TABLE OF CONTENTS

Abstract ii

1 Parallel Solvers **1**
- 1.1 Paritioning 1
- 1.2 Global and local index maps 2
- 1.3 Local node ordering 2
- 1.4 Two level parallelism 2
- 1.5 Distributed Solvers 2
  - 1.5.1 Node Partitioning 2
  - 1.5.2 Global and Local Indices 3
  - 1.5.3 Local Node Ordering 3
- 1.6 Overlapping Computation and Communication 6
- 1.7 Targeting the GPU 6
  - 1.7.1 Explicit Solvers 7
  - 1.7.2 Fragments (integrate above) 10
  - 1.7.3 Implicit Solvers 11

Bibliography 12

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

PARALLEL SOLVERS

Solving PDEs in a distributed computing environment requires three design decisions [8]. First, the domain is partitioned and distributed across compute nodes in some fashion. Intelligent partitioning impacts load balancing on processors to minimize computation to communication ratio; imbalanced computation across processors can cause excessive delay per iteration as processors wait to receive information. Second, one must decide what information each processor is able to access regarding node information, solution values, etc and establish index mappings that translate between a processor's local context and the global problem. In situations where processors are not aware of all nodes/solution values in the global domain, the index mappings are essential to maintaining solution consistency at each time-step. Last but not least, each processor can re-order nodes locally in an effort to improve solver efficiency and local system conditioning. Node ordering also allows us to minimize data transfer between CPU and GPU.

Parallelization of the RBF-FD method is achieved at two levels. First, the physical domain of the problem—in this case, the unit sphere—is partitioned into overlapping subdomains, each handled by a different CPU process. All CPUs operate independently to compute/load RBF-FD stencil weights, run diagnostic tests and perform other initialization tasks. A CPU computes only weights corresponding to stencils centered in the interior of its partition. After initialization, CPUs continue concurrently to solve the PDE. 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 (**??**) 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.

1.1 Paritioning

We partition the domain assuming that domains overlap. Processor views of the domain are limited to only nodes required by stencils under processor control. Subset of the grid. Subset of the stencils.

Preliminary decomposition is slicing horizontally in the domain.
Load balancing important and can be handled by METIS.

1.2 Global and local index maps

Due to the local view of each processor, local index maps are much shorter than global maps. They run from 0 to N_p.
STL Maps based

1.3 Local node ordering

The local index set is ordered as QmB, BmO, O, R
Domain boundary nodes appear at beginning of the list

1.4 Two level parallelism

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 [5], 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.

1.5 Distributed Solvers

Need partitioning, global and local index maps, and local ordering for a distributed [8]

1.5.1 Node Partitioning

Figure 1.1 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. Author's Note: Q: what is the percentage overlap for n? $\frac{1}{2}n^{\frac{1}{d}}$ gives depth into neighbor since n is uniformly sampled we expect a cube shape. (SHould be literature on this...no?)

The linear partitioning in Figure 1.1 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 [3] or icosahedral

geodesic grid [7], which can evenly balance the computational load across partitions. Other interesting partitionings can be generated with software libraries such as the METIS [4] family of algorithms, capable of partitioning and reordering directed graphs produced by RBF-FD stencils.

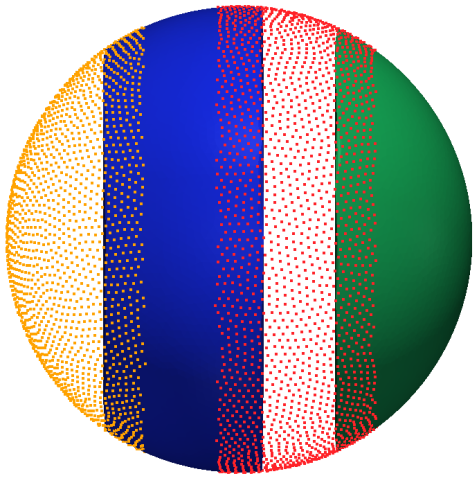

Figure 1.1: Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

Author's Note: Need to flesh detail on ParMETIS, two dimensional partitioning and case when a node value is required by multiple processors

1.5.2 Global and Local Indices

1.5.3 Local Node Ordering

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

It is convenient to break up the nodes on a given CPU into various sets according to whether they are sent to other processors, are retrieved from other processors, are permanently on the processor, etc. Note as well, that each node has a home processor since the RBF nodes are partitioned into multiple domains without overlap. Table 1.1, defines the collection of index lists that each CPU must maintain for both multi-CPU and multi-GPU implementations.

