperbolic PDEs even with local refinement of

nodes in time.

Unfortunately—ill-conditioning aside—collocation methods are prohibitively expensive to solve when scaled to a large number of nodes. Assuming the collocation matrix does not change in time, global methods, with their dense systems, scale at $O(N^3)$ operations for initial preconditioning/preprocessing followed by $O(N^2)$ operations every time-step. This complexity is consistent with any collocation scheme. However, by introducing sparsity into the system (e.g., using compactly supported RBFs), the complexity is somewhat reduced.

General Purpose GPU (GPGPU) computing is one of today's hottest trends within scientific computing. The release of NVidia's CUDA at the end of 2006 marked both a redesign of GPU architecture, plus the addition of a new software layer that finally made GPGPU accessible to the general public. The CUDA API includes routines for memory control, interoperability with graphics contexts (i.e., OpenGL programs), and provides GPU implementation subsets of BLAS and FFTW libraries [**?**]. After the undeniable success of CUDA for C, new projects emerged to encourage GPU programming in languages like FORTRAN (see e.g., HMPP [**?**] and Portland Group Inc.'s CUDA-FORTRAN [**?**]).

In early 2009, the Khronos Group–the group responsible for maintaining OpenGL–announced a new specification for a general parallel programming lanugage referred to as the Open Compute Language (OpenCL) [**?**]. Similar in design to the CUDA language—in many ways it is a simple refactoring of the predecessor—the goal of OpenCL is to provide a mid-to-low level API and language to control any multi- or many-core processor in a uniform fashion. Today, OpenCL drivers exist for a variety of hardware including NVidia GPUs, AMD/ATI CPUs and GPUs, and Intel CPUs.

This *functional portability* is the cornerstone of the OpenCL language. However, functional portability does not imply performance portability. That is, OpenCL allows developers to write kernels capable of running on all types of target hardware, but optimizing kernels for one type of target (e.g., GPU) does not guarantee the kernel will run efficiently on another target (e.g., CPU). With CPUs tending toward many cores, and the once special purpose, many-core GPUs offering general purpose functionality, it is easy to see that soon the CPU and GPU will meet somewhere in the middle as general purpose many-core architectures. Already, ATI has introduced the Fusion APU (Accelerated Processing Unit) which couples an AMD CPU and ATI GPU within a single die. OpenCL is an attempt to standardize programming ahead of this intersection.

Petascale computing centers around the world are leveraging GPU accelerators to achieve peak performance. In fact, many of today's high performance computing installations boast significantly more GPU accelerators than CPU counterparts. The Keeneland project is one such example, currently with 240 CPUs accompanied by 360 NVidia Fermi class GPUs with at least double that number expected by the end of 2012 [**?**].

Such throughput oriented architectures require developers to decompose problems into thousands of independent parallel tasks in order to fully harness the capabilities of the hardware. To this end, a plethora of research has been dedicated to researching algorithms in all fields of computational science. Of interest to us are methods for atmospheric- and geo-sciences.

CHAPTER 2

RBF METHODS

2.1 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.

$$\mathcal{L}u(\mathbf{x}) \mid_{\mathbf{x}=\mathbf{x}_j} \approx \sum_{i=1}^{n} w_i u(\mathbf{x}_i) + w_{n+1} p_0 \tag{2.1}$$

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 [**?**, **?**, **?**, **?**] 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:

$$\begin{pmatrix} \phi(\epsilon\,||\mathbf{x}_1 - \mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_1 - \mathbf{x}_2|| & \cdots & \phi(\epsilon\,||\mathbf{x}_1 - \mathbf{x}_n||) & 1 \\ \phi(\epsilon\,||\mathbf{x}_2 - \mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_2 - \mathbf{x}_2|| & \cdots & \phi(\epsilon\,||\mathbf{x}_2 - \mathbf{x}_n||) & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ \phi(\epsilon\,||\mathbf{x}_n - \mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_n - \mathbf{x}_2|| & \cdots & \phi(\epsilon\,||\mathbf{x}_n - \mathbf{x}_n||) & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix} = \begin{bmatrix} \mathcal{L}\phi(\epsilon\,||\mathbf{x} - \mathbf{x}_1||)|_{\mathbf{x}=\mathbf{x}_j} \\ \mathcal{L}\phi(\epsilon\,||\mathbf{x} - \mathbf{x}_2||)|_{\mathbf{x}=\mathbf{x}_j} \\ \vdots \\ \mathcal{L}\phi(\epsilon\,||\mathbf{x} - \mathbf{x}_n||)|_{\mathbf{x}=\mathbf{x}_j} \\ 0 \end{bmatrix}, \tag{2.2}$$

where w_{n+1} is ignored after the matrix in (2.2) 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 $\mathrm{O}(n^3 N)$ 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 (2.2), 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 [**?**] leading to the matrix structure in Figures 3.4 and 3.5. 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 [**?**, **?**].

2.1.1 Hyperviscosity

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

$$D = \alpha \frac{\partial}{\partial \lambda} + \beta \frac{\partial}{\partial \theta} \tag{2.3}$$

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 [**?**, **?**]. Stabilization of the RBF-FD method is achieved through the application of a hyperviscosity filter to Equation (2.3) [**?**]. By using Gaussian RBFs, $\phi(r) = e^{-(\epsilon r)^2}$, the hyperviscosity (a high order Laplacian operator) simplifies to

$$\Delta^k \phi(r) = \epsilon^{2k} p_k(r) \phi(r) \tag{2.4}$$

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 [**?**]). 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.

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

$$\frac{\partial u}{\partial t} = -Du + Hu, \tag{2.5}$$

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 4.0.1), 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 $N \to \infty$ [**?**]. 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 [**?**, **?**].

2.2 Cut from Paper1

Global RBF collocation methods pose the problem of interpolating a multivariate function $f : \Omega \to \mathbb{R}$ where $\Omega \subset \mathbb{R}^m$. Given a set of sample values $\{f(x_j)\}_{j=1}^N$ on a discrete set of nodes $X = \{x_j\}_{j=1}^N \subset \Omega$, an approximation $\hat{f}_N$ can be constructed through linear combinations of interpolation functions. Here, we choose univariate, radially symmetric functions based on Euclidean distance ($||\cdot||$), and use translates $\phi(x - x_j)$ of a single continuous real valued function ϕ defined on $\mathbb{R}$ and centered at x_j:

$$\phi(x) := \varphi(||x||).$$

Here, φ is a Radial Basis Function and ϕ the associated kernel. For simplification $\phi_j(x)$ refers to a kernel centered at x_j; i.e., $\varphi(||x - x_j||)$.

The interpolant $\hat{f}_N(x)$ requires a linear combination of translates:

$$\hat{f}_N(x) = \sum_{j=1}^{N} c_j \phi_j(x)$$

with real coefficients $\{c_j\}_{j=1}^N$. Assuming the interpolant passes through known values of f; i.e.,

$$\hat{f}_N(x_i) = f(x_i), \qquad 1 \le i \le N,$$

allows one to solve for coefficients if the following linear system is uniquely solvable:

$$\sum_{j=1}^{N} c_j \phi_j(x_i) = f(x_i), \qquad 1 \le i \le N.$$

This is true if the $N \times N$ matrix Φ produced by the linear system

$$\begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_N(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_N(x_2) \\ \vdots & \ddots & \ddots & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \cdots & \phi_N(x_N) \end{pmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_N) \end{bmatrix}$$

$$\Phi \vec{c} = \vec{f}$$

is nonsingular.

When choosing an appropriate basis function for interpolation, a subset of RBFs have been shown to produce symmetric positive definite Φ, while others are only conditionally positive definite (for more details see [**?**]). With the latter set, additional polynomial terms are added to constrain the system and enforce positive definiteness, resulting in this expanded system of equations:

$$\begin{aligned}
\sum_{j=1}^{N} c_j \phi_j(x_i) + \sum_{k=1}^{M} d_k p_k(x_i) &= f(x_i), \qquad 1 \le i \le N, \\
\sum_{j=1}^{N} c_j p_k(x_j) &= 0, \qquad 1 \le k \le M. \\
\begin{pmatrix} \Phi & P \\ P^T & 0 \end{pmatrix} \begin{bmatrix} c \\ d \end{bmatrix} &= \begin{bmatrix} \vec{f} \\ 0 \end{bmatrix}
\end{aligned} \tag{2.6}$$

where $\{p_k\}_{k=1}^{M}$ is a basis for Π_p^m (the set of polynomials in m variables of degree $\le p$) and

$$M = \binom{p+m}{m}.$$

Given the coefficients $\vec{c}$, the function value at a test point x is interpolated by

$$\begin{aligned}
\hat{f}_N(x) &= \sum_{j=1}^{N} c_j \Phi_j(x) + \sum_{l=1}^{M} d_l P_l(x) \\
&= \begin{bmatrix} \Phi & \mathbb{P} \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix} \\
&= \vec{\Phi}_x^T \vec{c} \\
&= \vec{\Phi}_x^T \Phi^{-1} \vec{f}
\end{aligned} \tag{2.7}$$

where $\vec{c}$ is substituted by the solution to Equation 2.6. In Equation 2.7 the term $\Phi_x^T \Phi^{-1}$, dependent only on node positions, can be evaluated prior to knowing f.

