PARALLEL-STREAMING ALGORITHMS FOR SOLVING PARTIAL DIFFERENTIAL EQUATIONS ON UNSTRUCTURED MESHES

by

Zhisong Fu

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

December 2013

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Zhisong Fu**

has been approved by the following supervisory committee members:

Ross T. Whitaker	, Cochair	**06/28/2013** Date Approved
Robert M. Kirby	, Cochair	**06/28/2013** Date Approved
Mary Hall	, Member	**06/28/2013** Date Approved
Robert S. MacLeod	, Member	**06/27/2013** Date Approved
Jonathan M. Cohen	, Member	**06/28/2013** Date Approved

and by **Alan Davis** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Partial differential equations (PDEs) are widely used in science and engineering to model phenomena such as sound, heat, and electrostatics. In many practical science and engineering applications, the solutions of PDEs require the tessellation of computational domains into unstructured meshes and entail computationally expensive and time-consuming processes. Therefore, efficient and fast PDE solving techniques on unstructured meshes are important in these applications. Relative to CPUs, the faster growth curves in the speed and greater power efficiency of the SIMD streaming processors, such as GPUs, have gained them an increasingly important role in the high-performance computing area. Combining suitable parallel algorithms and these streaming processors, we can develop very efficient numerical solvers of PDEs.

The contributions of this dissertation are twofold: proposal of two general strategies to design efficient PDE solvers on GPUs and the specific applications of these strategies to solve different types of PDEs. Specifically, this dissertation consists of four parts. First, we describe the general strategies, the domain decomposition strategy and the hybrid gathering strategy. Next, we introduce a parallel algorithm for solving the eikonal equation on fully unstructured meshes efficiently. Third, we present the algorithms and data structures necessary to move the entire FEM pipeline to the GPU. Fourth, we propose a parallel algorithm for solving the levelset equation on fully unstructured 2D or 3D meshes or manifolds. This algorithm combines a narrowband scheme with domain decomposition for efficient levelset equation solving.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I thank my advisors, Ross Whitaker and Mike Kirby for providing invaluable guidance, both technical and methodological, throughout my graduate studies. I also thank my committee members, Mary Hall, Rob McLeod, and Jonathan Cohen for their valuable feedback and discussions. Many thanks go to the Scientific Computing and Imaging Institute for providing a stimulating research environment at the University of Utah. Finally, I thank my family and my friends for their continuous support.

CHAPTER 1

INTRODUCTION

There are generally three types of PDEs: hyperbolic, elliptic, and parabolic [2]. Different types of PDEs have different properties that dictate the numerical methods appropriate to solve them, and different numerical methods require different algorithms and data structures to perform efficiently on GPUs. This dissertation presents a set of algorithms and data structures to efficiently solve the canonical equations of hyperbolic and elliptic PDEs on GPUs. Parabolic PDEs can typically be solved as elliptic equations with implicit temporal discretization.

1.1 Motivation

1.1.1 PDEs and Unstructured Meshes

PDEs are ubiquitous in science and engineering. They are used to model a wide variety of phenomena such as sound, heat, electrostatics, electrodynamics, fluid flow, and elasticity. Some PDEs can be solved analytically by separation of variables, but many of the PDEs associated with practical science and engineering problems are difficult or even impossible to solve in this way; they have to be solved numerically instead. To numerically solve a PDE, the solver typically needs to tessellate the computational domain into a structured or unstructured mesh. The studies in this dissertation are focused on PDE solutions on fully unstructured meshes (triangular meshes or tetrahedral meshes) for two reasons. First, unstructured meshes are widely used and have many advantages over structured meshes. One advantage of unstructured meshes is that they can handle complex computational domains accurately and efficiently. Many practical science and engineering problems need to solve PDEs on complex computational domains and require fully unstructured body-fitting meshes. For example, in cardiac simulations, the domain is a volume bounded by a smooth, curved surface, and

triangle meshing strategies for surfaces combined with tetrahedral meshing of the interior can accurately and efficiently capture these irregular domains [97] (*e.g.*, see Figure 1.1-left). Another advantage of using unstructured meshes is that vertices can be set on the domain boundary surfaces or interfaces of different regions/materials, allowing for greater accuracy later when applying a numerical method. For instance, geometric optics (Figure 1.1-right) or geophysics applications often require irregular unstructured meshes for accurate, efficient modeling of material discontinuities that are represented as triangulated surfaces embedded in a tetrahedral mesh. The other reason for focusing on unstructured meshes in this dissertation is that unstructured meshes pose several challenges for efficient PDE solution on GPUs that have not been addressed in the literature.

1.1.2 Numerical PDE Solution on GPUs and Challenges

Since the invention of computers, much research has been focused on the numerical solution of PDEs, and a large number of numerical methods are introduced in the literature to solve the PDEs numerically on computers [2]. One

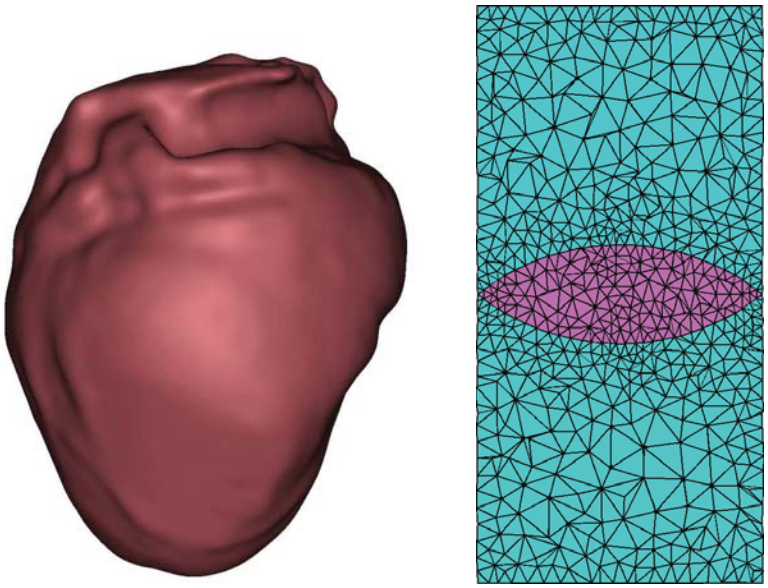


Figure 1.1. Examples of body-fitting meshes used for numerical simulation. On the left is the surface of a heart model mesh used for bioelectric computation. On the right is a cross-section of a lens model used for the simulation of geometric optics (the blue region denotes air while purple denotes the location of the lens).

of the most important metrics by which to measure these numeral methods is the efficiency in terms of running time. Developing efficient numerical solvers typically entails exploiting the computing capability of modern computer architecture and hardware and designing suitable algorithms and data structures accordingly.

Recent developments in computer hardware show that performance improvements will no longer be driven primarily by increased clock speeds, but by parallelism and hardware specialization. Single-core performance is leveling off, while hex-core CPUs are available as commodities; soon, conventional CPUs will have tens of parallel cores. Commodity multimedia processors such as the IBM Cell and graphics processing units (GPUs) are considered forerunners of this trend. These processors offer highly parallel *streaming architectures* that promise very large computational capabilities on computers that are affordable for single-person use. As an example, currently available GPUs can attain over a TeraFLOP in terms of peak double-precision performance and over three TeraFLOP for single-precision operations [69] on one's desktop machine! In addition, the faster growth curves in the speed and increased power efficiency of GPUs relative to CPUs have gained them an increasingly important role in the high performance computing. They are now widely used as floating point accelerators in supercomputers, which can be defined as devices that carry out arithmetic operations concurrently with or in place of the CPU. For example, Titan [66] is equipped with 18,688 GPUs that contribute over 90% of the peak performance. Developing efficient code for such accelerators is a very important building block of fully utilizing these supercomputers. Another reason for the increasing importance of GPUs is the emergence of general-purpose programming languages to facilitate implementing scientific applications on the graphics hardware. The practicability and performance of CUDA (compute unified device architecture) and OpenCL, an open-source standard of GPGPU programming, greatly help scientists to fully explore the large potential computing power of the GPUs.

However, GPUs' computing power does come at a cost; it requires a fairly restrictive computational model — a highly-constrained single-instruction multiple-datastream (SIMD) paradigm. These modern SIMD architectures offer many

parallel computing units (up to several thousand cores) in a tiered data-sharing structure, basic branching circuits, and ample memory bandwidth to limited caches. These functional restrictions are not a coincidence. They are considered an essential aspect of obtaining this raw computational power with conventional fabrication technologies, because they significantly simplify the logic required for synchronization and memory access. In addition, due to the relatively small cache space, compact representation and reuse of data is essential to performance.

The restrictions made by modern streaming architectures place significant demands on the design of numerical algorithms. Thus, algorithms that efficiently take advantage of these architectures are of significant interest. However, algorithms that achieve optimal performance on modern streaming architectures cannot be obtained by a straightforward mapping of numerical codes to these architectures. For example, an optimized version of a reduction operation is 30 times faster than the unoptimized version [68]. Typically, numerical algorithms that are efficient on streaming architectures should be specially designed for such architectures with suitable parallelism strategy and memory access patterns for the specific problem. However, the specialization of the algorithms can result in loss of problem generality, and this presents a challenge in the design of APIs to allow application scientists to easily take advantage of these new capabilities [42]. In practice, we imagine that template metaprogramming would be combined with an application specific API to allow the program to choose different optimization strategies based on classes of equations and parameters.

Because of the architecture restrictions of the GPU, solving PDEs on GPUs for unstructured meshes are particularly challenging. First, there is no natural partition of the domain for parallelism, and arbitrary decomposition of computations (*e.g.*, decompose by indices) usually leads to poor cache performance and an unbalanced workload. Second, for regular meshes, the valence of the nodes is the same, and hence nodal parallelism is typically employed that assigns each node to a thread. But for unstructured meshes, the nodes have variant valences that lead to irregular data structure and unbalanced workload. Third, it is much harder to handle the data exchange between partitions, and additional computations and a separate

data structure to find and store the boundary locations are typically needed to handle the boundary communications.

1.2 Contributions

In this dissertation, the research aims to develop efficient numerical PDE solvers, particularly on domains tessellated to body-fitting unstructured meshes. This goal is achieved by exploiting the huge computing power of the state-of-the-art SIMD streaming processors. Specifically, this dissertation explores the GPU-based, efficient solution of two types of PDEs, hyperbolic equations (the static and dynamic Hamilton-Jacobi equations) and elliptic equations (the Helmoholtz and Poisson equations), with different numerical methods. The contributions of this dissertation can be summarized into the following four categories.

1. General strategies. This dissertation presents two general strategies, the domain decomposition strategy and the hybrid gathering strategy, for designing efficient PDE solvers for unstructured meshes. We apply these strategies in all of our PDE solvers in this dissertation.

2. A fast iterative method for solving the eikonal equations on unstructured domains. This dissertation introduces a parallel algorithm for solving the eikonal equation with both isotropic and anisotropic speed functions on fully unstructured meshes. The method is appropriate for the type of fine-grained parallelism found on modern massively-SIMD architectures such as GPUs and takes into account the particular constraints and capabilities of these computing platforms. We have implemented the algorithm on a single CPU, as well as multicore CPUs with shared memory and a single GPU, with comparative results against state-of-the-art eikonal solvers. This is the first GPU implementation to solve the eikonal equation on unstructured meshes in the literature.

3. Architecting the finite element method pipeline for the GPU. This dissertation presents the algorithms and data-structures necessary to move the entire FEM pipeline to the GPU. Specifically, we propose an efficient GPU-based algorithm to generate local element information and to assemble the global linear system associated with the FEM discretization of an elliptic PDE. To solve the corre-

sponding linear system efficiently on the GPU, we have implemented a conjugate gradient method preconditioned with a geometry-informed algebraic multigrid (AMG) method preconditioner. We also introduce a new fine-grained parallelism strategy, a corresponding multigrid cycling stage, and efficient data mapping to the many-core architecture of GPU.

4. Fast parallel solver for the levelset equations on unstructured domains. This dissertation introduces a parallel algorithm for solving the levelset equation on fully unstructured 2D or 3D meshes or manifolds. We propose to combine the narrowband scheme and domain decomposition for efficient levelset equation solving. We also present the efficient narrowband fast iterative method (nbFIM) to compute the distance transform by solving an eikonal equation and the patched narrowband (patchNB) scheme to evolve the embedding. We apply the hybrid gathering parallelism strategy to enable regular and lock-free computations in both the nbFIM and patchNB.

1.3 Document Organization

Chapter 2 of this dissertation describes the general strategies of designing efficient PDE solvers. Chapter 3 introduces an efficient iterative method to solve the eikonal equation with isotropic speed functions on triangular meshes. Chapter 4 presents a fast solver for the eikonal equation with both isotropic and anisotropic speed functions on 3D tetrahedral meshes. These two chapters correspond to papers [37, 38], respectively. Next, a GPU-based pipeline for the finite element method is presented in Chapter 5, which is based on paper [39]. Then, an efficient parallel solver for the levelset equations is introduced in Chapter 6, and this chapter corresponds to paper [40]. I keep Chapters 3- 6 basically the same as the corresponding papers, and hence there are some redundancies. Finally, I wrap up the dissertation by summarizing of the proposed dissertation research and proposing future research directions in Chapter 7.

CHAPTER 2

ALGORITHM DESIGN STRATEGIES

This chapter develops two general strategies to design efficient algorithms to numerically solve the PDEs on GPUs. Both strategies are applied in all the PDE solvers introduced in this dissertation. The first strategy is *domain decomposition* that is used to decompose the computations among GPU streaming multiprocessors and cores. The other strategy is called *hybrid gathering* that we use to avoid contention without atomic operations that are expensive on GPUs.

2.1 Domain Decomposition

The term "domain decomposition methods" typically refers to a group of numerical methods that solve a boundary value problem by splitting it into smaller boundary value problems on subdomains and iterating to coordinate the solution between adjacent subdomains. The problems on the subdomains are independent, which makes domain decomposition methods suitable for parallel computing. Traditionally, this approach is used to provide coarse-grained parallelism for computation on multicore processors or clusters. The computation of each subdomain is assigned to a thread that is executed on a core of a multicore processor or a node of a cluster. The research in this dissertation uses domain decomposition to solve PDEs on unstructured meshes on GPUs. Here are some considerations when designing these algorithms:

1. Fine-grained parallelism and SIMD operation. Modern GPUs are equipped with up to tens of streaming multiprocessors (SM), with each of them having up to hundreds of cores. This architecture desires that the computations in a subdomain be further decomposed into finer tasks that are assigned to the threads of a block and executed by the cores of a SM. Ideally, these tasks should be *regular* so that they can be performed efficiently in a SIMD fashion. For

example, in a typical PDE solver for unstructured meshes, one can choose to use node-based parallelism or element-based parallelism. Usually, the element-based parallelism provide more regular computations. Therefore, when we perform domain decomposition, we typically decompose the computational domain into patches of elements and assign the computations on the elements of a patch to a thread block. However, in some problems, there is no natural way to decompose the computational domain that provides fine-grained parallelism, and in this case, we can *create* such decomposition by replacing the original computational primitives (*e.g.*, computation on an element) with patches. This *subdomain creation* idea is used in the solution of the eikonal equation, a hyperbolic PDE, presented in Chapters 3 and 4. One of the properties of the hyperbolic PDEs is that information propagates across the computational domain from the boundaries at a finite speed, and the solver needs to perform many iterations for the whole domain to converge. In each iteration, only the computations in the region around the information propagating front are useful, and this region is called the narrowband. To save computation, typically only the values of the nodes in the narrowband are updated in each iteration, and hence the actual computational domain is the narrowband. It is hard to decompose the narrowband, which has arbitrary shape, size, and topology. In addition, the narrowband is deforming with iterations and needs to be decomposed for every iteration, which is expensive. In this case, we employ a patched update scheme to "create" subdomains. This method decomposes the whole domain into patches, and treats these patches, instead of nodes, as computational primitives that are moved in and out of the narrowband. In each iteration, the solver assigns the patches inside the narrowband to thread blocks that are executed on GPU SMs. Now the narrowband is still changing every iteration, but there always exists a natural domain decomposition that provide fine-grained parallelism no matter how the narrowband changes.

2. Locality in the GPU memory hierarchy. The GPU architecture features a hierarchical memory space, consisting of the slow global memory accessible by all threads and the fast but small cache (*e.g.*, shared memory, registers) only accessible by a block or a thread. A typical kernel function consists of three steps:

loading data from global memory, performing computation, and storing data back to global memory. The global memory accesses have relatively high latency, and hence ideally, the computational density should be high enough to hide the latency. One way to increase computational density is to store the data in the fast cache and perform extra computations on the cache. In this case, these extra computations are fast and do not affect the overall performance, as long as the patch size is determined according to the cache size to make sure that the subdomain can fit into the cache. On the basis of this logic, when we design the algorithms for the PDEs solvers, we choose numerical schemes that can benefit from the extra fast computations, and when performing the domain decomposition, we determine the subdomain size according to the cache size. For example, in the AMG linear system solver introduced in Chapter 5, we choose to use block Jacobi method for the relaxations in place of the Jacobi method. The block Jacobi method more effectively smooths out high-frequency errors by performing multiple Jacobi iterations inside each patch, and hence increases the convergence speed of the global Conjugate Gradient method. Combined with our specially designed algorithm and data structures, this method can take advantage of the cheap computations and achieve great overall performance.

2.2 Hybrid Gathering Scheme to Avoid Contention

To solve PDEs on unstructured meshes, we can choose to use a node-based parallelism or element-based parallelism to decompose the computations. For meshes with unique-shape elements (*e.g.*, triangular mesh), element-based parallelism typically leads to more regular computations and is more suitable for SIMD operations. However, element-based parallelism may introduce contention since degrees of freedom typically live on the nodes, and hence multiple elements can be updating the same node at the same time. Typically, this contention problem is solved by using atomic operations, but such operations are expensive on GPUs, especially for double precision operations. Therefore, when designing the algorithms for GPU-based PDE solvers on unstructured meshes, our strategy is to avoid contention with some GPU-suited preprocessing instead of using atomic

operations. This strategy leads to a novel computation decomposition scheme that we call *hybrid gathering*. Actually, this scheme can be generalized to a broader class of problems on unstructured meshes or graphs that have the following properties:

1. The need to compute the values for a set of degrees of freedom whose relationships and data dependencies can be expressed by a graph (typically, specifically, an unstructured mesh);

2. The possibility to decompose a computation with different schemes according to different components of the graph and different associated operators;

3. A suitable decomposition scheme (for SIMD-type architectures), which introduces contention; and

4. Alternative decomposition schemes according to different graph components that are related by data dependency typically dictated by the topology of the graph.

For instance, in a finite element computation, the solution and the associated operators are approximated on elements or patches (which we collectively call "cells", *i.e.*, volume elements). This is a cell-wise computation where each cell has virtually the same amount of computation. Solutions in cells are often controlled by degrees of freedom at vertices, and thus solutions at vertices must accumulate the effects of adjacent cells. Valences may vary widely, and loads are not naturally balanced for this lighter-weight computation. The context switch between these two types of computation and the careful and efficient transfer of data are critical to efficient solvers.

The preference of the decomposition scheme depends on how the degrees of freedom are associated with the components of the unstructured mesh; here we consider *vertices*, *edges*, *faces*, and *cells*. If the degrees of freedom live on one component, the typical decomposition is to separately perform the local operators corresponding to this particular component in parallel. We call this decomposition scheme a *natural decomposition* where the decomposition is consistent with the degrees of freedom. This decomposition scheme can be represented as a sparse matrix-vector operation, as shown in Figure 2.1. The operator "$\bigotimes$" denotes a generic operation defined on the degrees of freedom that are given by the locations of "$\star$"s. The advantage of this scheme is that it naturally avoids contention

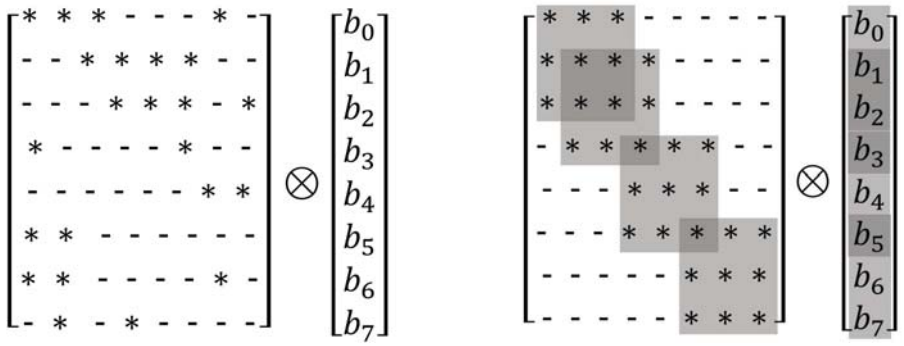


Figure 2.1. Matrix representations of the natural and regular decomposition schemes.

because each degree of freedom has an associated thread. However, it can introduce unbalanced load when the graph component associated with the degrees of freedom (usually vertices or edges) has widely varying valence. These irregular computations and data structures are not efficient on GPUs because of the logical branching structures they necessitate.

An alternative decomposition scheme is to decompose the computations according to another graph component, which is not directly tied to the degrees of freedom of the solution. We call such a decomposition scheme a *regular decomposition*; it is often more suitable for GPUs as it tends towards regular local operators and corresponding data structures. Such is the case with the cell-wise decomposition in FEM. Figure 2.1 depicts the matrix representation of this approach. In this decomposition, the matrix is grouped in terms of local operators according to the graph components. The groups can overlap each other, and the vector of degrees of freedom is segmented but has overlaps. Each group of matrix-vector operations represents a set of local computations that are performed by a thread. This decomposition scheme may result in contention as multiple threads may be updating the same degree of freedom due to the overlapping. The conventional solution to this problems is to use atomic operations. However, this is not suitable for GPUs as the atomic operations on GPUs are quite expensive, especially for double precision floating point.

We have developed the hybrid gathering scheme to combine the advantages of both the natural and regular decomposition schemes. In the hybrid gath-

ering decomposition scheme, the computation is decomposed into two stages (two matrix-vector operations): (1) performing local operations on the associated component (group matrix-vector operations) and stores intermediate result and (2) fetching data from the intermediate result according to the gathering matrix. The symbols $\bigotimes$ and $\bigodot$ in Figure 2.2 represent the operations in these two stages, respectively. In the first stage, the matrix groups and vector segments are not overlapping, and the group matrix-vector operations can be assigned to different threads and performed without contention. This stage decomposes the computations according to the graph component with regular local operators, and after this stage, each thread fetches data from the intermediate result according to the gathering matrix to assemble the value for the degrees of freedom. In practice, the two stages are implemented in one single kernel function, and fast GPU cache (shared memory or registers) is used to store the intermediate data. In this way, the gathering stage is very efficient.

The generation of the gathering matrix is a key part of the hybrid gathering decomposition scheme. The degrees of freedom live on one component of the graph denoted C_1, and the computations are performed in another component denoted C_2. Therefore, the gathering matrix represents a topological mapping from C_2 to C_1, and this mapping describes the data dependencies for each degree of freedom. In practice, this mapping from C_2 to C_1 is typically given as a C_2 list, denoted $\mathbb{E}$, which consists of C_1 indices. For instance, if the graph is a triangular

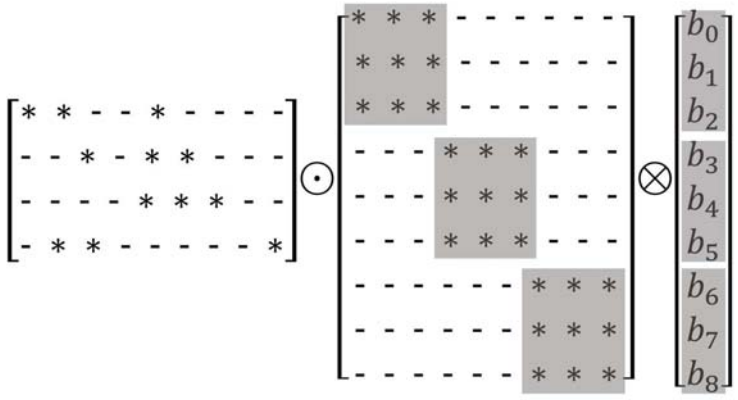


Figure 2.2. Matrix representations of the Hybrid Gathering scheme.

mesh, the topology information can be given as a list of triangles (C_2) consisting of indices of vertices (C_1). The location indices of this list usually correspond to memory location of the data that is needed by the threads. We create a sequence list $\mathbb{S}$ that records the memory locations of $\mathbb{E}$. Then, we sort $\mathbb{E}$ and permute $\mathbb{S}$ according to the sorting. Now, in the sorted list $\mathbb{E}'$, the C_1 indices are grouped, and the permuted sequence list, denoted $\mathbb{S}'$, stores the data memory location in original element list $\mathbb{E}$. The $\mathbb{E}'$ and $\mathbb{S}'$ together indicate the locations of the "$\star$"s in the gathering matrix and form the coordinate list (COO) sparse matrix representation of the gathering matrix. In this way, we virtually convert the contention problem to sorting problem. Here, the list to be sorted, $\mathbb{E}$, has fixed length keys that can be sorted very efficiently on GPUs with radix sorting. This contention to sorting transition is trying to avoid the weakness of the architecture of GPUs and take full advantage of their computing power.

CHAPTER 3

A FAST ITERATIVE METHOD FOR SOLVING THE EIKONAL EQUATION ON TRIANGULATED SURFACES

3.1 Introduction

The eikonal equation has a wide range of applications. In image analysis, for example, shortest paths defined by image-driven metrics have been proposed for segmentation [76] and tracking of white-matter pathways in the diffusion weighted images of the brain [51]. In seismology, the eikonal equation is used to calculate the travel time of the optimal trajectories of seismic waves [91]. The eikonal equation models the limiting behavior of Maxwell's equations [43] and is therefore useful in geometric optics. In computer graphics, geodesic distance on surfaces has been proposed for surface remeshing and mesh segmentation [92, 95]. The eikonal equation also has applications in medicine and biology. For instance, cardiac action potentials can be represented as moving interfaces and eikonal-curvature descriptions of wavefront propagation [56, 26]. For many of these applications described above, unstructured simplicial meshes, such as tetrahedra and triangles, are important for accurately modeling material interfaces and curved domains. This chapter addresses the problem of solving the eikonal equation on triangulated domains, which are approximations to either flat regions (subsets of $\mathfrak{R}^2$) or curved surfaces in $\mathfrak{R}^3$.

For many of these applications, there is a need for fast solutions to the eikonal equation—*e.g.*, run times of fractions of a second on large domains. For instance, solvers that can run interactively will allow scientists and mathematicians to explore parameter spaces of complex models and to reconfigure geometries and visualize their relationships to the solutions. In other cases, such as inverse

problems and remeshing, the algorithms require multiple solutions of the eikonal equation as part of the inner loop of an iterative process. Thus, there is a need for fast, efficient eikonal solvers.

Efficient solutions on state-of-the-art computer architectures place particular constraints on the data dependencies, memory access, and scale of logical operations for such algorithms. The trend in computer architecture is toward multicore CPUs (conventional processors) and massively parallel streaming architectures, such as graphics processing units (GPUs). Thus, parallel algorithms that run efficiently on such architectures will become progressively more important for many of these applications. Of particular interest are the massively parallel streaming architectures that are available as commodities on consumer-level desktop computers. With appropriate numerical algorithms, these machines provide computational performance that is comparable to the supercomputers of just a few years ago. For example, the most recent graphics processing units (GPUs), which cost only several hundred US dollars, can reach a peak performance of nearly 10^{12} floating point operations per second (TeraFLOPS); a performance equivalent to a top supercomputer a decade ago [107]. This computing power, however, is for a single-instruction multiple-datastream (SIMD) computational model, and most of the recent massively parallel architectures, such as GPUs [20], rely heavily on this paradigm. These modern SIMD architectures provide a large number of parallel computing units (up to several hundred cores) in a hierarchical data-sharing structure, rather simple branching circuits, and large memory bandwidth. As such, they place important restrictions on the algorithms that they can run efficiently. Addressing these constraints is an important aspect of this paper.

In the past several decades, many methods have been proposed to solve the eikonal equation on unstructured grids for both two-dimensional and three-dimensional domains. Iterative schemes, for example [84], rely on a fixed-point method that solves a quadratic equation at each grid point in a predefined update order and repeats this process until the solution on the entire grid converges. Some adaptive, iterative methods based on a label-correcting algorithm (from a similar shortest-path problem on graphs [13]) have been proposed [77, 16, 34, 35].

The fast marching method (FMM) by Sethian [88], a form of the algorithm first proposed in [80], is used widely and is the de facto state-of-the-art for solving the eikonal equation. FMM has an asymptotic worst case complexity of $O(N \log N)$, which is optimal. However, it uses a strict updating order and the min-heap data structure to manage the narrow band which represents a bottleneck that thwarts parallelization. Although the FMM has some parallel variants [46, 101] that use domain decompositions, they rely on a serial FMM within each subdomain, which is not efficient for massively parallel, SIMD architectures. Furthermore, these parallel variants are only for regular grids, and the extension to unstructured, triangular meshes, the topic of this chapter, is not straightforward.

For homogeneous speed functions on flat domains, the characteristics of the eikonal equation are straight lines. In such cases, one can solve the eikonal equation by updating solutions along specific directions without explicit checks for causality. Based on this observation, Zhao [110] and Tsai *et al.* [100] proposed the fast sweep method (FSM), which uses a Gauss-Seidel update scheme for the straight (grid-aligned) wavefront and proceeds across the domain in an incremental *sweep*. This method may converge faster than the Jacobi update methods, which update all grid points at once. However, the update scheme, which proceeds simultaneously for all nodes on the wavefront, still presents a bottleneck because it limits updates to a specific set of points in a predefined order. More importantly, previous work [52] has shown that the number of iterations or sweeps grows with the complexity of the *speed function*, and thus the method is only efficient for relatively simple (nearly homogeneous) inputs, where the characteristics are approximately straight. FSM has extensions to 2D and 3D unstructured meshes [79] whose update ordering is based on distances of grid nodes to some selective reference points. However, this extension cannot be easily used for surface meshes (*e.g.*, in $\mathfrak{R}^3$) because Euclidean distances between nodes are not consistent with geodesic distances on the mesh.

Jeong and Whitaker propose the fast iterative method [52, 51] (FIM) to solve the heterogeneous eikonal equation and anisotropic Hamilton-Jacobi equations efficiently on parallel architectures. The FIM manages the list of active nodes and iteratively updates the solutions on those vertices until they are consistent with

their neighboring vertices. Vertices are added to or removed from the list based on a convergence criterion, but the management of this list does not entail an extra burden of expensive ordered data structures or special updating sequences. Proper management of the list ensures consistency of the entire solution. This chapter builds on the FIM algorithm, and describes the application to unstructured meshes and an implementation on a streaming, SIMD parallel architecture.

In this chapter, we propose a new computational technique to solve the eikonal equation on triangulated surface meshes efficiently on parallel architectures; we call it the *mesh fast iterative method* (meshFIM), because it is an extension of the FIM method proposed in [52]. We describe a parallel implementation of meshFIM on shared memory parallel systems and propose a new data structure for the efficient mapping of unstructured meshes for parallel SIMD processors with limited high-bandwidth memory. The contributions of this chapter are twofold. First, we introduce the meshFIM algorithms for both single processor and shared memory parallel processors and perform a careful empirical analysis by comparing them to the state-of-the-art CPU-based method, the fast marching method (FMM), in order to understand the benefits and limitations of each method. Second, we propose a patch-based meshFIM solver, specifically for more efficient implementation of the proposed method on massively parallel SIMD architectures. We describe the detailed data structure and algorithm, present the experimental results of the patch-based meshFIM, and compare them to the results of the CPU-based methods to illustrate how the proposed method scales well on state-of-the-art SIMD architectures.

The chapter proceeds as follows. In the next section, we describe relevant work from the literature. In Section 3.2, we introduce the proposed method and its hierarchical implementation for SIMD parallel architectures. In Section 3.3, we show numerical results, including consistency and convergence, on several different examples with different domains and speed functions, and we compare the performance against the fast marching method. In Section 3.4, we summarize the chapter and discuss future research directions related to this work.

3.2 Fast Iterative Method (FIM) on Unstructured Meshes

3.2.1 Notation and Definitions

In this chapter, we consider the numerical solution of the eikonal equation 3.1, a special case of nonlinear Hamilton-Jacobi partial differential equations (PDEs), defined on a two-dimensional manifold with a scalar speed function

$$\begin{cases} H(\mathbf{x}, \nabla\phi) = \left|\nabla_{\S}\phi(\mathbf{x})\right|^2 - \dfrac{1}{f^2(\mathbf{x})} = 0 \quad \forall \mathbf{x} \in \S \subset \Re^3 \\ \phi(\mathbf{x}) = B(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{B} \subset \S \end{cases} \tag{3.1}$$

where $\S$ is a smooth two-dimensional manifold in $\Re^3$, $\nabla_{\S}$ is the gradient operator in the tangent plane to the manifold, $\phi(\mathbf{x})$ is the travel time or distance from the source, $f(\mathbf{x})$ is a positive speed function defined on $\S$, and $\mathcal{B}$ is a set of smooth boundary conditions, which adhere to the consistency requirements of the original equation. Of course, a two-dimensional, flat domain is a special case of this specification, and the proposed methods are appropriate for that scenario as well. The solution of the eikonal equation with an arbitrary speed function is sometimes referred to as a weighted distance [94] as opposed to a Euclidean distance for a constant speed function on flat domains. We approximate the solution on a triangulation of $\S$, denoted $\S_T$. The solution is represented point-wise on the set of vertices V in S_T, and interpolated across the triangles with linear basis elements. The ith vertex in V is denoted v_i and its position is a 3-tuple and denoted $\mathbf{x}_i = (x, y, z)$ where $x, y, z \in \Re$. An *edge* is a line segment connecting two vertices (v_i, v_j) in $\Re^3$ and is denoted $e_{i,j}$ while the vector from vertex v_i to vertex v_j is denoted $\mathbf{e}_{i,j}$ which equals to $\mathbf{x}_j - \mathbf{x}_i$. The angle between $e_{i,j}$ and $e_{i,k}$ is denoted $\angle_i$ or $\angle_{j,i,k}$.