Figure 1.2 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

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

Table 1.1: Sets defined for stencil distribution to multiple CPUs

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 1.2 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 $\mathcal{G}$,

$$\mathcal{G} = \{\mathcal{Q} \backslash \mathcal{B} \quad \mathcal{B} \backslash \mathcal{O} \quad \mathcal{O} \quad \mathcal{R}\}, \tag{1.1}$$

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 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

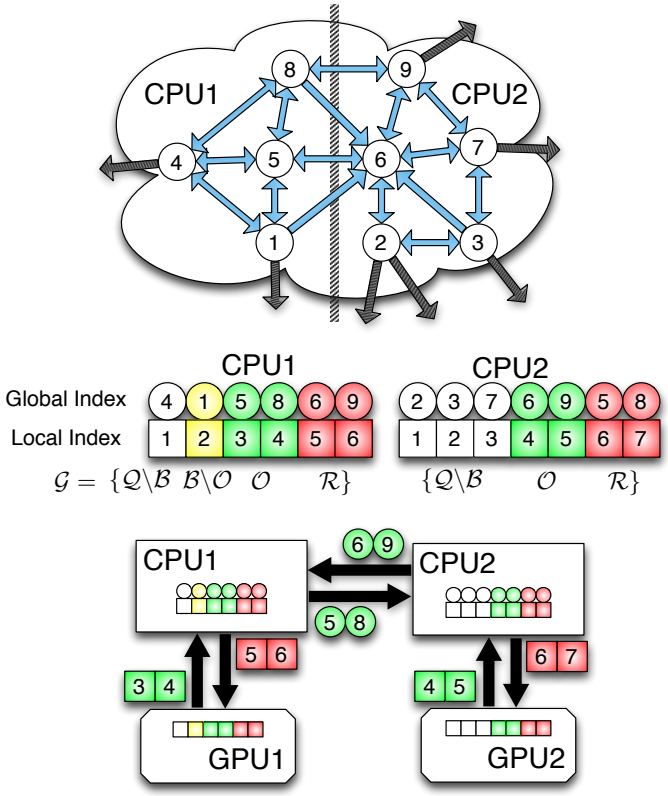

Figure 1.2: Partitioning, index mappings and memory transfers for nine stencils ($n = 5$) spanning two CPUs and two GPUs. Top: the directed graph created by stencil edges is partitioned for two CPUs. Middle: the partitioned stencil centers are reordered locally by each CPU to keep values sent to/received from other CPUs contiguous in memory. Bottom: to synchronize GPUs, CPUs must act as intermediaries for communication and global to local index translation. Middle and Bottom: color coding on indices indicates membership in sets from Table 1.1: $\mathcal{Q}\backslash\mathcal{B}$ is white, $\mathcal{B}\backslash\mathcal{O}$ is yellow, $\mathcal{O}$ is green and $\mathcal{R}$ is red.

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 1.2. Each GPU operates on the local indices ordered according to Equation (1.1). 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.

1.6 Overlapping Computation and Communication

Need to describe our approach to launch

In simplest form (& indicates an overlapped execution; overlapping is possible by non-blocking launches to the GPU before comm is started):

- $U'_{Q\setminus B} = A_{Q\setminus B}U_{Q\setminus B}$
- & MPI_alltoallv
- $U'_B = A_B U_B$

This form benefits because stencils are operated on in their entirety. For the GPU this is important because we maintain SIMD nature.