For the RBF-FD method, derivatives of $u(x)$ are weighted combinations of a small neighborhood of N_s nodes (i.e., a stencil with $N_s \ll N$):

$$\mathcal{L}u(x_1) \approx \sum_{j=1}^{N_s} c_j u(x_j)$$

where $\mathcal{L}u$ denotes a linear differential quantity over u (e.g., $\mathcal{L}u = \frac{du}{dx}$). Here, c_j are unknown, and obtained by enforcing the reconstruction:

$$\mathcal{L}\phi_j(x_1) = \sum_{i=1}^{N_s} c_i \phi_j(x_i) \qquad \text{for } j = 1, 2, ..., N_s$$

with $\mathcal{L}\phi_j$ provided by analytically applying the differential operator to the kernel function. This is equivalent to:

$$\begin{pmatrix} \phi_1(x_1) & \phi_1(x_2) & \cdots & \phi_1(x_{N_s}) \\ \phi_2(x_1) & \phi_2(x_2) & \cdots & \phi_2(x_{N_s}) \\ \vdots & \ddots & \ddots & \vdots \\ \phi_{N_s}(x_1) & \phi_{N_s}(x_2) & \cdots & \phi_{N_s}(x_{N_s}) \end{pmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} \mathcal{L}\phi_1(x_1) \\ \mathcal{L}\phi_2(x_1) \\ \vdots \\ \mathcal{L}\phi_{N_s}(x_1) \end{bmatrix} \tag{2.8}$$

$$\Phi_s \vec{c} = \vec{\Phi_{\mathcal{L}_s}}. \tag{2.9}$$

Solving Equation 2.9 for stencils weights, $\vec{c}$, and substituting into Equation 6.3 reveals the similarity between RBF-FD and the general problem of RBF interpolation (see Equation 2.7 and discussion):

$$\begin{aligned} \mathcal{L}u(x_1) &\approx (\Phi_s^{-1}\vec{\Phi}_{\mathcal{L}_s})^T \vec{u} \\ &\approx \vec{\Phi}_{\mathcal{L}_s}^T \Phi_s^{-T} \vec{u} \end{aligned}$$

Again, based on the choice of RBF, positive definiteness of the system is ensured by adding the same constraints as Equation 2.6, but note differential quantities for $\vec{\mathbb{P}}$ on the right hand side:

$$\begin{pmatrix} \Phi_s & \mathbb{P} \\ \mathbb{P}^T & 0 \end{pmatrix} \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} \vec{\Phi}_{\mathcal{L}_s} \\ \vec{\mathbb{P}}_{\mathcal{L}} \end{bmatrix}. \tag{2.10}$$

Equation 2.10 is strictly for the stencil centered at node x_1. To obtain weights for all stencils in the domain, a total of N such systems must be solved.

Stabilization of the RBF-FD method is achieved through the application of a hyperviscosity filter [**?**]. By assuming the use of Gaussian RBFs, $\phi(r) = e^{-(\epsilon r)^2}$, the hyperviscosity operator simplifies to

$$\Delta^k \phi(r) = \epsilon^{2k} p_k(r) \phi(r). \tag{2.11}$$

The multiples of generalized Laguerre polynomials, $p_k(r)$, are obtained through the following recursive relation:

$$\begin{cases} p_0(r) &= 1, \\ p_1(r) &= 4(\epsilon r)^2 - 2d, \\ p_k(r) &= 4((\epsilon r)^2 - 2(k-1) - \frac{d}{2}) p_{k-1}(r) - 8(k-1)(2(k-1) - 2 + d) p_{k-2}(r), \qquad k = 2, 3, ... \end{cases}$$

where d is the dimension of the problem. We assume $d = 2$ below when working on the surface of the sphere.

CHAPTER 3

TARGETING MULTIPLE GPUS

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 (2.5) 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 [**?**], 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.

3.0.1 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 perma-

$\mathcal{G}$	: 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 3.1: Sets defined for stencil distribution to multiple CPUs

nently 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 3.1, defines the collection of index lists that each CPU must maintain for both multi-CPU and multi-GPU implementations.

Figure 3.1 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 3.1 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 $\mathcal{G}$,

$$\mathcal{G} = \{\mathcal{Q} \backslash \mathcal{B} \quad \mathcal{B} \backslash \mathcal{O} \quad \mathcal{O} \quad \mathcal{R}\}, \tag{3.1}$$

is designed to simplify both CPU-CPU and CPU-GPU memory transfers by grouping nodes of similar type. 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 $\mathcal{R}$.

The structure of $\mathcal{G}$ offers two benefits: first, solution values in $\mathcal{R}$ and $\mathcal{O}$ are contiguous

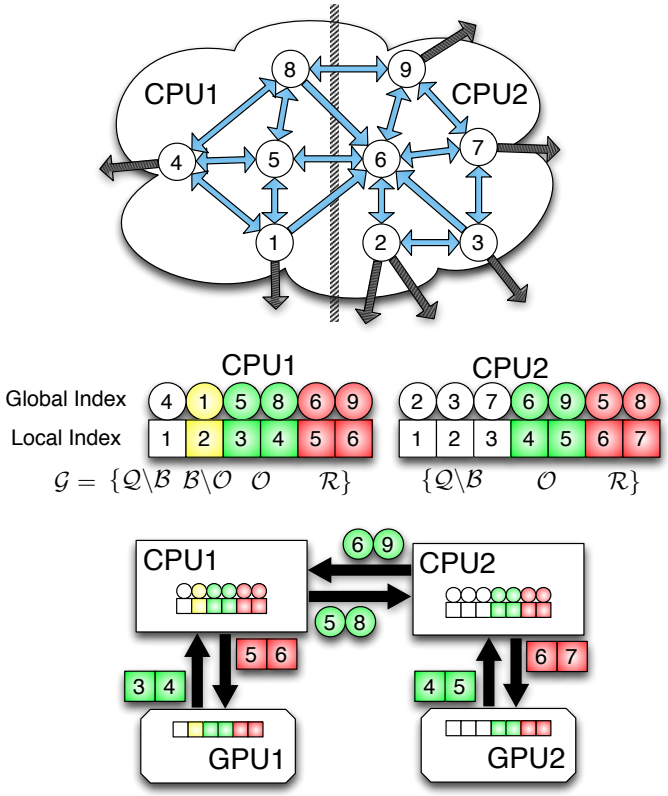


Figure 3.1: Partitioning, index mappings and memory transfers for nine stencils ($n = 5$) spanning two CPUs and two GPUs. Top: the directed graph created by stencil edges is partitioned for two CPUs. Middle: the partitioned stencil centers are reordered locally by each CPU to keep values sent to/received 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 3.1: $\mathcal{Q}\backslash\mathcal{B}$ is white, $\mathcal{B}\backslash\mathcal{O}$ is yellow, $\mathcal{O}$ is green and $\mathcal{R}$ is red.

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 3.1. Each GPU operates on the local indices ordered according to Equation (3.1). 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 3.2 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 3.2 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 [**?**] or icosahedral geodesic grid [**?**], which can evenly balance the computational load across partitions. Other interesting partitionings can be generated with software libraries such as the METIS [**?**] family of algorithms, capable of partitioning and reordering directed graphs produced by RBF-FD stencils.

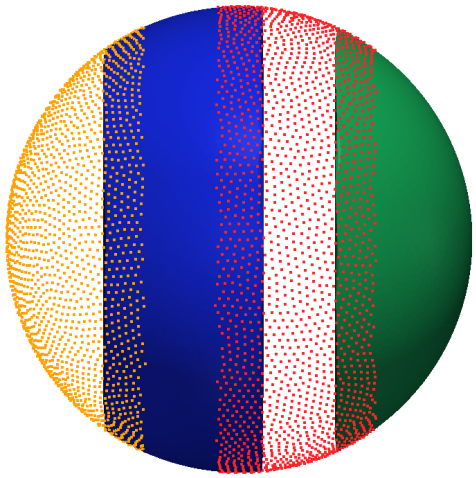


Figure 3.2: Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

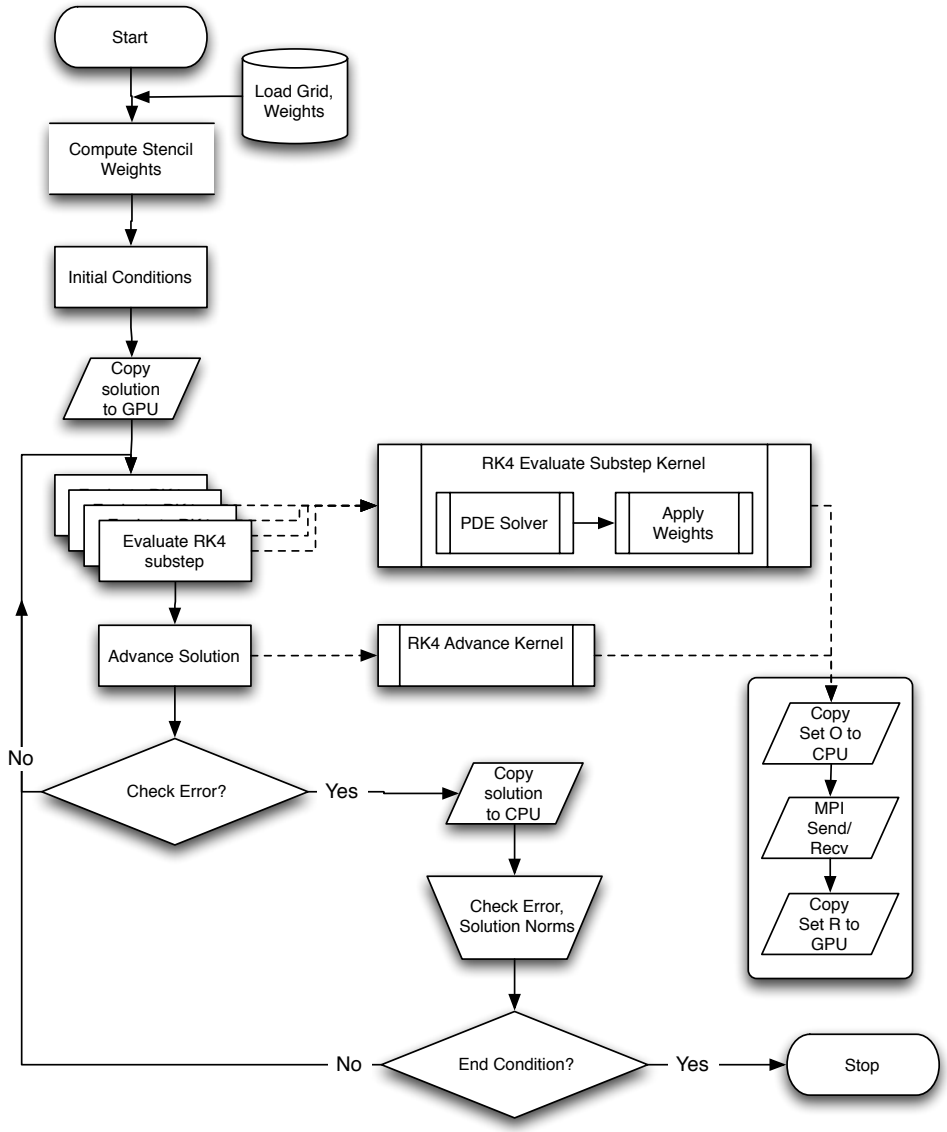


Figure 3.3: Workflow for RK4 on multiple GPUs.