The neighbors of a vertex are the set of vertices connected to it by edges. A triangle, denoted $T_{i,j,k}$, is a set of three vertices v_i, v_j, v_k that are each connected to the others by an edge. We assume the triangulation adheres to a typical criteria for consistency for 2D manifolds, *e.g.*, edges not on the boundary of the domain belong to two triangles, *etc.* We call the vertices connected to v_i by an edge the *one-ring neighbors* of v_i and the triangles sharing vertex v_i are the *one-ring triangles* of v_i. For example, in Figure 3.1-left, the vertex v_1 is the neighbor of vertex v_2 and vice-versa. Vertices v_2, v_3, v_4, v_5, v_6, v_7 constitute the one-ring of v_1, and triangles

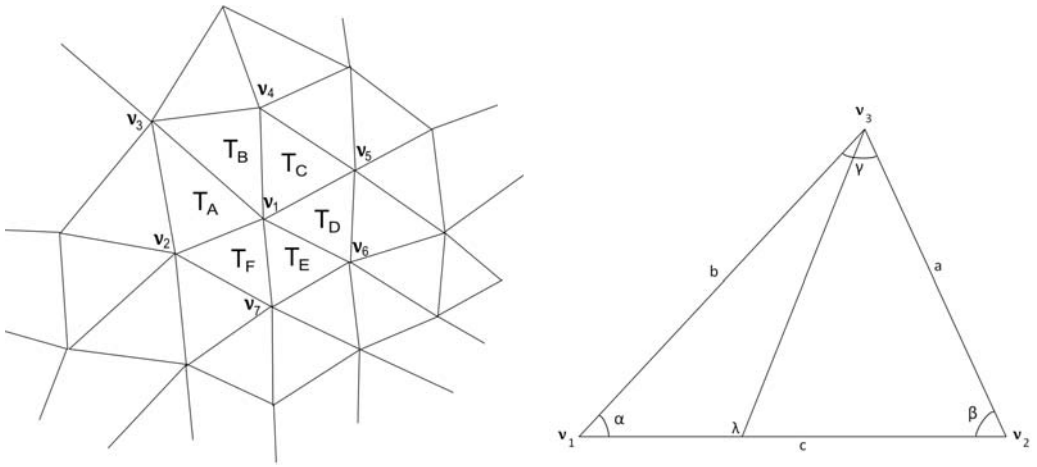


Figure 3.1. A triangulation $\S_T$ of surface § (left) and the local solver: update the value at vertex v_3 in a triangle (right)

$T_{1,2,3}, T_{1,3,4}, \ldots, T_{1,2,7}$ (which we will denote with capital letters for multi-indices T_A, …, T_F as in the figure) form the one-ring triangles of v_1. We define the discrete approximation to ϕ at vertex v_i to be Φ_i.

3.2.2 Local Solver

In Equation 3.1, domain § is a manifold for which we have a tessellation $\S_T$ and the numerical solution of the equation $\Phi(\mathbf{x})$ is defined on the vertices of the triangles of the tessellation. The solution at each vertex, sometimes referred to as the *travel time*, is computed from its current value and its one-ring neighbors (see Figure 3.1-left), using a linear approximation of the solution on each triangular facet. The formulation presented here is a constructive form of derivation in [79], which describes a Godunov approximation that picks an upwind direction of travel for the characteristics based on consistency of the resulting solution. For a single update of a single vertex v_i, a set of n potential solutions ($n = 6$ for v_1 in Figure 3.1-left) are calculated for the n one-ring triangles. Each of these triangle solutions represents the shortest path across that triangle from the boundary conditions, as described in the following paragraphs. The approximated solution at vertex v_i, $\Phi_i \approx \phi(\mathbf{x}_i)$, is set to be the minimum among the n values associated with each triangle in the one-ring. From a computational point-of-view, the bulk of the work is in the calculation of the n temporary or potential solutions from the adjacent triangles of

each vertex.

The specific calculation on each triangle is as follows. Considering a triangle $T_{1,2,3}$ in Figure 3.1-right. We use an upwind scheme to compute the solution Φ_3, from values Φ_1 and Φ_2 to comply to the causality property of the eikonal solution [79]. We consider a local scheme based on piecewise linear reconstructions within the triangle. The characteristics are perpendicular to the gradient of Φ, which is linear, and thus the travel time to v_1 must be determined by time associated with a line segment lying in the triangle $T_{1,2,3}$.

Because acute triangles are essential for proper numerical consistency [57], we consider only the case of acute triangles here and discuss obtuse triangles subsequently. For a triangle $T_{1,2,3}$ in Figure 3.1-right, we denote the angles formed by the triangular edges as $\angle_1 = \alpha$, $\angle_2 = \beta$, and $\angle_3 = \gamma$, and denote the edge lengths as $\|\mathbf{e}_{1,2}\| = c$, $\|\mathbf{e}_{1,3}\| = b$, and $\|\mathbf{e}_{2,3}\| = a$. We assign a constant speed f to each triangle, $T_{1,2,3}$, which is consistent with a symmetric (isotropic) speed and a linear solution on each element. We denote the difference in travel time between v_1 to v_2 as $\Phi_{1,2} = \Phi_1 - \Phi_2$.

If the vertices v_1 and v_2 are upwind of v_3, then there is a characteristic passing through v_3 that intersects edge $\mathbf{e}_{1,2}$ at position $\mathbf{x}_\lambda = \mathbf{x}_1 + \lambda\mathbf{e}_{1,2}$, where λ is unknown and $\lambda \in [0,1]$ in order for the characteristic to intersect the edge. The line segment that describes the characteristic across $T_{1,2,3}$ is $\mathbf{e}_{\lambda,3} = \mathbf{e}_{1,3} - \mathbf{e}_{1,\lambda} = \mathbf{e}_{1,3} - \lambda\mathbf{e}_{1,2}$. Thus the travel time from $\mathbf{x}_\lambda$ to $\mathbf{x}_3$ is $\Phi_{\lambda,3} = f\|\mathbf{e}_{\lambda,3}\| = f\|\mathbf{e}_{1,3} - \lambda\mathbf{e}_{1,2}\|$.

Because the approximation of the solution on the triangle $T_{1,2,3}$ is linear, we have $\Phi_\lambda = \Phi(\mathbf{x}_\lambda) = \Phi_1 + \lambda\Phi_{1,2}$. The solution at v_3 is the solution at $\mathbf{x}_\lambda$ plus the travel time from $\mathbf{x}_\lambda$ to the vertex v_3, and therefore

$$\Phi_3 = \Phi_\lambda + \Phi_{\lambda,3} = \lambda\Phi_{1,2} + \Phi_1 + f\|\mathbf{e}_{1,3} - \lambda\mathbf{e}_{1,2}\|. \tag{3.2}$$

All that remains is to find λ, and for this we observe that λ should minimize Φ_3 because the characteristic direction is the same as the gradient of the solution. Assigning zero to the derivative (with respect to λ) of Equation 3.2 gives a quadratic equation from which we solve for λ. To satisfy the causality condition, λ must be in the range of $[0,1]$. If the solved λ is in $[0,1]$, we compute Φ_3 from Equation 3.2,

else we compute two Φ_3's from Equation 3.2 assuming λ as 0 and 1, and take the smaller one.

Because the computation of the solution for linear, triangular elements have poor approximation properties when applied to obtuse triangles [82], we have to treat obtuse triangles as a special case. For this, we adopt the method used in [57]. As illustrated in Figure 3.2, if $\angle_3$ is obtuse, we connect v_3 to the vertex v_4 of a neighboring triangle and thereby cut the obtuse angle into two smaller angles. If these two angles are both acute, then we are done, as shown in the left picture of Figure 3.2; otherwise if one of the smaller angles is still obtuse, then we connect v_3 to the vertex v_5 of another neighboring triangle. This process is performed recursively, until all new angles at v_3 are acute, as shown in the right image of Figure 3.2. Note that algorithmically, these added edges and triangles are not considered part of the mesh; they are used only in the solver for updating the solution at v_3.

3.2.3 MeshFIM Updating Scheme

The original *fast iterative method* [52] for solving the eikonal equation was proposed for rectilinear grids. In this section, we extend the method to unstructured triangular meshes, called *meshFIM*, in a way that is appropriate for more general simplicial meshes. We begin with a serial (single-threaded) version of the algorithm, and then describe a parallel (multithreaded) version of meshFIM for

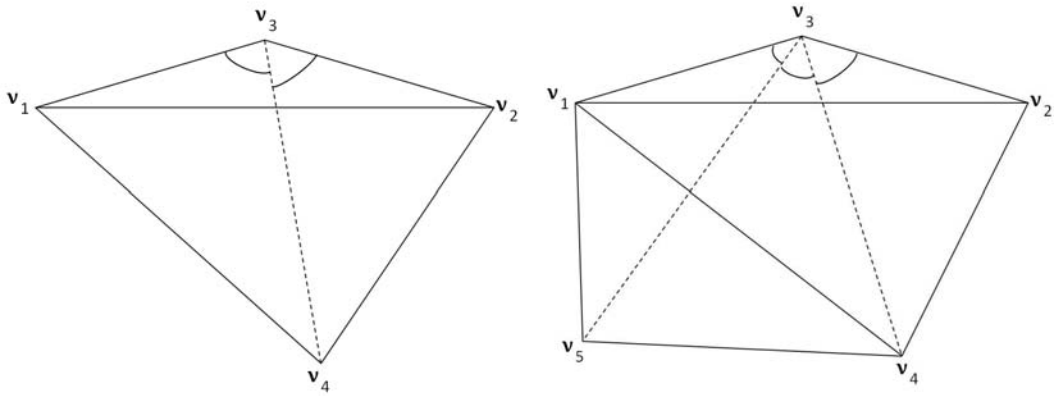


Figure 3.2. Strategy to deal with obtuse triangles

shared memory system. Finally, we describe the algorithm for SIMD, streaming architectures with limited (hierarchical) shared memory capabilities in detail. Here we mention the properties that make FIM suitable for parallel solutions of the eikonal equation, because they govern some of the subsequent design choices:

1. The algorithm does not impose a particular update sequence.

2. The algorithm does not use a separate, heterogeneous data structure for sorting.

3. The algorithm is able to simultaneously update multiple points.

The strategy of meshFIM is to solve the eikonal equation on triangular mesh vertices with lightweight data structures for easy mapping to SIMD architectures with fast access to limited amounts of high-speed memory. This is the basic model of state-of-the-art streaming architectures [20]. As in FIM [52], meshFIM maintains a data structure that represents a narrow computational band, a subset of the mesh, called the *active list*, for storing the vertices that are being updated. During each iteration, the list of active vertices/triangles is modified to remove vertices whose solutions are consistent with their neighbors and to include vertices that could be affected by the last set of updates. Thus, a vertex is removed from the active list when its solution is up-to-date with respect to its neighbors, and a vertex is appended to the list when the value of any potentially upwind neighbor has changed.

Convergence of the algorithm to a valid approximation of the eikonal equation is provable [52] if three conditions are met:

1. Any vertex whose value may be inconsistent with its neighbors (according to the local solver) must be appended to the active list.

2. A vertex is removed from the active list only when its value is consistent with its neighbors.

3. The algorithm terminates only when the active list is empty.

There are a variety of algorithms that meet these criteria. Indeed, FMM is a special case of this philosophy, which adopts a particular update order that guarantees that once a point is removed from the active list, it will never again need to be added (it is upwind of every subsequent update of vertex/grid values). In the remainder

of this section, we will discuss rules for updating vertex values and managing the active list that are efficient for arbitrary ordering of vertex-value updates, including update schemes that include both synchronous and asynchronous update of the active list.

Before the computation of the solution, any algorithm must compute certain static information about the mesh, including the speed for each triangle and values of the boundary conditions, and initialize the appropriate data structures, in this case the active list L, which is set to be all of the vertices adjacent to the boundary conditions. The computation of the speed function depends on the application, and the initialization of the active list is not a computationally important step; thus, we do not treat the initialization as an important aspect of the parallel algorithms presented in this chapter.

We begin with the basic algorithm, which assumes synchronous updates of the entire active list, and then introduce alternatives that take better advantage of asynchronous updates. In this context, an *iteration* is one loop through the entire active list. In the basic algorithm, for every vertex $v_j \in L$, we compute the new Φ_j from solutions on the one-ring. This solution puts each vertex into a consistent solution with the values of its neighbors from the previous iteration, and thus all vertices, nominally, are removed from the active list. Each updated vertex, however, triggers the activation of neighbors of greater value, which are potentially downwind. The algorithm would continue to update each subsequent active list until the active list is empty.

If we consider asynchronous updates, values that are potentially downwind of others in the active list may take advantage of updated values from the current iteration. Indeed, taken to the limit, the updates are done on individual nodes, one at a time, proceeding from the node of lowest value—which is the FMM algorithm. For parallel algorithms, the approach will be a mixture of synchronous updates among processors and asynchronous updates as each processor proceeds with a particular subset of the active list. The situation becomes more complicated when we consider the limited amount of communication that is available between processors or blocks of processors, which motivates processing multiple iterations on

subsets of the domain without exchanging data or updating boundary conditions. In such cases, it is sometimes a more effective use of computational resources to run multiple iterations on the same set of active nodes, not removing each one from the list after updating, so that they can take advantage of updates of neighbors. The particular choice of updating strategy depends on the architecture, and in the sections that follow these choices are described for three different computational scenarios.

3.2.4 Algorithms for CPU

The criteria for a correct algorithm would suggest that a vertex could be removed from the list and its neighbors activated after a single update—knowing that it will be reactivated as needed. However, in the absence of a strict or approximate sorting of values in the active list, it is efficient to reconcile the values of vertices on the current wavefront (active list), before retiring updated vertices and including new ones. From this insight, we derive the proposed algorithm, which is as follows. Nodes on the active list are updated one at a time. After each node is updated, its value is consistent of its upwind neighbors, and each update is immediately transferred to the solution to be used by subsequent updates. The algorithm loops through the active list, continuously updating values, and when it reaches the last element of the list simply starts again at the beginning—thus, there is effectively no beginning or end to the list. A vertex remains on the active list until the difference between its old value and new value is below some error tolerance—effectively, it does not change from the last update. We refer to a vertex that does not change value (to within tolerance ϵ) as *ϵ-converged*. Each ϵ-converged vertex is removed from the active list. As the converged vertex is removed from the active list, all of its potentially downwind neighbors (neighbors of greater value) undergo one update step. If their values are not ϵ-converged (*i.e.*, they change significantly), they are appended to the active list. The algorithm continues looping through the active list until the list is empty.

Table 3.1 compares the number of solution updates between FMM, strict synchronous and asynchronous relabeling schemes, and the proposed mesh fast iter-

Table 3.1. Average number of local solver calls per vertex with the FMM, synchronous relabeling scheme, asynchronous relabeling scheme, and meshFIM for two different meshes—one simple and one complex (sphere and dragon described below).

	FMM	Synchronous	Asynchronous	meshFIM
Simple mesh	18.1	737.8	177.0	19.6
Complex mesh	18.3	671.7	175.2	59.2

ative method (meshFIM). The FMM is optimal (although run times will be slightly offset by the time involved in managing the heap), and the synchronous and asynchronous schemes perform very poorly. The asynchronous scheme depends, in principle, on update order, but these results are consistent across a set of experiments with random permutations of the active list. This table also shows that the update strategy of the FIM, while not optimal, provides numbers of updates that are much closer to FMM, and showed a robustness to the ordering of the active list.

Because the serial algorithm does not depend significantly on the ordering of updates, the extension to multiple processors is immediate. We simply divide the active list arbitrarily into N sublists, assign the sublists to the N threads, and let each thread use an asynchronous update for the vertices within the sublist. These updates are done by applying the updating step in Algorithm 3.1 to each subactive list.

3.2.5 Algorithm for GPU with SIMD Parallel Architecture

In this section, we describe the implementation of meshFIM for SIMD parallel architecture that we call patchFIM.

To make good use of the GPU performance advantage, we propose a variant of meshFIM, called patchFIM, that scales well on SIMD architectures, using a patch-based update scheme. The main idea is splitting the computational domain (mesh) into multiple nonoverlapping patches (sharing only boundary vertices), and treating each patch, which will be processed in a SIMD fashion in a single block, as a computing primitive, corresponding logically to a vertex in the original meshFIM algorithm.

Algorithm 3.1 meshFIM(V,B,L)

```
1: Comment 1. Initialization (V : all vertices, L : active list, B: seed vertices)
2: for all v ∈ V do
3:    if v ∈ B then
4:       Φ_v ← 0
5:    else
6:       Φ_v ← ∞
7:    end if
8: end for
9: for all v ∈ V do
10:   if any 1-ring vertex of v ∈ B then
11:      add v to L
12:   end if
13: end for
14: Comment 2. Update vertices in L
15: while L is not empty do
16:   for all v ∈ L do
17:      p ← Φ_v
18:      q ← Update(v)
19:      if |p − q| < ε then
20:         for all adjacent neighbor v_nb of v do
21:            if v_nb is not in L then
22:               p ← Φ_{v_nb}
23:               q ← Update(v_nb)
24:               if p > q then
25:                  Φ_{v_nb} ← q
26:                  add v_nb to L
27:               end if
28:            end if
29:         end for
30:         remove v from L
31:      end if
32:   end for
33: end while
```

The active list maintains a set of *active patches* instead of active vertices, and a whole active patch is moved from global memory to a block and updated for several SIMD iterations, which we call *internal iterations*. A set of internal iterations comprises a single *iteration* for that patch. Thus, for each patch iteration, the data for that patch are copied to the shared memory space, and internal iterations are executed to update the solution on that patch. Of course, multiple computing

blocks can process multiple patches simultaneously, while other patches wait in global memory to be swapped out to blocks.

This patch strategy is meant to take advantage of the SIMD parallelism, but it introduces some inefficiencies. For instance, an entire patch must be activated any time a vertex in an adjacent patch gets updated. A patch must remain active as long as any of vertices are still active. The number of internal iterations is required to offset of the cost of transferring data between memory caches; however, vertices within a patch are updated without communication with adjacent patches, and thus boundary conditions lag and may be out of date as the internal iterations proceed.

These inefficiencies must be justified by an effective SIMD algorithm for the patches. There are two challenges. First is to provide SIMD processing on the unstructured mesh, and second is to keep the computational density sufficiently high. The parallelism is obtained by introducing a data structure for SIMD computing on unstructured meshes, which we call the *cell-assembly* data structure (terminology adapted from the finite element method (FEM) literature). Specifically, the *cell-assembly* data structure includes three arrays, labeled mnemonically **GEO**, **VAL**, and **NBH**. **GEO** is the array storing per-triangle geometry and speed information required to solve the eikonal equation. It is divided into subsegments with a predefined size that is determined by the largest patch among all. Each subsegment stores a set of four floats for each triangle, *i.e.*, three floats for triangle edge lengths and one float for the speed value. **VAL** is the array storing per-triangle values of solution of the eikonal equation. It is divided into subsegments, similar to **GEO**, but instead of geometric information, solutions on three vertices are stored. We use two **VAL** arrays, one is for input and the other is for output, to avoid memory conflicts. To deal with boundaries across patches, we simply duplicate and store the exterior boundary vertices for each patch and treat the data on those vertices as fixed boundary conditions for each patch iteration. The **NBH** array stores indices to **VAL** for the per-vertex solution. Figure 3.3 depicts the data structure introduced above.

A single inner iteration on a patch proceeds in two steps. In the first step, all of

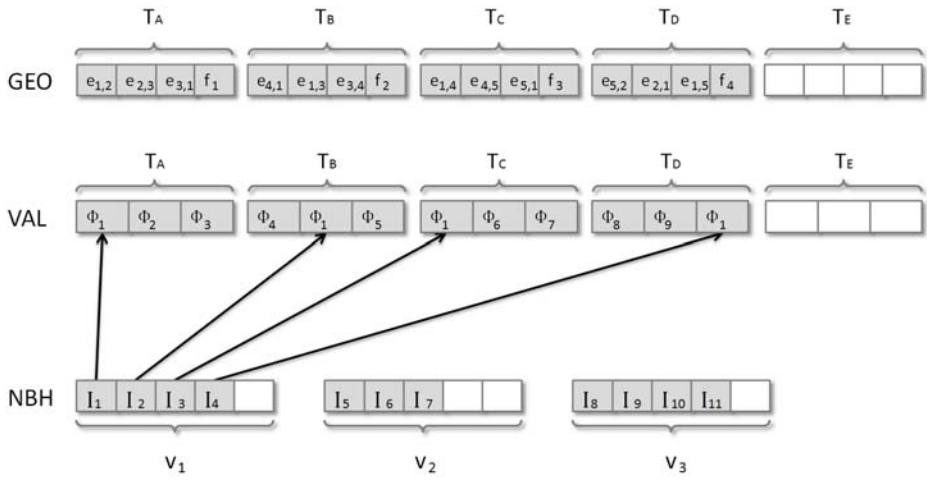


Figure 3.3. Data structure: in this figure, T_i is a triangle, $e_{i,j}$ represents the edge length, f_i is the inverse of speed in a triangle. Φ_i means the value of the i_{th} vertex. I_i in NBH represents the data structure for the i_{th} vertex, each of which has q indices pointing (shown as arrows) to the value array.

the triangles produce the arrival time for the solution for each vertex of the triangle from the opposite edge, with special values for invalid results, as above. The triangle solutions all undergo the same computation, with some minor branching in the determination of valid solutions. In the second step, all vertices are updated by referring back to the appropriate data in triangle solutions and performing a min operation on the valid solutions (assembly). The vertex computation must loop through all of the triangles in the one-ring, and thus the run-time of this step is determined by the vertex with highest valence in the patch. Thus, SIMD efficiency favors meshes with relatively consistent valences.

3.2.5.1 Preprocessing

The patchFIM algorithm requires some preprocessing before the iterations begin. First, we must partition the mesh into patches. We use the multilevel partitioning scheme described in [55]. It partitions the vertices of a mesh into roughly equal patches, such that the number of edges connecting vertices in different parts is minimized. The particular algorithm for mesh partitioning is not important to the proposed algorithm, except that efficiency is obtained for patches with similar numbers of vertices/triangles and relatively few vertices on

the boundaries.

In this step, we also calculate the static mesh information, including dealing with the obtuse triangles. We use the same idea as in meshFIM to treat obtuse triangles. However, instead of adding *virtual edge*, we also add *virtual triangles* generated by splitting the obtuse triangle to the corresponding cell-assembly data structures. Figure 3.4 demonstrates this, where $\angle_{1,3,2}$ is obtuse, and adding a *virtual edge* $e_{3,4}$ will generate two "virtual triangles" $T_{1,3,4}$ and $T_{2,3,4}$. If one of $\angle_{1,3,4}$ and $\angle_{2,3,4}$ is still obtuse, the algorithm would split again. The last thing in this step is to initialize values of each vertices and the active list. Instead of keeping a narrow band active vertices, we maintain list of active patches. If any of the vertices in a patch is adjacent to a seed point, this patch is added to the initial active list.

3.2.5.2 Iteration step

In this step, each patch is treated just like a vertex in meshFIM. The main iteration continues until the active list becomes empty. Each patch in the active list is assigned to a SIMD computing unit where all vertices value in this patch are updated several times. After every update, the assembly stage reconciles the different solutions for a vertex. This is done with a loop over the **NBH** to find the minimum value. If a patch is convergent, meaning all vertices in this patch

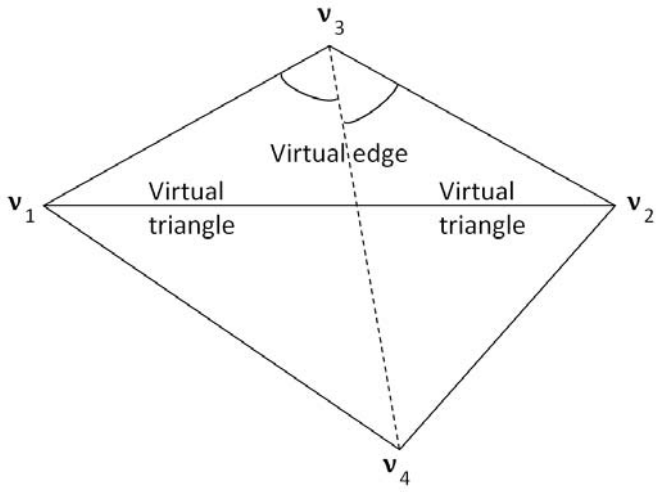


Figure 3.4. Virtual edge and virtual triangles

are convergent, it is removed from the active list and its nonconvergent neighbor patches are added to the active list.

Checking the patch convergence can be simply updating the entire patch once and checking if there exists a vertex whose solution has changed by the update. To do this, we use a reduction operator, which is commonly used in the streaming programming model to reduce a larger input stream to a smaller output stream. For SIMD architectures, parallel reduction can be implemented using an iterative method. In each iteration, we adopt a tree-based method in which every thread reads two Boolean values from the convergence array of current patch and writes back the result of the AND operation of two values. The number of the threads to participate in this reduction is halved in the successive iteration, and this is repeated until only one thread is left. In this way, for a block of size n, only $O(\log_2 n)$ computations are required to reduce a block. In the pseudo-code to follow, $C(p)$ is a Boolean value representing the convergence status of a patch p (per-patch convergence), and $C_v(p)$ is a set of Boolean values where each value represents the convergence status of the vertices in the patch p (per-vertex convergence). The pseudo-code for patchFIM is given in Algorithm 3.2, where the pseudo-code for each subroutine in the patchFIM is given in Algorithm 3.3, 3.4, and 3.5, respectively.

Algorithm 3.2 patchFIM($\mathbf{VAL}_{\text{in}}$, $\mathbf{VAL}_{\text{out}}$, L, P)

```
1: Comment: L: active list of patches, P: set of all patches
2: while L is not empty do
3:    MainUpdate(L, C_v, VAL_in, VAL_out)
4:    CheckNeighbor(L, C_v, C, VAL_in, VAL_out)
5:    UpdateActiveList(L, P, C)
6: end while
```

3.3 Results and Discussion

In this section, we discuss the performance of the proposed algorithms in realistic settings compared to the most popular FMM-based algorithm. We have conducted systematic empirical tests with a set of different meshes with various speed functions. First, we show the result of the single-threaded (serial) CPU

Algorithm 3.3 MainUpdate(L, C_v, $\mathbf{VAL}_{\text{in}}$, $\mathbf{VAL}_{\text{out}}$)

```
1: 1. Main iteration
2: for all p ∈ L in parallel do
3:   for i = 1 to n do
4:     for all t ∈ p in parallel do
5:       VAL_out(t) ← LocalSolver(VAL_in(t))
6:       reconcile solutions in t
7:     end for
8:     update C_v(p)
9:     swap VAL_in(t) and VAL_out(t)
10:    reconcile solutions in
11:  end for
12: end for
```

Algorithm 3.4 CheckNeighbor(L, C_v, C, $\mathbf{VAL}$_in, $\mathbf{VAL}_{\text{out}}$)

```
1: 2. Check neighbors
2: for all p ∈ L in parallel do
3:   C(p) ← reduction(C_v(p))
4: end for
5: for all p ∈ L in parallel do
6:   if C(p) == TRUE then
7:     for all adjacent neighbor of p_nb of p do
8:       add p_nb to L
9:     end for
10:  end if
11: end for
12: for all p ∈ L in parallel do
13:   for all t ∈ p in parallel do
14:     VAL_out(t) ← LocalSolver(VAL_in(t))
15:     reconcile solutions in t
16:   end for
17:   update C_v(p)
18:   swap VAL_in(t) and VAL_out(t)
19:   reconcile solutions in p
20: end for
21: for all p ∈ L in parallel do
22:   C(p) ← reduction(C_v(p))
23: end for
```

implementation of meshFIM and FMM, and discuss the intrinsic characteristics relative to existing algorithms. Second, we provide the result of multithreaded CPU implementation to discuss scalability of the proposed algorithm on shared memory

Algorithm 3.5 UpdateActiveList(L, P, C)

```
3. Update active list
clear(L)
for all p ∈ P do
    if C(p) = FALSE then
        insert p to L
    end if
end for
```

multiprocessor computer systems. Last, we show the GPU implementation to demonstrate the performance of the proposed method on SIMD parallel architectures. Single precision is used in all experiments throughout the entire chapter. We have carefully chosen four triangular meshes with increasing complexity to compare the performance of each method. In addition, we used two different speed functions, a constant and correlated random speed, to elaborate how the heterogeneity of the speed function affects the performance of each method.

The meshes for the experiments in this section are:

Mesh 1: A regularly triangulated flat square mesh with 1,048,576 vertices (1024 by 1024 regular grid),

Mesh 2: An irregularly triangulated flat square mesh with 1,181,697 vertices and 2,359,296 triangles,

Mesh 3: A sphere with 1,023,260 vertices and 2,046,488 triangles (Figure 3.5-left), and

Mesh 4: Stanford dragon with 631,187 vertices and 1,262,374 triangles (Figure 3.5-right).

The speed functions $f(x)$ are: **Speed 1** — a constant speed of one, and **Speed 2** — correlated random noise.

3.3.1 Serial CPU Results

We have tested our CPU implementation on a Windows Vista PC equipped with an Intel i7 920 CPU running at 2.66 GHz. First, we focus only on the performance of FMM and the single-threaded implementation of our method (meshFIM-ST) on different meshes with a constant speed (Speed 1). Rows 1 and 2 of Table 3.2 show the experimental results for the serial implementations.

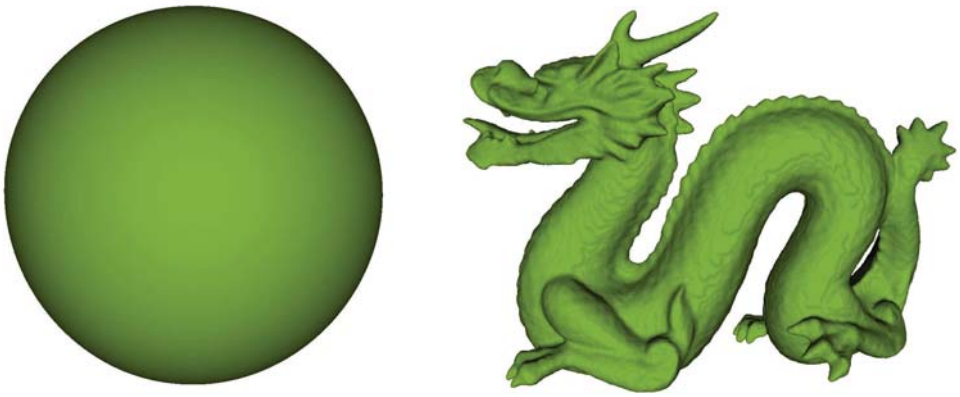


Figure 3.5. Sphere and Stanford dragon meshes

Table 3.2. Running time (millisecond) of FMM, single-threaded FIM (meshFIM-ST), and multithreaded FIM (meshFIM-MT) on Meshes 1, 2, 3, and 4 with a constant speed (Speed 1).

	Mesh 1	Mesh 2	Mesh 3	Mesh 4
FMM	5092	7063	6362	3612
meshFIM-ST	6562	9354	8591	4331
meshFIM-MT	2198	3151	2846	1487

The eikonal equation with the speed function of constant one ($f(\mathbf{x}) = 1$) is the simplest test, and the easiest to perform well. However, it is useful in real-world applications because the solution is the geodesic distance on a surface to the initial source boundary. In this experiment, we use a single point as the source for all four meshes so that the r-level set of the solution Φ is a curve that is a collection of all points on the surface whose distance to the source point is r. As shown in Table 3.2, FMM outperforms the single-threaded meshFIM slightly on all the test cases. Although FMM has the overhead of managing the heap data structure, the cost related to computing distance becomes the major bottleneck for the eikonal equation on the mesh. Because meshFIM usually requires more iterations per vertex than FMM (which is optimal in this respect), meshFIM runs slower than FMM for serial execution.

To further elaborate the difference of two methods, we conducted the experiment on Mesh 3 using both speed functions. As shown in Table 3.3, the

Table 3.3. Running time (millisecond) of FMM and meshFIM (single and multi-threaded) on Mesh 3 and both speed functions (Speed 1 and 2).

	Speed 1	Speed 2
FMM	6362	6435
meshFIM-ST	8591	11960
meshFIM-MT	2846	4362

performance of FMM is not affected by the choice of the speed functions, which clearly demonstrates the advantage of the worst-case-optimal algorithm. On the other hand, the running time for meshFIM increased significantly from Speed 1 to Speed 2 because the total number of iterations (vertex updates) is significantly increased for Speed 2 due to the huge variance of the speed.