A more complex form which might overlap more comm and comp depending on the size of B (i.e., might be beneficial for large stencil sizes):

- $U'_{Q\setminus B} = A_{Q\setminus B}U_{Q\setminus B}$
- & MPI_alltoallv
- & $U'_B = A_{B\setminus R}U_{B\setminus R}$ (partial sparse vector product)
- $U'_B += A_R U_R$

Note that there is no non-blocking version of MPI_alltoallv within the MPI standard, so the above two algorithms will not be overlapped when computing on the CPU. To overlap on the CPU, [**?**] refers to the non-blocking collective as MPI_ialltoallv. With a collective like this, we could enforce a pattern like:

- MPI_alltoallv
- & $U'_{Q\setminus B} = A_{Q\setminus B}U_{Q\setminus B}$
- & $U'_B = A_{B\setminus R}U_{B\setminus R}$ (partial sparse vector product)
- MPI_wait
- $U'_B += A_R U_R$

1.7 Targeting the GPU

Author's Note: The basic primitives necessary to parallelize our problems: SpMV, AXPY and

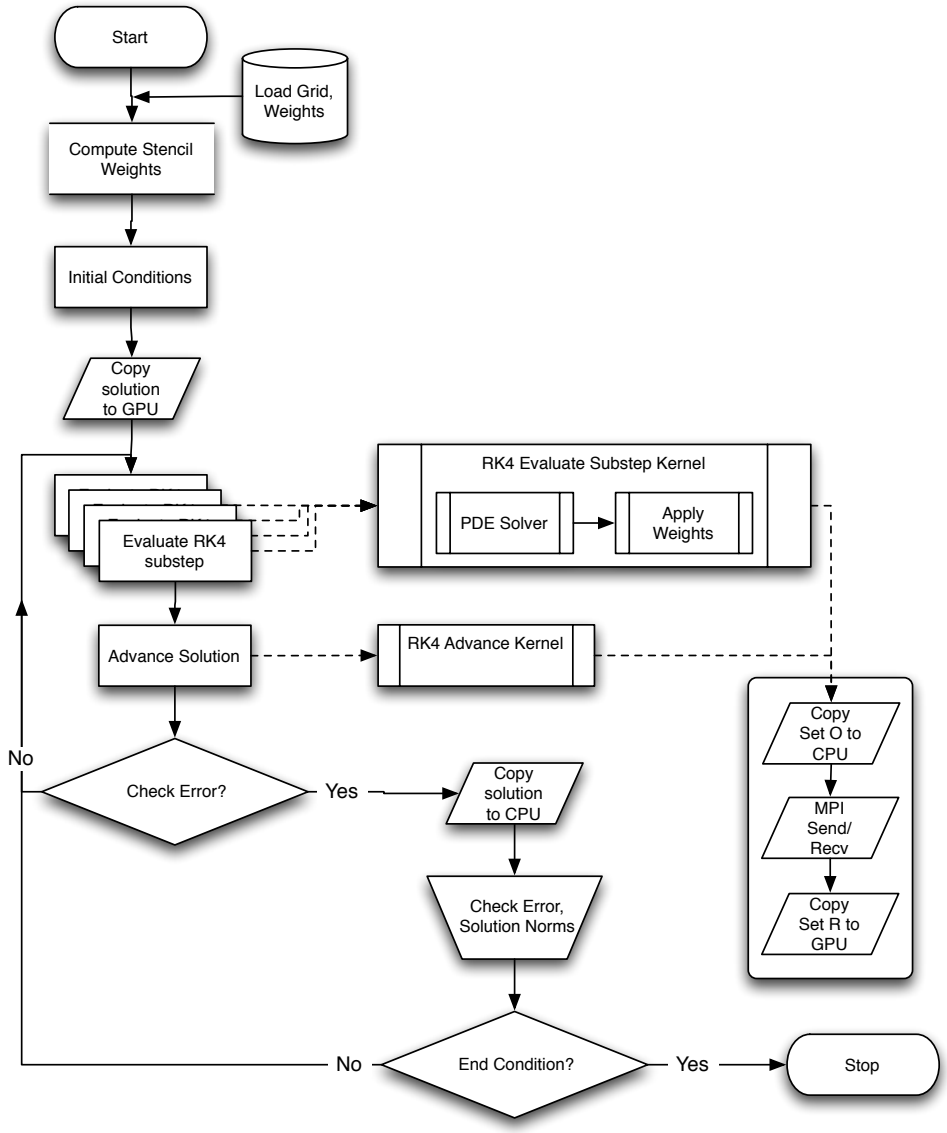

Figure 1.3: Workflow for RK4 on multiple GPUs.