3.0.2 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 3.3 summarizes the time advancement steps for the multi-CPU/GPU implemen-

tation. The RK4 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 $\mathbf{k}_{1\to4}$ 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 (2.5) 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 [**?**] 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 [**?**]. OpenCL assumes a tiered memory hierarchy that provides fast but small *local memory* space that is shared within a work-group [**?**]. Local memory on Fermi GPUs is 48 KB per multiprocessor [**?**]. 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 [**?**], 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.

Naive Approach: One thread per stencil. In this first implementation, each thread computes the derivative at one stencil center (Figure 3.4). 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 [**?**]. 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.

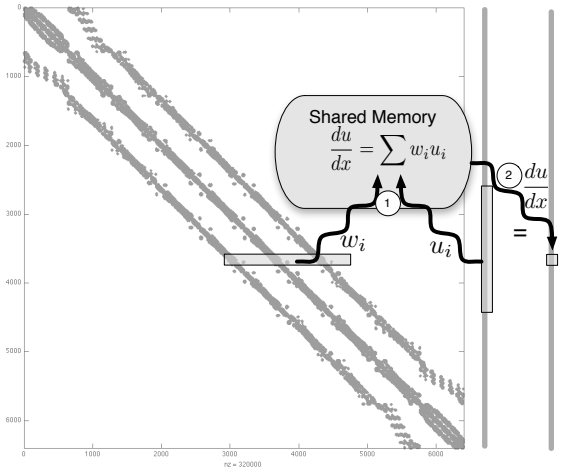


Figure 3.4: Naive approach to sparse matrix-vector multiply. Each thread is responsible for the sparse vector dot product of weights and solution values for derivatives at a single stencil.

Alternate Approach: One warp per stencil. An alternate approach, illustrated in Figure 3.5, 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 threads

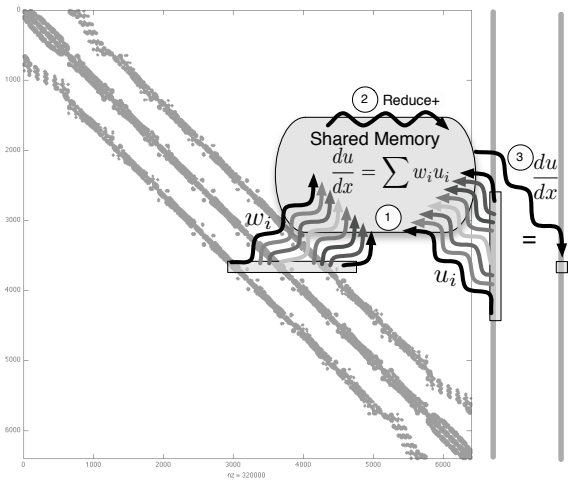


Figure 3.5: Alternative approach. A full warp (32 threads) collaborate to apply weights and compute the derivative at a stencil center.

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., [**?**] and [**?**]).

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.

3.1 Cut from Paper1

While the nomenclature used in this paper is typically associated with CUDA programming, the names *thread* and *warp* are used to clearly illustrate kernel execution in context of the NVidia specific hardware used in tests. OpenCL assumes a lowest common denominator of hardware capabilities to provide functional portability. However, intimate knowledge of hardware allows for better understanding of performance and optimization on a target architecture. For example, OpenCL assumes all target architectures are capable at some level

of SIMD (Single Instruction Multiple Data) execution, but CUDA architectures allow for Single Instruction Multiple Thread (SIMT). SIMT is similar to traditional SIMD, but while SIMD immediately serializes on divergent operations, SIMT allows for a limited amount of divergence without serialization.

At the hardware level, a *thread* executes instructions on the GPU. On Fermi level GPUs, groups of 32 threads are referred to as *warps*. A warp is the unit of threads executed concurrently on a single *multi-processor*. In OpenCL (i.e., software), a collection of hardware threads performing the same instructions are referred to as a *work-group* of *work-items*. Work-groups execute as a collection of warps constrained to the same multiprocessor. Multiple work-groups of matching dimension are grouped into an *NDRange*. The *kernel* provides a master set of instructions for all threads in an NDRange [**?**].

NVidia GPUs have a tiered memory hierarchy related to the grouping of threads described above. In multiprocessors, each computing core executes a thread with a limited set of registers. The number of registers varies with the generation of hardware, but always come in small quantities (e.g., 32K shared by all threads of a multiprocessor on the Fermi). Accessing registers is free, but keeping data in registers requires an effort to maintain balance between kernel complexity and the number of threads per block. Threads of a single work-group can share information within a multiprocessor through *shared memory*. With only 48 KB available per multiprocessor [**?**], shared memory is another fast but limited resource on the GPU. OpenCL refers to shared memory as *local memory*. Sharing information across multiprocessors is possible in *global device memory*—the largest and slowest memory space on the GPU. To improve seek times into global memory, Fermi level architectures include L1 on each multiprocessor and a shared L2 cache for all multiprocessors.

CHAPTER 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 [**?**, **?**] and 2) solid body rotation [**?**]. 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 [**?**, **?**, **?**, **?**]. 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 [**?**] for details) in order to be consistent with previously published results (see e.g., [**?**] and [**?**]). 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 [**?**].

4.0.1 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

$$\frac{\partial h}{\partial t} + \frac{u}{\cos\theta}\frac{\partial h}{\partial \lambda} = 0 \tag{4.1}$$

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 (4.1) 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

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

Table 4.1: 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

where $\rho(\theta) = \rho_0 \cos\theta$ is the radial distance of the vortex with $\rho_0 = 3$. The exact solution to (4.1) 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 (4.1) is

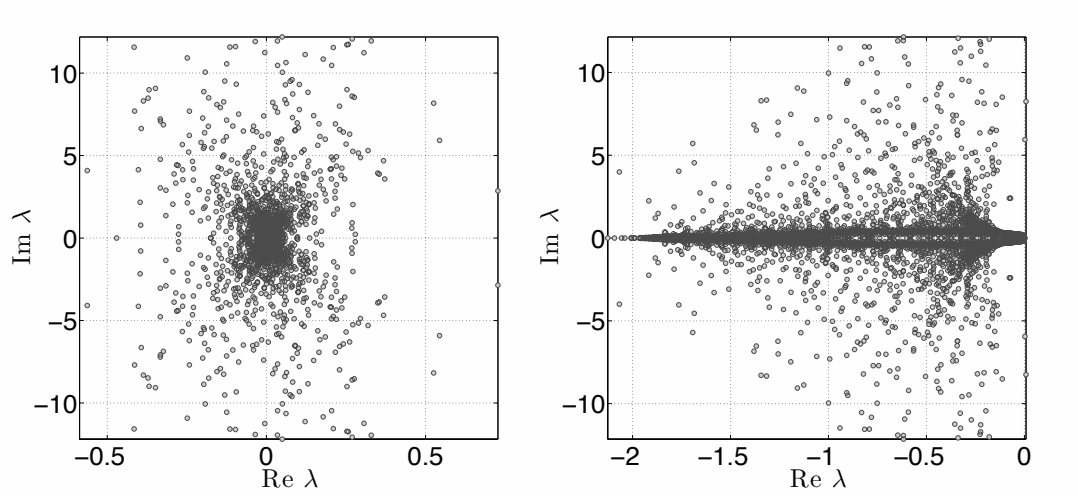
$$\frac{d\mathbf{h}}{dt} = -\text{diag}(\omega(\theta))D_\lambda \mathbf{h}. \tag{4.2}$$

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 (4.2) in the form given in (2.5). The scaling parameter γ_c and the order of hyperviscosity k are given in Table 4.1. 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 4.1b shows the effect of hyperviscosity on the eigenvalues of the DM, $-diag(\omega(\theta))D_\lambda$, in (4.2).

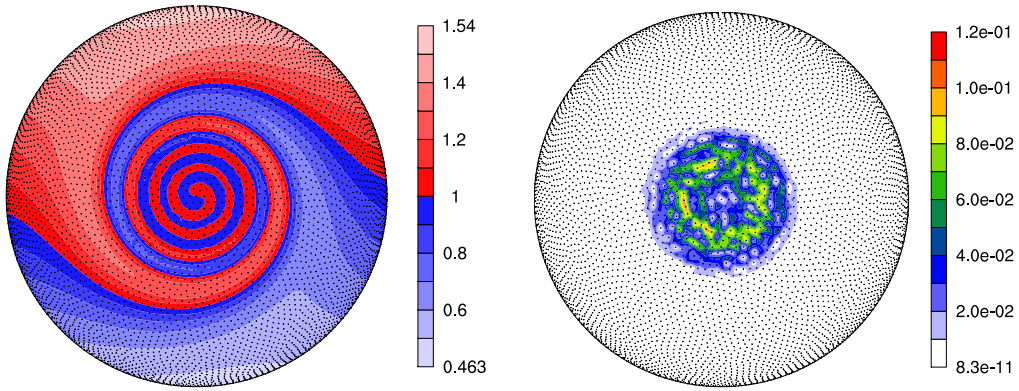
In order to scale to large node sets, the RBF shape parameter, ϵ, is chosen such that the mean condition number of the local RBF interpolation matrices $\bar{\kappa}_A = \frac{1}{N}\sum_{j=1}^{N}(\kappa_A)_j$ is kept constant as N increases ($(\kappa_A)_j$ is the condition number of the interpolation matrix in (2.2), representing the j^{th} stencil). For a constant mean condition number, ϵ varies linearly with $\sqrt{N}$ (see [**?**] 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 [**?**].

Figure 4.2 shows the solution to Equation (4.1) 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 4.3. The time step $\Delta t = 0.05$ for all resolutions.



(a) No Hypervsiscosity (b) With Hyperviscosity

Figure 4.1: 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 $\gamma_c = 40$.