The meshFIM algorithm is designed for parallelism, and the results on the multithreaded system bear this out. The third row in Table 3.3 shows the running time of multithreaded meshFIM using four CPU cores. Because FMM is a serial algorithm (a strict ordering of the updates on vertices requires this), there is no benefit of using multiple threads. In contrast, meshFIM scales well on multicore systems. On a quad-core processor, we observed a nearly three times speedup from meshFIM-ST to meshFIM-MT on all cases. This result suggests that meshFIM is a preferred choice for such shared memory systems.

3.3.2 GPU Implementation Result

To show the performance of meshFIM on SIMD parallel architectures, we have implemented and tested patchFIM (Algorithm 3.2) on an NVIDIA GT200 GPU using NVIDIA CUDA API [68]. The NVIDIA GeForce GTX 275 graphics card is equipped with 896 MBytes of memory and 30 microprocessors, where each microprocessor consists of eight SIMD computing cores that run at 1404 MHz. Each computing core has 16 KBytes of on-chip shared memory for fast access to local data. The 240 cores run in parallel, but the preferred number of threads running on a GPU is much larger because cores are time-shared by multiple threads to maximize the throughput and increase computational intensity. Computation on the GPU entails running a kernel with a batch process of a large group of fixed size

thread blocks, which maps well to the patchFIM algorithm that uses patch-based update methods. A single patch is assigned to a CUDA thread block, and each triangle in the patch is assigned to a single thread in the block. In order to balance the GPU resource usage, *e.g.*, registers and shared memory, and the number of threads running in parallel, we fix the thread block size to result in the maximum occupancy [1] and adjust the maximum number of triangles among all patches to conform that.

Table 3.4 shows the performance comparison of patchFIM with two single-threaded CPU implementations (*i.e.*, FMM and meshFIM) on the same meshes and speed functions, and shows the speedup factors of patchFIM over the CPU methods. Communication times between CPU and GPU, which are only about one tenth of the running times in our experiments, are not included for patchFIM to give a more accurate comparison of the methods . As shown in this result, the patchFIM algorithm maps very well to the GPU and achieves a good performance gain over both the serial and multithreaded CPU solvers. On a simple case such as Mesh 1 with Speed 1, patchFIM runs about 33 times faster than meshFIM-ST and 25 times faster than FMM. On other more complex cases, patchFIM runs up to 15 times faster than FMM. In addition, on the heterogeneous media using Mesh 3 with Speed 2, where meshFIM-ST runs roughly half as fast as FMM on the CPU, patchFIM still runs about 14 times faster than FMM.

As shown in this result, SIMD efficiency of the meshFIM algorithm depends on the input mesh configuration, more specifically, the average vertex valence relative to the highest valence. Thus, Mesh 1 is the most efficient set up because almost all vertices have valence six. In contrast, Mesh 2 shows the worst performance due to the highest vertex valence of 11. Meshes 3 and 4 have a maximum valence of 8. Moreover, Mesh 2 has the largest percentage of high valence (greater than 6) vertices. Mesh 3 and 4 are commonly found set up where valences follow a tight, symmetric distribution centered valence six. In summary, patchFIM implemented on the GPU runs faster than any existing CPU-based solver on all examples we tested, with the effectiveness depending on mesh configuration and distribution of valences of vertices. Many applications based on time-consuming eikonal equation

Table 3.4. Running times (milliseconds) and speedups (factor) for different algorithms and architectures.

	Mesh 1 with Speed 1	Mesh 2 with Speed 1	Mesh 3 with Speed 1	Mesh 4 with Speed 1	Mesh 3 with Speed 2
FMM	5092	7063	6362	3612	6435
meshFIM-ST	6562	9354	8591	4331	11960
patchFIM	201	910	415	287	459
Speedup over FMM	25×	8×	15×	13×	14×
Speedup over meshFIM-ST	33×	10×	21×	15×	28×

solvers can run at real-time or interactive rates using the proposed method.

In patchFIM, there are two user-defined parameters: the size of patch and the iteration number within an active patch update. In our experiments, the empirically optimal patch size is 64 vertices, which means the maximum number of vertices among all patches is 64. There is a trade-off here. On the one hand, the smaller patch sizes efficiently concentrate vertex updates on the wavefront. This is because we update all the vertices of a patch each iteration, while only the updates for the vertices on the wavefront are useful. For smaller patch sizes, the average ratio of number of vertices inside the wave front to the total number of vertices in this patch is higher; hence, there is less percentage of useless computation. On the other hand, the SIMD architecture requires the patch size to be large enough to take advantage of the large number of processors and to hide the hardware latency [68] associated with memory transfers. A small parameter study of different patch sizes showed 64 vertices to be an effective compromise and that this parameter is consistent across different meshes.

3.3.3 Analysis of Results

In the previous section, the performance of the eikonal solvers are compared based on the running time on different architectures. Because running time can be affected by many factors, such as implementation schemes and hardware performance, we measure the number of local solver calls for a more precise performance analysis in this section. We also briefly discuss the accuracy of the proposed method, and introduce parameter optimization techniques for GPU implementation.

3.3.3.1 Asymptotical Cost Analysis

The most time-consuming operation for the eikonal equation solver is the update of the solution on a vertex with its one-ring triangles, and each update includes N local solver calls where N is the valence of this vertex. Table 3.5 compares the average number of local solver calls per vertex on different meshes with different speed functions.

As can be seen from Table 3.5, FMM requires approximately 18 local solver calls in all cases. This can be explained as follows. For FMM, the solutions of the vertices on the wavefront may be computed multiple times. Each vertex has six neighbors on the average, and statistically half of the neighbors are potentially upwind. Thus, each vertex is updated roughly three times, and each time requires a solve for the six triangles in the one-ring. This explains the characteristic 18 solves per vertex, independent of the meshes and speed functions. In comparison, the average number of local solver calls for meshFIM depends largely on the speed function, which can be noticed when comparing Speed 2 with Speed 1. In addition, the average number of local solver calls for meshFIM-ST is more than that of FMM on all the experiment settings. This difference in number of calls is offset, but only slightly, by the extra work of FMM in maintaining the heap. The multithreaded CPU version (meshFIM-MT) needs more updates because of the extra computation associated with simultaneous updates in the red-black Gauss-Seidel iteration scheme. This explains, to some extent, why we get about three times speedup on a quad-core CPU. The patchFIM method incurs an extra computation associated with patch-based updates. This factor of 5–20 is consistent with the run times we see. Roughly, if we have 200 processors operating at approximately

Table 3.5. Average number of local solver calls per vertex for different algorithms.

	Mesh 1 with Speed 1	Mesh 2 with Speed 1	Mesh 3 with Speed 1	Mesh 4 with Speed 1	Mesh 3 with Speed 2
FMM	17.9	19.5	18.1	18.3	18.1
meshFIM-ST	18.0	23.3	24.4	19.6	59.2
meshFIM-MT	18.0	26.6	46.1	23.1	83.1
patchFIM (GPU)	105.0	595.5	290.9	251.2	334.1

half the clock rate, we would expect, ideally, a 100× advantage. However, with the efficiencies shown in Table 3.5, we would expect a 5–20× speed advantage on the GPU (relative to FMM), which is consistent with data in Table 3.4. This result also provides evidence that the CUDA implementation achieves a computational density that is high enough to offset latency and memory management overhead.

We can asymptotically compare the computational costs of the FMM and meshFIM algorithms as follows [52]. Let k_1 and k_2 be the costs for a local solver and a heap updating operation, respectively. Suppose P_{FMM} and P_{FIM} are the average number of local solver calls per vertex in FMM and meshFIM-ST, respectively (as in Table 3.5). Let h be the average heap size. The total costs for FMM and meshFIM-ST on a mesh with N vertices can be defined asymptotically as follows:

$$\begin{cases} C_{FMM} = N(k_1 P_{FMM} + k_2 P_{FMM} log_2 h) = N k_1 P_{FMM}(1 + \frac{k_2}{k_1} log_2 h) \\ C_{FIM} = N k_1 P_{FIM}. \end{cases}$$

The value $\frac{k_2}{k_1}$ is empirically measured to be about 0.02. Hence, the ratio for the costs of meshFIM and FMM is: $\frac{C_{FIM}}{C_{FMM}} = \frac{P_{FIM}}{P_{FMM}(1+0.02 log_2 h)}$.

For the setting with Mesh 1 and Speed 1, the average heap size h (which is proportional to the arc length of the expanding wavefront) is 1302 for FMM and $\frac{P_{FIM}}{P_{FMM}}$ is approximately 1.67, as can be derived from Table 3.5. Therefore, $\frac{C_{FIM}}{C_{FIM}} \approx 1.38$ in this case, which is consistent with the experimental results in Table 3.4.

As shown in the above analysis, $k_1 >> k_2$ in C_{FMM}, so the impact of the update operations on the performance of FMM is much more significant than that of the heap operations for moderately sized meshes. This is juxtaposed with the lower cost of computing node updates on regular grids, which makes FIM more competitive with FMM in that circumstance, even for serial implementations [52]. It can also be seen that, with a larger mesh (which means larger h), the performance difference between single-threaded meshFIM (C_{FIM}) and FMM (C_{FMM}) will be less. Of course the design goal of meshFIM is that it can be mapped well to parallel architectures. Even with some performance degradation from Gauss-Seidel iteration in meshFIM-ST to red-black Gauss-Seidel in meshFIM-MT, we can still get large performance gain from running on multiple core CPUs.

The performance of meshFIM is determined by the number of updates (or the number of local solver calls), which depends heavily on the heterogeneity of the speed function. The following experiment systematically characterizes how the speed function affects the performance of these algorithms. First, we generate white noise for the initial speed function, and then apply a mesh Laplacian operator [109] N times to the initial speed function to make the speed function less heterogeneous for increasing N. Figure 3.6 shows the result of this experiment. The x axis is the number of total vertices in the mesh and the y axis is the number of local solver calls. As N becomes bigger, and the speed function more homogeneous or smooth, the plot becomes less steep, and the results become closer to the meshFIM results with a constant speed function. We also see that FMM increases linearly with number of vertices, as expected.

3.3.3.2 Error Analysis

To show that the proposed algorithm achieves the first-order accuracy we would expect from the linear elements introduced in the solver, we performed a convergence analysis. We use seven regularly triangulated square meshes, representing a 16×16 patch of $\Re^2$, with the number of vertices ranging from 256 to

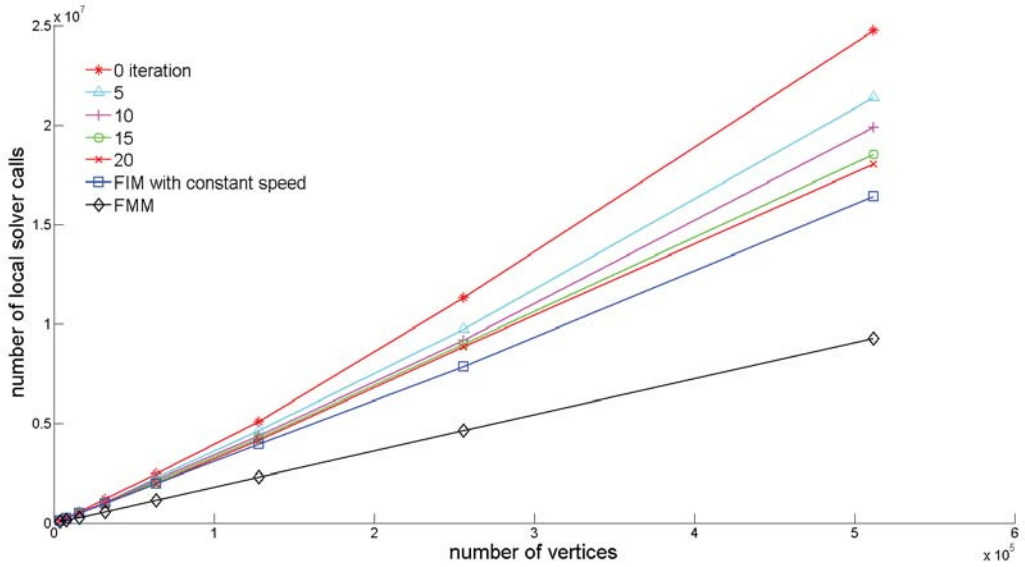
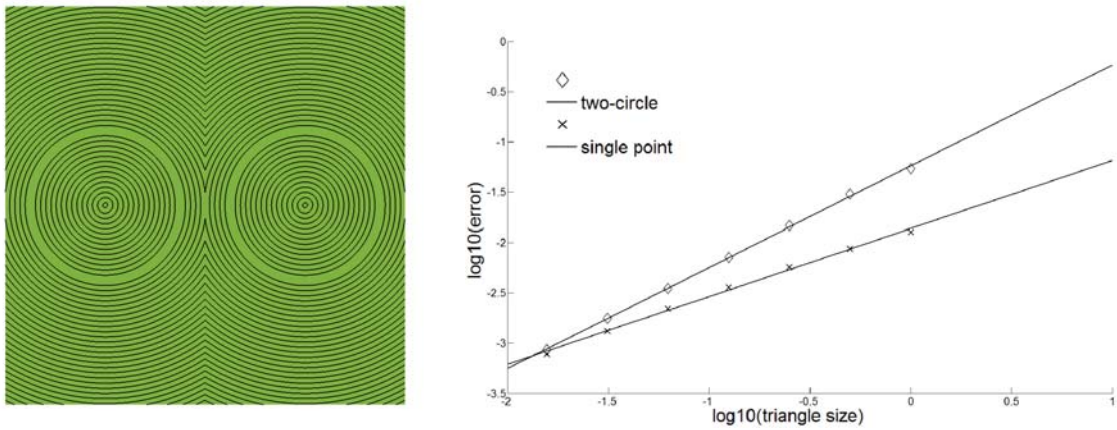


Figure 3.6. Laplacian experiment results

1,048,576. We considered two cases of boundary conditions. In the first case, we used a pair of isolated points and in the second case, we used a pair of circles of radius 3, where the domain is 16×16. Boundary conditions were projected onto the grid using the nearest vertices to the circles or points. We then solve for the distances to these boundaries for the entire domain using the patchFIM eikonal solver and compare against analytical results at the vertices using the average squared error (L_2)—similar plots result from *sup* error. Figure 3.7(a) shows the level sets of a solution to the circular boundary conditions. Finally, we can plot these errors against the size of triangles, as shown in Figure 3.7(b). For the circular boundary conditions, the slope of this graph is 1.0, which is consistent to our claim that meshFIM is first-order accurate. For the point boundary conditions, the slope is less—showing the method is not first-order accurate for nonsmooth boundaries, which are inconsistent with the governing equations.

3.3.3.3 Parameter Optimization

As for the iteration number within an active patch update, every active patch is updated multiple times before its convergence is checked. There are two motivations. First, not all the vertices in a patch reach a consistent configuration



(a) (b)

Figure 3.7. Results for different boundaries: a) The level sets of the solution of the eikonal equation which represents distance to two circular boundaries. b) The error as a function of resolution shows first-order convergence for smooth boundary conditions.

with a single update. This is clear if we imagine a wavefront of active vertices initiated at one side of a patch propagating to the other side. The check for convergence requires communication with the CPU, and we would like to make maximum use of the fast on-chip shared memory space without communicating with the main memory. However, if the number of iterations per patch n is too large, the algorithm executes useless extra updates after reaching a consistent configuration. Generally, n is proportional to the patch diameter, which is related to the number of iterations it takes for a wavefront to propagate across a patch. The optimal choice of n depends not only on the size of the patch but also on the input speed function. In general, according to our experiments, the best n can be around 7 for most cases for patches of approximately 64 vertices. The running times for $n < 7$ can be quite good, but are not stable across different data sets and speed functions. However, for $n > 7$ the running time becomes stable and gradually increases as n increases. This is because patches with 64 vertices usually converge in about seven updates, and therefore, wavefront propagation is almost identical with $n > 7$ iterations.

3.4 Conclusions

In this chapter, we propose a fast and easily parallelizable algorithm to solve the eikonal equation on unstructured triangular meshes on a single-core CPU and on parallel, streaming architectures with restrictions on local memory. The proposed algorithms are based on the fast iterative method with modifications to accommodate unstructured triangular grids. The method employs a narrow band method to keep track of the mesh vertices to be updated and iteratively updates vertex values until they converge. Instead of using an expensive sorting data structure to ensure the causality, the proposed method uses a simple list to store active vertices and updates them asynchronously, using an ad-hoc ordering, which can be determined by the hardware. The vertices in the list are removed from or added to the list based on the convergence, which is a measure of consistency with neighboring vertices. The method is easily portable to parallel architectures, which is difficult or infeasible with many existing methods.

CHAPTER 4

A FAST ITERATIVE METHOD FOR SOLVING THE EIKONAL EQUATION ON TETRAHEDRAL DOMAINS

4.1 Introduction

The eikonal equation and its variations (forms of the static Hamilton-Jacobi and levelset equations) are used as models in a variety of applications, ranging from robotics and seismology to geometric optics. These applications include virtually any problem that entails the finding of shortest paths, possibly with inhomogeneous or anisotropic metrics (*e.g.*, due to material properties). In seismology, for example, the eikonal equation describes the travel time of the optimal trajectories of seismic waves traveling through inhomogeneous anisotropic media [81]. In cardiac electrophysiology [74], action potentials on the heart can be represented as moving interfaces that can be modeled with certain forms for the eikonal equation [26, 56]. The eikonal equation also describes the limiting behavior of Maxwell's equations [43], and is therefore useful in geometric optics (*e.g.*, [25, 72]).

As described in [10], many of these cases present a clear need to solve such problems on fully unstructured meshes. In particular, in this work, the use of unstructured meshes is motivated by the need for body-fitting meshes. In certain problems, such as cardiac simulations, the domain is a volume bounded by a smooth, curved surface, and triangle meshing strategies for surfaces combined with tetrahedral meshing of the interior can accurately and efficiently capture these irregular domains (*e.g.*, see Figure 1.1-left). In other problems, such as in the case of geometric optics (Figure 1.1-right) or in geophysics applications, irregular unstructured meshes allow for accurate, efficient modeling of material discontinuities that are represented as triangulated surfaces embedded in a tetrahedral mesh.

While solutions of the eikonal equation are used in their own right in many physical problems, such solutions are also used as building blocks in more general computational schemes such as in remeshing and in image/volume analysis (*e.g.*, [3, 5, 23, 87]). When used as part of a more general computational pipeline, it is essential that effort be expended to minimize the computational cost of this component in an attempt to optimize the time of employing the pipeline. There is a clear need for the development of fast algorithms that provide solutions of the eikonal equation on unstructured 3D meshes.

Recent developments in computer hardware show that performance improvements will no longer be driven primarily by increased clock speeds, but by parallelism and hardware specialization. Single-core performance is leveling off, while hex-core CPUs are available as commodities; soon, conventional CPUs will have tens of parallel cores. Commodity multimedia processors such as the IBM Cell and graphics processing units (GPUs) are considered forerunners of this trend. To obtain solutions in an efficient manner on these state-of-the-art Single-Instruction-Multiple-Data (SIMD) type computer architectures places particular constraints on the data dependencies, memory access, and scale of logical operations for such algorithms.

Building an efficient three-dimensional tetrahedral eikonal solver for multicore and SIMD architectures poses many challenges, some unique to working with three-dimensional data. First of all, as in two dimensions, the update scheme of the solver needs to be easily parallelizable and pose no data dependencies for the active computational domain, which will change as the solution progresses. Secondly, representing the topology of an unstructured 3D mesh imposes a significant memory footprint compared to its two-dimensional counterpart, creating challenges in achieving the *computational density* necessary to make use of the limited memory, registers, and bandwidth on massively parallel SIMD machines. Thirdly, the vertex valences of the three-dimensional unstructured meshes can be both quite high and can be highly variable across the mesh, posing additional challenges in SIMD efficiency.

In the past several decades, many methods have been proposed to efficiently

solve the eikonal equations on regular and unstructured grids. The fast marching method (FMM) by Sethian [57] (a triangular mesh extension of [88]) is often considered the *de facto* state-of-the-art for solving the eikonal equation; its asymptotic worse case complexity, $O(N \log N)$, was shown to be optimal. It attains optimality by maintaining a heap data structure with a list of active nodes, on a moving front, that are candidates for updating. The node with the shortest travel time is considered to be solved, removed from the list, and never visited again. This active list contains only a (relatively small) subset of the nodes within the entire mesh. Though it provides worst-case optimality for the serial case, the use of a heap data structure greatly limits the parallelization of the approach. Zhao [110] and Tsai *et al.* [100] introduced an alternative approach, the fast sweeping method (FSM), which uses a Gauss-Seidel style update strategy to progress across the domain in an incremental grid-aligned *sweep*. This method does not employ the sorting strategy found in FMM, and hence is amenable to coarse-grained parallelization [46, 101, 111]. The Gauss-Seidel style sweeping approach of FSM, however, is a significant limitation when attempting to build a general, efficient fine-grained parallel eikonal solver over tetrahedral meshes. Although one can do as is traditionally done in parallel computing and employ coloring techniques (*e.g.*, red-black) to attempt to mitigate this issue [93], one cannot push this strategy to the levels needed for the fine-grain parallelization required on current streaming architectures. Furthermore, any gains through parallelism must offset any suboptimal behavior; previous work has shown that FSM introduces a large amount of excess computation for certain classes of realistic input data [52].

In this chapter, we put forward a new local solver specially designed for tetrahedral meshes and anisotropic speed functions, propose a data compaction strategy to reduce the memory footprint (and hence reduce costly memory loads) of the local solver, design new data structures to better suit the high valence numbers typically experienced in three-dimensional meshes, and also propose a GPU-suitable sorting-based method to generate the gather-lists to enable a lock-free update. We also propose a new computational method to solve the eikonal equation on three-dimensional tetrahedral meshes efficiently on parallel

streaming architectures; we call our method the *tetrahedral fast iterative method* (tetFIM). The framework is conceptually similar to the previously proposed FIM methodology [37, 52] for triangle meshes, but the move to three-dimensions for solving realistic physics-based problems requires two significant extensions. First is a principles-based local solver which handles anisotropic material (which is needed for realistic three-dimensional physics-based simulations such as in geometric optics and seismology). Second is the corresponding re-evaluation and redesign of the computational methodology for triangles in order to fully exploit streaming hardware in light of the additional mathematical complexities required for solving the eikonal equation in inhomogeneous, anisotropic media on fully three-dimensional tetrahedralizations. This chapter also provides algorithmic and implementation details, as well as a comparative evaluation, for two data structures designed to efficiently manage three-dimensional unstructured meshes on GPUs. The data-structure issue is particularly important in 3D, because of the increased connectivity of the mesh and the need to mitigate the cost of loading three-dimensional data to processor cores in order to keep the computational density high.

The remainder of the chapter proceeds as follows. In Section 4.2, we present the mathematical and algorithmic description of the fast iterative method for solving the inhomogeneous anisotropic eikonal equation on fully unstructured tetrahedral domains. We then in Section 4.3 describe how the proposed algorithm can be efficiently mapped to serial and multithreaded CPUs and to streaming architectures such as the GPU. In Section 4.4, we provide results that compare both our CPU and GPU implementations against other widely-used methods and discuss the benefits of our method. We present conclusions and future work in Section 4.4.6.

4.2 Mathematical and Algorithmic Description

In this section, we describe the mathematics associated with the eikonal equation and the corresponding algorithm we propose for its solution. The main building blocks of the method are a new local solver and the active list update scheme. The local solver, upon being given a proposed solution of the eikonal equation on three

of the four vertices of a tetrahedron, updates the fourth vertex value in manner that is consistent with the characteristics of the solution. The update scheme is the management strategy for the active list, consisting of the rules for when vertices are to be added, removed, or remain on the list. We refer to the combination of these two building blocks as *tetFIM*.

4.2.1 Notation and Definitions

The eikonal equation is a special case of the nonlinear Hamilton-Jacobi partial differential equations (PDEs). In this chapter, we consider the numerical solution of this equation on a 3D domain with an inhomogeneous, anisotropic speed function:

$$\begin{cases} H(\mathbf{x}, \nabla\phi) = \sqrt{(\nabla\phi)^T M(\nabla\phi)} = 1 \quad \forall \mathbf{x} \in \Omega \subset \mathbb{R}^3 \\ \phi(\mathbf{x}) = B(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{B} \subset \Omega \end{cases} \tag{4.1}$$

where Ω is a 3D domain, $\phi(\mathbf{x})$ is the travel time at position $\mathbf{x}$ from a collection of given (known) sources within the domain, $M(\mathbf{x})$ is a 3×3 symmetric positive-definite matrix encoding the speed information on Ω, and $\mathcal{B}$ is a set of smooth boundary conditions which adhere to the consistency requirements of the PDE. We approximate the domain Ω by a planar-sided tetrahedralization denoted by Ω_T. Based upon this tetrahedralization, we form a piecewise linear approximation of the solution by maintaining the values of the approximation on the set of vertices V and employing linear interpolation within each tetrahedral element in Ω_T. We let M be constant per tetrahedral element, which is consistent with a model of linear paths within each element. v_i denotes the ith vertex in V whose position is denoted by a 3-tuple $\mathbf{x}_i = (x, y, z)^T$ where $x, y, z \in \mathbb{R}$. An *edge* is a line segment connecting two vertices (v_i, v_j) in $\mathbb{R}^3$ and is denoted by $e_{i,j}$. Two vertices that are connected by an edge are neighbors of each other. $\mathbf{e}_{i,j}$ denotes the vector from vertex v_i to vertex v_j and $\mathbf{e}_{i,j} = \mathbf{x}_j - \mathbf{x}_i$. The angle between $e_{i,j}$ and $e_{i,k}$ is denoted by $\angle_i$ or $\angle_{j,i,k}$.

A tetrahedron, denoted $T_{i,j,k,l}$, is a set of four vertices v_i, v_j, v_k, v_l that are each connected to the others by an edge. A tetrahedral face, the triangle defined by vertices v_i, v_j and v_k of $T_{i,j,k,l}$, is denoted $\Delta_{i,j,k}$. The solid angle ω_i at vertex v_i subtended by the tetrahedral face v_j, v_k, v_l is given by $\omega_i = \xi_{j,k} + \xi_{k,l} + \xi_{l,j}$, where $\xi_{j,k}$ is the dihedral angle between the planes that contain the tetrahedral faces $\Delta_{i,j,l}$ and

$\Delta_{i,k,l}$ and define $\xi_{k,l}$ and $\xi_{l,j}$ correspondingly. We define a tetrahedron as an *acute tetrahedron* when all its solid angles are smaller than 90 degrees while we define an *obtuse tetrahedron* as one in which one or more of its solid angles is larger than 90 degrees. We note that one can define both an acute and obtuse tetrahedron in terms of *dihedral angle*, which is equivalent to the proposed definition. We call the vertices connected to vertex v_i by an edge the *one-ring neighbors* of v_i, and the tetrahedra sharing vertex v_i are called the *one-ring tetrahedra* of v_i. We denote the discrete approximation to the true solution ϕ at vertex v_i by Φ_i.

4.2.2 Definition of the Local Solver

One of the main building blocks of the proposed algorithm is the *local solver*, a method for determining the arrival time at a vertex assuming a linear characteristic across a tetrahedron emanating from the planar face defined by the other three vertices—whose solution values are presumed known. In this section, we define the actions of the local solver for both acute and obtuse tetrahedron.

Given a tetrahedralization Ω_T of the domain, the numerical approximation, which is linear within each tetrahedron, is given by $\Phi(\mathbf{x})$ and is defined by specifying the values of the approximation at the vertices of the tetrahedra. The solution (*travel time*) at each vertex is computed from the linear approximations on its one-ring tetrahedra. From the computational point-of-view, the bulk of the work is in the computation of the approximations from the adjacent tetrahedra of each vertex—work accomplished by the local solver.

Because acute tetrahedra are essential for proper numerical consistency [57], we consider the case of acute tetrahedra first and then discuss obtuse tetrahedra subsequently. The specific calculation on each acute tetrahedron is as follows. Considering the tetrahedron $T_{1,2,3,4}$ depicted in Figure 4.1, we use an upwind scheme to compute the solution Φ_4, assuming the values Φ_1, Φ_2, and Φ_3 comply with the causality property of the eikonal solutions [79]. The speed function within each tetrahedron is constant, so the travel time to v_4 is determined by the time/cost associated with a line segment lying within the tetrahedron $T_{1,2,3,4}$, and this line segment is along the wave front normal direction that minimizes the value at v_4. The key step is to determine the normal direction $\mathbf{n}$ of the wavefront and establish

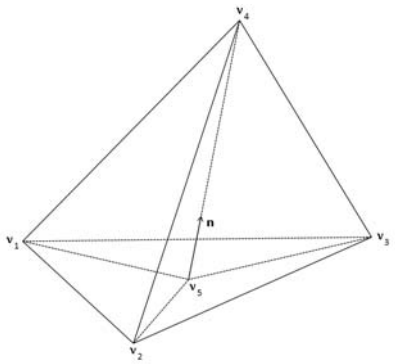


Figure 4.1. Diagram denoting components of the local solver. We compute the value of the approximation at the vertex v_4 from the values at vertices v_1, v_2, and v_3. The vector **n** denotes the wave propagation direction that intersects with the triangle $\Delta_{1,2,3}$ at v_5.

whether or not the causality condition is satisfied. The ray that has a direction **n** and passes through the vertex v_4 must fall inside the tetrahedron $T_{1,2,3,4}$ in order to satisfy the causality condition. To check such a causality condition numerically, we first compute the coordinates of the point v_5 at which the ray passing through v_4 with direction **n** intersects the plane spanned by v_1, v_2, and v_3 and then then check to see whether or not v_5 is inside the triangle $\Delta_{1,2,3}$.

We denote the travel time for wave to propagate from the vertex v_i to the vertex v_j as $\Phi_{i,j} = \Phi_j - \Phi_i$, and therefore, the travel time from v_5 to v_4 is given by $\Phi_{5,4} = \Phi_4 - \Phi_5 = \sqrt{\mathbf{e}_{5,4}^T M \mathbf{e}_{5,4}}$, according to the Fermat principle as it applies to Hamilton-Jacobi equations [100]. An alternative derivation of this principle from the perspective of geometric mechanics is given in [47]. Using the linear model within each cell and barycentric coordinates $(\lambda_1, \lambda_2, \lambda_3)$ to denote the position of v_5 on the tetrahedral face, we can express the approximate solution at v_5 as $\Phi_5 = \lambda_1\Phi_1 + \lambda_2\Phi_2 + \lambda_3\Phi_3$, where the position is given by $\mathbf{x}_5 = \lambda_1\mathbf{x}_1 + \lambda_2\mathbf{x}_2 + \lambda_3\mathbf{x}_3$. Here, λ_1, λ_2, λ_3 satisfy that $\lambda_1 + \lambda_2 + \lambda_3 = 1$. This gives the following expression for Φ_4:

$$\Phi_4 = \lambda_1\Phi_1 + \lambda_2\Phi_2 + (1 - \lambda_1 - \lambda_2)\Phi_3 + \sqrt{\mathbf{e}_{5,4}^T M \mathbf{e}_{5,4}}. \tag{4.2}$$

The goal is to find the location of v_5 that minimizes Φ_4. Thus, we take the partial derivatives of Equation 4.2 with respect to λ_1 and λ_2 and equate them with zero to obtain the conditions on the interaction of the characteristic and the opposite face:

$$\begin{cases} \Phi_{1,3}\sqrt{\mathbf{e}_{5,4}^T M \mathbf{e}_{5,4}} = \mathbf{e}_{5,4}^T M \mathbf{e}_{1,3} \\ \Phi_{2,3}\sqrt{\mathbf{e}_{5,4}^T M \mathbf{e}_{5,4}} = \mathbf{e}_{5,4}^T M \mathbf{e}_{2,3}. \end{cases} \tag{4.3}$$

If $\Phi_{1,3}$ and $\Phi_{2,3}$ are not both zero, we have the following linear equation:

$$\Phi_{2,3}(\mathbf{e}_{5,4}^T M \mathbf{e}_{1,3}) = \Phi_{1,3}(\mathbf{e}_{5,4}^T M \mathbf{e}_{2,3}). \tag{4.4}$$

We must now solve Equation 4.4 and either one of Equation 4.3 for λ_1 and λ_2. If no root exists, or if λ_1 or λ_2 falls outside the range of $[0,1]$ (that is, the characteristic direction does not reside within the tetrahedron), we then apply the 2D local solver used in [37] to the faces $\Delta_{1,2,4}$, $\Delta_{1,3,4}$, and $\Delta_{2,3,4}$ and select the minimal solution from among the three. The surface solutions allow for the same constraint, and if the minimal solutions falls outside of the tetrahedral face, we consider the solutions along the edges for which we are guaranteed a minimum solution exists. Because the quantity being minimized, there can be only one minimum, and the optimal solution associated with that element must pass through the tetrahedron or along one of its faces/edges.