1.7.1 Explicit Solvers

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 1.3 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 (**??**) 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 [5] 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 [6]. OpenCL assumes a tiered memory hierarchy that provides fast but small *local memory* space that is shared within a work-group [5]. Local memory on Fermi GPUs is 48 KB per multiprocessor [6]. 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 [1], 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 1.4). 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 [6]. 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.

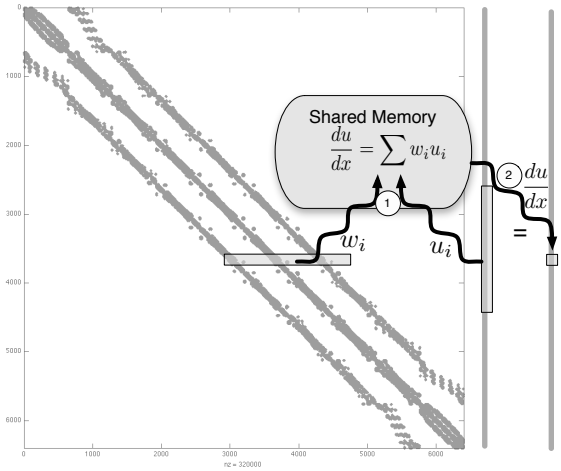

Figure 1.4: Naive approach to sparse matrix-vector multiply. Each thread is responsible for the sparse vector dot product of weights and solution values for derivatives at a single stencil.

Alternate Approach: One warp per stencil. An alternate approach, illustrated in Figure 1.5, 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

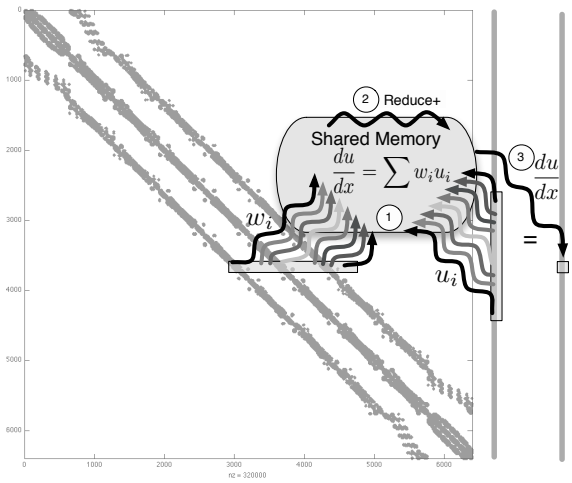

Figure 1.5: Alternative approach. A full warp (32 threads) collaborate to apply weights and compute the derivative at a stencil center.

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 [2]).

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.

dirichlet: kernel copies dirichlet values into place Author's Note: Need to dust off code

1.7.2 Fragments (integrate above)

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 [5].

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 [6], 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.

1.7.3 Implicit Solvers

Perhaps this section should move up. I think it might be best to discuss GMRES and iterative methods before Distributed Solvers (that way we can say the general solution form is "matrix form". but in parallel we need to use distributed algorithms.

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

BIBLIOGRAPHY

[1] 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. 9

[2] 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. 10

[3] 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. 2

[4] 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. 3

[5] Khronos OpenCL Working Group. *The OpenCL Specification (Version: 1.0.48)*, October 2009. 2, 8, 11

[6] NVidia. *NVIDIA CUDA - NVIDIA CUDA C - Programming Guide version 4.0*, March 2011. 8, 9, 11

[7] DA Randall, TD Ringler, and RP Heikes. Climate modeling with spherical geodesic grids. *Computing in Science & Engineering*, pages 32–41, 2002. 3

[8] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, second edition, 2003. 1, 2