(a) Computed Solution (b) Relative Absolute Error

Figure 4.2: 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 $1.25(10^{-2})$

4.0.2 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

$$\frac{\partial h}{\partial t} + \frac{u}{\cos\theta}\frac{\partial h}{\partial \lambda} + v\frac{\partial h}{\partial \theta} = 0, \tag{4.3}$$

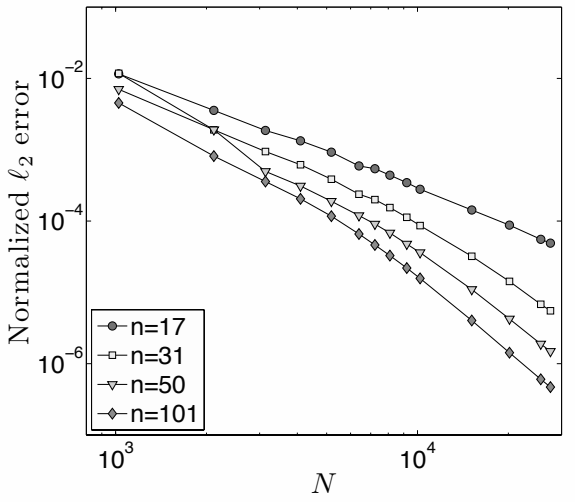


Figure 4.3: Convergence plot for vortex roll-up at $t = 3$.

with velocity field,

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

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 [**?**, **?**].

The discretized form of (4.3) is

$$\frac{d\mathbf{h}}{dt} = -\text{diag}\left(\frac{u}{\cos\theta}\right) D_\lambda \mathbf{h} - \text{diag}(v) D_\theta \mathbf{h} \tag{4.4}$$

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 (4.4) 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 (4.3) 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 [**?**, **?**], applying the entire spatial operator to the right hand side of Equation 2.2 generates a single DM that analytically removes the singularities at poles.

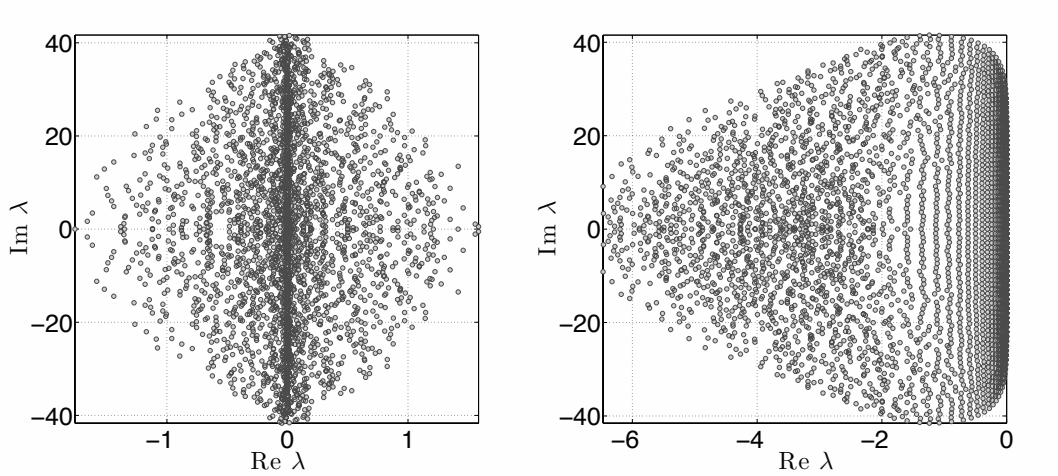
We will advect a C^1 cosine bell height-field given by

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

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 4.2: 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	$5 * 10^{-2}$



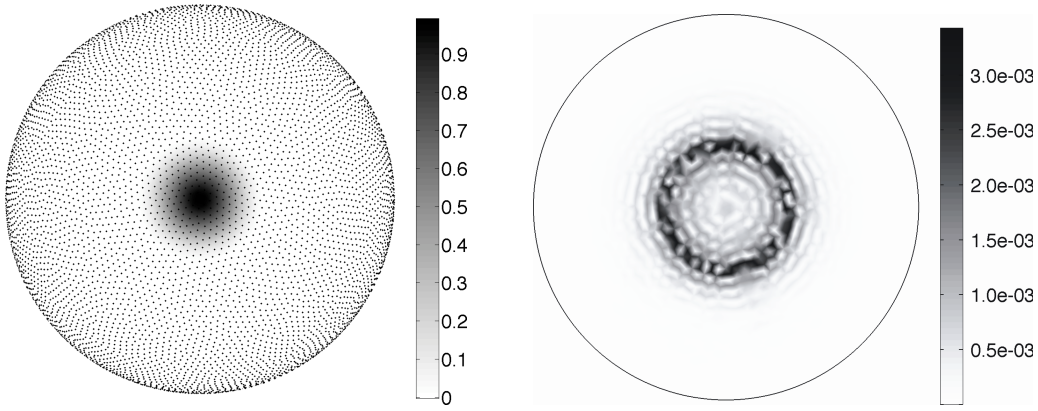
(a) No Hyperviscosity

(b) With Hyperviscosity

Figure 4.4: Eigenvalues of (4.4) 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 4.4 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 4.2.

Figure 4.5 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 [**?**]). After ten



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

Figure 4.5: Cosine bell solution after 10 revolutions with $N = 10201$ nodes and stencil size $n = 101$. Hyperviscosity parameters are $k = 8$, $\gamma_c = 5(10^{-2})$.

revolutions, the cosine bell is still intact. The majority of the absolute error (Figure 4.5b) appears at the base of the C^1 bell where the discontinuity appears in the derivative. At ten revolutions, Figure 4.6 illustrates the convergence of the RBF-FD method. All tests in Figure 4.6 assume 1000 time-steps per revolution (i.e., $\Delta t = 1036.8$ seconds).

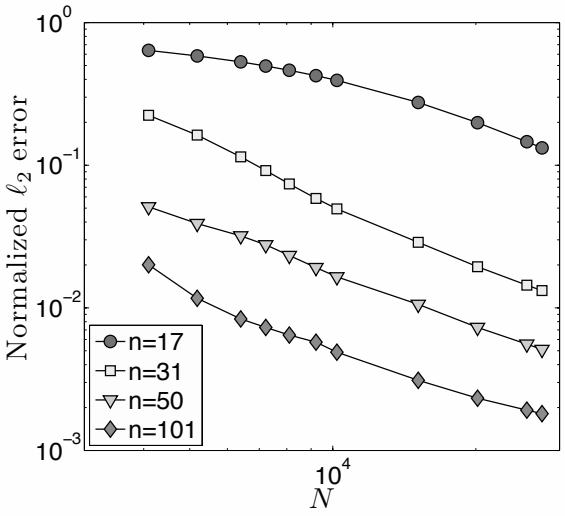


Figure 4.6: Convergence plot for cosine bell advection. Normalized ℓ_2 error at 10 revolutions with hyperviscosity enabled.

4.1 Cut from Paper1

To verify our multi-CPU and multi-CPU+GPU implementations, two hyperbolic PDEs on the surface of the sphere are tested. For both cases a spherical coordinate system is used in terms of latitude λ and longitude θ:

$$\begin{aligned} x &= \rho \cos\lambda \cos\theta, \\ y &= \rho \sin\lambda \cos\theta, \\ z &= \rho \sin\theta. \end{aligned}$$

Node sets are the Maximum Determinant point distributions on the sphere [**?**] consistent with previously published results (see e.g., [**?**] and [**?**]).

Hyperviscosity parameters γ_c and k depend on the RHS of the PDE. For this test case, hyperviscosity scaling parameters are listed in Table 4.1. Linear functions to choose the RBF support parameter ϵ are also provided. The parameters γ_c and k were obtained via trial-and-error parameter searching on $N = 4096$ nodes. The goal when choosing parameters is to push all eigenvalues to the left half-plane, and then tweak γ_c up or down to condense the eigenvalues as near to the imaginary axis as possible. We try to keep the range of filtered eigenvalue real parts within twice the width of the unfiltered range, so hyperviscosity does not cause too much diffusion in the solution.

For the cosine bell we use the initial conditions

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

where the bell of radius $R = \frac{a}{3}$ is centered at (λ_c, θ_c) and provided by the expression,

$$\rho = a \arccos(\sin\theta_c \sin\theta + \cos\theta_c \cos\theta \cos(\lambda - \lambda_c)).$$

We assume $a = 1$, $h_0 = 1$, and $(\lambda_c, \theta_c) = ({}^{3\pi}/_{2}, 0)$. The angle $\alpha = {}^{\pi}/_{2}$ is chosen to transport the bell over the poles of the coordinate system, and $u_0 = {}^{2\pi a}/_{1036800}$.

Author's Note: Include figures from NCL or Paraview

CHAPTER 5

PERFORMANCE BENCHMARKS

Author's Note: First I need to describe the hardware used. This includes: Troi, Spear, Keeneland; their gpus and RAM, etc. Discuss interconnects (why can't we run multiple kernels on one node of spear)

We propose the implementation of an efficient multi-node, multi-GPU RBF PDE package to run on a heterogeneous cluster of compute nodes, each with one or more GPUs attached. Specifically, we will utilize hardware available in the Department of Scientific Computing: the ACM cluster. As will be explained, the ACM cluster is moderately sized, but for true scaling tests a larger cluster such as the TeraGrid Lincoln cluster will be required. For a survey of heterogeneous clusters that leverage co-processors like GPUs, refer to [**?**].

5.1 ACM Cluster (FSU)

The ACM cluster at Florida State University is funded by SCREM-NSF 0724273 titled "SCREMS: High Performance Computing and Visualization", with PI Gordon Erlebacher of the Department of Scientific Computing, and other faculty in the Department of Mathematics. The cluster of one head node and 20 compute nodes is composed of homogenous CPUs. Each node contains two dual-core Opteron 2.2 GHz CPUs (4 cores total) and 4 GB of RAM. Nodes are connected via 4X DDR Infiniband.

The homogeneity of the cluster is lost when computing on with GPUs (see Table 5.1). With each Tesla C1060 capable of roughly 933 GFLOPS in single precision, the peak performance of ACM in multi-core compute mode is about 16 TFLOPS. Double precision is reduced to roughly 78 GFLOPS on the C1060s [**?**]. At 20 compute nodes, ACM is a small cluster sufficient for development, but not large enough to thoroughly test scaling properties.

The Compute Capability listed in Table 5.1 specifies the GPU's support for advancements such as atomic functions (introduced in 1.1) and double IEEE754 floating point precision (introduced in 1.3) [**?**]. When programming in CUDA, we will only use the homogeneous component of the cluster. On the other hand, we might experiment with homogeneous computing with the OpenCL version of the code.