In the case of parallel architectures with limited high-bandwidth memory, the memory footprint of the local solver becomes a bottleneck to performance. The smaller the memory footprint of the local solver, the higher the computational density one can achieve on the streaming processors, and the closer one gets to the 100-200× raw improvement in processing power (relative to a conventional CPU). Here we explore the algebra a little more carefully to reduce these computations to their fundamental degrees of freedom. Solving Equations 4.3–4.4 directly requires storing all the coordinates of the vertices and the components of M, which is 18 floating point values in total. In practice, we can reduce the computations and memory storage based on the observation that $\mathbf{e}_{5,4}$ can be reformatted as: $\mathbf{e}_{5,4} = \mathbf{x}_4 - \mathbf{x}_5 = \mathbf{x}_4 - (\lambda_1\mathbf{x}_1 + \lambda_2\mathbf{x}_2 + \lambda_3\mathbf{x}_3) = [\mathbf{e}_{1,3}\ \mathbf{e}_{2,3}\ \mathbf{e}_{3,4}]\lambda$, where $\lambda = [\lambda_1\ \lambda_2\ 1]^T$. Hence, we obtain

$$\mathbf{e}_{5,4}^T M \mathbf{e}_{5,4} = \lambda^T [\mathbf{e}_{1,3}^T\ \mathbf{e}_{2,3}^T\ \mathbf{e}_{3,4}^T]^T M [\mathbf{e}_{1,3}\ \mathbf{e}_{2,3}\ \mathbf{e}_{3,4}]\lambda = \lambda^T M' \lambda \tag{4.5}$$

where $M' = [\alpha\ \beta\ \theta]$ with

$$\begin{cases} \alpha = [\mathbf{e}_{1,3}^T M \mathbf{e}_{1,3} \;\; \mathbf{e}_{2,3}^T M \mathbf{e}_{1,3} \;\; \mathbf{e}_{3,4}^T M \mathbf{e}_{1,3}]^T \\ \beta = [\mathbf{e}_{1,3}^T M \mathbf{e}_{2,3} \;\; \mathbf{e}_{2,3}^T M \mathbf{e}_{2,3} \;\; \mathbf{e}_{3,4}^T M \mathbf{e}_{2,3}]^T \\ \theta = [\mathbf{e}_{1,3}^T M \mathbf{e}_{3,4} \;\; \mathbf{e}_{2,3}^T M \mathbf{e}_{3,4} \;\; \mathbf{e}_{3,4}^T M \mathbf{e}_{3,4}]^T \end{cases} \tag{4.6}$$

and

$$\begin{cases} \mathbf{e}_{5,4}^T M \mathbf{e}_{1,3} = \lambda^T \alpha \\ \mathbf{e}_{5,4}^T M \mathbf{e}_{2,3} = \lambda^T \beta. \end{cases} \tag{4.7}$$

Plugging Equation 4.5, 4.6, and 4.7 into Equations 4.3 and 4.4, we obtain

$$\begin{cases} \Phi_{1,3} \sqrt{\lambda^T M' \lambda} = \lambda^T \alpha \\ \Phi_{2,3} \lambda^T \beta = \Phi_{1,3} \lambda^T \alpha. \end{cases} \tag{4.8}$$

Solving Equation 4.8 only requires storing M', which is symmetric so only requires six floats per tetrahedron.

Having defined the acute tetrahedron local solver, we now discuss the case of obtuse tetrahedra. The computation of the solution for linear approximations on tetrahedral elements has poor approximation properties when applied to obtuse tetrahedra [82]. The issue of dealing with *good versus bad meshes* is not the main focus of this chapter or the proposed algorithm, but limited incidences of obtuse tetrahedron can be addressed within the local solver. To accomplish this, we extend the method proposed in [57], originally designed for triangular meshes to work for tetrahedral meshes. As shown in Figure 4.2 where ω_4 is obtuse, we connect v_4 to the vertex v_5 of a neighboring tetrahedron and thereby cut the obtuse solid angle into three smaller solid angles. If these three solid angles are all acute, then the process stops, as shown in the left images of Figure 4.2; otherwise, if one of the smaller solid angles is still obtuse, then we connect v_4 to the vertex v_6 of another neighboring tetrahedron. This process is performed recursively until all new solid angles at v_4 are acute as shown in the right image of Figure 4.2, or the opposite triangular faces coincides with a boundary. Note that algorithmically, these added edges and tetrahedra are not considered part of the mesh; they are considered virtual and only used within the local solver for updating the solution at v_4. We cannot prove the convergence of this refinement algorithm, and the above recursion could propagate extensively throughout the mesh in extraordinary cases. In practice, the

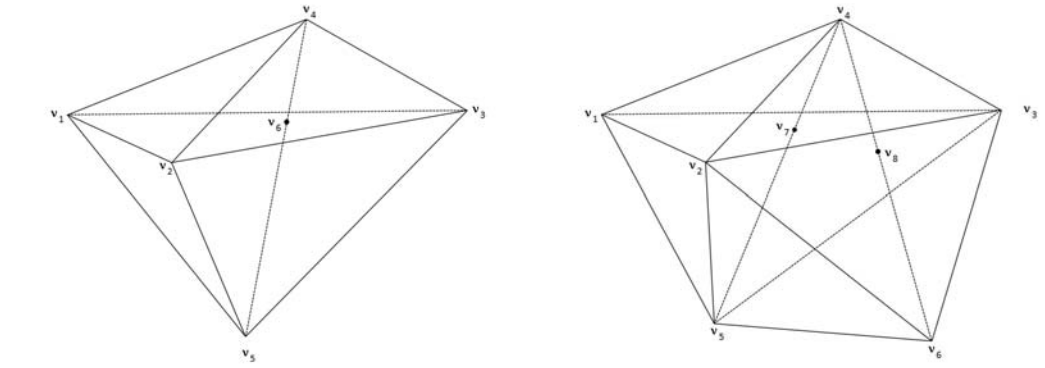


Figure 4.2. Diagram denoting the strategy used to deal with obtuse tetrahedra. We split the obtuse angle ω_4 to create three virtual tetrahedra used within the local solver.

algorithm would be forced to terminate after a fixed number of splits emanating from a single vertex — in all of the meshes in this chapter, the algorithm recursed no more than once.

4.2.3 Active List Update Scheme

The proposed algorithm uses a modification of the active list update scheme as presented in [37, 52] combined with the new local solver described above designed for unstructured tetrahedral meshes with inhomogeneous anisotropic speed functions.

The algorithm is iterative, but for efficiency, the updates are limited to a relatively small domain that forms a collection of narrow bands that form wavefronts of values that require updating. This narrow banding scheme uses a data structure, called *active list*, to store the vertices or tetrahedra slated for revision and these vertices/tetrahedra are called active vertices/tetrahedra. During each iteration, active vertices/tetrahedra can be updated in parallel and after the updates of all the active vertices/tetrahedra, the active list is modified to eliminate vertices whose solutions are consistent with their neighbors and to include vertices that could be affected by the last set of updates. Convergence of the algorithm to a valid approximation of the eikonal equation was proven in [52].

4.3 TetFIM Serial and Parallel Implementations

In this section, we provide implementation details in terms of methods and data structures necessary for the efficient instantiation of our local solver and active list update scheme on serial CPUs, multithreaded CPUs, and streaming SIMD parallel architectures.

4.3.1 Implementation on Serial and Multithreaded CPUs

The proposed method builds on the fast iterative method proposed for structured meshes [52], which operates as follows. Nodes on the active list are revised individually, and the corresponding values remain consistent with their upwind neighbors. Then, each updated value immediately overwrites the previous solution. The algorithm runs through the active list, constantly revising values, and at the end of the list, it loops back to the beginning. As such, the list has no real beginning or end. A vertex is removed from the active list when the difference between its old and revised values is below a predetermined tolerance —effectively, the value at the vertex does not change within the range of the prescribed tolerance from the previous update. We specify a vertex whose value remains unchanged (within some tolerance ϵ) as *ϵ-converged*. As each ϵ-converged vertex is removed from the active list, all of its potentially downwind neighbors (neighbors with larger value) are updated. If their values are not ϵ-converged (*i.e.*, they deviate significantly), they are included in the active list. The algorithm keeps updating the vertices in the active list until the list is empty.

The update of an active vertex does not depend on the other updates; hence, we can extend the single-threaded algorithm to shared memory multiprocessor systems by simply partitioning arbitrarily, at each iteration, the active list into N sublists and assigning the sublists to N threads. Each thread asynchronously update the vertices within the sublist. These updates are done by applying the updating step to each partition of the active list. In practice, we choose N to be twice the number of CPU cores to take full advantage of Intel's hyper-threading technology. At the beginning of an iteration, if there are n nodes in the active list, the sublist size M is given by $M = \lceil \frac{n}{N} \rceil$. The active list is evenly divided into N sublists, each containing M consecutive active nodes except for the last sublist

which may contain fewer than M active nodes. These N sublists are then assigned to N threads.

4.3.2 Implementation on Streaming SIMD Parallel Architectures

To exploit the GPU performance advantage, we propose a variation of tetFIM, called tetFIM-A, that adapts well on SIMD architectures by combining an agglomeration-based update strategy that is divided across blocks and carefully designed data structures for 3D tetrahedral meshes. In this method, the computational domain (mesh) is split into minimally overlapping agglomerates (sharing only one layer of tetrahedra) and each agglomerate is treated with logical correspondence to a vertex in the original tetFIM. The vertices in each agglomerate are updated in a SIMD fashion on a block, and the on-chip cache is employed to store the agglomerate data and the intermediate results. Similar to the CPU variants of tetFIM, a narrow banding scheme is used to focus the computation in terms of the necessary computational region. The active list consists of a set of active agglomerates instead of active vertices.

In an iteration, each active agglomerate is loaded from the global memory to a block, and the values of all vertices in this agglomerate are updated by a sequence of SIMD iterations which we call *internal iterations*. The agglomerate data are copied to the on-chip memory space, and the internal iterations are performed to revise the solutions of the vertices in that agglomerate. In general, the whole computation consists of two steps: the preprocessing and the iteration.

4.3.2.1 Preprocessing

The tetFIM-A requires setup or preprocessing before the computation of the solution. First, we divide the mesh into agglomerations through a multilevel partitioning scheme described in [55]. The specific algorithm for mesh partitioning is not essential to the suggested algorithm, except that efficiency is achieved for agglomerates with matching numbers of vertices/tetrahedra and relatively few vertices on the agglomerate boundaries. We also precompute the static mesh information, including the extra information associated with the obtuse tetrahedra, and prepare the necessary data for the iteration step, including compaction of the

speed and geometric data and generation of the *gather-lists* which will be described below.

4.3.2.2 Iteration step

In this step, each agglomerate is treated just like a vertex in tetFIM, and the main iteration continues until the active list becomes empty. The main iteration consists of three stages as outlined below. First, each agglomerate in the active list is assigned to a SIMD computing unit. Second, once the agglomerate is updated, we check to see if the agglomerate is ϵ-converged, *i.e.*, all vertices in an agglomerate are ϵ-converged. Checking the agglomerate convergence entails updating the entire agglomerate once and seeing if there exists a vertex with a changed solution. This is done with a reduction operation, which is commonly employed in the streaming programming model to efficiently produce aggregate measures (sum, max, *etc.*) from a stream of data [75]. Finally, we deal with the effects of an update on the active list. If an agglomerate is not ϵ-converged, we add it into the active list; otherwise, we add its neighboring agglomerates to the active list and then go to the first stage and repeat the update again (see Algorithm 4.1).

This agglomeration strategy is meant to exploit the high computing power from modern SIMD processors. However, the 3D tetrahedral mesh and anisotropy of the speed function pose some challenges for this strategy to achieve good performance. First, representing the topology of an unstructured 3D mesh and storing the speed matrices imposes a large memory footprint. In juxtaposition to this, high local memory residency and sufficient computational density are desired to hide the memory access latency. Due to the large memory footprint, the agglomerate size must be small enough so that the limited on-chip fast memory space of the SIMD processor can accommodate all the agglomerate data. However, small agglomerate sizes leads to a larger boundary and more global communication which is slow for SIMD architectures. In addition, unstructured 3D meshes can have large and highly variant vertex valances which result in uneven workload for the threads and an incoherent memory access pattern that affects the achieved bandwidth. To address all these challenges, it is essential to carefully design the data structure

Algorithm 4.1 meshFIM(A,L) (A: set of agglomerates, L: active agglomerate list)

```
comment: initialize the active list L
for all a ∈ A do
  for all v ∈ a do
    if any v ∈ S then
      add a to L
    end if
  end for
end for
comment: iterate until L is empty
while L is not empty do
  for all a ∈ L do
    update the values of the node in each a
  end for
  for all a ∈ L do
    check if a is converged with reduction operation
  end for
  for all a ∈ L do
    if a is converged then
      add neighboring agglomerates of a into a temporary list L_temp
    end if
  end for
  clear active list L
  for all a ∈ L_temp do
    perform 1 internal iteration for a
  end for
  for all a ∈ L_temp do
    check if a is converged with reduction operation
  end for
  for all a ∈ L_temp do
    if a is converged then
      add a into active list L
    end if
  end for
end while
```

used for the agglomeration strategy so that the data structure is compact and regular. We explore here two different data structures for representing tetrahedral agglomeration yielding high computational density for the SIMD processing of tetrahedral meshes on blocks. We call these two representations the *one-ring-strip* and the *cell-assembly* data structures.

4.3.2.3 Description of One-ring-strip Data Structure

The one-ring-strip data structure is efficient only for the case of isotropic speed functions because its run-time effectiveness is offset by the memory footprint of the geometric and speed information in the anisotropic case. We discuss it here as it provides better performance for this very important special case. As in tetFIM, the update for one vertex includes computing solutions from its one-ring tetrahedra and taking the minimum solution as the new updated value. In order to minimize memory usage, we store for each vertex its one-ring tetrahedra by storing the outer-facing triangles on the polyhedron formed by the union of the one-ring tetrahedra. To further improve memory usage, these triangles are stored in "strips" as commonly used in computer graphics [14]. Specifically, for a given vertex within the mesh, the faces of its one-ring tetrahedra that are opposite of the vertex form a triangular surface (see Figure 4.3) from which we generate a triangular strip and store this strip instead of storing the entire one-ring tetrahedra list.

In practice, the one-ring-strip data structure consists of four arrays: **VAL**, **STRIP**, **GEO**, and **SPEED**. **GEO** is the array storing the per-vertex geometry information required to solve the eikonal equation. It is divided into subsegments with a predefined size that is determined by the largest agglomeration among all the agglomerates. Each subsegment stores a set of three floating point variables (floats) for the vertex coordinates of each vertex. **VAL** is the array storing the per-vertex values of the solution of the eikonal equation. It is also divided into subsegments,

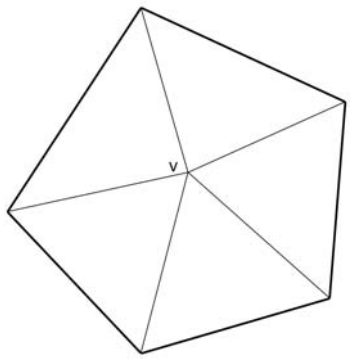


Figure 4.3. 2D representation of the outer surface of vertex v formed by the one-ring tetrahedra: the polygon formed by the bold line segments is analogous to the outer triangular surface in tetrahedral mesh.

and solutions on the vertex are stored. The algorithm requires two **VAL** arrays, one for the input and the other for the output, in order to avoid memory conflicts. Vertices on the boundaries between agglomerates are duplicated so that each agglomerate has access to vertices on neighboring agglomerates, which are treated as fixed boundary conditions for each agglomerate iteration. The **STRIP** array stores both indices to **GEO** and **VAL**, respectively, for the geometric information and the current solution at each vertex within the strip. The **SPEED** array stores per-tetrahedron speed values corresponding to the tetrahedral strip of a vertex. This data structure is not suited for the anisotropic case since the speed matrix requires significant memory. Anisotropic speed functions require that six floating point numbers of the speed matrix be stored for each adjacent tetrahedron of a node, while isotropic speed functions require only one floating point number per adjacent tetrahedron. Figure 4.4 depicts the data structure introduced above. In a single internal iteration on an agglomerate, the one-ring-strip data structure employs a vertex-based parallelism, *i.e.*, each thread in a block is in charge of the update of a vertex which includes computing the potential values from the one-ring tetrahedra of this vertex and then taking the minimum as the final result.

4.3.2.4 Description of Cell-assembly Data Structure

The *cell-assembly* data structure is an extension of the data structure described in [37] for triangular meshes. However, especially for the tetrahedral meshes, we have designed a new data compaction scheme to combine the anisotropic speed matrices with the geometric information. In addition, instead of using a fixed length array **NBH** to store the memory locations for a thread to gather data, we use a more compact data structure to store these locations. Also, we propose a lock-free strategy to generate the gather-lists which are needed in the computation to find the minimum of the potential values of each node. The *cell-assembly* works for both the isotropic and anisotropic cases, although it is slightly less efficient in terms of run-time performance for some isotropic cases than the one-ring-strip data structure.

The cell-assembly data structure includes four arrays, labeled **GEO**, **VAL**, **OFF-**

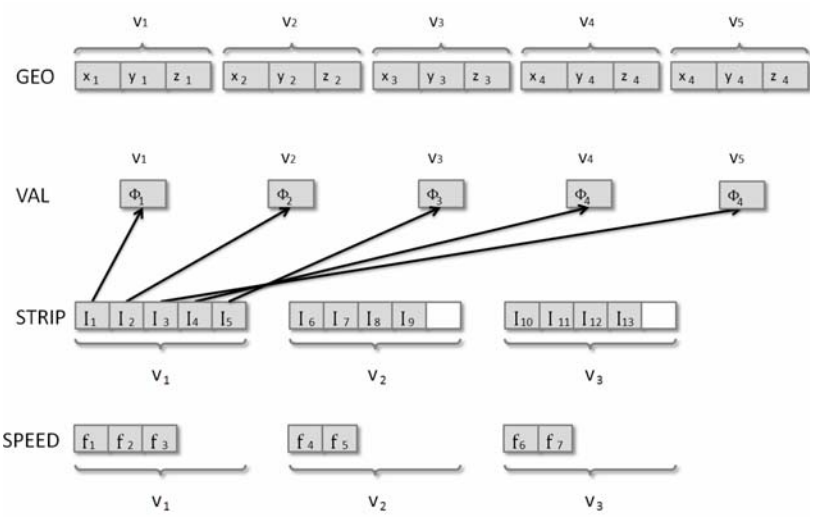


Figure 4.4. One-ring-strip data structure: in this figure, T_i is a tetrahedron, x_i, y_i, and z_i represent the coordinates of the i_{th} vertex, f_i is the inverse of speed on a vertex. Φ_i denotes the value of the solution at the i_{th} vertex. I_i in STRIP represents the data structure for the one-ring-strip of the i_{th} vertex each of which has q indices pointing (shown as arrows) to the value array.

SETS, and **GATHER**. **GEO** stores compacted geometry and speed information, and the compaction scheme is described below. This is different from the cell-assembly for the 2D meshes described in [37] which stores the speed and geometric information separately. **GEO** is also divided into subsegments with a predefined size that is determined by the largest agglomeration. **VAL** stores per-tetrahedron values of solution of the eikonal equation. As with the one-ring-strip, we simply duplicate and store the exterior boundary vertices for each agglomeration and treat the data on those vertices as fixed boundary conditions for each agglomerate iteration to deal with agglomerate boundaries. The **GATHER** array stores concatenated per-vertex gather-lists which are the indices to **VAL** for the per-vertex solution, and the **OFFSETS** array indicates the starting and ending of the gather-list of each node in the **GATHER** array. These gather-lists are stored differently because a tetrahedral mesh may have very various valence, and the fixed length data structure used in [37] may waste a lot of memory space and bandwidth for the sentinel values.

For cell-assembly, the updates of the intermediate (potential) vertex values in an agglomerate employ tetrahedron-based parallelism. Each thread of a block

is responsible for updating all four vertex values of a tetrahedron, and the intermediate results are stored in the **VAL** array. Then we need to find the final value of a node, which is the minimum of its potential values which are stored in the per-tetrahedron **VAL** array. Typically, an atomic minimum operation is then needed to find the minimum for each node in parallel. However, atomic operations are costly on GPUs, and we avoid them by switching to a vertex-based parallelism strategy using gather-lists. A gather-list stores indices to **VAL** and tells the thread where to fetch potential values in the **VAL** array for a node. A gather-then-scatter like operation is then used to find the minimum value of a vertex from its one-ring tetrahedra and reconcile all the values of this vertex according to the gather-lists. Generating the gather-lists efficiently on GPUs is not a trivial task, given only the geometric information of the mesh — the element list and the node coordinate list. We use a sorting strategy to achieve this. Given a copy of the element list **ELE** which stores the vertex indices of each tetrahedron, we create an auxiliary array **AUX** of the same size and fill it with an integer sequence. Specifically, if the size of **ELE** is n, **AUX** is initialized to $\{0, 1, 2, \ldots \ n-1\}$. We sort **ELE** and permute **AUX** according to the sorting. Now **AUX** stores the concatenated gather-lists all the nodes, but we need to know the starting and ending positions of the gather-list of each node, which is achieved by a reduction and a scan operation on the **ELE** array. These operations – sorting, reduction, and scan – are all very efficient on GPUs, and we use the CUDA thrust library [67] in our implementation. Now **ELE** and **AUX** are respectively the **OFFSETS** and **GATHER** arrays we need.

Next, we describe how we combine the speed matrix and geometric information in practice. As shown in Section 4.2.2, the local solver for updating a vertex requires six floats to store the symmetric speed matrix M', so a total of 24 floats are needed to update all four vertices on a tetrahedron. However, based on the topology of the tetrahedron and some algebra reductions, we have:

$$\mathbf{e}_{i,j} = \mathbf{e}_{i,k} + \mathbf{e}_{k,j}, \tag{4.9}$$

$$\mathbf{v_1}^T M \mathbf{v_2} = \mathbf{v_2}^T M \mathbf{v_1} \text{ and} \tag{4.10}$$

$$\mathbf{v_1}^T M \mathbf{v_2} + \mathbf{v_1}^T M \mathbf{v_3} = \mathbf{v_1}^T M (\mathbf{v_1} + \mathbf{v_2}) \tag{4.11}$$

where $\mathbf{v_1}$, $\mathbf{v_2}$, and $\mathbf{v_3}$ are arbitrary vectors. According to these properties, we can calculate all the four M' elements from the six values: $\mathbf{e}_{1,3}^T M' \mathbf{e}_{2,3}$, $\mathbf{e}_{2,3}^T M' \mathbf{e}_{3,4}$, $\mathbf{e}_{1,3}^T M' \mathbf{e}_{3,4}$, $\mathbf{e}_{1,4}^T M' \mathbf{e}_{2,4}$, $\mathbf{e}_{1,3}^T M' \mathbf{e}_{1,4}$, and $\mathbf{e}_{2,3}^T M' \mathbf{e}_{2,4}$. Precomputing these values, we need only store six floats for each tetrahedron which are stored in the **GEO** array.

Compared to the one-ring-strip data structure, the advantage of cell-assembly is that the computational work is almost the same for each SIMD thread independent of the valances of the vertices, while for one-ring-strip, the computational work per thread is determined by the valences of the vertices. More homogeneity in the valances of the vertices results in better load balancing for the different threads. However, the one-ring-strip data structure has a smaller memory footprint and higher computation density since each SIMD thread computes the local solver on each tetrahedron of a one-ring-strip. We evaluate the performance each data structure empirically in the next section.

4.4 Results and Discussion

In this section, we discuss the performance of the proposed algorithms in realistic settings compared to two widely-used competing methods: the fast marching method (FMM) and the fast sweeping method (FSM). Serial CPU implementations were generated which strictly follow the algorithms as articulated in the (previously) cited references. We rely on a collection of unstructured meshes having variable complexities to illustrate the performance of each method. For this set of meshes, we examine how the performance of these methods is affected by four different speed functions—a homogeneous isotropic speed, a homogeneous anisotropic speed, a heterogeneous anisotropic random speed, and a speed function for the geometric optics/lens example. We first show the error analysis of the proposed first-order numerical scheme. Next, we show the results of the single-threaded (serial) CPU implementation of tetFIM, FMM, and FSM, and review the typical performance characteristics of the existing algorithms. We then detail the results of our multithreaded CPU implementation and discuss the scalability of the proposed algorithm on shared memory multiprocessor computer systems. Finally, we present the results of our GPU implementation to demonstrate the performance

of the proposed method on massively SIMD streaming parallel architectures. For consistency of evaluation, single precision was used in all algorithms and for all experiments presented herein.

The meshes and speed functions for the experiments in this section [1] are as follows:

Mesh 1: A regularly tetrahedralized cube with 1,500,282 tetrahedra ($63\times63\times63$ regular grid) whose maximum valence is 24;

Mesh 2: An irregularly tetrahedralized cube with 197,561 vertices and 1,122,304 tetrahedra whose maximum valence is 54;

Mesh 3: A heart model with 437,355 vertices and 2,306,717 tetrahedra whose maximum valence is 68 (Figure 1.1-left);

Mesh 4: A lens model with 260,908 vertices and 1,561,642 tetrahedra whose maximum valence is 58 (Figure 1.1-right);

Mesh 5: A 3-D model with irregular geometries, which we call *blobs*, with 437,355 vertices and 2,306,717 tetrahedra whose maximum valence is 88 (Figure 4.5).

Speed 1: A homogeneous isotropic speed of constant 1.0,

Speed 2: A homogeneous anisotropic diagonal speed tensor with diagonal entries 1.0, 4.0, and 9.0,

Speed 3: A heterogeneous anisotropic correlated random symmetric positive-definite speed tensor,

Speed 4: A heterogeneous isotropic speed for lens model, and

Speed 5: A heterogeneous isotropic speed for lava lamp model.

4.4.1 Error Analysis

To show that the proposed algorithm achieves the first-order accuracy we would expect from the piecewise linear approximation used within the solver, we performed a convergence analysis on a problem with a known solution. We use six regularly tetrahedralized cube meshes, representing a 256×256×256 block within $\mathbb{R}^3$, with the number of vertices on each side ranging from 17 to 513. We use an ellipse octant (placing the center of the ellipse at the corner of the cube domain)

[1] Files containing the mesh and speed function definitions can be found at: http://www.sci.utah.edu/people/zhisong.html

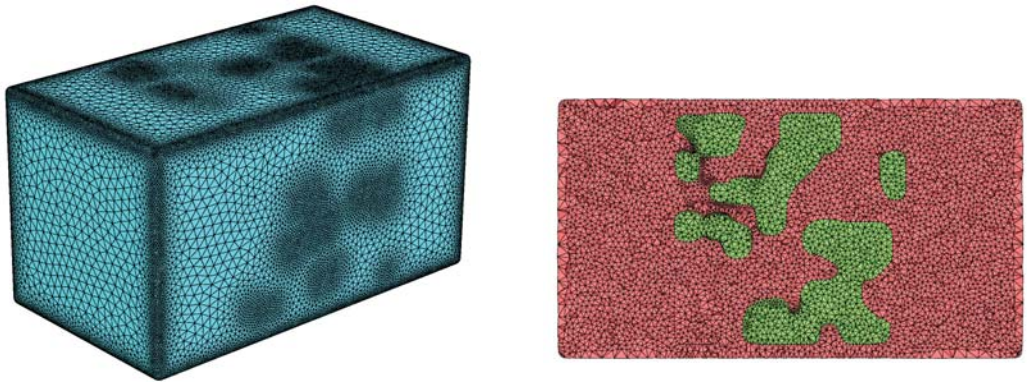


Figure 4.5. *Blobs* mesh and its cross section. The different colors in the cross section represent different materials indices of refraction (speed functions).

of the form $x^2 + 4y^2 + 9z^2 = R^2$, where $R = 40$ as the source. Boundary conditions were projected onto the vertices using the nearest vertices to the sphere. We then solve for the distances to these boundaries for the entire domain using the tetFIM eikonal solver with an anisotropic diagonal speed matrix with diagonal numbers 1, 4, and 9 and compare against analytical results at the vertices using the L_1 error. L_1 errors are computed in this way. First, for each tetrahedron, take the average of the errors at the vertices and multiply by the volume of the tetrahedron. We then sum up the products over all tetrahedra and divide the sum by the volume of the whole domain. Finally, we calculate the error orders of any two consecutive meshes. The results are presented in Table 4.1. The table shows that the order of the error is approaching 1.0 with increasing resolution, which is consistent to our claim that tetFIM is asymptotically first-order accurate.

4.4.2 CPU Implementation Results and Performance Comparison

We have tested our CPU implementation on a Windows 7 PC equipped with an Intel i7 965 Extreme CPU running at 3.2 GHz. All codes were compiled with Visual Studio 2010 using compiler options /O2 and /arch:SSE2 to enable SIMD instructions. (we accomplished a comparison using the Intel Sandy Bridge CPU to run some of the tests. The results show the Sandy bridge CPU is around twice as fast as the i7 965. All results presented herein can be scaled appropriately to interpret the results against the Sandy Bridge processor). First, we focus on the performance the CPU

Table 4.1. Table presenting our convergence results (L_1 error) and the order of convergence as computed from subsequent levels of refinement.

	Speed 1		Speed 2	
Mesh sizes	L_1 Error	Order	L_1 Error	Order
17	8.073934	—	15.399447	—
33	4.688324	0.78	9.232588	0.74
65	2.606537	0.85	5.347424	0.79
129	1.396091	0.90	2.967363	0.85
257	0.721630	0.95	1.558972	0.93
513	0.362584	0.99	0.789725	0.98

implementations of our tetFIM method compared against serial FMM and FSM on three different meshes with differing complexities (Mesh 1, Mesh 2, and Mesh 3) using various speed functions. The anisotropic version of FMM [90] is no longer local in nature (as it requires a larger multi-element upwind stencil) and hence, we did not include anisotropic FMM in our comparisons. We call the serial version of our method CPU method tetFIM-ST and the multithreaded version tetFIM-MT (in all cases, we use four threads). In all these experiments, a single source point is selected at around the center of the cube. For the FSM, we select the reference points to be the eight corners of the cube and the run-time for FSM does not include the sorting time required to sort vertices according to their Euclidean distances to the reference points. Tables 4.2, 4.3, and 4.4 show the computational results for this set of experiments.

As shown in Tables 4.2, 4.3, and 4.4, FMM outperforms both tetFIM and FSM on all isotropic cases. This is to be expected as FMM is a worst-case optimal method

Table 4.2. Run-time (in seconds) of FMM, FSM, single-threaded tetFIM (tetFIM-ST), and multithreaded tetFIM with four threads (tetFIM-MT) on Meshes 1 with Speeds 1, 2, and 3.

	Speed 1	Speed 2	Speed 3
FMM	69	—	—
FSM	213	216	680
tetFIM-ST	80	81	107
tetFIM-MT	27	28	41

Table 4.3. Run-time (in seconds) of FMM, FSM, tetFIM-ST, and tetFIM-MT on Mesh 2 with Speeds 1, 2, and 3.

	Speed 1	Speed 2	Speed 3
FMM	42	—	—
FSM	407	409	674
tetFIM-ST	60	59	175
tetFIM-MT	22	23	55

Table 4.4. Run-time (in seconds) of FMM, FSM, tetFIM-ST, and tetFIM-MT on Mesh 3 with Speeds 1, 2, and 3.

	Speed 1	Speed 2	Speed 3
FMM	71	—	—
FSM	807	823	1307
tetFIM-ST	113	122	173
tetFIM-MT	46	48	56

and its performance is not significantly impacted by the complexity of the mesh or the speed function, as observed previously in [52] and [37]. FIM outperforms the FSM on all the test cases. For simpler speed functions like Speeds 1 and 2, the FSM requires only two iterations to converge, because the characteristics are well captured thanks to the reference point choice. FSM, however, requires the update of all the vertices in the mesh according to their distance to each reference point in both ascending order and descending order. So for the eight reference points in these experiments, FSM needs to update all vertices 16 times in one iteration, which amount to 32 total updates for each vertex. On the other hand, tetFIM needs less updates for the mesh vertices when the wavefront passes through the whole domain from the source in the direction of the characteristics. Indeed, the average valance of the mesh is 24, and assuming that half of the neighbors of a vertex are fixed when a vertex is being updated, each vertex needs to be updated only 12 times on average. As pointed out in [51], when the speed function becomes more complex (*i.e.*, characteristics change frequently), FSM performs even worse when compared to FIM, which can be shown in our Speed 3 case where FSM needs six iterations to converge and tetFIM runs about seven times faster. Moving to

the more complex Mesh 2, FSM's performance is dramatically degraded, needing five iterations for simpler Speeds 1 and 2 and eight iterations for Speed 3. The tetFIM's performance, however, is inconsequentially impacted by the complexity of the mesh.

The tetFIM algorithm is designed for parallelism, and the results on the multithreaded system bear this out. The fourth rows in Tables 4.2, 4.3, and 4.4 show the run-times of multithreaded tetFIM using four CPU cores. Note that tetFIM scales well on multicore systems. On a quad-core processor, we observe a nearly three times speedup from tetFIM-ST to tetFIM-MT on all cases. The reduction from perfect scaling can be attributed to the fact that due to the partitioning of the active list at each time step, the multithreaded version accomplishes more updates per vertex than the serial version. In the single-threaded version, a single active list implies that updated information is available immediately once a computation is done, analogous to a Gauss-Seidel iteration; in the multithreaded case, the active list partitioning enforces a synchronization in terms of exchange of information between threads, analogous to a red-black Gauss-Seidel iteration.