# of Nodes	# GPUs per Node	GPU Name	Global Device Memory Size	Compute Capability
16	1	Tesla C1060	4 GB	1.3
3	1	Tesla C870	1.5 GB	1.0
1	1	GeForce 8500 GT	0.25 GB	1.1
1	0	(ACM head-node)	N/A	N/A

Table 5.1: Heterogeneous configuration of ACM cluster. The last entry represents the ACM head node which has no GPU attached.

5.2 Lincoln

One of the large heterogeneous GPU clusters made available through the TeraGrid is NCSA's Lincoln [**?**]. Lincoln sports 192 compute nodes containing Dell PowerEdge 1950 dual-socket nodes with quad-core Intel Harpertown 2.33GHz processors (8 cores per node) and 16GB of memory. This gives Lincoln a total of 1536 total CPU cores. Attached to the compute nodes are 96 NVIDIA Tesla S1070 accelerator units each containing 4 Tesla GPUs (compute capability 1.3) and 16 GB of Global Device Memory for a total of 384 GPU units. The peak performance of a Tesla S1070 is roughly 4 TFLOPS for single precision and 345 GFLOPS for double precision. Each of the 192 compute nodes in Lincoln is connected to two of the Tesla S1070s via PCI-e Gen2 X8. Compute nodes are connected via Infiniband SDR. The peak performance of Lincoln considering both GPU and CPU computation is 47.5 TFLOPS [**?**].

Lincoln is a heterogeneous multi-core compute platform, with heterogeneity arising from the ability to compute on both the CPU and GPU. Göddeke et al. [**?**] call this *local heterogeneity*, in contrast to ACM where the hardware attached to individual nodes is heterogeneous and called *global heterogeneity*.

5.3 Future Hardware

In Spring 2010, NVidia will publicly release a new architecture code named "Fermi" [**?**]. The new hardware will support many features of interest, the most important being 8x faster double precision than the older Tesla C1060 (GT200 architecture). It will also allow for concurrent kernel execution (for up to 16 small kernels) making it easier to keep the GPU saturated with computation. Table 5.2 considers some of the more monumental differences between the Fermi and GT200 architectures.

While there has been no word yet as to the date of availability (only speculation for Q1 2010), the Fermi architecture is anticipated to cost around $2,500 to $4,000 for the Tesla C2050 (3 GB Global Device Memory) and C2070 (6 GB Global Device Memory) units respectively [**?**].

Additionally, the FSU HPC group (supported by the Department of Scientific Computing) has several grants in play, which may lead to the acquisition of an additional GPU cluster.

	Fermi	GT200
# of concurrent kernels	16	1
Warps scheduled concurrently	2	1
Clock cycles to dispatch instruction to warp	2	4
Caching on Global Device Memory	Yes	No
Shared memory	64 KB	16 KB
Shared memory banks	32	16
Bank conflict free FP64 access	Yes	No
Cycles to read FP64 (from shared memory)	2	32
Max allowed warps	48	32

Table 5.2: Comparison of NVidia's new Fermi architecture to the GT200 architecture used for GTX 280, Tesla C1060 and other GPUs in use today.

CHAPTER 6

STOKES

6.1 Introduction

We consider herein the solution to steady-state viscous Stokes flow on the surface of a sphere, governed by:

$$\boldsymbol{\nabla} \cdot [\eta(\boldsymbol{\nabla}\mathbf{u} + (\boldsymbol{\nabla}\mathbf{u})^T)] + RaT\hat{r} = \boldsymbol{\nabla}p \tag{6.1}$$

$$\boldsymbol{\nabla} \cdot \mathbf{u} = 0 \tag{6.2}$$

where the unknowns $\mathbf{u}$ and p represent the vector velocity- and scalar pressure-field respectively, η is the viscosity tensor, Ra is the non-dimensional Rayleigh number, and T is an initial temperature profile. Many practical applications in sciences such as geophysics, climate modeling, and computational fluid dynamics must solve variations of the Navier-Stokes equations. The focus of this paper is on the implicit solve component for viscous (Stokes) flow, which amounts to the steady-state problem described by Equations 6.1 and 6.2.

This article introduces the first (to our knowledge) parallel approach to solve the steady-state equations on the surface of the unit sphere with the Radial Basis Function-generated Finite Differences (RBF-FD) method. Building on our work in [**?**], which parallelized explicit RBF-FD advection, our goal is to integrate both explicit and implicit components within a larger transient flow model.

Author's Note: follow with details of RBF methods and novelty of RBF-FD.

For decades, the demand for fast and accurate numerical solutions in fluid flow has lead to a plethora of computational methods for various geometries, discretizations and dimensions. On the sphere in $\mathbb{R}^3$ popular discretizations include the standard latitude-longitude grid, cubed-sphere [**?**], yin-yang overlapping grid [**?**], icosahedral grid [**?**] and centroidal voronoi tessellations [**?**]. Associate with each discretization is a mesh—specific to the choice of numerical method—that indicates connectivity of nodes for differentiation.

Two decades ago [**?**], meshless methods based on Radial Basis Functions (RBFs) were introduced for problems that require geometric flexibility with scattered node layouts in d-dimensional space, plus natural extensions to higher dimensions with minimal change in programming complexity [**?**, **?**]. These RBF methods tout competitive accuracy with other state-of-the-art and high-order methods [**?**, **?**, **?**, **?**, **?**], as well as stability for large time steps. A survey of RBF methods is provided by [**?**].

Author's Note: Change first sentence RBF-generated finite differences (RBF-FD) hold a promising future in that they share many advantages of global RBF methods, but reduce computational complexity to $O(N)$ and exhibit increased parallelism. The method was first suggested in 2000 [**?**], but made its grand debut a few years later in the simultaneous, yet independent, efforts of [**?**, **?**, **?**, **?**]. It has been successfully employed for a variety of problems including Hamilton-Jacobi equations [**?**], convection-diffusion problems [**?**, **?**], incompressible Navier-Stokes equations [**?**, **?**], transport on the sphere [**?**], and the shallow water equations [**?**].

Compared to classical finite differences (FD) which calculate differentiation weights with one-dimensional polynomials, RBF-FD leverages d-dimensional RBFs as test functions. This allows for generalization to d-dimensional space on completely scattered node layouts. For both methods, a stencil of size n neighboring nodes determines the accuracy of the approximation. However, contrary to the regular stencil node distributions from FD, RBF-FD allows for completely scattered node distributions. In comparison with global RBF methods, spectral accuracy is sacrificed in exchange for computational speed and parallelism. Still, high-order accuracy is possible with RBF-FD—6th- to 10th-order accuracy is common.

Until now, most of the focus in RBF-FD has been on explicit methods. However, many practical applications in sciences such as geophysics, climate modeling, and computational fluid dynamics must solve variations of the navier stokes equations which include an implicit solve component. This paper develops multi-GPU algorithms for implicit RBF-FD systems toward the goal of integration within transient flow problems. The explicit component of transient flow is a natural extension of our work in [**?**].

Author's Note: Flesh these points out and integrate them in surrounding paragraphs (until "end") Speed not the issue. Need less RBFs for given accuracy of steady state less nodes implies less memory general geometries are supported better distributions of nodes on spheres. competition like CitComS, etc. use cubed sphere, yinyang grids and triangular meshes in combination with low order methods (2nd and 3rd order). Increasing the order of the method or dimension can significantly increase the complexity of the algorithms. RBF-FD naturally extends to higher dimensions and increasing the order is as simple as increasing the number of nodes in the stencil. Author's Note: end

Related work (list references): RBF methods for Elliptic PDEs

- Global [**?**]
- Compact Support
- divergence-free spherical radial basis Glerkin method for Stokes on the unit sphere [**?**]
- RBF-FD
  - Incompressible Navier-Stokes using explicit (Euler) and semi-implicit (Crank Nicholson) time step. Small problem size ($61x61$) and small stencil sizes ($n = 9$); ghost nodes beyond boundary strategy. Both RBF-FD and RBF-HFD tested. [**?**]

Related work on Preconditioned iterative methods for Stokes Flow

- Survey of preconditioners used for Stokes flow problems [?] (limited applicability since they do not assume non-SPD matrices)
- Multi-GPU Jacobi iteration for Navier stokes flow in cavity http://scholarworks.boisestate.edu/cgi/viewcontent.cgi?article=1003&context=mecheng_facpubs

For decades, a major push has been executed in science to parallelize algorithms and leverage resources available on the increasingly capable supercomputers and clusters. Perhaps as soon as the next decade, we will see exascale level architectures ([?]). Given today's technology, those architecture will surely achieve their performance with the help of accelerator hardware such as GPUs.

To prepare the RBF community for the future, we develop algorithms employing two levels of parallelization: first the MPI standard spans computation across multiple CPUs, and second computation is distributed across the many processing cores of GPU accelerators.

The two level parallelization can even be extended to three level parallelism with pThreads or OpenMP [?]. While OpenCL provides the means to target parallelism on either multi-core CPUs or many-core GPUs, it does not allow a parallel kernel on one hardware interact with a parallel kernel on the another. That is to say, an OpenCL kernel on the CPU cannot launch kernels on the GPU s

Our goal is to demonstrate to the geosciences that RBF-FD can function well on both hyperbolic and elliptic problems. In [?] we introduced a multi-GPU implementation of RBF-FD and demonstrated the method's strong ability to stably advect solid bodies on the sphere. In this paper we continue toward the goal of RBF-FD solutions for fluid flow problems with a multi-GPU Poisson solver for steady-state Stokes flow. In this context, speed is not a paramount issue.

Related work on RBF methods targeting the GPU is quite limited. Schmidt et al. [?] solve an implicit system for a global RBF method using Accelerys Jacket in Matlab. Our work in [?] introduced the first parallel implementation of RBF-FD for explicit advection capable of spanning multiple CPUs as well as multiple GPUs.

Related work on multi-CPU or multi-GPU RBFs

- CPU [?] [?]
- single-GPU [?]
- multi-GPU
  - Preconditioned BiCGStab for Navier Stokes, Finite Element method [?]

While RBF-FD differentiation matrices are applied in the same fashion as standard FD methods, they are unique in that they are asymmetric, non-positive definite and potentially have high condition numbers. To solve an implicit system therefore, we requie an iterative krylov solver like GMRES or BiCGStab which are applicable to matrices of this type. Additionally, preconditioned variants of these methods are required to reduce the complexity of the solution process.

Within this paper we implement a preconditioned GMRES method for RBF-FD on multiple GPUs.

Parallel GMRES

- CPU only: PETSc [?], Hypre [?]
- Parallel GMRES on single GPU available in ViennaCL [?] and CUSP [?]
- Parallel GMRES on Multiple GPUs [?]
- Reduced Communication with increased computation [?]

This article continues our effort with an implementation of RBF-FD on both single and multiple-GPUs for elliptic PDEs. In the next section we introduce

6.2 RBF-FD Weights

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

$$\mathcal{L}u(\mathbf{x}) \mid_{\mathbf{x}=\mathbf{x}_j} \approx \sum_{i=1}^{n} w_i u(\mathbf{x}_i) \tag{6.3}$$

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 the RBF center and the evaluation point measured in the standard Euclidean 2-norm. It has also been shown through experience and studies [?, ?, ?, ?] that better accuracy is gained by the interpolant being able to reproduce a constant. Hence, the constraint $\sum_{i=1}^n c_k = \mathcal{L}1|_{\mathbf{x}=\mathbf{x}_j} = 0$ is added, where w_{n+1} is ignored after the matrix in (6.4) is inverted. That is,

$$\begin{pmatrix} \phi(\epsilon\,||\mathbf{x}_1 - \mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_1 - \mathbf{x}_2||) & \cdots & \phi(\epsilon\,||\mathbf{x}_1 - \mathbf{x}_n||) & 1 \\ \phi(\epsilon\,||\mathbf{x}_2 - \mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_2 - \mathbf{x}_2||) & \cdots & \phi(\epsilon\,||\mathbf{x}_2 - \mathbf{x}_n||) & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ \phi(\epsilon\,||\mathbf{x}_n - \mathbf{x}_1||) & \phi(\epsilon\,||\mathbf{x}_n - \mathbf{x}_2||) & \cdots & \phi(\epsilon\,||\mathbf{x}_n - \mathbf{x}_n||) & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} \mathcal{L}\phi(\epsilon\,||\mathbf{x} - \mathbf{x}_1||)|_{\mathbf{x}=\mathbf{x}_j} \\ \mathcal{L}\phi(\epsilon\,||\mathbf{x} - \mathbf{x}_2||)|_{\mathbf{x}=\mathbf{x}_j} \\ \vdots \\ \mathcal{L}\phi(\epsilon\,||\mathbf{x} - \mathbf{x}_n||)|_{\mathbf{x}=\mathbf{x}_j} \\ 0 \end{bmatrix} \tag{6.4}$$

This small system solve is repeated N times for each $\mathbf{x}_j$, $j = 1...N$, to form the DM associated with one derivative quantity. As an example, if $\mathcal{L}$ is the identity operator then the above procedure will lead to RBF-FD interpolation. If $\mathcal{L}$ is $\frac{\partial}{\partial x}$, it will lead to the DM that approximates the first derivative in x. While Equation 6.4 is symmetric, the added constraints for coefficient c_{n+1} detracts from the system's positive definite-ness. In lieu of this, a direct LU factorization solves for the weights. Also, observe that multiple right hand sides can be employed to efficiently obtain weights corresponding to all required derivative quantities (i.e., $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, ∇^2, etc.) in one system solve per stencil center.

6.3 GPU Based Solver

Author's Note: We should only discuss the ViennaCL implementation. Unless I can get my ILU preconditioner implemented for CUSP.

Our implementation leverages existing libraries for sparse matrix-vector operations on the GPU. Two libraries exist for sparse matrix linear algebra on the GPU: CUSP [**?**] and ViennaCL [**?**]. By implementing our algorithm in the context of ViennaCL, we directly benefit from improvements to the performance of underlying sparse matrix-vector product, vector dot vector and other linear algebra primitives. Also, ViennaCL provides seamless interoperability with the Boost::UBLAS, EIGEN and MTL libraries via C++ templates. We test the performance of our algorithm on one or more CPUs with the Boost::UBLAS library.

Table comparing CUSP and ViennaCL features

The GMRES algorithm was introduced in 1986 by Saad and Schultz [**?**]. The iterative solver support general matrix structures, whereas methods like Conjugate Gradient require symmetric positive definiteness.

6.3.1 GMRES Algorithm

At the core of the GMRES algorithm is an Arnoldi (orthogonalization) process. Author's Note: significance of orthogonalization.

Multiple variants of GMRES exist Author's Note: cite refs that utilize unique orthogonalization steps. The motivation behind alternative Arnoldi processes is to save both memory and operation counts. In some cases stability of the GMRES iterations can also be improved Author's Note: I recall a paper saved on my laptop. Saad [**?**] introduced a practical implementation of the GMRES method that utilizes Given's rotations to compute an implicit QR factorization. The Given's based algorithm is part of the CUSP library; ViennaCL implements the Householder reflection algorithm.

We have implemented the Given's rotation algorithm within ViennaCL, because it is simpler to implement in parallel and increases parallelism Author's Note: verify statement with reference.

[**?**] does not describe their use of rotations.)

Note that the application of a preconditioner such as ILU0 introduces an additional call to MPI_Alltoallv before everywhere M^{-1} is present in Algorithm 6.1.

6.3.2 Multiple GPUs

Domain Decomposition. A restricted additive Schwarz [**?**] domain decomposition is applied. Traditional additive Schwarz methods construct a restriction matrix R to constrain processor computation to a subdomain and R^T project it back onto the global domain. In contrast, restricted additive Schwarz replaces replaces the restriction matrix R with some restriction operator R_P^0 where $R_P^0 \subset R$, but continues to use R^T to project the solution back onto the global domain. The main idea behind re , restricted additive schwarz assigns all nodes to exactly one subdomain. Regular additive Schwarz would allow nodes to be in one or more subdomains.

Algorithm 6.1 Left-preconditioned GMRES(k) with Given's Rotations

ε (tolerance for the residual norm r), x_0 (initial guess), and set $convergence = false$
MPI_Alltoallv(x_0)
while $convergence == false$ **do**
    $r_0 = M^{-1}(b - Ax_0)$
    MPI_Alltoallv(r_0)
    $\beta = ||r_0||_2$ ▷ MPI_Allreduce($< r_0, r_0 >$)
    $v_1 = r_0/\beta$
    for $j = 1$ to k **do**
        $w_j = M^{-1}Av_j$ ▷ MPI_Alltoallv(w_j)
        for $i = 1$ to j **do**
            $h_{i,j} =< w_j, v_i >$ ▷ MPI_Allreduce
            $w_j = w_j - h_{i,j}v_i$
        end for
        $h_{j+1,j} = ||w_j||_2$ ▷ MPI_Allreduce
        $v_{j+1} = w_j/h_{j+1,j}$
    end for
    Set $V_k = [v_1, \cdots, v_k]$ and $\bar{H}_k = (h_{i,j})$ an upper Hesssenberg matrix of order $(m + 1) \times m$
    Solve a least-square problem of size m: $\min_{y \in \mathbb{R}^k} ||\beta e_1 - \bar{H}_k y||_2$
    $x_k = x_0 + V_k y_k$
    if $||M^{-1}(b - Ax_k)||_2 < \varepsilon$ **then**
        $convergence = true$
    end if
end while

$$\mathbf{u} = \sum_{p=0}^{numprocs} R'_p (A(R_p^0)^T)^{-1} R_P^0 R_P F \tag{6.5}$$

where R_p is a restriction matrix (identity on diagonals for stencils operated on by a processor, zero elsewhere). In our case, the partitions are determined by node coordinates, so all non-zeros for a stencil row end up on the same processor (i.e., the decomposition of the matrix does not allow multiple domain blocks per row).

MPI Communication. Mention communication using MPI_AlltoAllv, MPI_Allreduce. (Show communication points in algorithm—be sure to explicitly show Givens rotations in algorithm. [**?**] did list rotations. We also use the CPU for Gram Schmidt process.)

6.4 Multiple GPUs

Scaling GMRES across multiple GPUs requires a domain decomposition. Domain decompositions can be interpreted as partitioning of nodes or cuts along

Domain Decomposition. The domain is decomposed naïvely by cutting the domain into domain bounding box into P equal sized partitions. Each processor is assigned one partition. Partitions may contain unequal number of nodes. Future work will integrate a domain partitioning library such as ParMETIS [**?**] or SCOTCH [**?**] to improve load balancing for processors.

$$\mathbf{u} = \sum_{p=0}^{numprocs} R_p (A R_p^T)^{-1} F \tag{6.6}$$

where R_p is a restriction matrix (identity on diagonals for stencils operated on by a processor, zero elsewhere). In our case, the partitions are determined by node coordinates, so all non-zeros for a stencil row end up on the same processor (i.e., the decomposition of the matrix does not allow multiple domain blocks per row).

The restriction matrix R_p^0 contains only zeros and ones. Cai et al. [**?**] let R_p^0 contain ones on the diagonal to indicate rows of the system for which a processor p is responsible. Our implementation similarly places I on the diagonal, but we then apply a general permutation matrix P to compress $R_p^0 A$ such that processors assemble an $N_p \times m$ local matrix (where $N_p \ll N$ and $m \ll N$). Likewise, processors allocate vectors for the reduced problem size. The solution is $N_p \times 1$ and RHS is $m \times 1$. Recall that the domain decomposition is determined by node partitioning, which does not guarantee rows of the global system corresponding to stencils in a processor's partition will be a contiguous block of the full system.