4.4.3 GPU Implementation Results

To demonstrate the performance of tetFIM on SIMD parallel architectures, we have implemented and tested tetFIM-A on an NVIDIA Fermi GPU using the NVIDIA CUDA API [68]. The NVIDIA GeForce GTX 580 graphics card is has 1.5 GBytes of global memory and 16 microprocessors, where each microprocessor consists of 32 SIMD computing cores that run at 1.544 GHz. Each computing core has configurable 16 or 48 KBytes of on-chip shared memory, for quick access to local data. Computation on the GPU entails running a kernel with a batch process of a large group of fixed size thread blocks, which maps well to the tetFIM-A algorithm that employs agglomeration-based update methods. A single agglomerate is assigned to a CUDA thread block. For the one-ring-strip data structure, each vertex in the agglomerate is assigned to a single thread in the block, while in cell-assembly data structures, each tetrahedron is assigned to a thread. These two variants of the tetFIM-A algorithm are called tetFIM-A-ORS and tetFIM-A-CA, respectively.

The agglomerate scheme seeks to place the agglomerated data into the GPU

cache (registers and shared memory). However, the GPU cache size is very limited, and hence, we have to use agglomerates with smaller diameters compared to what can be used in triangular mesh cases. This implies that we perform fewer internal iterations in the 3D case versus the 2D case, which leads to lower computational density. On the other hand, performing fewer internal iterations reduces the number of redundant internal iterations caused by outdated boundary information. In addition, the local solver for tetrahedral mesh requires more computations. Table 4.5 demonstrates that our agglomerate scheme balances the trade-off between the agglomerate size, the number of internal iterations and computational density very well on the GPU; the speedup values increase in 3D over previously published 2D results [37]. In addition, our GPU implementations perform much better than all the CPU implementations. Section 4.4.5 provides detailed analysis of the parameter choice.

Table 4.5 also shows the performance comparison of the two tetFIM-A variants, tetFIM-A-ORS, and tetFIM-A-CA with the single-threaded CPU implementation (tetFIM) on the same meshes and the isotropic speed function, and shows the speedup factors of tetFIM-A over the CPU method. Communication times between CPU and GPU, which are only about one tenth of the run-times in our experiments, are not included for tetFIM-A to give a more accurate comparison of the methods. As shown in this result, tetFIM-A-ORS performs better than tetFIM-A-CA for Mesh 1, which is a regularly tetrahedralized cube. This is because one-ring-strip data structure consumes less shared memory so as to allow larger agglomerates. Large agglomerates need more inside iterations to converge; hence, the computational density is increased due to fast shared memory usage for inside iterations. While for the more complex irregular meshes like Mesh 3 in this comparison, tetFIM-A-CA has a performance advantage. The reason is that for irregular meshes, the valence of the vertices vary greatly; hence, the computational density of tetFIM-A-ORS for each thread is sufficiently unbalanced that computing power is wasted when faster threads are waiting for the slower ones to finish. On the other hand, the two tetFIM-A algorithms achieve a good performance gain over both the serial and multithreaded CPU solvers. On a simple case such as Mesh 1 with Speed

1, tetFIM-A-ORS runs about 201 times faster than tetFIM-ST while tetFIM-A-CA runs about 131 times faster than tetFIM-ST. On the other more complex cases, tetFIM-A-ORS runs up 23 times faster than tetFIM-ST while tetFIM-A-CA is 37 times faster. See Figure 4.6 for visualizations of the resulting solutions.

We also observe that SIMD efficiency of the tetFIM algorithm depends on the input mesh configuration (*i.e.*, the average vertex valence relative to the highest valence). As seen from Table 4.5, both GPU implementations achieve the highest speedups on Mesh 1 compared to the CPU implementation while achieving the lowest speedups on Mesh 3 which has much greater maximum vertex valance. This is because the highly unstructured mesh, *e.g.*, Mesh 3, leads to unbalanced word load and waste of memory bandwidth on SIMD architectures.

Next, we show the tetFIM-A applied to the anisotropic cases. Because the one-ring-strip data structure is not suitable for this case, we include only the performance result of cell-assembly data structure variant tetFIM-A-CA. Table 4.6 clearly shows that the tetFIM-A which is implemented on the GPU performs much better than the CPU implementation on all the examples we experimented,

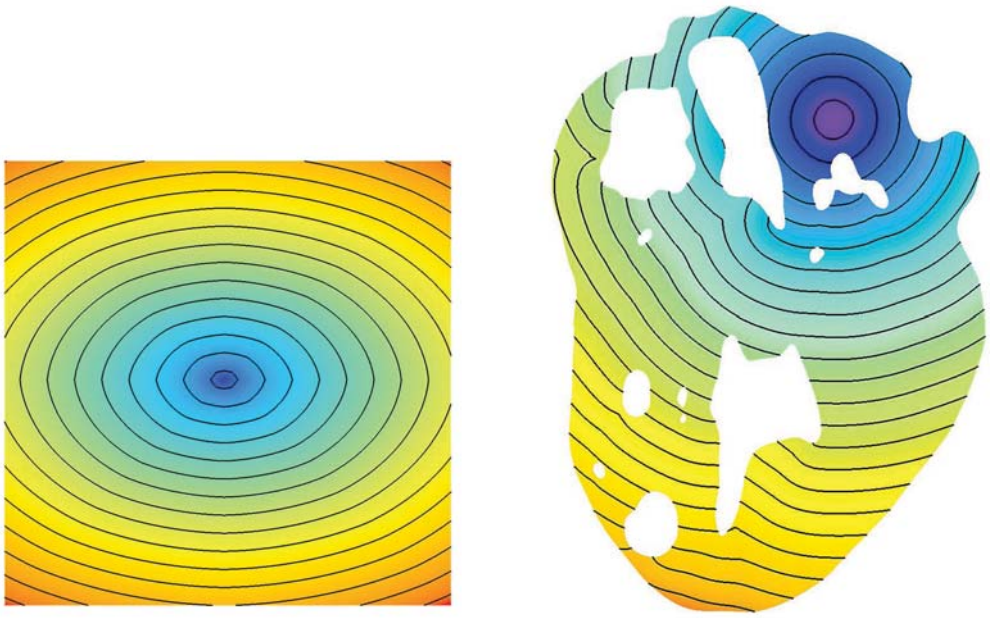


Figure 4.6. Color maps and level curves of the solutions on the cube and heart meshes. Left: the ellipse speed function (Speed 2). Right: the isotropic constant function (Speed 1).

Table 4.5. Run-times (in seconds) and speed-up factors (against tetFIM-ST) for the different algorithms and architectures on all meshes with Speed 1. Data in first row are from Tables 4.2, 4.3, and 4.4.

	Mesh 1	Mesh 2	Mesh 3
tetFIM-ST	80	60	113
tetFIM-MT	27	22	46
tetFIM-A-ORS	0.396	1.412	2.694
tetFIM-A-CA	0.587	0.939	1.911
Speedup 1	202×	42×	42×
Speedup 2	136×	64×	59×

Table 4.6. Run-times (in seconds) and speed-up factors for the different algorithms and architectures. Data in first row are from Tables 4.2, 4.3, and 4.4.

	Mesh 1 Speed 2	Mesh 2 Speed 2	Mesh 3 Speed 2	Mesh 1 Speed 3	Mesh 2 Speed 3	Mesh 3 Speed 3
tetFIM-ST	81	59	113	107	175	173
tetFIM-A-CA	0.580	0.958	1.986	1.356	2.079	2.413
Speedup	140×	62×	57×	79×	84×	72×

regardless of the mesh configuration and speed function.

Finally, Table 4.7 shows the preprocessing time for Meshes 1, 2, and 3. The preprocessing is performed on the GPU and includes permuting the geometric information (element list and vertex coordinate list) according to the mesh partition using METIS and generating the gather-lists for the cell-assembly data structure. The graph partitioning and triangle strip generation time are not included since they are not essential parts of our algorithm.

4.4.4 Meshes for Complex Surfaces

We have also tested this method on meshes with more complex conformal surfaces (Meshes 4 and 5) to show that the proposed method works correctly

Table 4.7. Run-times (in seconds) of the preprocessing step for Mesh 1, 2, and 3.

Mesh 1	Mesh 2	Mesh 3
0.150	0.120	0.209

when applied to scenarios that resemble physical simulation associated with target applications. Figures 4.7 and 4.8 show the results of the simulation on the lens model and the *blobs* model. The green region in the lens model (Figure 1.1-right) has a speed function of 1.0, which represents the refractive index of air, and the red region models a lens with refractive index of 2.419. Similarly, in the *blobs* model, the red and green regions have constant speed functions of 1.0 and 10.0, respectively. Table 4.8 shows the performance of all the methods for Meshes 4 and 5.

4.4.5 Analysis of Results

In this section, we discuss the analysis of our results in terms of asymptotic cost and parameter optimization choices.

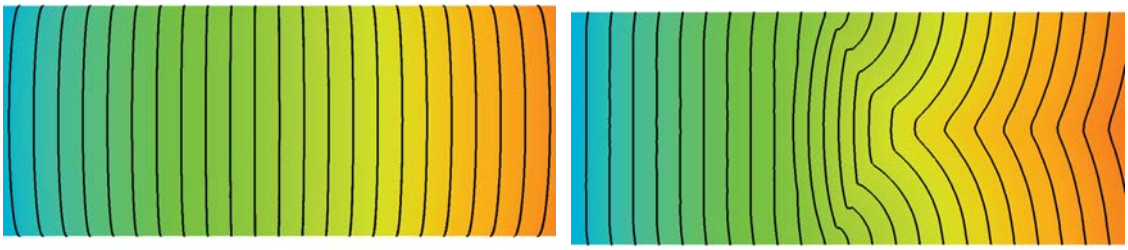


Figure 4.7. Color maps and level curves of the solutions on the lens model with boundary as given by the figure in the left-side image.

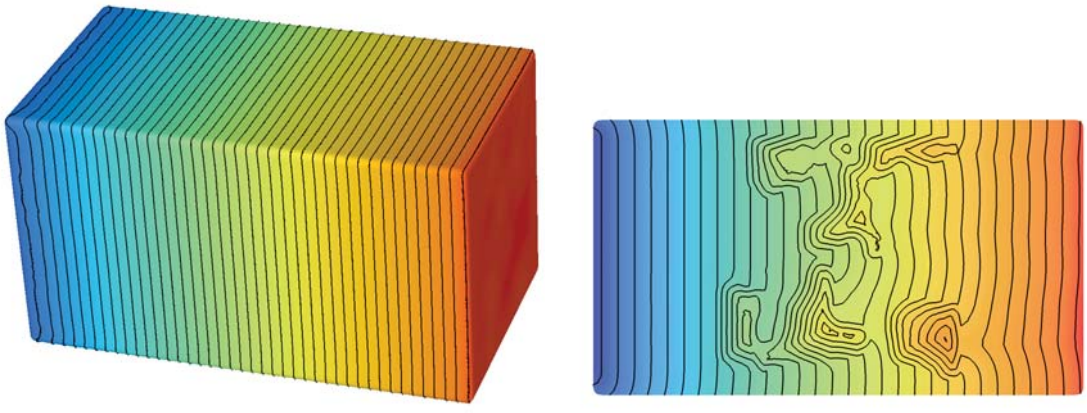


Figure 4.8. Color maps and level curves of the solutions on the *blobs* model with boundary as given by the figure in the left-side image.

Table 4.8. Run-time (in seconds) of all methods on Meshes 4 and 5. The "Speedup VS. FMM" column lists the speedup of all methods compared to FMM with negative numbers denoting that the method is slower than FMM.

	Mesh 4	Speedup VS. FMM	Mesh 5	Speed-up VS. FMM
FMM	43	1	51	1
FSM	378	-8.8	517	-10.1
tetFIM-ST	74	-1.7	62	-1.2
tetFIM-MT	22	2.0	21	2.4
tetFIM-A-ORS	2.372	18.1	2.032	25.1
tetFIM-A-CA	1.801	23.9	1.538	33.2

4.4.5.1 Asymptotic Cost Analysis

We accomplished an asymptotic cost analysis that measures the number of iterations and number of updates per vertex for our proposed serial CPU version tetFIM-ST and GPU version tetFIM-A. We used four meshes with different sizes to show that our method scales very well against mesh size for a given speed function (see Table 4.9).

4.4.5.2 Parameter Optimization

In tetFIM-A, there are two parameters that need to be specified: the agglomerate size and the internal iteration number. The agglomeration scheme provides fine-grained parallelism that is suitable for SIMD architectures by partitioning the mesh into agglomerates that are mapped to different computational blocks. During the internal iterations on the agglomerate accomplished per block, the boundary

Table 4.9. Asymptotic cost analysis: # iter is the number of iterations needed to converge and # up is the average number of updates per vertex.

	tetFIM-ST				tetFIM-A			
	Speed 2		Speed 3		Speed 2		Speed 3	
Mesh sizes	# iter	# up	# iter	# up	# iter	# up	# iter	# up
17	37	11	44	13	48	29	69	51
33	70	12	81	15	103	29	119	49
65	139	12	170	16	206	32	265	51
129	276	11	326	15	403	31	510	50

conditions are lagged. Hence, taking an excessive number of internal iteration is wasteful as it merely drives the local solution to an incorrect fixed-point (in the absence of boundary condition updates). For this reason, it may seem ideal to have smaller agglomerate sizes which tend to need fewer internal iterations for the agglomerate to converge (and thus less computation is wasted). However, smaller agglomerates result in a large boundary and more global communication among blocks. In addition, we need also take into account the size of the limited hardware resources, *e.g.*, GPU shared memory and registers. We want to fit the agglomerate into the fast on-chip (shared) memory space to increase the computational density. Based upon our experiments, the best agglomerate size is around 64 vertices. For the internal iteration number, our experiments show that the ideal number is approximately three when agglomerates are of this size.

4.4.6 Conclusions

In this chapter, we have presented a variant of the *fast iterative method* appropriate for solving the inhomogeneous anisotropic eikonal equation over fully unstructured tetrahedral meshes. Two building blocks are required for such an extension: the design and implementation of a local solver appropriate for tetrahedra with anisotropic speed information, and algorithmic extensions that allow for rapid updating of the active list used within the FIM method in the presence of the increased data footprint generated when attempting to solve PDEs on three-dimensional domains. After describing these two building blocks, we make the following computational contributions. First, we introduce our tetFIM algorithms for both single processor and shared memory parallel processors and perform a careful empirical analysis by comparing them to two widely-used CPU-based methods, the state-of-the-art fast marching method (FMM) and the fast sweeping method (FSM), in order to understand the benefits and limitations of each method. Second, we propose an agglomeration-based tetFIM solver, specifically for more efficient implementation of the proposed method on massively parallel SIMD architectures. We then described the detailed data structures and algorithms, present the experimental results of the agglomeration-based tetFIM, and compare

them to the results of the CPU-based methods to illustrate how well the proposed method scales on state-of-the-art SIMD architectures. In comparison to [37], we have demonstrated that careful management of data allows us to maintain high computational density on streaming SIMD architectures – yielding significantly greater speedup factors than seen when solving two-dimensional eikonal problems on GPUs.

In future work, we envisage extending this technique to time-dependent Hamilton-Jacobi problems in 2D and 3D. Specifically, we will seek to address how one might solve the level-set equations over unstructured meshes on current streaming GPU hardware.

CHAPTER 5

ARCHITECTING THE FINITE ELEMENT METHOD PIPELINE FOR THE GPU

5.1 Introduction

The finite element method (FEM) is a numerical technique for finding approximate solutions of partial differential equations (PDEs). FEM naturally handles complex geometries through the use of unstructured meshes and because of this and other provable numerical properties, FEM is widely used for the simulation of physical phenomena in many disciplines such as continuum mechanics, fluid dynamics, and biophyisics. In general, the FEM is implemented as a pipeline consisting of three computationally intensive tasks: computation of the elemental local operators, assembly of the local operators into a system of linear equations for the global unknown degrees of freedom, and solving of the system of equations [49, 54]. In this chapter, we refer to these tasks as the element computation step, the assembly step, and the linear solve step, respectively. The element computation step is application dependent and, in general, embarrassingly parallel. Correspondingly, this step will be mentioned but not highlighted in this chapter. The other two steps, however, require careful consideration when attempting to optimize their corresponding algorithms for parallel architectures. The assembly step uses the mesh topology information to gather information from multiple elements to form the FEM linear system representing the relationship between the global degrees of freedom. This system is then solved using computational linear algebra techniques that are appropriate for the type of the matrix formed.

In many of FEM applications, the FEM method is part of a much larger scientific or engineering undertaking. In many cases, the FEM solve is done multiple times on very large datasets in order to explore parameters spaces, fit measured data, or solve an inverse problem. One way to accelerate the FEM pipeline is by exploiting ad-

vances in modern computational hardware. In recent years, the rapid advancement of many-core processors, and in particular graphical processing units (GPUs), has sparked a broad interest in porting numerical methods to these architectures, thanks to their low cost and very high computing capacity. With appropriate numerical algorithms, modern GPUs demonstrate very strong computational performance comparable to supercomputers of just a few years ago.

The single instruction multiple thread (SIMT) architecture used in GPUs places particular constraints on both the design and implementation of algorithms and data structures, making the porting of existing numerical strategies often difficult, inefficient, or even impossible. The architecture provides a large number of parallel computing units (up to several hundred cores) with a hierarchical data-sharing structure. For example, current NVIDIA GPUs are composed of up to 16 streaming multiprocessors (SMs) each containing a number of streaming processor cores (SPs) and on-chip memory. All SMs have access to global memory, the off-chip memory (DRAM), which has a high latency of several hundred clock cycles. The on-chip memory of each SM includes a space partitioned into registers for individual threads, shared memory which can be accessed by multiple threads and general data cache which is not user controllable. The on-chip memory has very low latencies of only 20-30 clock cycles [68]. These architectural features place important restrictions on algorithms if one wants them to run efficiently on such hardware. Addressing these constraints in the context of the finite element method is one important aspect of this chapter.

Another reason for the increasing popularity of GPU computing is the emergence of consistent, relatively simple GPU computing models, such as the Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL), and associated APIs compatible with several general purpose programing languages. In this chapter, we use CUDA extensions to C for our GPU implementation. In CUDA, a CPU program instantiates a collection of *kernels*, each of which runs as a SIMT computation that is executed in parallel. Kernels are organized into *blocks*, and each block of threads in the grid is executed on a single streaming multiprocessor on the GPU. Threads in the same block may

communicate via *shared memory* and synchronization primitives, with low latency. Alternatively, threads between blocks must communicate via *global memory*, which has high latency. When sequentially numbered threads access sequential data in global memory, the memory access of up to 128 bytes may be performed as a single transaction, a process referred to as *coalescing*. Since global memory accesses have high latency, global memory coalescing is important for performance optimization if the kernel is memory bound. Access to shared memory is banked, and if two threads executing the same instruction attempt to access different words of data from the same bank, a conflict will occur and the accesses must be performed sequentially in conflict-free subsets. In summary, the most optimized kernels minimize global memory transactions, avoid shared memory bank conflicts, and minimize register and shared memory usage to fully occupy the arithmetic logic and floating point units.

For experimental results in this chapter, we use a standardized prototypical problem—the elliptic Helmholtz equation solved over a nontrivial domain—to demonstrate the algorithmic and data structure modifications that must be made in order to gain efficiency of the FEM pipeline on the GPU. In particular, we focus our attention on the two nontrivial tasks: the global assembly step and the global linear solve step. Because the local matrices are already formed in the element computation step, the global assembly step usually includes first allocating and initializing a memory space for the global matrix, then finding the location in the global matrix for each local matrix value and finally assembling (summing) these values to the location in the global matrix. A number of strategies [64, 30, 31, 32, 24] have been proposed to port this step to the GPU (*e.g.*, graph coloring and reduction lists) in a way that one gains the benefits of fine-grain parallelism. However, these strategies need significant preprocessing that does not easily port to the GPU. We propose an alternative method that minimizes the preprocessing and at the same time achieves great performance on GPU.

For solving the global linear system that comes as a consequence of FEM assembly, numerous methods have been proposed in the literature. The most popular group of methods within the FEM community are the (iterative) Krylov

subspace methods such as the conjugate gradient method [99, 27]. The number of iterations of the method is bounded by the rank of the matrix; the particular convergence rate with respect to a given linear system is determined by the eigenspace structure of the operator (often expressed in terms of the condition number of the matrix). Thus, a preconditioner that improves the structure of the eigenspace often helps accelerate the convergence rate of these methods. The global linear system that we seek to solve is both symmetric and positive definite. Considering this and the need for a preconditioning method that maps effectively to the GPU, we propose a solver that used the conjugate gradient method (CGM), preconditioned with a geometry-informed, algebraic multigrid (AMG) method.

In this chapter, we present the algorithms and data structures necessary to execute on the GPU the full FEM pipeline as a PDE solver over unstructured tessellations. Our proposed GPU global assembly step requires very little pre-processing and shows a significant performance boost compared to an optimized CPU implementation. For the solving of the global linear system, we propose a geometry-informed algebraic multigrid method and present novel fine-grained parallelism strategies and corresponding data structures to suit GPU architecture. GPU-based MG methods typically use the Jacobi or polynomial methods for the relaxation as these are based on easily parallelizable sparse matrix vector multiplication (SPMV) [11, 44]. However, these methods do not make full use of GPU computing power, because SPMV is generally a memory-bound operation with low computational density. In this chapter, we propose a relaxation method that operates on a novel data structure and has higher computational density and demonstrates better performance. We also analyze the performance of our strategy and data structures in different problem scenarios, compared against state-of-the-art GPU and CPU linear solvers. In our AMG method, the set-up stage needs extra work compared to typical AMG implementations so its performance is slightly worse than the setup of other state-of-the-art GPU implementations, but our solving stage is significantly faster. This makes our method particularly suitable for some applications, such as in bidomain problems [78], where the mesh is fixed and the linear system solving needs to be performed many times or for

ill-conditioned problems where linear solving takes a long time compared to the assembly and AMG set-up.

The remainder of this chapter is organized as follows. In Section 5.2, we describe the related previous work from the literature. In Section 5.3, we introduce the problem definition that we have selected as the canonical problem for this work, and will present the basics of the finite element method discretization methodology. In Section 5.4, we present our GPU-based computing strategy for the FEM assembly step. In Section 5.5, we present the details of how we solve the global linear system on the GPU – namely, we present our GPU-focused mesh-informed algebraic multigrid method used to precondition a conjugate gradient linear system solver. In Section 5.6, we show numerical results related to several different engineering scenarios. We analyze different GPU implementation strategies and data structures and explain the optimizations that were required to achieve performance under the austere constraints of the GPU. For completeness, we compare our performance against other alternative GPU and CPU linear solvers. In Section 5.7, we summarize the chapter and discuss future research directions related to this work.

5.2 Previous Work

In the past decade, there have been a multitude of studies that have the explicit goal of porting part or all of the finite element pipeline to many-core architectures. In our review, we will focus on the two compute-intense and challenging components of the pipeline: the global linear system *assembly* step and the global system *solve* step.

For the assembly step, early works [15, 83] present relatively simple assembly strategies designed in light of their specific applications. They compute in parallel each nonzero value in the global linear system independently, which suits many-core architectures very well. However, these methods are based on special characteristics of their applications which allow them to derive simple expressions for the nonzero values not available for use in the general FEM context.

Some more general, but more complicated, GPU assembly strategies have recently been proposed. For instance, [59, 60] employ graph coloring to partition

elements into nonoverlapping sets so that all elemental matrices of one set can be accumulated to the global matrix in parallel without conflicts. Similarly, graph partitioning and reduction list strategies are proposed in [24] to optimize the assembly performance on GPU. These strategies, however, need significant preprocessing such as the generation of a graph coloring, graph partitioning, and/or a reduction list based upon the graph induced by the mesh being used. Information derived from this preprocessing is used in the generation of the data-structured used on the GPU. Many of these preprocessing steps in and of themselves are not easily parallelizable; in addition, their serial implementations take significant running time.

Recently, Markall *et al.* [64] compare several different assembly strategies on different architectures; they propose a local matrix approach for their assembly and demonstrate that this approach is efficient on many-core architectures for 2D meshes. Their method stores all the local matrices of the elements in a large block matrix instead of storing an assembled global matrix. The matrix vector multiplication is performed in three stages: a spreading operation, a local matrix vector-multiplication, and a gather operation as done in high-order finite element methods [104]. The local matrices typically have the same size and use the same data structure for their storage, so the local matrix vector-multiplication has a regular memory access pattern amenable to GPUs. In addition, this method requires very little preprocessing to accomplish the assembly operation. The authors in [58] introduce a similar approach for GPU-based FEM which computes the local matrices on the fly. The local matrices, however, need much more memory space than the fully assembled global matrix, especially for 3D meshes. Our experiment shows the matrix operations using this approach perform worse than using assembled global matrix in 3D meshes, consistent with the CPU study in [22]. Some recent studies [31, 32], conducted in parallel to this chapter, propose to assembly the global matrix into a Coordinate list (COO) format and then convert the matrix to compressed sparse row (CSR) format by removing duplicate nonzero entries. We propose an agglomeration strategy for the assembly step. The proposed strategy decreases the memory footprint by removing data duplication which,

when combined with a novel compact sparse matrix data structure, enables the method to avoid the preprocessing used by others, which rely on search operations and atomic addition operations in the fast on-chip memory.

The linear system of equations that comes from the use of the finite element methodology is often sparse, symmetric, and positive definite [49]. Consequently, Krylov subspace methods such as the conjugate gradient method are amongst the most widely used numerical linear algebra techniques used with FEM analysis. In practice, the conjugate gradient methods are almost always preconditioned to help improve their convergence rate [99, 27]. The simplest preconditioner is the diagonal preconditioner which is very simple to apply but is usually of marginal benefit, because it takes as the approximate inverse merely the inverse of the diagonal of the original matrix.

Incomplete LU factorization (ILU) is a widely used preconditioning method which computes a sparse lower triangular matrix L and sparse upper triangular matrix U such that $A = LU + R$. When the system satisfies certain conditions, the matrix $M = LU$ can be used as an effective preconditioner for conjugate gradient [85]. ILU, however, depends on triangular solves which are sequential in nature and hence particularly difficult to parallelize/optimize for large sparse matrices because of the fill-in of nonzero elements. Thus, ILU preconditioning is not particularly well-suited to GPUs [62]. Another popular preconditioner is the block Jacobi preconditioner, which is easy to parallelize and implement on GPU. In the block Jacobi preconditioner, one partitions the domain into blocks on which one does Jacobi iterations independent of the other blocks with some timed synchronization strategy. The problem with this kind of preconditioner is that it usually requires a large number of iterations to be effective (*e.g.*, converge), so the benefits of improved parallelism may be outweighed by the increased work in iterations [62]. We have elected to use the a variant of the multigrid method [65, 17] as the preconditioner for our conjugate gradient solver. The multigrid method is a widely used preconditioner and has been shown to be very effective on systems resulting from FEM. Multigrid methods, by employing grids of different mesh sizes (levels), provide rapid convergence rates by reducing

low frequency error through coarse grid correction and removing high frequency error via fine grid smoothing. Research has shown that multigrid methods scale very well when applied to parallel computing and are very fast for many practical problems [15, 44, 45, 11, 103, 6, 8, 102].

Recently, some effort has been made to port the preconditioned Krylov subspace method with multigrid preconditioner to many-core architectures. Representative works include [44] and [11]. In [44], the authors present a GPU implementation of a preconditioned conjugate gradient method with a multigrid preconditioner. They use an algebraic multigrid similar to boomerAMG [45] and the interleaved compressed sparse row (ICSR) data structure for sparse matrix storage in an attempt to coalesce the global memory accesses. As pointed out in [12], however, ICSR (the same as the Ellpack data structure described in [12]) is not suitable for unstructured meshes where their nodes have highly variable valance. The authors of [11] also presents a parallel algebraic multigrid method which exposes substantial fine-grained parallelism in both the construction of the multigrid hierarchy as well as the cycling or solve stage. In both works, the Jacobi method is used in the most expensive multigrid step, the relaxation at each resolution. This method is easy to parallelize but is not very effective as the relaxation step [8]. Additionally, the Jacobi method depends on the sparse matrix-vector multiplication operation, which has low computational density and is generally memory bandwidth bounded. In [41, 28, 29], the authors introduce GPU-based linear solvers with multigrid methods. The solvers use the ELLpack sparse matrix data structure for their specific problems, which is not efficient when number of nonzero entries per row varies largely. Their proposed approach also rely on sparse matrix-vector multiplication which has low computational density as previously mentioned. A recent work [105] proposes to use a auxiliary grid to construct the grids that dramatically speeds up the setup stage and improves convergence rate. This work is developed in parallel to our work. In this chapter, we propose to combine a geometry-informed algebraic multigrid solver as the preconditioner to the Krylov-based conjugate gradient method. To better exploit GPU hardware, we will employ block Jacobi relaxation as part of our preconditioner.

5.3 Problem Definition and FEM Discretization

We use as our canonical problem the generalized elliptic Helmholtz problem, given in the strong form as:

$$-\nabla \cdot (\sigma(\mathbf{x})\nabla u(\mathbf{x})) + \lambda u(\mathbf{x}) = f(\mathbf{x}) \qquad \mathbf{x} \in \Omega \tag{5.1}$$

with zero Neumann (*i.e.*, natural) boundary conditions on the boundary of the domain Ω. In Equation 5.1, $u(\mathbf{x})$ is the solution over a domain Ω, $f(\mathbf{x})$ is a (given) right-hand side forcing function, $\sigma(\mathbf{x})$ is a symmetric, positive definite matrix and λ is a strictly positive constant. This problem has been chosen as it is representative of the type of system found in many engineering applications such as solid and fluid mechanics [49, 54]. Although Neumann conditions have been selected for simplicity, nothing presented in this chapter strongly depends on this choice; Dirichlet or mixed (Robin) conditions could equally have been chosen.

In traditional finite element analysis, the weak form of Equation 5.1 is formed through integration by part and the resulting equation then discretized. Let us define our approximation space $\mathcal{V}$ based upon a piecewise tessellation of Ω denoted Ω_T, which contains E elements and N nodes. We seek to find an approximation $\tilde{u} \in \mathcal{V}$ such that for all $v \in \mathcal{V}$:

$$(\nabla v, \sigma \nabla u) + \lambda (v, u) = (v, f) \tag{5.2}$$

where $(\cdot, \cdot)$ denotes the L_2 inner product over the domain. Following [49], we express our function space in terms of a basis of global piecewise linear tent functions $\phi_i(\mathbf{x})$ where i denotes a vertex index within our triangulation of the computational domain. With this choice of the discretizing trial and test functions, we arrive at the following system of equations:

$$\sum_{j=1}^{N} (\nabla \phi_i, \sigma \nabla \phi_j) \tilde{u}_j + \lambda \sum_{j=1}^{N} (\phi_i, \phi_j) \tilde{u}_j = (\phi_i, f), \tag{5.3}$$

where $\tilde{u}_j$ denotes the approximation of u on node v_j and i ranges from $1, \ldots, N$. We can rewrite the above equation in matrix form:

$$\begin{cases} \mathbf{A}\underline{u} = \underline{b}, \\ \mathbf{A} = \mathbf{S} + \lambda \mathbf{M}, \end{cases} \tag{5.4}$$

where $\underline{b}$ is the forcing vector formed from the right-hand side of Equation 5.3, $\mathbf{S}$ is the stiffness matrix given by $S_{ij} = (\nabla\phi_i, \sigma\nabla\phi_j)$ and $\mathbf{M}$ is the mass matrix given by $M_{ij} = (\phi_i, \phi_j)$. Given $\lambda > 0$, $\mathbf{A}$ is a symmetric, positive-definite matrix.

In practice, each entry A_{ij} of the matrix $\mathbf{A}$ is assembled from all elements that contain both nodes v_i and v_j and similarly, each entry b_i of the vector $\underline{b}$ is assembled from all elements that contain v_i.

A standard approach used to form the global mass and stiffness matrices is to form the *local* mass and stiffness matrices associated with each element and to assemble them based upon the mesh topology. For a triangulated 2D domain $\Omega \subset \mathcal{R}^2$, considering a triangle $e \in \Omega_T$, the local matrix $\mathbf{A}^e$ is computed as $A^e_{l_i l_j} = S^e_{l_i l_j} + \lambda M^e_{l_i l_j}$ where l_i and l_j denote the local indices of the vertices v_i and v_j in triangle e and the entries of $\mathbf{S}^e$ and $\mathbf{M}^e$ are computed by $S^e_{l_i l_j} = (\nabla\phi_i, \nabla\phi_j)$ and $M^e_{l_i l_j} = (\phi_i, \phi_j)$, respectively. The integrals are computed with numerical quadrature over the triangle (using a mapping and Gaussian integration [54]). The matrix entries $A^e_{l_i l_j}$ can then be accumulated to the i^{th} row and j^{th} column of the global matrix $\mathbf{A}$, *i.e.*, $A_{ij} += A^e_{l_i l_j}$. The forcing vector can be computed in a similar manner. The entry b_i of $\underline{b}$ is the integral of the basis function at v_i and the forcing function, *i.e*, $b_i = (\phi_i, f)$. The integral over each element is computed first and then accumulated to its corresponding location in b as done in the formation of A. The serial algorithm for the general assembly step to compute A is show in Algorithm 5.1.

Algorithm 5.1 Assembly(Ω_T)

```
1: Initialize A to zeros;
2: for all element e ∈ Ω_T do
3:    Compute A^e and b^e;
4:    for all node v_i ∈ e do
5:       for all node v_j ∈ e do
6:          A_ij += A^e_{l_i l_j};
7:       end for
8:    end for
9: end for
```

Once the global matrix **A** and the forcing vector $\underline{b}$ are formed, a linear solver is used to solve the system $\mathbf{A}\underline{u} = \underline{b}$ for $\underline{u}$. Of the three steps main steps within the finite element method, the elemental computation step is embarrassingly parallel once the data are ready. To save memory access, this step is combined with the assembly step in our FEM pipeline. In the sections to follow, we focus on the details of our assembly and linear system solution strategies and the elemental computation step outlined above will be mentioned in the assembly step description.

5.4 FEM Assembly on the GPU

Generally, a parallel assembly algorithm would proceed as follows. First, one forms the *empty* global matrix according to the given mesh, using a sparse matrix representation, and sets all entries of the matrix to zeros. One then loads the data needed for the elemental computation (node indices and coordinates) from global memory and performs all elemental computations in parallel. Finally, one accumulates the local matrix entry values to the proper locations in the precomputed empty matrix. To find the proper locations, one needs to perform the searching operations before the accumulation.

This algorithm is simple and needs minimal preprocessing, but it is not, in this direct form, well-suited to GPU architectures. This is because the global memory accesses of the nodal coordinates and the loading of needed data for each element are not coalesced. Also, each node's coordinates are shared by multiple elements so the coordinates, residing in global memory, are accessed redundantly. When a thread is trying to accumulate the computed element matrix to the global matrix, it needs to search for the memory location. This search operation is expensive to accomplish using global memory. Finally, the accumulation operations are done in parallel which can cause race conditions. This requires that atomic add operation be used to do the accumulation; such operations are also expensive when accomplished using global memory.

To address these challenges, we propose a patch-based hierarchical assembly strategy. With the proposed strategy, global memory accesses are coalesced, redundant global memory loads are avoided, and the global matrix entry accumulation

is performed in a hierarchical way. Binary search and accumulation are done in shared memory, and the accumulated values are written back to global memory as a block. The details of the algorithm for this strategy are described as follows.

The algorithm begins with a data preparation step. Given a mesh including a node coordinate list, an element list, and an adjacency (neighboring nodes) information, we first partition the node set of the mesh into mutually disjoint subsets that we call patches. We assign the elements to the patches based on the patch assignments of their first nodes. In this way, each patch consists of a set of elements that do not overlap with other patches. We then rearrange the node coordinate list and element list according to this decomposition. The node indices are changed after rearrangement so that the node indices of each element and the adjacency information of the mesh are also changed accordingly. The x, y, z coordinates of the node list are, in practice, stored in three separate arrays for coalesced global memory access. For the same reason, the node indices of the element list are also stored in separate arrays. For instance, we use four arrays to store the node indices of the tetrahedral elements with array i storing the indices of the i^{th} node of each element. This decomposition operation does not add to the total running time of the FEM solve, because this decomposition is also use by the linear system solver in subsequent parts of the algoirthm.

Next, we form the global empty matrix from the adjacency information of the mesh as the nonzero entry column indices of row i corresponds to the index of node v_i (diagonal entry) and the indices of v_i's neighbors. Because the global matrix is symmetric, we build and store only the upper half (including the diagonal) of the matrix. We choose to use the compressed sparse row (CSR) format to store this matrix. CSR consists of three arrays: $row\_offsets$, $column\_indices$, and $values$ where $values$ is an array of the (left-to-right, then top-to-bottom) nonzero values of the matrix; $column\_indices$ is the column indices corresponding to the values; and $row\_offsets$ is the list of indices where each row starts. We then fill the $row\_offsets$ and $column\_indices$ arrays according to the mesh adjacency information and all entries of the $values$ with zero.

With the node coordinate list, the element list, and an empty global matrix

prepared, the assembly process consists of the following six steps:

1. The coordinate data for each patch are loaded into shared memory. Specifically, assuming that patch i has N_i nodes, we use each of the first N_i threads to load the coordinates of one node. By this procedure, the global memory accesses are coalesced.

2. Assuming that patch i has E_i elements, each thread loads the coordinates needed by an element and stores them into registers. For the elements on the boundary of a patch, some of their nodes are outside of the patch. In this case, the node indices are not available in shared memory so data has to be loaded from global memory. Fortunately, the boundary node number takes a small portion of the whole node set so the global memory access does not significantly affect the performance.

3. Each thread executes the elemental computation to construct the local (elemental) matrices.

4. The *column_indices* and *values* arrays of the CSR global matrix are loaded into shared memory, overwriting the shared memory space used for node coordinates in the first step. Shared memory has a limited size, which is not enough to store all the data (*i.e.*, coordinates, *column_indices*, and *values*) for our typical patch size so the shared memory for coordinates is overwritten to save shared memory. In this situation, preserving the ordering in which data are loaded into shared memory is essential to guarantee correctness, *i.e*, the loading of *column_indices* and *values* must be accomplished after the coordinates are loaded into local storage (registers or local memory) for all elements of this patch. The *values* array in shared memory is initialized to zero.

5. Local matrix entries are accumulated (with atomic add being used) to the proper location in the *values* array in shared memory. The proper location is found by a binary search on the *column_indices* array in the shared memory. Specifically, considering an element e, $A^e_{l_i l_j}$ must be accumulated to row i and column j in the global matrix. Array segment $column\_indices[row\_offsets[i]]$ to $column\_indices[row\_offsets[i+1]]$ contains all the column indices of the nonzero entries of row i. However, it is not known where index j is inside this segment.

We use a binary search to find the location of index j, which is also the location in *values* where we should accumulate $A^e_{l_i l_j}$ to. For patch boundary elements where the element node is outside of the patch, a binary search and atomic adds have to be used on global memory.

6. The *values* array in shared memory is written into global memory in a coalesced manner. Note that the shared memory *values* array write back can conflict with other patches that are processing boundary elements. Because of this, a temporary *values_B* array in global memory is used to store the boundary element accumulation. After the whole assembly kernel function has completed, *values_B* is added to *values* array.

5.5 Solution of the FEM Linear System

In this section, we present our GPU-aware conjugate gradient solver preconditioned with a geometry-informed algebraic multigrid solver used for the solution of the linear system produced through the FEM method described previously.

5.5.1 Method Description

The matrix from our canonical problem, discretized using the finite element method, produces a sparse, symmetric positive-definite matrix. Therefore, we choose a preconditioned conjugate gradient (PCG) algorithm to solve the linear system $Au = b$, as shown in Algorithm 5.2.

We use a geometry-informed algebraic multigrid (AMG) solver as a preconditioner for the conjugate gradient method (PCG-AMG), in order to achieve an efficient and robust linear system solver for finite element problems. In this section, we describe in detail our parallelism scheme and data structures used to adapt our PCG-AMG to the GPU architecture. The proposed AMG solver is based on the smoothed aggregation multigrid (SAMG) method described in [103]. As in most other AMG methods, SAMG constructs the graph corresponding to the interconnectivity of the degrees of freedom from the matrix. The proposed AMG method constructs the graph and corresponding meshes (the primary mesh and coarsened structure) directly from the mesh, and therefore, we call it *geometry-informed*. In this way, we can save the computation that converts a mesh to a graph and use the

Algorithm 5.2 Preconditioned Conjugate Gradient(A, b, u_0)

$r_0 \leftarrow b - Au_0$
$z_0 \leftarrow M^{-1}r_0$
$p_0 \leftarrow z_0$
$k \leftarrow 0$
while *true* **do**
  $\alpha_k \leftarrow \frac{r_k^T z_k}{p_k^T A p_k}$
  $u_{k+1} \leftarrow u_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
  if $\|r_{k+1}\| < \epsilon$ **then**
    exit loop
  end if
  $z_{k+1} \leftarrow M^{-1}r_{k+1}$
  $\beta_k \leftarrow \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
  $p_{k+1} \leftarrow z_{k+1} + \beta p_k$
  $k \leftarrow k + 1$
end while
return u_{k+1}

geometry information to measure the quality of the aggregation or patches that are used in our AMG method.

The PCG-AMG method consists of two stages: the set-up stage and the iteration stage. The set-up stage includes the following steps: grid construction, prolongator generation, and coarse-level operator generation. This stage prepares the data for the multigrid method and is executed only once. The iteration stage includes the CG iteration, as shown in Algorithm 5.2. A multigrid *V-cycle* is performed once as the preconditioner for each CG iteration. In the following subsections, we describe in detail the proposed GPU-based PCG-AMG method.

5.5.1.1 Set-up Stage

The set-up stage begins with the construction of the AMG meshes from the mesh. This construction starts with the decomposition of the nodes into small mutually disjoint subsets. This decomposition process is called *aggregation* and the node subsets are called *aggregates*. The aggregation, as in [6] and [102], relies on a maximal independent set (MIS) of mesh nodes to define roots of aggregates and then groups each root and its neighbors into one aggregate. After this process, any

ungrouped nodes are assigned to the nearest aggregate. After the aggregation of one level, the algorithm builds an induced graph from the aggregation by treating each aggregate as a node in the coarser level and adding an edge between two aggregates (nodes in the coarser level) if any of their nodes are connected in the finer level. Then the algorithm performs the aggregation again on the coarser level graph. The algorithm continues until the number of nodes in the graph is smaller than a certain threshold. In practice, because our relaxation method requires the graphs of each level be partitioned into larger *patches*, we propose the *double partitioning strategy* which will be described in Section 5.5.2.

With the meshes constructed, the tentative prolongator at level l, $\tilde{P}^l$, is given by:

$$\tilde{P}^l_{ij} = \begin{cases} 1 \; if \; i \in C^l_j \\ 0 \; otherwise, \end{cases} \tag{5.5}$$

where C^l_j denotes the aggregate to which node j belongs in level l. The actual prolongator is a smoothed version of the $\tilde{P}$. We choose the weighted Jacobi method as the smoother, thus yielding a prolongator matrix given by:

$$P^l = (I - \omega D^{-1} A^l)\tilde{P}^l, \tag{5.6}$$

where ω is a positive constant (scaling), I is the identity matrix, D is the matrix given by the diagonal of A^l, which is the grid operator matrix of level l. Given the prolongator at the level l, its coarser level $l+1$ operator (matrix) is formed variationally. Firstly, we compute the restrictor which is the transpose of the prolongator: $R^l = {P^l}^T$ and then compute the coarser-level operator by $A^{l+1} = R^l A^l P^l$.

5.5.1.2 Iteration Stage

The iteration stage includes the PCG-AMG iterations as shown in Algorithm 5.2. In each iteration, one AMG *V-cycle* is performed as the preconditioner. From the computational point of view, in this stage, the AMG *V-cycle* is actually the bulk of the work. Here, I describe our *V-cycle* algorithm in detail.

A *V-cycle* is generally composed of these steps: prerelaxation to smooth the values, computation of the residual, restriction of the residual to a higher level, recursively calling the *V-cycle* procedure until the coarsest level is reached, solution

of the coarsest level linear system, prolongation of the value to finer level and postrelaxation to smooth the value again. The detailed algorithm is as follows:

Algorithm 5.3 V-cycle(A^k, R^k, P^k, b^k, u^k)

1: **if** *level k is the coarsest level* **then**
2: *solve* $A^k u^k = b^k$ *and return* u^k
3: **else**
4: $u^k \leftarrow \text{pre-relax}(A^k, u^k, b^k)$
5: $r^k \leftarrow b^k - A^k r^k$
6: $r^{k+1} \leftarrow R^k r^k$
7: $e^{k+1} \leftarrow \text{V-cycle}(A^{k+1}, R^{k+1}, P^{k+1}, r^{k+1})$
8: $e^k \leftarrow P^k e^{k+1}$
9: $u^k \leftarrow u^k + e^k$
10: $u^k \leftarrow \text{post-relax}(A^k, u^k, b^k)$
11: **end if**

The relaxations are the most time-consuming part of all the *V-cycle* steps, so a suitable relaxation method and optimized implementation are essential for overall performance. In our case, a good relaxation method should effectively smooth out the high-frequency errors and be easily parallelized for GPU. The relaxation is usually implemented as a Jacobi smoothing (see Equation 5.7) since it is very easy to implement for parallel architectures. Indeed, both [11] and [44] use this method in their respective AMG GPU implementations.

$$u = u + \omega D^{-1}(b - Au). \tag{5.7}$$

However, the Jacobi method is not ideal for multigrid relaxation in terms of convergence rate [8]. Its implementation depends on the matrix-vector multiplication, which generally has low computational density and does not efficiently use resources on the GPU. In this chapter, we propose to use a variant of weighted block Jacobi method for relaxation. This method gives significantly better convergence rate than the Jacobi method and can achieve fine-grained parallelism and high computational density by taking advantage of the hierarchical memory layout on GPU.

The standard weighted block Jacobi is defined as follows. Let $\mathcal{N} = \{1, ..., n\}$ be the set of all the nodes in the domain and consider decomposing $\mathcal{N}$ into p nonoverlapping patches,

$$\mathcal{N} = \bigcup_{1}^{p} \mathcal{N}_k.$$

Let A be partitioned into blocks A_{ij} of size $n_i \times n_j$ where the rows of A_{ij} are in $\mathcal{N}_i$ and the columns are in $\mathcal{N}_j$. The weighted block Jacobi method takes the matrix form:

$$u = u + \omega M^{-1}(b - Au), \tag{5.8}$$

where ω is a positive constant (scaling), M is a block diagonal matrix with $M = diag\{A_{kk}\}$ with $diag\{A_{kk}\}$ denoting the block diagonal matrix with blocks A_{kk}.

Because $M^{-1} = diag\{A_{kk}^{-1}\}$, block Jacobi computes $g_k = A_{kk}^{-1}$ in parallel with each processor solving for one of the g_k either directly or iteratively. We do not precisely compute M^{-1}, but instead we use multiple weighed Jacobi iterations to approximate g_k. That is, we iterate $\tilde{g}_k^{n+1} = \tilde{g}_k^n + \omega D_{kk}^{-1} r_k$ multiple times. D_{kk} is the diagonal matrix of A_{kk} and r_k denotes the residual values corresponding to $\mathcal{N}_k$. With this method, we can use low-latency GPU memories (shared memory and registers) to store the diagonal matrices and do the weighted Jacobi iterations on these fast memory spaces to achieve high performance. Our experiments (see Section 5.6) show that this method is very effective as the relaxation for multigrid in terms of overall convergence rate.

5.5.2 Implementation and Data Structures

We now present the implementation details and data structures needed to effectively use the GPU's streaming multiprocessors.

5.5.2.1 Set-up Stage

The block Jacobi method requires that the domain be partitioned into patches with each patch containing a group of connected nodes. This task is challenging for the several reasons. First, we want to map the patches to the CUDA blocks; thus, the patches should be small enough so that they can fit into limited hardware resources. Second, the patches should be large enough so that patch partitioning does not

result in too many edge cuts, because this increases interactions between patches and undermines the effectiveness of the Jacobi updates. Third, the SAMG that we are mimicking needs a finer partition of the mesh into aggregates as mentioned in Section 5.5.1. It is important that the patch partition does not cut through aggregates in order to achieve the best convergence rate.

We propose the bottom-up double partitioning strategy to generate the aggregates and patches. The double partitioning strategy includes three steps: (1) generation of the aggregates (aggregate partition), (2) building an induced graph from the aggregates, and (3) generation of the patches by partitioning the induced graph again (patch partition).

Both of the partitions rely on maximal independent sets (MIS) or k-MIS, an extension of MIS where k specifies the radius of independence of the set. An MIS is a set of nodes in the graph no two of which are connected by an edge, a k-MIS is a set of nodes in the graph no two of which are connected by a path of length k or less. Both MIS and k-MIS have the property that no node in the graph can be added to the set without violating the independence property. Since regularity of aggregate size is important to the convergence of the solver, we have found it necessary to take steps to control the aggregate sizes to improve the distribution.

Our partition method takes as input the graph representation of the mesh and produces the permutation necessary to re-order the nodes of the input graph according to their patch and aggregate membership, the indices for the start of each aggregate and partition in the permuted graph, and the graph representation of the next coarser mesh. The aggregate partition is performed as follows:

1. Find a k-MIS for the graph, where the value of k is chosen to control the number and size of generated aggregates. Higher values of k result in sparser sets of root nodes and therefore larger aggregates.

2. Number the nodes in the k-MIS sequentially to index the aggregates.

3. Add other nodes to aggregates iteratively. Each node in the graph checks its neighbors to see which aggregate they are in. If all neighbors are in the same aggregate, the current node will add itself to the same aggregate. If the neighbors are members of more than one aggregate, the node selects the aggregate with which

it shares the highest adjacency. This repeats until all nodes are allocated.

4. After the initial allocation is completed, find the number of nodes in each aggregate, remove aggregates below a certain size by labeling all nodes in these aggregates as unallocated, and then re-index the remaining aggregates.

5. Repeat the allocation process to add the nodes from eliminated aggregates to remaining aggregates.

The patch partition consists of performing the partition defined above on the induced graph, which includes a weight for each node that is the number of nodes in the corresponding aggregate. The size control mechanisms applied therefore use the total weight of nodes rather than their count.

To control the size of patches, we remove patches below a threshold weight and re-allocate their aggregates as detailed above. Then we iteratively exchange nodes between patches to improve the size distribution. The patches exchange aggregates as follow:

1. Compute the weighted size for each patch.

2. Each aggregate that could move to another patch calculates the most desirable exchange for itself.

3. Every patch for which it is desirable to give up a node(s) performs the most desirable exchange (this limit is to damp oscillations of patch size that could be caused by multiple exchanges).

4. Recalculate the weighted sizes for each patch, and if the largest patch is smaller than the threshold value, the process terminates, otherwise another iteration begins.

Our experiments show that for 3D tetrahedral meshes, $k = 2$ is best for the aggregate partition and $k = 1$ for the patch partition, as the resulting aggregates and patches generated are of the appropriate size. Our parallel k-MIS algorithm is similar to [7, 102, 11] and implemented on GPU. We know of one other k-MIS implementation in the publicly available CUSP library, which according to our experiments has comparable performance (in terms of computing time) to our implementation.

The partitions described above form a permutation array that maps the indices

of the nodes in the original mesh (graph) to a re-ordered index list in which nodes belonging to the same patch are grouped together and within each patch, nodes belonging to the same aggregate are grouped together. Once the partitions are done, we permute the matrix of each multigrid level according to the permutation array and the tentative prolongator $\tilde{P}$ is constructed according to Equation 5.5. Then $\tilde{P}$ is smoothed with one weighed Jacobi iteration as described in Equation 5.6. $\tilde{P}$ is a sparse matrix with a special sparse pattern that each row has only one nonzero value which is set to one. We use a special version of parallel sparse matrix-matrix multiplication as described in [11] to compute $A\tilde{P}$ needed in the prolongator smoothing process. Lastly, the restrictor and the matrix for the next level is computed as described in the previous section. We use the matrix transpose and matrix-matrix multiplication functions in the CUSP library to compute the restrictor and the matrix for next level.

5.5.2.2 Iteration Stage

The iteration stage performs the PCG iterations. As depicted in Algorithm 5.2, one iteration of PCG consists of a preconditioning step (one *V-cycle*), a matrix vector multiplication, and some vector operations. Of all these operations, the preconditioning step (*V-cycle*) is the most expensive. As mentioned above, the *V-cycle* consists of prerelaxation, residual computation, restriction, coarsest level solution, prolongation, error correction, and postrelaxation. The prerelaxation, residual computation, and postrelaxation steps are the bulk of the work since each of them needs to access the operator matrix of a level. Our proposed *V-cycle* pipeline combines the prerelaxation and residual computation steps to save one costly matrix access. Next, we will describe the data structures we propose for our AMG preconditioner and the *V-cycle* pipeline in detail.

An appropriate data structure is essential to fully harness the potential computing power of the GPU. The GPU has limited fast memory space (shared memory and registers) in addition to global memory. When local data of a kernel are too large to fit in the fast memory space, the data spill over to the local memory, which is as slow as global memory. So a compact data structure is desired to save storage

and memory accesses. The data structure determines also the memory access pattern, which is particularly important for global memory accesses because of their high latency. Block Jacobi requires domain decomposition (patches) and the matrix is permuted accordingly to bear a blocked pattern. Each edge in the domain corresponds to a nonzero value in the matrix.

We propose a novel sparse matrix data structure specially designed for the block Jacobi method, which we call patch sparse matrix format (patchSPM). This proposed data structure is composed of three parts: the patch inside A_I, the patch boundary A_B, and the diagonal A_D. Thus, $A = A_I + A_B + A_D$. A_I is composed of all the entries A_{ij} of A such that i and j belong to the same patch. A_B is defined as the opposite and A_D stores the diagonal values in an array. Matrix A_I is a diagonally blocked matrix, and all the matrix blocks are symmetric, sparse matrices. These matrix blocks are concatenated in the GPU global memory with each of them in a sparse matrix format. An integer array is used to store the beginning offset for each matrix block.

The patch inside matrix A_I is typically much denser than A_B, and each of its matrix blocks is loaded into shared memory and accessed many times during the block Jacobi inner updates, as described in Section 5.5.1.2. Therefore, the data format of its matrix blocks has a significant impact on the performance. We considered three potentially appropriate sparse matrix formats: Ellpack (ELL), CSR, and Symmetric Coordinate list (SymCOO). ELL format stores a M by N matrix nonzero values in a dense M by K array *values*, where K is the maximum number of nonzeros entries per row. Similarly, the corresponding column indices are stored in another M by K array *indices*. The rows that have fewer than K nonzero values are padded with a sentinel value. ELL format is regular resulting in coalesced global memory accesses, but it stores useless data to pad the unstructured matrix to be rectangular, which wastes bandwidth and undermines GPU performance. In addition, due to the useless padding data, ELL data structure is not compact enough to fit into the fast memory space (shared memory and registers). For many meshes where the maximum valance is high, the data spill over into slow local memory and inner Jacobi iterations become very expensive. CSR, as described in

Section 5.4, is compact but irregular, which leads to uncoalesced global memory access. The SymCOO format is a variant of COO for symmetric matrices. It consists of three arrays: *row_indices*, *column_indices*, and *values*. The *row_indices* and *column_indices* arrays store the row index and column index of each nonzero entry of the upper half of the matrix. The *values* array stores the values of those nonzero entries. SymCOO is the most compact data structure since it stores only half of the matrix and it is regular. The drawback of SymCOO is that it typically requires atomic operations in the relaxation step. We alleviate this drawback by performing the atomic operation in the faster GPU memory space (shared memory space). Our experiments show that using SymCOO format for the matrix blocks of A_I has the best overall performance. The boundary matrix A_B is very sparse and stored in general COO format. Figure 5.1 shows the patchSPM data structure.

In our *V-cycle* pipeline, as mentioned before, we combine the prerelaxation and residual steps, *i.e*, we use only one CUDA kernel function, which we call *prerelax-residual*, for these two steps. We now describe this kernel in detail as follows.

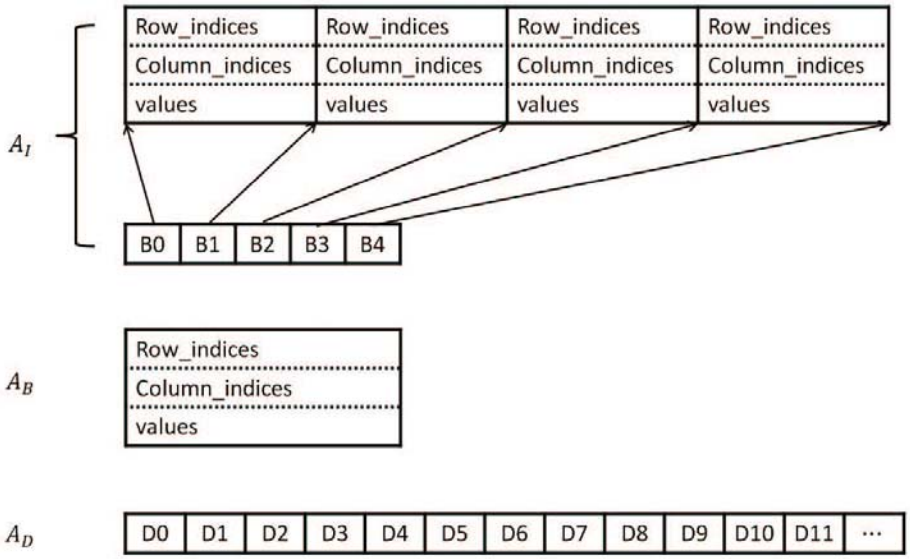


Figure 5.1. The patchSPM data structure consists of three parts: A_I, A_B, and A_D. A_I includes a concatenated list of SymCOO formats and an integer array indicating the beginning and ending of each matrix block in the list, A_B is in COO format, and A_D is an array of diagonal values.

1. Each CUDA block loads a segment of A_D, b, and a matrix block of A_I (corresponding to a patch) into the shared memory and registers.

2. The kernel allocates two arrays *s-Ax* and *s-u* in shared memory and initializes their elements to zeros. These arrays are used to store the block matrix vector multiplication result and the temporary result after each inner Jacobi iteration, respectively. The kernel synchronizes here to make sure the matrix block of A_I is loaded and *s-Ax* is initialized within a CUDA block before execution of next instruction.

3. The kernel performs multiple inner Jacobi iterations in the shared memory registers to obtain the final u result now in *s-u* and the final result is written back to global memory after synchronization.

4. One more block matrix vector multiplication is performed to compute the partial residual $\tilde{r}$.

Here, the computed residual is incomplete because the computation takes into account only the values of the inside matrix A_I and the diagonal matrix A_D, *i.e*, the computed residual from this kernel is $\tilde{r} = b - (A_I + A_D)u$. The real residual should be $r = b - Au$ so after this kernel call, we need to "compensate" the residual by subtracting $A_B x$ from $\tilde{r}$, and then the real residual is $r = \tilde{r} - A_B u$. Similarly, before the postrelaxation, $A_B x$ should be subtracted from b to get the real right-hand side for the block Jacobi iteration, as described in Equation 5.8. The *postrelax* kernel is quite similar to the *prerelax-residual*, but it does not have the residual computation step. Since A_B is relatively sparse compared to A_I, the running time needed to compute $A_B u$ is relatively short. On the whole, we have a different *V-cycle* pipeline (Algorithm 5.4) for our multigrid method from the typical pipeline shown in Algorithm 5.3.

5.5.3 Mixed-Precision Computation

In numerical computing on the GPU, there is a fundamental performance advantage in using single precision floating point data format over double precision. Due to a more compact representation, twice the number of single precision data elements can be stored at each level of the memory hierarchy, including the register file, caches, and main memory. By the same token, handling single precision values

Algorithm 5.4 *V-cycle-new*(A_I^k, A_B^k, R^k, P^k, b^k, u^k)

1: **if** *level k is the coarsest level* **then**
2: *solve* $A^k u^k = b^k$ and
3: **return** u^k
4: **else**
5: $u^k, \tilde{r}_k \leftarrow$ pre-relax-residual(A_I, u^k, b^k)
6: $r^k \leftarrow \tilde{r}^k - A_B^k u^k$
7: $r^{k+1} \leftarrow R^k r^k$
8: $e^{k+1} \leftarrow$ V-cycle($A_I^{k+1}, A_B^{k+1}, R^{k+1}, P^{k+1}, r^{k+1}$)
9: $u^k \leftarrow P^k e^{k+1} + u^k$
10: $\tilde{b}^k \leftarrow b^k - A_B u^k$
11: $u^k \leftarrow$ post-relax($A_I, u^k, \tilde{b}^k$)
12: **end if**

consumes less bandwidth between different memory levels. In addition, many modern processor architectures, including GPUs, have much better throughput for single precision operations than for double precision operations. For example, NVIDIA's Fermi GPUs' single precision operations are around twice as fast as double precision [71]. Thus, researchers have been trying to find ways to use single precision operations as much as possible without sacrificing the overall accuracy. [21] and [33] point out that for a preconditioned Krylov-subspace method, the preconditioner can be single precision without affecting the accuracy. This is important for our proposed solver since we are trying to load the matrices associated with the multigrid levels into fast, but limited size, local memory spaces. In our implementation, the multigrid associated matrices and floating point vector are in single precision while all other floating point numbers are in double precision. We store an extra copy of the finest level matrix A_{fine} in double precision, and this matrix is used for the matrix vector multiplication in the PCG iteration. A_{fine} is not used for any blocked operation so it is stored in a general sparse matrix data structure called Hybrid, which is particularly efficient for unstructured sparse matrix (*e.g.*, [12]).

5.6 Numerical Results

To show the characteristics of our proposed method and the performance of the implementation, we conduct a set of systematic experiments with various

unstructured meshes and numerical set-ups. We compare our implementation against our optimized serial CPU version for the assembly step and compare our linear system solver against the state-of-the-art multigrid-based GPU and CPU solvers, namely the CUSP [70] and Hypre [63] libraries. We refer to these solvers as *CUSP-PCGAMG* and *Hypre-PCGAMG*, respectively, and we call our solver *patchPCGAMG*. All experiments are executed on a Linux (OpenSuse 11.4) computer equipped with an Intel i7 965 Extreme CPU running at 3.2 GHz and a NVIDIA GeForce GTX 580 GPU. The GPU is equipped with 1.5 GBytes of memory and 16 streaming multiprocessors, where each multiprocessor consists of 32 SIMD computing cores that run at 1.544 GHz. Each streaming multiprocessor has configurable 16 or 48 KBytes of on-chip shared memory for quick access to local data. Computation on the GPU means running a kernel with a batch process of a large group of fixed size thread blocks. NVCC 4.0.1 and gcc 4.3 are used to compile the CUDA and CPU codes, respectively, and -O3 flag is used in the compiling.

The unstructured meshes we use in our tests are listed in Table 5.1. The Regular mesh is generated by the following process: subdivide a $4\times4\times4$ cube into 512 $0.5\times0.5\times0.5$ small cubes and then cut each small cube into six tetrahedra resulting in an initial tetrahedral mesh containing 729 nodes and 3072 elements. We then subdivide each tetrahedron of this initial tetrahedral mesh into eight smaller tetrahedra by connecting the midpoints of the edges. We perform this midpoint subdivision three times to produce the final Regular mesh shown in Table 5.1. In this process, a series of tetrahedral meshes is generated with each finer mesh doubling the resolution of the coarser mesh. This series of meshes is used in our scalability experiment in Section 5.6.2.2. The Irregular mesh is generated by tetrahedralizing a $4\times4\times4$ cube. The Heart and Brain meshes are visualized in Figure 5.2. The Blobs mesh has two regions, inside of the blobs and outside of the blobs, which are color coded differently in Figure 5.3. This mesh is used in the heterogeneous domain experiment in Section 5.6.2.4 where the two regions have different coefficients (σ in Equation 5.4). The proposed assembly and linear system solution methods extend naturally to 2D triangular meshes with some parameter tuning. Therefore, we only report the 3D tetrahedral mesh result in this section.

Table 5.1. The meshes used in our experiments.

mesh names	node number	element number	min valance	max valance
Regular	274,625	1,572,864	3	18
Irregular	197,561	1,122,304	3	25
Heart	437,355	2,306,717	5	36
Brain	322,497	1,805,242	6	34
Blobs	277,657	1,650,105	5	46

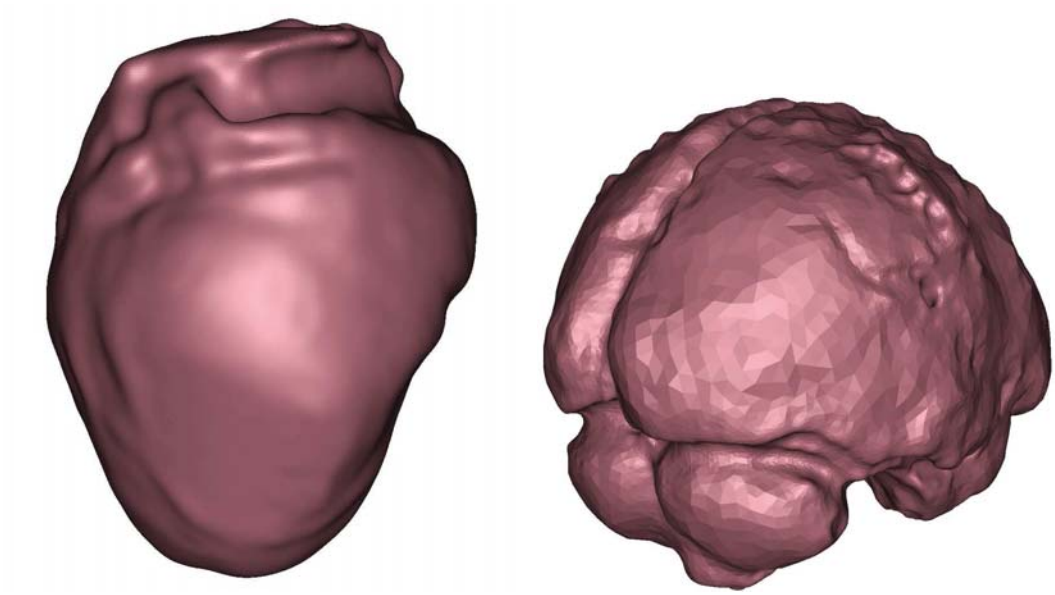


Figure 5.2. Surface rendering of the exterior surfaces of the Heart and Brain meshes.