6.4.1 Solution Ordering

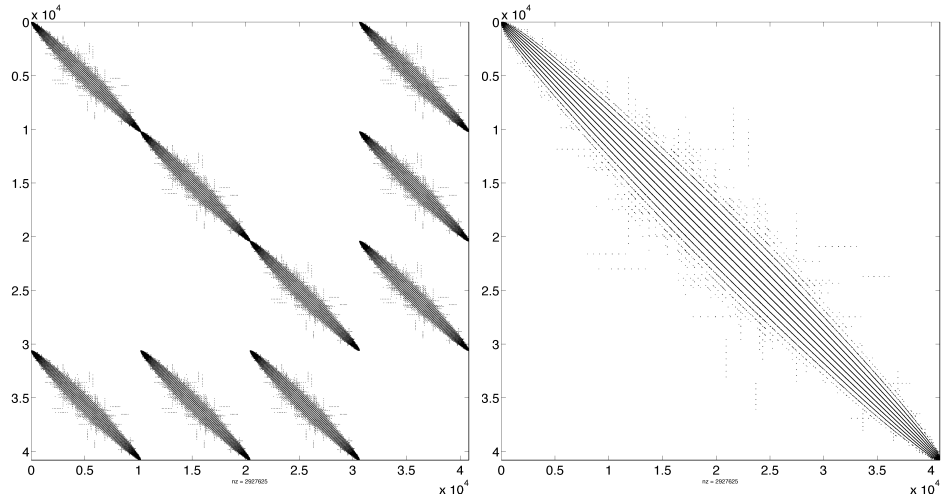
To simplify distribution of solution values, we apply a reordering to the system to interleave components of the solution and group solutions by node. This allows us to directly copy a double4 for each

The DM assembly depends on all dimensions of solution values for a single node to be consecutive in memory. While Equation **??** has all components of U, V, W and P grouped together, the values of u_1, v_1, w_1 and p_1 correspond to node (x_1, y_1, z_1).

Figure **??** demonstrates the effect of interleaving our solution. The sparsity pattern of the original DM with solutions grouped by component is shown in Figure 6.1a. Well defined blocks of non-zeros are filled with RBF-FD weights from Equation **??**. Figure **??** presents the sparsity pattern for interleaved solution components. The pattern is similar to a single block of Figure 6.1a, but the sub-matrix $(10 : 50) \times (10 : 50)$ of each solution ordering, shown in Figures **??** and **??**, illustrate that non-zeros in Figure **??** are small 4×4 blocks with the structure of Equation **??**.

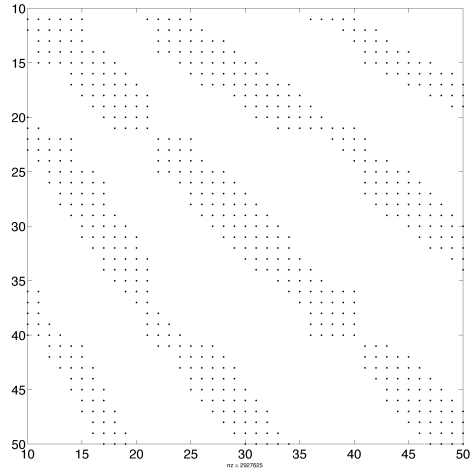
Node Order. Author's Note: Should we discuss node ordering and its implications on convergence? if so, I should state:

- Nodes are read from file
- Their order is either: unmodified or modified

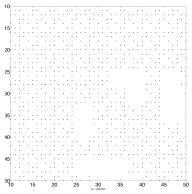


(a) Grouped by Solution Component

(b) Interleaved Solution Components



(c) Grouped Submatrix $(10:50) \times (10:50)$



(d) Interleaved Submatrix $(10:50) \times (10:50)$

Figure 6.1: Sparsity pattern of linear system in Equation **??**. Solution values are either ordered by component (e.g., $(u_1, \cdots, u_N, v_1, \cdots, v_N, w_1, \cdots, w_N, p_1, \cdots, p_N)^T$) or interleaved (e.g., $(u_1, v_1, w_1, p_1, \cdots, u_N, v_N, w_N, p_N)^T$).

- If modified, our goal is to improve memory access patterns by reducing the bandwidth of the DM. A bandwidth of n is the best case scenario where the solution vector is accessed linearly. The worst case scenario is when bandwidth is N and the solution is accessed randomly via stencils.

- Ordering nodes according to a space filling curve can reduce the "randomness" of solution access by placing elements of stencils nearby (sometimes sequential) in memory.

- Many space filling curves exist. Raster (IJK), Snake, Morton, Hilbert, U, X, etc. We

consider Raster here, with other orderings left for future work.

- Our restriction operator for domain decomposition might reduce the bandwidth for each processor compared to the global DM. How does the combination of restriction and reordering work?

- Reordering the DM has implications on its condition number and the convergence rate of the GMRES algorithm. We should provide bandwidth of matrix before and after reordering, as well as the condition number before and after. We should monitor convergence with and without reordering, and compare the number of GMRES iterations per minute/second.

MPI Communication. The majority of communication within our implementation consists of two MPI routines: MPI_Alltoallv and MPI_Allreduce.

6.5 Governing Equations

We solve the PDE on the surface of the sphere as an example for one and multiple GPUs. Boundary conditions detract from the accuracy of RBF-FD and introduce other issues such as Runge phenomena [**?**], so we first verify the solution without boundaries.

Author's Note: Need to reference work that solves problem in two steps and justify our approach to solve in one step. Golub paper might be good for this. Or the Stoke preconditioners paper

Assuming η is a constant (i.e., $\boldsymbol{\nabla}\eta = 0$), our system simplifies to

$$\begin{pmatrix} -\eta\nabla^2 & 0 & 0 & \frac{\partial}{\partial x_1} \\ 0 & -\eta\nabla^2 & 0 & \frac{\partial}{\partial x_2} \\ 0 & 0 & -\eta\nabla^2 & \frac{\partial}{\partial x_3} \\ \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ p \end{pmatrix} = \frac{RaT}{\sqrt{x_1^2 + x_2^2 + x_3^2}} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{pmatrix}. \tag{6.7}$$

where the ∇^2 operator in spherical polar coordinates for $\mathbb{R}^3$ is:

$$\nabla^2 = \underbrace{\frac{1}{\hat{r}}\frac{\partial}{\partial \hat{r}}\left(\hat{r}^2\frac{\partial}{\partial \hat{r}}\right)}_{\text{radial}} + \underbrace{\frac{1}{\hat{r}^2}\Delta_S}_{\text{angular}}. \tag{6.8}$$

Here Δ_S is the Laplace-Beltrami operator—i.e., the Laplacian constrained to the surface of the sphere with radius $\hat{r}$. This form nicely illustrates the components of the ∇^2 corresonding to the radial and angular terms.

On the surface of the unit sphere the radial term vanishes, so we are left with:

$$\nabla^2 \equiv \Delta_S. \tag{6.9}$$

The following RBF operator from [**?**]—Equation (20) can be applied to the RHS of Equation 6.4 to generate Laplace-Beltrami RBF-FD weights:

$$\Delta_S = \frac{1}{4}\left[\left(4 - r^2\right)\frac{\partial^2}{\partial r^2} + \frac{4 - 3r^2}{r}\frac{\partial}{\partial r}\right], \tag{6.10}$$

where r is the Euclidean distance between nodes of an RBF-FD stencil and is independent of our choice of coordinate system.

Additionally following [**?**, **?**], the off-diagonal blocks in Equation **??** must be constrained to the sphere via the projection matrix:

$$P = I - \mathbf{x}\mathbf{x}^T = \begin{pmatrix} (1-x_1^2) & -x_1x_2 & -x_1x_3 \\ -x_1x_2 & (1-x_2^2) & -x_2x_3 \\ -x_1x_3 & -x_2x_3 & (1-x_3^2) \end{pmatrix} = \begin{pmatrix} P_{x_1} \\ P_{x_2} \\ P_{x_3} \end{pmatrix} \tag{6.11}$$

where $\mathbf{x}$ is the unit normal at the stencil center, and [**?**, **?**] show that with a little manipulation weights can be directly computed with these operators for Equation 6.4:

$$P\frac{\partial}{\partial x_1} = (x_1\mathbf{x}^T\mathbf{x}_k - x_{1,k})\frac{1}{r}\frac{\partial}{\partial r}|_{\mathbf{x}=\mathbf{x}_j} \tag{6.12}$$

$$P\frac{\partial}{\partial x_2} = (x_2\mathbf{x}^T\mathbf{x}_k - x_{2,k})\frac{1}{r}\frac{\partial}{\partial r}|_{\mathbf{x}=\mathbf{x}_j} \tag{6.13}$$

$$P\frac{\partial}{\partial x_3} = (x_3\mathbf{x}^T\mathbf{x}_k - x_{3,k})\frac{1}{r}\frac{\partial}{\partial r}|_{\mathbf{x}=\mathbf{x}_j} \tag{6.14}$$

6.5.1 Constraints

Due to the lack of boundary conditions on the sphere, the family of solutions that satisfy the PDE in Equation **??** includes four free constants (one for each u_1, u_2, u_3 and p).

One way to close the null-space of the solution is to augment Equation **??** with the following constraints:

$$\left(\begin{array}{cccc:cccc} -\eta\nabla^2 & 0 & 0 & \frac{\partial}{\partial x_1} & 1_{N\times1} & 0 & 0 & 0 \\ 0 & -\eta\nabla^2 & 0 & \frac{\partial}{\partial x_2} & 0 & 1_{N\times1} & 0 & 0 \\ 0 & 0 & -\eta\nabla^2 & \frac{\partial}{\partial x_3} & 0 & 0 & 1_{N\times1} & 0 \\ \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} & 0 & 0 & 0 & 0 & 1_{N\times1} \\ \hdashline 1_{1\times N} & 0 & 0 & 0 & & & & \\ 0 & 1_{1\times N} & 0 & 0 & & 0_{4\times4} & & \\ 0 & 0 & 1_{1\times N} & 0 & & & & \\ 0 & 0 & 0 & 1_{1\times N} & & & & \end{array}\right) \left(\begin{array}{c} u_1 \\ u_2 \\ u_3 \\ p \\ \hdashline c_1 \\ c_2 \\ c_3 \\ c_4 \end{array}\right) = \frac{RaT}{\sqrt{x_1^2+x_2^2+x_3^2}} \left(\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ 0 \\ \hdashline \int_\Omega u_1 \partial\Omega \\ \int_\Omega u_2 \partial\Omega \\ \int_\Omega u_3 \partial\Omega \\ \int_\Omega p \partial\Omega \end{array}\right). \tag{6.15}$$

Author's Note: Need the integral of my manufactured solution on the RHS. ℓ_1 norm does not converge to 0, so it will screw solve with constraints where the subscript on $1_{N\times1}$ indicates a $N \times 1$ vector of ones. These constraints add the unknowns (c_1, c_2, c_3, c_4), which should solve to be zero. The four rows on the bottom require that the solution satisfy the integral over the domain for each solution component. In combination with the four added columns, the constraints indicate that the solution components must satisfy integrals using the same constant value. This is only possible if the constants are zero. Our added constraints are not chosen for physical significance, but for algebraic significance. When solving Equation **??** with GMRES, the constraints improve conditioning of the system and increase the rate of convergence.