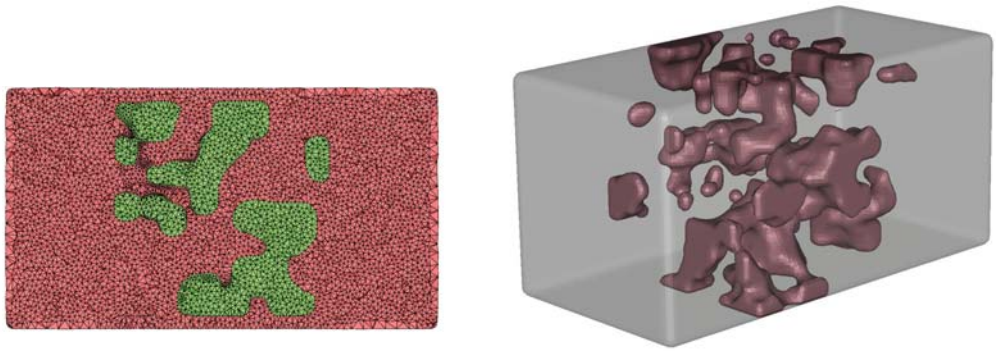


Figure 5.3. A cross section and the volume visualization of the Blobs mesh.

5.6.1 Assembly Performance

We show the performance of our GPU assembly by assembling for the linear system of the Helmholtz equation (Equation 5.1 with $\lambda = 1$) from all the meshes mentioned above and comparing the running time against our optimized serial CPU implementation which is based on Algorithm 5.1. Both implementations compute the global matrix A as in Equation 5.4 using double precision. The results are shown in Table 5.2. Our GPU implementation of the assembly step is up to 87 time faster than the CPU implementation.

5.6.2 Linear System Solution Numerical Experiments

We conduct a series of experiments to show the properties of our method and the performance of our implementation. We compare the result against the state-of-the-art GPU and CPU multigrid-based linear solver: CUSP-PCGAMG and Hypre-PCGAMG. For Hypre-PCGAMG, the hybrid Gauss-Seidel method is used for the relaxation and PMIS is chosen for coarsening. We use mixed precision strategy for patchPCGAMG and CUSP-PCGAMG and double precision for Hypre-PCGAMG as our experiment shows that single precision and double precision performance difference is very small on CPU. The CUSP-PCGAMG uses the same smoothed aggregation multigrid method as ours while the Hypre library uses the BoomerAMG-based multigrid preconditioner. For all the experiments, the solution is considered converged if the relative error $\epsilon = \frac{||r||}{||b||} < 1e-8$, where r is the residual and b is the right-hand side of the linear system whose entries are all set to one, *i.e*, $(\phi_i, f) = 1$ in Equation 5.3. $||\mathbf{x}||$ denotes the l^2 norm of a vector $\mathbf{x}$. We show the result of tolerance $1e-8$, but the trend is the same for smaller tolerances.

Table 5.2. Assembly performance (double precision): GPU and CPU running time (in seconds) comparison.

meshes	GPU	CPU	speedup
Regular	0.0298	1.080	36
Irregular	0.0229	1.010	44
Heart	0.0465	3.114	67
Brain	0.0355	3.077	87
Blobs	0.0319	2.525	79

5.6.2.1 Multigrid Set-up Stage Performance

Table 5.3 shows the running time of the multigrid set-up stage for all meshes with the result compared to the set-up stages of CUSP-PCGAMG and Hypre-PCGAMG. S1 and S2 are the speedups in contrast with Hypre-PCGAMG and CUSP-PCGAMG, respectively, and negative values denote when patchPCGAMG is slower.

Compared to CUSP-PCGAMG, our AMG set-up stage has an extra partitioning step as described in Section 5.5.2.1, and hence its performance is worse. As shown in the table above, patchPCGAMG is 1.2× to 1.3× slower. On the other hand, patchPCGAMG achieves up to 3.2× speedup for the set-up stage when compared with Hypre-PCGAMG.

5.6.2.2 Scalability with Problem Size

Multigrid-preconditioned Krylov subspace methods are known to have linear scalability with the matrix size for structured problems, and thus, the convergence rate (number of CG iterations) should not change with the matrix size. In this section, we show how our AMG preconditioned CG linear system solver scales when the mesh resolution increases using the series of regular tetrahedral meshes mentioned before. FEM is used to solve the Helmholtz equation with natural boundary condition on these meshes. We solve the associated linear system with our AMG preconditioned CG linear solver and show the scalability of the solver by measuring the number of global (PCG) iterations needed to converge. The result is compared to the other two AMG preconditioned CG linear solvers(CUSP-

Table 5.3. Multigrid set-up stage running time in seconds. S1 and S2 are the speedups comparing patchPCGAMG to Hypre-PCGAMG and CUSP-PCGAMG. Speedup number is in parentheses when patchPCGAMG is slower.

meshes	patchPCGAMG	Hypre-PCGAMG	S1	CUSP-PCGAMG	S2
Regular	0.519	1.10	2.0	0.420	(1.2)
Irregular	0.385	0.665	1.7	0.292	(1.3)
Heart	0.813	2.51	3.2	0.604	(1.3)
Brain	0.591	1.68	2.9	0.451	(1.3)
Blobs	0.569	1.27	2.1	0.431	(1.3)

PCGAMG and Hypre-PCGAMG) and a pure CG solver (CUSP-CG) (see the plot in Figure 5.4).

As shown in Figure 5.4, our solver demonstrates good scalability with problem size although not perfectly linear. It is slightly better than the other two PCG-AMG solvers. In addition, our solver needs only about half the number of iterations to converge compared to CUSP-PCGAMG. This difference is mainly due to the difference of the relaxation method since both are using SAMG method and the aggregate partition strategy is similar. The inexact block Jacobi relaxation we use shows clear advantage over the Jacobi method used by CUSP-PCGAMG. We can also see from the plot that all three PCG-AMG solvers scales much better than the pure CG solver, which confirms the claim that MG generally has good scalability with problem size.

5.6.2.3 Inner Iteration Influence on Convergence Rate

As mentioned earlier, we use the inexact weighted block Jacobi method for the relaxation step in the multigrid method. Multiple inner Jacobi iterations are performed to approximate the inverse of the matrix block A_{ii} according to patch i.

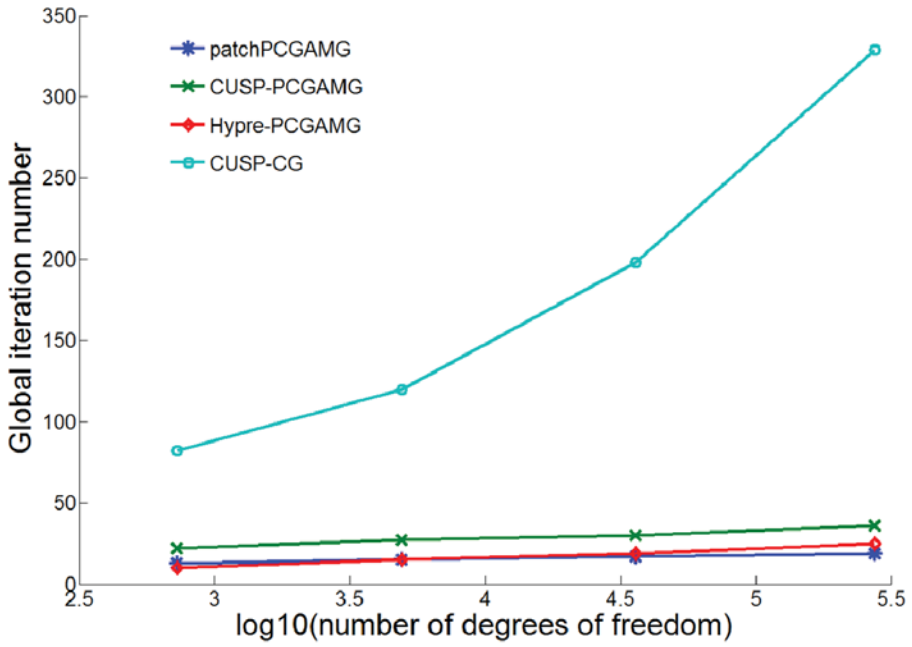


Figure 5.4. The plot for number of degrees of freedom against global iteration number.

The reasons why we compute the inverse inexactly are two-fold: first, the inverse computation for the matrix blocks are different if we compute exactly, which leads to imbalanced work loads for the CUDA blocks. Second, we are using the block Jacobi as the relaxation to smooth out the high-frequency error and there is no need to compute the inverse exactly. Figure 5.5 shows how the number of inner iterations is related to the global (PCG) iteration number needed to converge for the meshes. It can be seen from the plot that larger inner iteration number generally lead to less global iterations but after around three inner iterations, the global iteration number does not change any more or changes very little. Although the inner iteration is relatively cheap as we load the matrix blocks into fast memory space (registers or shared memory), it is not totally free. Larger inner iteration number leads to poorer per-iteration relaxation performance. Our experiments show that three inner iteration is generally the sweet spot for overall performance.

5.6.2.4 Heterogeneous Media Influence on Convergence Rate

This experiment shows how the method performs when the domain is heterogeneous, *i.e.*, the coefficients of the Laplacian operator in Equation 5.2 $\sigma = \sigma(\mathbf{x})$ is not the same for all $\mathbf{x}$. This happens when a simulation is done on a multimaterial

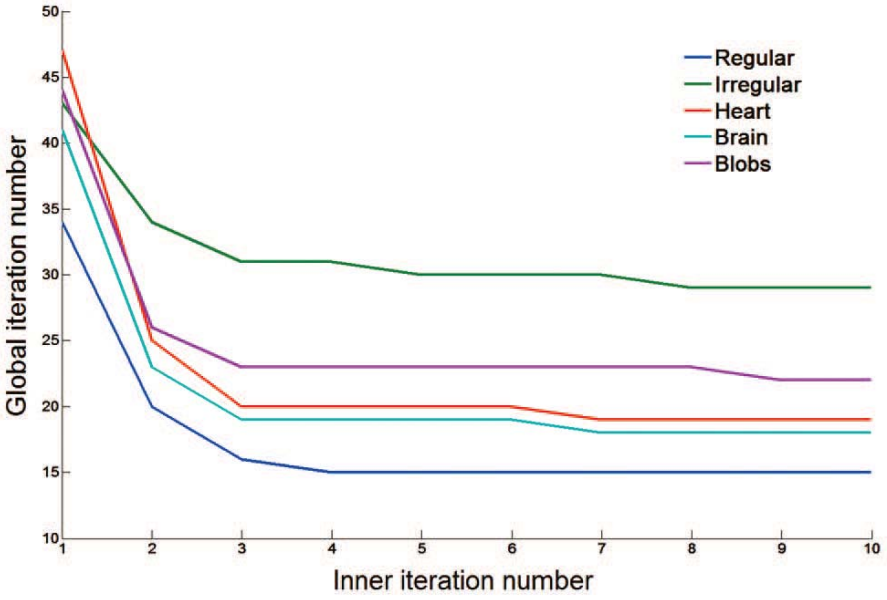


Figure 5.5. Plot of inner iteration number against global iteration number.

domain as the σ is usually different in different materials. Table 5.4 shows how the method performs when the meshes have two different materials and one of the materials has $\sigma = 1$ and the other material's σ is $1, 10, 100$, respectively, and compares to CUSP and Hypre and the unpreconditioned conjugate gradient method from CUSP library which we call *CUSP-CG*. As shown in the table, all methods converge slower with increased heterogeneity. The patchPCGAMG and CUSP-PCGAMG are becoming worse at roughly the same rate (the convergence rate ratio of the two is roughly the same with different heterogeneity). This means the patch partition used patchPCGAMG is not affecting the performance for heterogeneous problem.

5.6.2.5 Running Times for All Meshes Comparison

Table 5.5 compares the running times and (number of iterations) for our linear solver along with CUSP-PCGAMG and Hypre-PCGAMG. We also include two pure (unpreconditioned) conjugate gradient implementations: a GPU implementation from the CUSP libraray (CUSP-CG) and a CPU implementation in Hypre (Hypre-CG). S1 and S2 are the speedups of patchPCGAMG compared to Hypre-PCGAMG and CUSP-PCGAMG. S3 is the speedup of the CUSP-CG compared to the Hypre-CG.

Also shown in Table 5.6, the patchPCGAMG achieves up to 51× speedup compared to the state-of-the-art CPU PCG-AMG implementation Hypre-PCGAMG while porting the pure CG method to GPU gains only up to 9× speedup. This is indicative that CG is not particularly well-suited for the GPU many-core architectures. Although adding AMG as the preconditioner makes the solver much more complicated than pure CG, it is worth the extra effort considering the performance

Table 5.4. Heterogeneous media performance comparison for the Blobs mesh: (m,n) means the σ values for the two materials in the domain are m and n, respectively. The numbers reported are the global iteration numbers.

Methods	(1,1)	(1,10)	(1,100)
patchPCGAMG	23	31	60
CUSP-PCGAMG	50	60	122
Hypre-PCGAMG	28	30	40
CUSP-CG	1048	2419	7071

Table 5.5. Running times in seconds (global iteration number) for all meshes: S1 and S2 are the speedups of patchPCGAMG compared to Hypre-PCGAMG and CUSP-PCGAMG. S3 is the speedup of the CUSP-CG compared to the Hypre-CG.

meshes	patch PCGAMG	Hypre-PCGAMG	S1	CUSP-PCGAMG	S2	CUSP-CG	Hypre-CG	S3
Regular	0.139(19)	3.86(25)	28	0.175(36)	1.3	0.680(329)	3.73(329)	5
Irregular	0.167(31)	3.02(29)	18	0.216(56)	1.3	2.43(1639)	14.8(1639)	6
Heart	0.218(20)	11.2(31)	51	0.631(46)	2.9	4.64(1148)	33.8(1131)	7
Brain	0.165(19)	7.78(27)	47	0.432(45)	2.6	8.15(1838)	60.4(1810)	9
Blobs	0.172(23)	5.70(28)	33	0.409(50)	2.4	3.34(1048)	16.0(1030)	5

Table 5.6. Per global iteration running times in milliseconds for all meshes: S1 and S2 are the speedups of patchPCGAMG compared to Hypre-PCGAMG and CUSP-PCGAMG. Speedup number is in parentheses when patchPCGAMG is slower.

meshes	patch PCGAMG	Hypre-PCGAMG	S1	CUSP-PCGAMG	S2
Regular	7.31	154	21	4.68	(1.6)
Irregular	5.40	104	19	3.86	(1.4)
Heart	10.9	361	33	13.7	1.3
Brain	8.71	288	33	9.60	1.1
Blobs	7.49	204	27	8.18	1.1

improvement on the GPU. In addition, the patchPCGAMG achieves 1.3× to 2.9× speedup comparing to the CUSP-PCGAMG on the same GPU. The global iteration numbers in the table demonstrate that our block Jacobi relaxation greatly improves the convergence rate compared to Jacobi method used in CUSP-PCGAMG. Table 5.6 shows the per global iteration performance of the three PCGAMG methods. Comparing to the CUSP-PCGAMG, the per iteration running time of the patchPCGAMG is comparable although the block Jacobi relaxation used in the patchPCGAMG performs much more computation than the Jacobi method. This confirms our claim that our relaxation method increases the computational density and better balances the memory bandwidth and computations. It can also be noted from Table 5.6 that for the simpler meshes, Regular and Irregular, where the valance is relatively not variable, the CUSP has better per iteration performance because it uses the Hybrid sparse matrix data structure that performs better when the matrix is regular ([11]). For the other meshes, which are more representative of

real-life data, the patchPCGAMG performs similarly or even better. As expected, both GPU implementations have much better per iteration performance than the Hypre-PCGAMG.

5.7 Conclusions and Future Work

In this chapter, we present the complete pipeline of a parallel FEM solver for unstructured meshes that performs very well on the many-core parallel processors. The proposed GPU assembly performs up to 87× better than an optimized CPU implementation, and the proposed multigrid preconditioned CG solver achieves a speedup of up to 51× compared to the state-of-the-art CPU implementations. These speed ups compare very favorably against other attempts at GPU-accelerated linear solvers, many of which report lackluster results [19]. The algorithms and data structures need not be changed to run on newer generation hardware (*e.g.*, Kepler GPU) efficiently. However, some parameter might need to be tuned to obtain the best performance, such as the patch size and inner iteration number.

We choose to use a geometry-informed AMG as the preconditioner for the CG method to solve the linear system from the FEM. The proposed AMG preconditioner dramatically speeds up the convergence rate of the CG method and changes the computational bulk of the work from the CG iteration to the AMG preconditioner—a solver methodology which adapts very well to the many-core parallel architecture with proposed parallelism scheme and data structures. This is juxtaposed with the typical CG implementation on the GPU, which suffers from excessive communication and low computational density. This is borne out in the experimental data, which shows dramatically better speed ups for AMG on the GPU vs the CPU. Thus, the corresponding improvements in AMG performance on the GPU make it a particularly attractive option for taking advantage of the significant compute power offered by these devices.

Unfortunately, AMG presents some challenges, particularly in the aggregation, restriction, and prolongation methods, that are sometimes problem-dependent; thus, it is more difficult to imagine a completely general software solution for the linear solve, as one would typically expect with a CG solver. We have

included some preliminary results for the heterogeneous media and those results are very encouraging, but further investigation is needed to fully understand how the heterogeneity influences the performance when the partitions (aggregate and patch) do not align with the heterogeneity. Anisotropy is likely to present further challenges. In this chapter, we focused on solving the FEM problems with a single GPU. However, there are circumstances that single GPU is not enough for a given problem, either because the problem size is too large to fit into the memory of a single GPU, or the performance of the problem on a single GPU is not satisfactory. Therefore, an important area of future work would be solvers that use an out-of-core paradigm for memory handling/shuffling to the GPU or solvers that scale across multiple GPUs or GPU clusters.

CHAPTER 6

AN EFFICIENT PARALLEL ALGORITHM FOR SOLVING THE LEVELSET EQUATIONS ON UNSTRUCTURED DOMAINS

6.1 Introduction

The levelset method uses a scalar function $\phi = \phi(\mathbf{x}(t), t)$ to implicitly represent a surface or a curve, $S = \{x(t) \mid \phi(\mathbf{x}(t))\} = k$, hereafter referenced as a *surface*. The surface or curve deformation is captured by numerically solving the associated nonlinear partial differential equation (levelset equation) on ϕ. The levelset method has a wide array of application areas, ranging from geometry, fluid mechanics, and computer vision to manufacturing processes [89] and virtually any problem that requires interface tracking. The method was originally proposed by Osher and Sethian [73] for regular grids, and early levelset computations used finite difference approximations on fixed, logically rectilinear grids. Such techniques have the advantages of a high degree of accuracy and programming ease. However, in some situations, a triangulated domain with finite element type approximations is more appropriate. Barth and Sethian have cast the levelset method into the finite element framework and extended it to unstructured meshes in [9]. Since then, the levelset method has been widely used in applications that involve complex geometries and require the use of unstructured mesh for simulation. For example, in medical imaging, the levelset method on brain surfaces is used for automatic sulcal delineation, which is important for investigating brain development and disease treatment [53]. In computer graphics, researchers have been using the levelset methods for feature detection and mesh cutting via geodesic curvature flow [108]. Yet another application of the levelset method on unstructured domains is the simulation of solidification and crystal growth processes [98].

Many studies have been conducted to develop efficient levelset solvers. In [4], Adalstein and Sethian propose a narrowband scheme to speed up the computation. This approach is based on the observation that one is typically only interested in a particular interface, and in this case, only the computation around the interface is necessary. Whitaker proposes the sparse field method in [106], which employs the narrowband concept and maintains a narrowband containing only the wavefront nodes and their neighbors to further save computation. Some other studies in the literature are focused on memory efficiency of the levelset method. Bridson proposes the sparse block grid method in [18] to dynamically allocate and free memory and achieves suboptimal storage complexity. Strain [96] proposes the octree levelset method that is also efficient in terms of storage. Houston *et al.* [48] apply the run-length encoding (RLE) scheme to compress regions away from the narrowband to adjust their sign representation while storing the narrowband with full precision. This scheme further improves the storage efficiency over the octree approach. A number of recent works [61, 50, 36] address the parallelism strategies for solving the levelset equation on CPU-based and GPU-based parallel systems. However, these works have been focused on regular grids, and the parallelism schemes do not readily extend to unstructured meshes.

Recently, there has been growing interest in floating point accelerators. These accelerators are devices that perform arithmetic operations concurrently with or in place of the CPU. Two solutions received special attention from the high-performance computing community: GPUs, originally developed for video cards to render graphics, that are now used for very demanding computational tasks, and the newly released Intel Xeon Phi, which employs very wide (512 bit) SIMD vectors on the same X86 architecture as other Intel CPUs and promises high-performance and little programming difficulty. Relative to CPUs, the faster growth curves of these accelerators in the speed and power efficiency have spawned a new area of development in computational technology. Now, many of the top supercomputers are equipped with accelerators such as the current top one, Titan [66]. Developing efficient code for these accelerators is a very important building block of fully utilizing these supercomputers. In this chapter, we present efficient parallel

algorithms for solving the levelset equations on unstructured meshes on both CPU-based and GPU-based parallel processing systems.

The use of unstructured meshes makes the levelset method more flexible with respect to computational domains. However, solving the levelset equation on unstructured meshes poses a number of challenges for efficient parallel computing. First, there is no natural partition of the domain for parallelism, and the use of a graph partitioner to decompose the mesh may result in uneven partition sizes, which in turn leads to a load balancing problem. Second, for regular meshes, the valence of the nodes is the same, and hence nodal parallelism is typically employed that assigns each node to a thread. However, for unstructured meshes, the nodes have varying valences and neighborhood structures, which leads to irregular data structures and unbalanced workload for nodal parallelism. Third, the boundary communications among partitions typically require additional computations and separate data structures to find and store the boundary locations.

In this chapter, we present a new parallelism strategy for solving the levelset equation on unstructured meshes that combines a narrowband scheme and domain decomposition. We propose the narrowband fast iterative method (nbFIM) to compute the distance transform by solving an eikonal equation in a narrowband around the wavefront and the patched narrowband scheme (patchNB) to evolve the levelset. We use unified domain partitioning for both distance transform and levelset evolution to ensure minimal setup time. For unstructured meshes, the update of the value on each node depends on values of its neighboring nodes, and the different valences can leads to load balancing issues. This is especially inefficient for GPUs and other streaming architectures, which employ SIMD-like architecture and prefer regular computations. To address this, we propose elemental parallelism instead of nodal parallelism to mitigate load balancing problem. However, the elemental parallelism approach may lead to contention, because multiple elements will try to update the value of the same node simultaneously. Typically, atomic operations are used to solve this problem. However, atomic operations are expensive especially on GPUs, and can result in significant numbers of threads blocking while waiting for access to variables. Therefore, we propose a new lock-free algorithm

(and associated data structures) to enforce data compactness and locality for both shared memory CPU systems and GPUs. We call this approach *hybrid gathering*. Our algorithm converts the contention problem to a sorting problem that is efficient on parallel systems, including GPUs [86]. Both the distance transform (part of maintaining the narrowband) and the levelset evolution benefit from this lock-free update scheme. In this chapter, we describe the data structure and algorithm, and present systematic experimental results to demonstrate the efficiency of the proposed method on both shared memory CPU system and the GPU.

This chapter proceeds as follows. In Section 6.2, we introduce levelset equation and the proposed methods and algorithms. In Section 6.3, we discuss implementation details and data structures. In Section 6.4, we discuss the performance of both CPU and GPU implementations of the proposed method, using several 2D and 3D examples as a benchmark. In Section 6.5, we summarize the results and discuss future research directions related to this work.

6.2 Mathematical and Algorithmic Description

In general, the levelset equation solver with narrowband scheme has two main building blocks: the distance transform recomputation (reinitialization) and interface evolution according to the levelset equation (evolution). In this section, we give the mathematical description of the levelset equations, and we describe the numerical algorithms. We first introduce the necessary notation and definitions and then describe the narrowband scheme and the associated reinitialization algorithm. Finally, we present the numerical scheme for the evolution step and lastly present the novel *hybrid gathering* parallelism scheme and lock-free update algorithm.

6.2.1 Notation and Definitions

The levelset method relies on an implicit representation of a surface by a scalar function

$$\phi : \Omega(\boldsymbol{x}) \rightarrow \mathbb{R}, \tag{6.1}$$

where $\Omega \in \mathbb{R}^n, n \in \{2,3\}$ is the *domain* of the surface model, which can be a 2D plane, a 3D volume or a manifold. Thus, a surface S is

$$S = \{\boldsymbol{x} \mid \phi(\boldsymbol{x}) = k\}. \tag{6.2}$$

The choice of k is arbitrary, and we call ϕ the *embedding*. The surface S is referred to as an *isosurface* of ϕ. Surfaces defined in this way partition Ω into inside and outside, and such surfaces are always closed provided they do not intersect the boundary of the domain. The embedding ϕ is approximated on a discrete tessellation of the domain. The levelset method uses a one-parameter family of embeddings, *i.e.*, $\phi(\boldsymbol{x}, t)$ changes over time t with $\boldsymbol{x}$ remaining on the k levelset of ϕ as it moves, and k remaining constant. The behavior of ϕ is obtained by setting the total derivative of $\phi(\boldsymbol{x}(t), t) = k$ to zero. Thus,

$$\phi(\boldsymbol{x}(t), t) = k \quad \Longrightarrow \quad \frac{\partial \phi}{\partial t} + \nabla\phi \cdot \boldsymbol{v} = 0. \tag{6.3}$$

Let $\mathbf{F}$ denote the normal speed, $\mathbf{F} = \boldsymbol{v} \cdot \frac{\nabla\phi}{|\phi|}$. The level set equation, given by Equation 6.3, then can be written as

$$\frac{\partial \phi}{\partial t} + \mathbf{F}|\nabla\phi| = 0. \tag{6.4}$$

We define the initial condition as $\phi(\boldsymbol{x}, t = 0) = \mathbf{g}(\boldsymbol{x})$. In general, $\mathbf{F}$ can be a more complicated function of $\boldsymbol{x}$ and $\nabla\boldsymbol{x}$: $\mathbf{F} = \mathbf{F}(\boldsymbol{x}, \nabla\phi, \nabla^2\phi, ...)$. In this chapter, we consider the levelset equation with

$$\mathbf{F} = \boldsymbol{\alpha}(\boldsymbol{x}) \cdot \nabla\phi + \epsilon(\boldsymbol{x})|\nabla\phi| + \beta(\boldsymbol{x})\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}|\nabla\phi|, \tag{6.5}$$

where $\boldsymbol{\alpha}(\boldsymbol{x})$, $\epsilon(\boldsymbol{x})$, and $\beta(\boldsymbol{x})$ are user defined coefficient functions. We call these three terms of $\mathbf{F}$ the advection term, eikonal term, and curvature term, respectively. This form of levelset equation is used widely in many applications such as image processing, computer vision, *etc*.

We approximate the domain Ω by a triangulation Ω_T, which consists of non-overlapping simplices that we call elements. Based upon this triangulation, we form a piecewise linear approximation of the solution by maintaining the values of the approximation on the set of vertices V and employing linear interpolation within each element in Ω_T. The total number of vertices in V is denoted $|V|$, and the total number of elements in Ω_T is denoted $|\Omega_T|$. We use v_i to denote the ith vertex

in V An *edge* is a line segment connecting two vertices (v_i, v_j) in $\mathbb{R}^d, d \in \{2,3\}$ and is denoted by $e_{i,j}$. The vector from vertex v_i to vertex v_j is denoted by $\mathbf{e}_{i,j} = \mathbf{x}_j - \mathbf{x}_i$.

In this chapter, we consider both 2D and 3D cases, and Ω_T consists of triangles or tetrahedra, respectively. A triangle, denoted $T_{i,j,k}$, is a set of three vertices v_i, v_j, v_k that are pairwise connected by an edge. Similarly, a tetrahedron is denoted $T_{i,j,k,l}$. The set of vertices adjacent to vertex v_i is called *one-ring neighbors* of v_i and is denoted by $\mathcal{N}_i$, while the set of adjacent elements is called *one-ring elements* of v_i and is denoted by $\mathcal{A}_i$. We denote the discrete approximation of the solution ϕ at vertex v_i by Φ_i. The area or volume of an element T is denoted meas(T).

6.2.2 Narrowband Scheme and Distance Transform Recomputation

Many applications require only a single surface model. In these cases, solving the equation over the whole domain for every time-step is unnecessary and computationally inefficient. Fortunately, levelsets evolve independently (to within the error introduced by the discrete triangulation) and are not affected by the choice of embedding. Furthermore, the evolution of ϕ is important only in the vicinity of that levelset. Thus, one should perform calculations for the evolution of ϕ only in a neighborhood of the surface expressed by Equation 6.2. In the discrete setting, there is a particular subset of mesh nodes whose values define a particular levelset. Of course, as the surface moves, that subset of mesh nodes must change according to the new position of the surface.

In [4], Adalsteinson and Sethian propose a narrowband scheme that follows this line of reasoning. The narrowband scheme constructs an embedding of the evolving curve or surface via a signed distance transform. The distance transform is truncated: computed over a finite number of nodes that lie no further than a specified distance from the levelset. This truncation defines the *narrowband* and the remaining points are set to constant values to indicate that they lie outside the narrowband. The evolution of the surface is computed by calculating the embedding only within the narrowband. When the evolving levelset approaches the edge of the narrowband, the new distance transform and the new embedding are calculated, and the process is repeated. This algorithm relies on the fact that

the embedding is not a critical aspect of the evolution of the levelset. That is, the embedding can be transformed or recomputed at any point in time, as long as such transformation does not change the position of the kth levelset, and the evolution will be unaffected by this change in the embedding. Following the strategy in [106], most inplementations keep a list of nodes in the narrowband. However, this approach is not efficient for GPUs and unstructured meshes because the nodes in the narrowband will have arbitrary order, and the memory access is effectively random. We propose the *patched narrowband* (patchNB) scheme to enforce memory locality and improve performance on GPUs. This scheme keeps a list of patches inside the narrowband instead of nodes, and each patch is assigned to a GPU streaming multiprocessor with values of nodes in each patch being updated in parallel by GPU cores. In this way, the data (geometry information, values, intermediate data) of each patch can be stored in fast shared memory, and global memory access is coalesced and reduced. We describe the scheme in more detail in Section 6.3.

This narrowband scheme requires the computation of the distance transform (reinitialization). We propose a modified version of the patched fast iterative method [37], that we call nbFIM, to compute the distance transform by solving the eikonal equation with the value of speed function set to one. The nbFIM restricts the computational domain to the a narrowband around the levelset that significantly reduce computational burden. Also, we propose new algorithm and data structures to further improve the performance.

Specifically, the nbFIM employs a domain decomposition scheme that partitions the computational domain into patches and iteratively updates the node values of the patches near the levelset until all patches are either converged (not changing anymore) or far away from the levelset. The algorithm maintains an active list that stores the patches that require an update. The active list initially contains the patches that intersect with the levelset, and then it is updated by removing convergent patches and adding their neighboring patches if they are within a certain distance from the levelset. The distance of a patch is defined as the minimal value of all the node values in this patch. In this way, the patches that are far

from the levelset are not updated at all, which saves computation. Also, it is guaranteed that all the nodes with values smaller than narrowband width are in the new narrowband list. The details of the implementation will be described in Section 6.3.

The node value update process calculates the new value of a node by computing potential values from the one-ring elements and take the minimal of the potential values as the new value. We call the computation of the potential value of a node the *local_solver*, which requires geometric information of one of the one-ring elements and the values of other nodes in the element.

6.2.3 Levelset Evolution and PatchNB

The numerical scheme we use to discretize Equation 6.4 in space is based on [9]. We adopt the positive coefficient scheme for the first-order terms of the levelset equation (the advection and the eikonal terms):

$$\mathbb{H}_j(\nabla\phi) = \frac{\sum_{l=1}^{|\Omega_T|} \tilde{\alpha}_j^l \mathbb{H}(\nabla\phi)}{\sum_{l=1}^{|\Omega_T|} \tilde{\alpha}_j^l \text{meas}(T_l)}, \tag{6.6}$$

where $\mathbb{H}$ denotes the advection or eikonal term, which are first order homogeneous functions of $\nabla\phi$, and $\tilde{\alpha}_j^l$ are non-negative constant coefficients that are defined as follows:

$$\tilde{\alpha}_j^l \leftarrow \frac{\max(0, \alpha_j^l)}{\sum_{k=1}^{d+1} \max(0, \alpha_k^l)}. \tag{6.7}$$

The coefficients α_j^l are defined as:

$$Q_j \leftarrow \nabla\mathbb{H} \cdot \nabla N_i \text{meas}(T), \tag{6.8}$$

$$Q_j^- \leftarrow \min(0, Q_j), \tag{6.9}$$

$$Q_j^+ \leftarrow \max(0, Q_j), \tag{6.10}$$

$$\delta_j \leftarrow Q_j^+ (\sum_{l=1}^{d+1} Q_l^+)^{-1} \sum_{i=1}^{d+1} Q_i^- (\phi_i - \phi_j), \tag{6.11}$$

$$\alpha_j^l \leftarrow \frac{\delta_j}{\mathbb{H}_{T_l}}, \tag{6.12}$$

where $\mathbb{H}_{T_l} = \int_{T_l} \mathbb{H} d\boldsymbol{x}$, and N_i is the linear basis function satisfying $N_i(\boldsymbol{x}_j) = \delta_{ij}$.