We also investigate the use of GMRES without constraints. This increases the number of iterations required to converge, but allows increased parallelism (decreased data sharing). Author's Note: Perhaps we can iterate without constraints until convergence slows then "restart" the problem on a single GPU with constraints included? Limits scalability but would allow more parallelism for part of iterations while also reasonable convergence.

6.5.2 Manufactured Solution

To verify our implementation, we manufacture a solution that satisfies the continuity equation. Using the identity

$$\boldsymbol{\nabla} \cdot (\boldsymbol{\nabla} \times g(\mathbf{x})) = 0, \tag{6.16}$$

for any function $g(\mathbf{x})$, we can easily manufacture a solution by choosing some vector function $g(\mathbf{x})$, projecting it onto the sphere via $P_x g(x)$ and applying the curl projection, Q_x:

$$Q_x = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}. \tag{6.17}$$

Then, a manufactured solution that satisfies both momentum and continuity conditions on the surface of the sphere is given by:

$$\mathbf{u} = Q_x(g(\mathbf{x})) \tag{6.18}$$

Typically, on the surface of the sphere, the projection operator from Equation **??** must be applied to an arbitrary $g(\mathbf{x})$. Here, we choose the components of $g(\mathbf{x})$ to be various spherical harmonics in Cartesian coordinates where $P_x Y_l^m = Y_l^m$. Consequently, application of P_x can be ignored.

We select $g(x)$ to be:

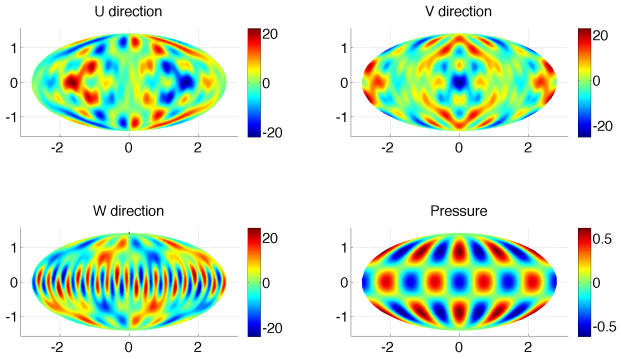
$$g(x) = 8Y_3^2 - 3Y_{10}^5 + Y_{20}^{20} \tag{6.19}$$

and the pressure function:

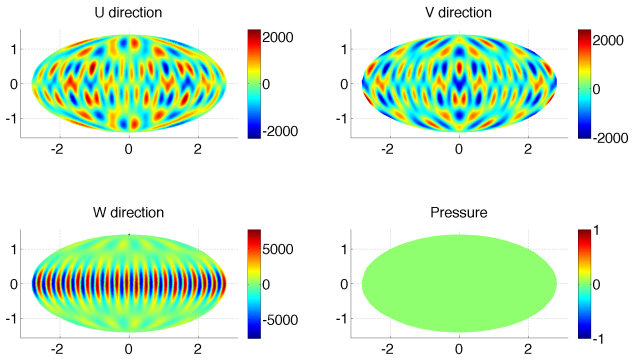
$$P = Y_6^4 \tag{6.20}$$

The manufactured solution is shown in Figure **??**. A Mollweide projection [**?**] maps the sphere onto the plane.

Convergence. As the problem size N increases, we expect the approximation to the solution to converge on the order of $\sqrt{n}$ where n is the choice of stencil size. Figure **??** demonstrates the convergence of our solution with respect to $\sqrt{N}$ for stencil sizes $n = 31{,}101$ Author's Note: update when figure is complete.



(a) Solution $Q_x(g(x))$ with $g(x) = 8Y_3^2 - 3Y_{10}^5 + Y_{20}^{20}$



(b) Right Hand Side (Constant Viscosity)

Figure 6.2: A divergence free field is manufactured for the sphere.

6.6 Preconditioning

GMRES is slow to converge when used without a preconditioner.

The differentation matrix produced by RBF-FD is asymmetric, non-positive definite, and non-diagonally dominant. Thus, many of the popular choices for preconditioning provided by CUSP and ViennaCL do not apply.

Our tests show that an incomplete LU factorization with zero fill-in [**?**] functions well.

We also find that a large Krylov subspace must be saved. GMRES converges best when approximately 250 dimensions are saved between restarts.

Plot comparing residual of GMRES without precond and with ILU0

Author's Note: Need to comment on the conditioning of the system and how stencils can influence convergence .

Need a table/plot comparing convergence of various preconditioners (ILU0, ILUT, AMG, etc.). We will justify our use of ILU0 even if it is the most basic and frequently least beneficial approach.

Demonstrate ILU0 is best for converging between

- Jacobi
- ILU0 on block $(1 : 3 * N)$
- ILU0 on full matrix
- ILUT
- AMG

Algorithm 6.2 Incomplete LU Factorization with Zero Fill-in (ILU0)

1: **for** i **do** $= 0$
2: $\quad a_{i,i} = a_{i,i}/a_{i,i}$
3: **end for**

6.7 GMRES Results

We want to express benchmarks in terms of the number of GMRES iterations per second, and the number of iterations required to converge. Readers wont care what percentage of peak we are getting, just how fast we get to the solution.

- One GPU
  - Convergence for stokes steady
  - GMRES iteration plot (assuming 1e-8 and restart=60)
- Multi-GPU
  - Number of iterations without constraints
  - Number of iterations with constraints
  - GMRES iteration plot
  - plot: number of GMRES iterations per second w.r.t. number of processors

CHAPTER 7

COMMUNITY OUTREACH

- Evan F. Bollig, Natasha Flyer, and Gordon Erlebacher. Using Radial Basis Function Finite Difference (RBF-FD) for PDE Solutions on the GPU. *submitted to Elsevier J. Comput. Phys.*, 2011
-
-
-
-
-

need Survey of goals stated in prospectus and whether they were completed or not (Probably part of slides, not actual dissertation)

BIBLIOGRAPHY

[1] HMPP Data Sheet. http://www.caps-entreprise.com/upload/article/fichier/64fichier1.pdf, 2009.

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

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

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

[5] Evan F. Bollig, Natasha Flyer, and Gordon Erlebacher. Using Radial Basis Function Finite Difference (RBF-FD) for PDE Solutions on the GPU. *submitted to Elsevier J. Comput. Phys.*, 2011.

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

[7] J. C. Carr, R. K. Beatson, B. C. McCallum, W. R. Fright, T. J. McLennan, and T. J. Mitchell. 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*, pages 119–ff, New York, NY, USA, 2003. ACM.

[8] Tom Cecil, Jianliang Qian, and Stanley Osher. Numerical Methods for High Dimensional Hamilton-Jacobi Equations Using Radial Basis Functions. *JOURNAL OF COMPUTATIONAL PHYSICS*, 196:327–347, 2004.

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

[10] P P Chinchapatnam, K Djidjeli, P B Nair, and M Tan. 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):275–290, March 2009.

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

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

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

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

[15] Qiang Du, Vance Faber, and Max Gunzburger. Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM Rev.*, 41(4):637–676, 1999.

[16] Gregory E. Fasshauer. *Meshfree Approximation Methods with MATLAB*, volume 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, Singapore 596224, 2007.

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

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

[19] Natasha Flyer, Erik Lehto, Sebastien Blaise, Grady B. Wright, and Amik St-Cyr. Rbf-generated finite differences for nonlinear transport on a sphere: shallow water simulations. *Submitted to Elsevier*, pages 1–29, 2011.

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

[21] Natasha Flyer and Grady B. Wright. A Radial Basis Function Method for the Shallow Water Equations on a Sphere. In *Proc. R. Soc. A*, volume 465, pages 1949–1976, December 2009.

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

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

[24] Bengt Fornberg and Natasha Flyer. Accuracy of Radial Basis Function Interpolation and Derivative Approximations on 1-D Infinite Grids. *Adv. Comput. Math*, 23:5–20, 2005.

[25] Bengt Fornberg, Elisabeth Larsson, and Natasha Flyer. Stable Computations with Gaussian Radial Basis Functions. *SIAM J. on Scientific Computing*, 33(2):869—-892, 2011.

[26] Bengt Fornberg and Erik Lehto. Stabilization of RBF-generated finite difference methods for convective PDEs. *Journal of Computational Physics*, 230(6):2270–2285, March 2011.

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

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

[29] E. J. Fuselier and G. B. Wright. A High-Order Kernel Method for Diffusion and Reaction-Diffusion Equations on Surfaces. *ArXiv e-prints*, May 2012.

[30] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 55–64, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

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

[32] A. Iske. *Multiresolution Methods in Scattered Data Modeling*. Springer, 2004.

[33] L. Ivan, H. De Sterck, S. A. Northrup, and C. P. T. Groth. Three-Dimensional MHD on Cubed-Sphere Grids: Parallel Solution-Adaptive Simulation Framework. In *20th AIAA CFD Conference*, number 3382, pages 1325–1342, 2011.

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

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

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

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

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

[39] Elisabeth Larsson and Bengt Fornberg. A Numerical Study of some Radial Basis Function based Solution Methods for Elliptic PDEs. *Comput. Math. Appl*, 46:891–902, 2003.

[40] Shaofan Li and Wing K. Liu. *Meshfree Particle Methods*. Springer Publishing Company, Incorporated, 2007.

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

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

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

[44] Jia Pan and Dinesh Manocha. Fast GPU-based Locality Sensitive Hashing for K-Nearest Neighbor Computation. *Proceedings of the 19th ACM SIGSPATIAL GIS '11*, 2011.

[45] Portland Group Inc. *CUDA Fortran Programming Guide and Reference*, 1.0 edition, November 2009.

[46] DA Randall, TD Ringler, and RP Heikes. Climate modeling with spherical geodesic grids. *Computing in Science & Engineering*, pages 32–41, 2002.

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

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

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

[50] C. Shu, H. Ding, and K. S. Yeo. 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):941 – 954, 2003.

[51] Ian H. Sloan and Robert S. Womersley. Extremal systems of points and numerical integration on the sphere. *Adv. Comput. Math*, 21:107–125, 2003.

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

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

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

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

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

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

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

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

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

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