We adopt the Finite Element Method (FEM) for the curvature term that is approximated at each node v_i as:

$$(K|\nabla\phi|)_i = \frac{\sum_{j\in\mathcal{N}_i} W_j^i(\phi_j - \phi_i)}{\sum_{T\in\mathcal{A}_i}\text{meas}(T)}\left|\frac{\sum_{T\in\mathcal{A}_i}\text{meas}(T)\sum_{j=1}^{d+1}\nabla N_j\phi_j}{\sum_{T\in\mathcal{A}_i}\text{meas}(T)}\right|, \tag{6.13}$$

where $W_j^i = \sum_{\{T|e_{ij}\in T\}}\frac{\nabla N_i\cdot\nabla N_j\text{meas}(T)}{|\nabla\phi|}$.

Equations 6.6–6.13 show that the computation of the new value ϕ_i at node v_i requires values from its one-ring neighbors and geometric data from the one-ring elements. Algorithm 6.1 shows a typical serial algorithm, and AH1, AH2, AV1, AV2, V, PV, CV are temporary arrays that store the intermediate results. Similar to the reinitialization, we use elemental parallelism for better load balancing and hybrid gathering scheme to avoid contention.

6.2.4 Hybrid Gathering Parallelism and Lock-free Update

In both of the reinitialization and the evolution steps, we need to update the values of the nodes in the mesh, and these updates can be performed independently. The natural way to parallelize the computation is to assign each nodal update to a thread. We call this approach *nodal parallelism*, and it can be represented as a sparse matrix-vector operation, as shown in Figure 6.1 (left). The operator $\bigotimes$ denotes a generic operation defined on the degrees of freedom corresponding to $\star$'s. The advantage of this scheme is that it naturally avoids contention, because each nodal computation has an associated thread. However, it can introduce unbalanced load when the nodes have widely varying valance. These irregular computations and data structures are not efficient on GPUs that use SIMD streaming architecture. An alternative parallelism scheme is to distribute the computations among threads according to the elements. We call such approach *elemental parallelism*; it is more suitable for GPUs, because it gives regular local operators and corresponding data structures. Figure 6.1 (right) depicts the matrix representation of this approach. In this matrix representation, the matrix is a block matrix in terms of local operators according to the elements. The matrix blocks can overlap each other, and the vector of degrees of freedom is segmented but has overlaps. Each block matrix-vector operation represents a set of local computations that are performed by a

Algorithm 6.1 Evolution(Φ, B) (Φ: values of the nodes, B: narrowband)

for all $p \in B$ **do**
  for all $T_{i,j,k} \in p$ **do**
    compute meas(T)
    compute ∇N_i, ∇N_j, ∇N_k
    $\nabla\phi = \nabla N_i * \phi_i + \nabla N_j * \phi_j + \nabla N_k * \phi_k$
    compute Q for the advection term: Q_i^1, Q_j^1, Q_k^1
    compute Q for the eikonal term: Q_i^2, Q_j^2, Q_k^2
    $(\mathbb{H}^1)_T = Q_i^1 * \phi_i + Q_j^1 * \phi_j + Q_k^1 * \phi_k$
    $(\mathbb{H}^2)_T = Q_i^2 * \phi_i + Q_j^2 * \phi_j + Q_k^2 * \phi_k$
    compute Q^+ and Q^-
    compute $\tilde{\alpha}_i^1$, $\tilde{\alpha}_j^1$, $\tilde{\alpha}_k^1$, $\tilde{\alpha}_i^2$, $\tilde{\alpha}_j^2$, $\tilde{\alpha}_k^2$
    AH1[i]+ = $\tilde{\alpha}_i^1 * (\mathbb{H}^1)_T$, AH[j]+ = $\tilde{\alpha}_j^1 * (\mathbb{H}^1)_T$, AH[k]+ = $\tilde{\alpha}_k^1 * (\mathbb{H}^1)_T$
    AH2[i]+ = $\tilde{\alpha}_i^2 * (\mathbb{H}^2)_T$, AH[j]+ = $\tilde{\alpha}_j^2 * (\mathbb{H}^2)_T$, AH[k]+ = $\tilde{\alpha}_k^2 * (\mathbb{H}^2)_T$
    AV1[i]+ = $\tilde{\alpha}_i^1 * \text{meas}(T)$, AV[j]+ = $\tilde{\alpha}_j^1 * \text{meas}(T)$, AV[k]+ = $\tilde{\alpha}_k^1 * \text{meas}(T)$
    AV2[i]+ = $\tilde{\alpha}_i^2 * \text{meas}(T)$, AV[j]+ = $\tilde{\alpha}_j^2 * \text{meas}(T)$, AV[k]+ = $\tilde{\alpha}_k^2 * \text{meas}(T)$
    V[i]+ = meas(T), V[j]+ = meas(T), V[k]+ = meas(T)
    PV[i]+ = $\nabla\phi * \text{meas}(T)$, PV[j]+ = $\nabla\phi * \text{meas}(T)$, PV[k]+ = $\nabla\phi * \text{meas}(T)$
    CV[i]+ = $\frac{\nabla N_i \cdot \nabla N_i * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_i \cdot \nabla N_j * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_i \cdot \nabla N_k * \text{meas}(T)}{|\nabla\phi|}$
    CV[j]+ = $\frac{\nabla N_j \cdot \nabla N_j * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_j \cdot \nabla N_i * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_j \cdot \nabla N_k * \text{meas}(T)}{|\nabla\phi|}$
    CV[k]+ = $\frac{\nabla N_k \cdot \nabla N_k * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_k \cdot \nabla N_i * \text{meas}(T)}{|\nabla\phi|} + \frac{\nabla N_k \cdot \nabla N_j * \text{meas}(T)}{|\nabla\phi|}$
  end for
  for all $v_i \in B$ **do**
    $\phi_i - = \alpha \frac{\text{AH1[i]}}{\text{AV1[i]}} + \epsilon \frac{\text{AH2[i]}}{\text{AV2[i]}} + \beta \frac{|\text{CV[i]}|\text{PV[i]}}{\text{V[i]}^2}$
  end for
end for

thread. This parallelism scheme may result in contention as multiple threads may be updating the same degree of freedom due to the overlapping. The conventional solution to this problems is to use atomic operations. However, this is not suitable for GPUs as the atomic operations on GPUs are quite expensive, especially for double precision floating number operations, which are widely used in scientific computing.

We have developed a novel parallelism scheme that we call hybrid gathering to combine the advantages of both the nodal and elemental parallelism schemes. In the hybrid gathering parallelism scheme, the computation is decomposed into

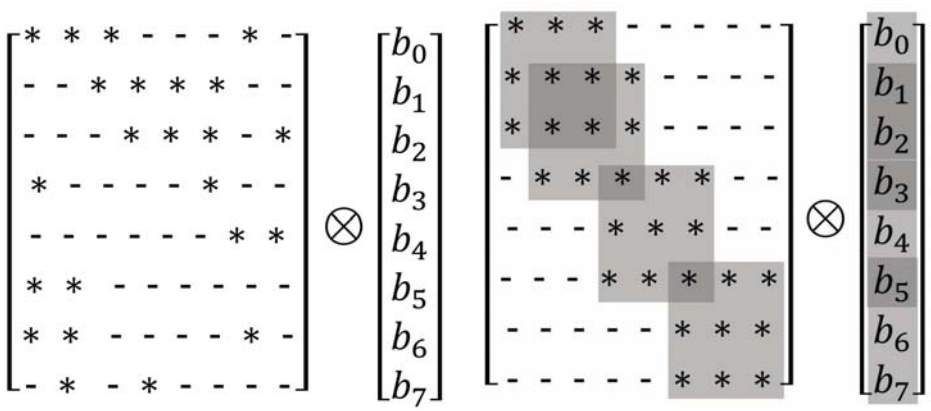


Figure 6.1. Matrix representations of the parallelism schemes. On the left, we present the nodal parallelism scheme. The $\star$'s denote nonzero values or some operators. On the right, we present the elemental parallelism scheme. The matrix is blocked, and the blocks can be overlapping each other.

two stages (two matrix-vector operations): (1) performs local operations on the associated element and stores intermediate result and (2) fetches and assembles intermediate result data according to the gathering matrix. The symbols $\bigotimes$ and $\bigodot$ in Figure 6.2 represent the operations in these two stages, respectively. In the first stage, matrix blocks and vector segments are not overlapping, and the matrix-block-vector-segment operations can be assigned to different threads, thus avoiding the contention. This stage decomposes the computations according to the elements with regular local operators, after which each thread fetches intermediate result data according to the gathering matrix to assemble the value for the degrees of freedom. In practice, the two stages are implemented in a single kernel function, and fast GPU cache (shared memory or registers) is used to store the intermediate data, which makes the gathering stage very efficient.

The computation of the gathering matrix is a key part of the hybrid gathering parallelism scheme. The degrees of freedom are associated with the nodes, and the computations are performed on elements. Therefore, the gathering matrix should represent a topological mapping from elements to nodes, and this mapping describes the data dependency for each degree of freedom. In practice, the mapping from elements to nodes is typically given as an element list, denoted $\mathbb{E}$, which consists of node indices. The location indices of this list correspond to the memory

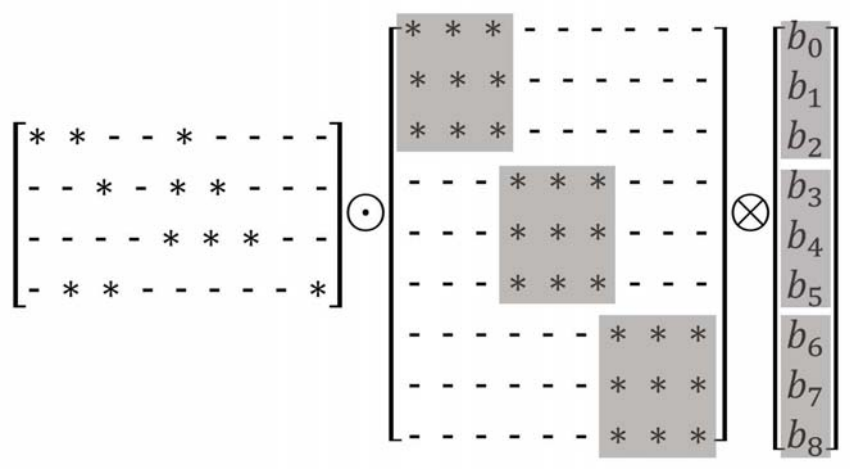


Figure 6.2. Matrix representations of the Elemental Gathering scheme.

location of the data that is required by the threads. We create a sequence list $\mathbb{S}$ that records the memory locations of $\mathbb{E}$. Then, we sort $\mathbb{E}$ and permute $\mathbb{S}$ according to the sorting. Now, in the sorted list $\mathbb{E}'$, the node indices are grouped, and the permuted sequence list, denoted $\mathbb{S}'$, stores the data memory location in original element list $\mathbb{E}$. The $\mathbb{E}'$ and $\mathbb{S}'$ together indicate the locations of the $\star$'s in the gathering matrix and form the coordinate list (COO) sparse matrix representation [12] of the gathering matrix. In this way, we convert the contention problem to a sorting problem. Here, list $\mathbb{E}$ has fixed length keys, which allows it to be sorted very efficiently on GPUs with radix sorting [86]. Essentially, sorting allows us to take full advantage of the GPU computing power and avoid the weakness of the architecture in the form of addressing contention.

6.3 Implementation

In this section, we describe the details of the implementation of our method to solve the levelset equation on parallel systems, including shared memory CPU-based computers and GPU-based systems. The pipeline consists of two stages: the set-up stage and the time-stepping stage. The set-up stage includes the partitioning of the mesh into patches, preparation of the geometrical data for the following computation, and generation of the gathering matrix Λ. We choose the METIS software package [55] to perform the partitioning. METIS partitions the mesh into

nonoverlapping node patches and outputs a list showing the partition index of each node. The data preparation step permutes the node coordinate list and rearranges the element list according to the partitioning. The time-stepping stage iteratively updates the node values until the desired number of time steps is reached. In each iteration, a reinitialization and multiple evolution steps are performed. We describe the detailed implementations and data structures for the set-up stage, the reinitialization step, and the evolution step, respectively, in the following subsections.

6.3.1 Set-up

During the set-up stage, the mesh is partitioned into patches, each of which consists of a set of nodes and a set of elements according to METIS output. The sets of nodes are mutually exclusive, and the element sets are one-layer overlapping: the boundary elements are duplicated. The vertex coordinate list and the element list are then permuted according to the partitioning so that the vertex coordinates and the element vertex indices are grouped together, and hence the global memory access is coalesced.

As described in Section 6.2, we use the hybrid gathering scheme to decompose the computation. Therefore, we must generate the gathering matrix Λ during the set-up stage. Here, we describe a simple example, with a triangular mesh, to demonstrate the generation of the matrix Λ generation. First, consider the simple mesh displayed in Figure 6.3. This mesh consists of two triangles, e_0 and e_1, which include four degrees of freedom in the solution: ϕ_0 through ϕ_3. During the solution process, the thread corresponding to element e_0 will be updating the values of ϕ_0, ϕ_1 and ϕ_2, while the thread corresponding to element e_1 will be updating the values of ϕ_0, ϕ_3 and ϕ_1. The corresponding data flow is shown in Figure 6.4. We store the intermediate data in a separate array that has one-to-one correspondence to the element list. Therefore, the element list indices are the same as the memory locations, from which the degrees of freedom require data. For our example, computations involving ϕ_0 require data from the element list memory locations 0 and 3. If we create an auxiliary sequence list {0,1,2,3,4,5}, sort the

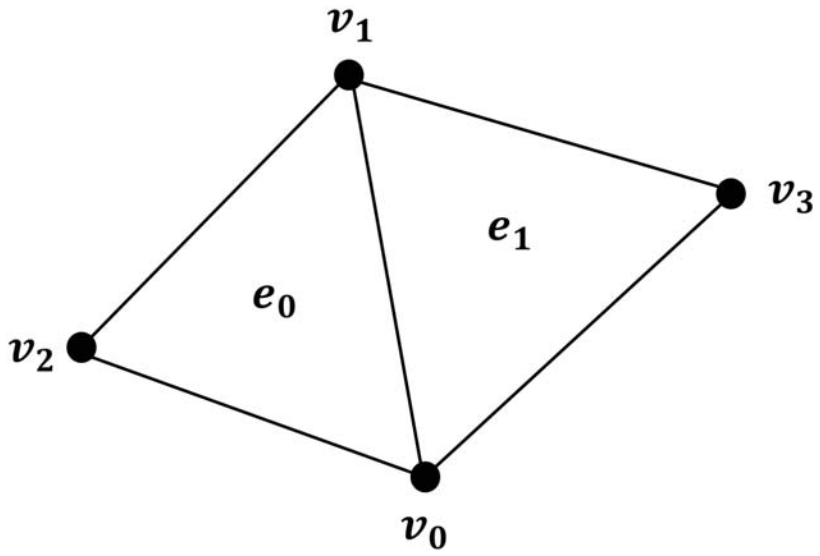


Figure 6.3. Mesh with two elements: e_0 and e_1.

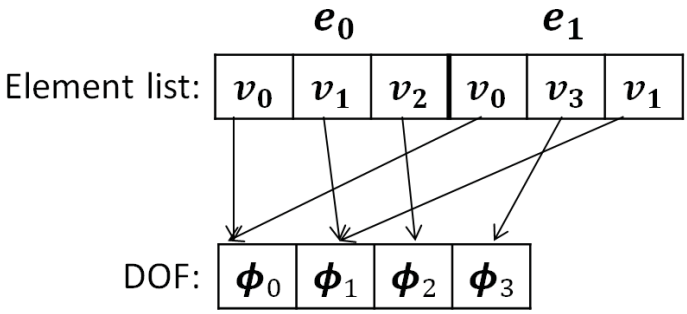


Figure 6.4. Data flow for the simple mesh example.

element list, and permute the sequence list according to the sort, we get two new lists: sorted element list {0,0,1,1,2,3} and permuted auxiliary list {0,3,1,5,2,4}. These two lists now contain the row indices and column indices of the $\star$ entries of matrix Λ. In practice, we convert this to compressed sparse row (CSR) matrix format [12] for storage by a reduction operation and a prefix summation operation on the sorted element list. The $\star$ entries in the matrix actually represent certain operators: minimum and summation in the reinitialization and evolution stages correspondingly. Hence, we do not have the value array in a typical CSR format. The final CSR representation of the gathering matrix for our two-triangle example is shown in Figure 6.5.

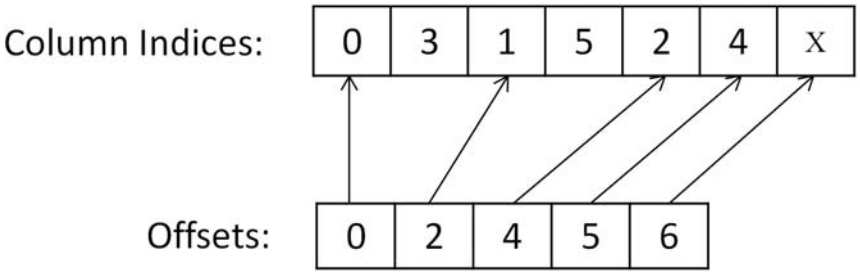


Figure 6.5. The CSR representation of gathering matrix for the two-triangle example. The box containing "X" denotes a memory location outside the bounds of the column indices array.

6.3.2 Reinitialization

In the reinitialization step, we use the nbFIM algorithm to compute the distance transform of the levelset. In this algorithm, the node value update step comprises the bulk of the work. This step updates the values of the nodes in the active patches multiple times (we call these *inner updates*) using the hybrid gathering scheme and records the convergence status of each active node. During each of the inner updates, each thread computes all the values on the nodes of the corresponding element and stores values in an intermediate array in the fast GPU shared memory. Then, we apply the gathering matrix to the intermediate array to compute the new value of each node. In this operation, each row of the gathering matrix corresponds to a node in the mesh, and column indices of the $\star$ entries in each row represent the location of required data in the intermediate array. We assign each row to a thread, which fetches the data from the intermediate array and calculates the minimum that is taken as the new value of the node. In practice, the elemental update and the gathering operation are performed in one kernel so that the intermediate values do not need to be written back to or read from global memory, which is expensive. As mentioned before, the node coordinates and element list are grouped according to the partitioning, and stored in an interleaved linear array so that the memory access is coalesced. Before the update computation, each thread needs to fetch node coordinates and old values for the corresponding element, and the memory access is virtually random. We use the fast shared memory to hold these data temporarily, and then each thread reads data from the shared memory instead of directly from

the global memory.

We use an improved *cell-assembly* data structure in the nbFIM, originally proposed in [37]. The new cell-assembly data structure includes five arrays, labeled **COORD**, **VAL**, **ELE** , **OFFSETS**, and **COL**. **COORD** and **VAL** store per-node coordinates and per-node value, respectively. **ELE** stores the per-element node indices. **OFFSETS** and **COL** form the CSR sparse matrix representation of the gathering matrix, except that we do not have the value array like in a regular CSR matrix.

In summary, assuming that every patch has N nodes and M elements (normally $M > N$), the reinitialization kernel function on the GPU (or SIMD parallelism) proceeds as follows:

1. If thread index $i < N$, load the coordinates and value of node i into shared memory array **SHARE**.

2. If thread index $i < M$, load the node indices for element i from **ELE** into registers. Fetch the node coordinates and values from **SHARE** to registers.

3. If thread index $i < M$, write node values of element i to shared memory **SHARE**.

4. If thread index $i < M$, call *local_solver* routine to compute the potential values of each node in element i, and store these values in **SHARE**.

5. If thread index $i < N$, load the column indices for the ith row of the gathering matrix, **COL**[**OFFSETS**[i]] through **COL**[**OFFSETS**[$i+1$]]. Then fetch data from **SHARE**, compute the minimal value, and broadcast the minimal value to **SHARE** according to the column indices.

6. If thread index $i < N$, if the minimal value is the same as the old value (within a tolerance), node i is labelled convergent.

7. Repeat steps 4 through 6 multiple times.

8. If thread index $i < N$, write the minimal value back to global memory **VAL**.

For the CPU-based shared memory parallel system, the implementation is generally similar but we make several modifications from the GPU implementation to suit the CPU architecture. First, we still use the nbFIM scheme, but we maintain an active segment list instead of an active patch list. Each segment in the list

stores the active nodes in its corresponding patch. The patched update strategy is suitable for GPU because it provides the fine-grained parallelism required by GPU architecture, but it also leads to extra computation [37]. The active segment list stores segments of active nodes, and each segment corresponds to a patch. Second, in the computation that updates the active node values, we assign each segment to a thread that updates all the active node values in the segment. In addition, we use the nodal parallelism to avoid any contention. Each thread computes the potential values of an active node from the one-ring elements, and then calculates the minimal value among the potential values. Specifically, the value update function in the reinitialization proceeds in the following steps for each thread t (here P denotes the number of patches):

1. Load the coordinates and value ϕ_a of an active node a of the t-th patch into registers.

2. Find one of the one-ring elements of a and load the coordinates and values into registers.

3. Call the *local_solver* to compute a potential new value ϕ_{tmp} of node a and perform $\phi_a = \min(\phi_a, \phi_{tmp})$.

4. Repeat step 2 and 3 until all one-ring elements of a are processed and write the final ϕ_a back to memory.

5. Repeat step 1 – 4 until all active nodes in patch t are processed.

6.3.3 Evolution

The evolution step updates the values of the nodes in the narrowband according to the equations presented in Section 6.2. As shown in Algorithm 6.1, we compute the approximation of the three terms of F and then update the node values. Similar to the reinitialization step, we need to deal with mixed types of parallelism: nodal parallelism and elemental parallelism as elemental computations are suitable for GPUs, but the degrees of freedom we want to solve for live on the nodes. The hybrid gathering scheme is again used to solve this problem. The update of a node value depends on multiple elemental computations corresponding to speed function **F**, and the computations for the three terms of **F** all require the same

geometric information and node values. Therefore, we want to perform all the computations in one kernel function to avoid repeated global memory accesses.

The hybrid gathering scheme is based on the assumption that the gathering can be performed on fast shared memory, and using one single kernel means that we need to store all the elemental intermediate results (**AH1**, **AH2**, **AV1**, **AV2**, **V**, **PV**, **CV** arrays in Algorithm 6.1) in shared memory, which usually is not large enough to accommodate all this data. We solve this problem by reusing the shared memory space and carefully arranging the memory load order so that the data memory footprint is small enough to fit in the GPU shared memory. Specifically, we store the elemental intermediate results in **SAH1**, **SAH2**, **SAV1**, **SAV2**, **SV**, **PV**, **SCV** arrays for each patch, which are all in shared memory, and then assemble the intermediate results according to the gathering matrix and store them in fast registers. The evolution kernel function processes in the following steps (assuming every patch has N nodes and M elements):

1. If thread index $t < N$, load the coordinates and value of node t into shared memory array **SHARE**.

2. If thread index $t < N$, load the column indices for the tth row of the gathering matrix, **COL**[**OFFSETS**[t]] through **COL**[**OFFSETS**[$t+1$]] into registers.

3. If thread index $t < M$, load the node indices for element t from **ELE** into registers. Fetch the node coordinates and values from **SHARE** to registers.

4. If thread index $t < M$, perform elemental computation of triangle $T_{i,j,k}$ for $(\mathbb{H}^1)_T$, $(\mathbb{H}^2)_T$ and $\tilde{\alpha}_i^1$, $\tilde{\alpha}_j^1$, $\tilde{\alpha}_k^1$, $\tilde{\alpha}_i^2$, $\tilde{\alpha}_j^2$, $\tilde{\alpha}_k^2$.

5. If thread index $t < M$, compute SAH1[t*M+0] = $\tilde{\alpha}_i^1 * (\mathbb{H}^1)_T$, SAH1[t*M+1] = $\tilde{\alpha}_j^1 * (\mathbb{H}^1)_T$, SAH1[t*M+2] = $\tilde{\alpha}_k^1 * (\mathbb{H}^1)_T$.

6. If thread index $t < N$, fetch data from **SAH1** according to the column indices for the tth row of the gathering matrix, compute the summation, and store the result in registers.

7. If thread index $t < M$, compute **SAH2**[t*M+0] = $\tilde{\alpha}_i^2 * (\mathbb{H}^2)_T$, **SAH2**[t*M+1] = $\tilde{\alpha}_j^2 * (\mathbb{H}^2)_T$, **SAH2**[t*M+2] = $\tilde{\alpha}_k^2 * (\mathbb{H}^2)_T$. Note here that if **SAH2** overlap **SAH1** in the shared memory space, the values of **SAH2** are completely rewritten. In this way, the shared memory footprint is not increased as the size of **SAH2** is the same as

SAH1.

8. If thread index $t < N$, fetch data from **SAH1** according to the column indices for the tth row of the gathering matrix, compute the summation, and store the result in registers.

9. Repeat step 7 through 8 for **SAV1**, **SAV2**, **SV**, **SPV**, **SCV** arrays.

10. If thread index $t < N$, compute the new value of node t.

Similarly, for the CPU implementation of the evolution step, we make some modification from the GPU implementation to suit the CPU architecture. We keep a list of elements that are inside the narrowband and perform the computation only on these elements. This is different from the GPU implementation, which updates all elements in a patch as long as any element in the patch is inside the narrowband. We assign the computations of the elements of each patch to a thread instead of each node to a thread to provide a coarse-grained parallelism for CPU. Also, we find that for CPU, atomic operations are efficient enough, so hybrid gathering scheme is not used.

6.3.4 Adaptive Time-step Computation

After each reinitialization step, we perform n update steps for the levelset evolution. In this process, we need to make sure that the evolving levelset does not cross the boundary of the narrowband. According to the Courant-Friedrichs-Lewy condition, the levelset evolution distance of each time-step $\Delta \boldsymbol{x} \leq 2\max_{i<M}(r_i)$, where r_i denotes the inscribed circle or sphere of the ith element and M is the number of elements in the narrowband. Denoting the narrowband width as w, we take a conservative n as $\frac{w}{4\max_{i<M}(r_i)}$ so that the levelset evolves at most half of w. Since the narrowband is changing, $\max(r_i)$ is also changing. Hence, we compute the $\max(r_i)$ adaptively at the beginning of each reinitialization step. r_i are precomputed and stored in an array and the $\max(r_i)$ is computed with a reduction operation. In the evolution step, the time-step Δt is dictated by the three terms of $\mathbf{F}$ in Equation 6.4. We define the time-step as:

$$\Delta t = \min_{i<M}\left(\frac{2r_i}{d(|\boldsymbol{\alpha}| + |\epsilon|)}, \frac{2r_i^2}{d|\kappa|}\right). \tag{6.14}$$

6.4 Results and Discussion

In this section, we present numerical experiments to demonstrate the performance of the proposed algorithms. We use a collection of 2D and 3D unstructured meshes having variable sizes and complexity to illustrate the performance of both CPU and GPU implementations. The performance data as well as implementation related details are provided in the following order: 1) CPU implementation discussion, 2) GPU implementation discussion, followed by 3) the comparison of the two. For consistency of evaluation, double precision is used in all algorithms and for all experiments presented below.

The meshes for the numerical experiments are as follows:

RegSquare: A 2D 512 by 512 square domain regularly triangulated with 524,288 triangles whose maximum valence is six;

IrregSquare: An 2D 512 by 512 square domain irregularly triangulated with 1,181,697 vertices and 2,359,296 triangles whose maximum valence is 14;

Sphere: A triangulated sphere surface with 1,023,260 vertices and 2,046,488 triangles whose maximum valence is 11;

Brain: Triangulated left hemisphere of human brain cortex surface with 631,187 vertices and 1,262,374 triangles (Figure 6.6) whose maximum valence is 19;

RegCube: A 3D regularly tetrahedralized cube with 1,500,282 tetrahedra ($63 \times 63 \times 63$ regular grid) whose maximum valence is 24; and

IrregCube: A 3D irregularly tetrahedralized cube with 197,561 vertices and 1,122,304 tetrahedra whose maximum valence is 54.

These meshes include 2D planar meshes, manifold (surface) meshes, and 3D meshes. They exhibit different geometrical complexity, mesh quality, and maximum nodal valance. Using various meshes allows us to assess the effect that mesh properties have on the algorithm performance.

The numerical simulation setup is as follows: we solve the levelset equation

$$\begin{cases} \dfrac{\partial \phi}{\partial t} + \boldsymbol{\alpha}(\boldsymbol{x}) \cdot \nabla \phi + \epsilon(\boldsymbol{x}) |\nabla \phi| + \beta(\boldsymbol{x}) \nabla \cdot \dfrac{\nabla \phi}{|\nabla \phi|} |\nabla \phi| = 0, \\ \phi(\boldsymbol{x}, t = 0) = g(\boldsymbol{x}), \end{cases} \tag{6.15}$$

where $\boldsymbol{\alpha}(\boldsymbol{x})$ is a user-defined vector function and $\epsilon(\boldsymbol{x})$ and $\beta(\boldsymbol{x})$ are user-defined scalar functions. $g(\boldsymbol{x})$ is the initial condition, which defines the values in the

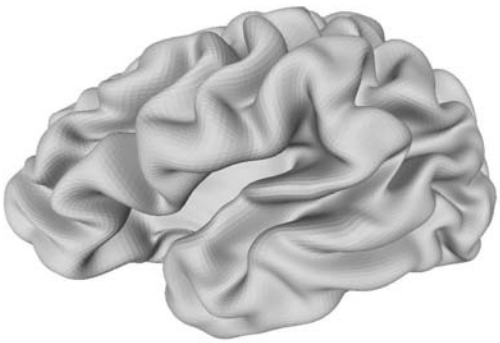


Figure 6.6. Left hemisphere of human brain cortex surface mesh.

domain when time $t = 0$. Computationally, the choice of these constant coefficients makes little difference. In the following numerical experiments, we set the constant coefficients α, ϵ, and β to be (1,0,0), 0.0, and 0.0, respectively, for nonmanifold meshes (RegSuqare, IrregSquare, RegCube, IrregCube). Figure 6.7 shows the result for the RegSquare mesh with these coefficients. The color map indicates the signed distance from the interface. Because the advection term is not well defined on the manifolds, we set the coefficients to be (0,0,0), 0.0, and 1.0 for manifold meshes (Sphere and Brain). Solving the levelset equation with these coefficients gives the geodesic curvature flow, which is widely used in many image processing and computer vision applications [108, 53]. Figure 6.8 shows the geodesic curvature flow on a human brain cortex. The left image demonstrates the initial interface and the right image shows the interface after evolution. We use the numerical scheme presented in [108] to discretize the curvature term on manifolds. Computationally, this scheme is almost the same as the numerical scheme we use for 2D and 3D meshes.

6.4.1 CPU Implementation Results and Performance Analysis

We conduct systematic experiments on a CPU-based parallel system to show the effectiveness and characteristics of our proposed method. We test our CPU implementation on a workstation equipped with two Intel Xeon E5-2640 CPU (12 cores in total) running at 2.5 GHz with turbo boost and hyperthreading enabled

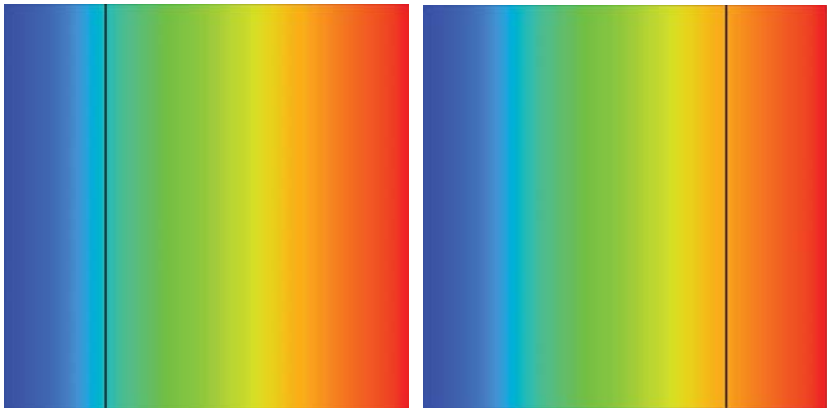


Figure 6.7. The interface on the RegSquare mesh. The left image shows the initial interface and the right image shows the interface after evolution.

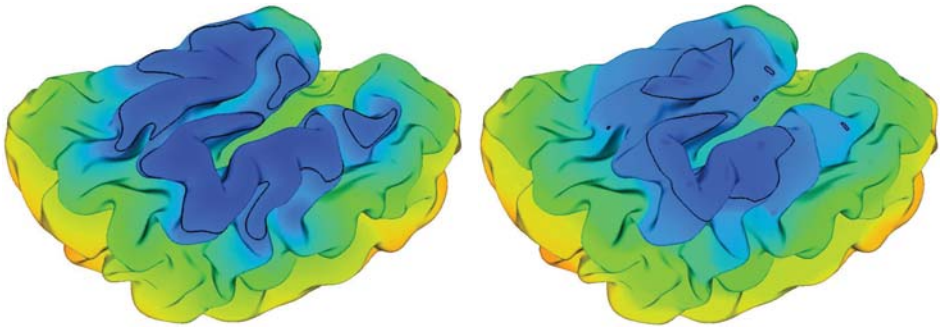


Figure 6.8. The interface on the Brain mesh. The left image shows the initial interface and the right image shows the interface evolution.

and 32GB DDR3 memory shared by the CPUs. The computer is running openSuse 11.4, and the code is compiled with gcc 4.5 using optimization option -O3. Firstly, we run our multithreaded CPU implementation as described in Section 6.3 on the workstation to demonstrate scalability of the proposed method. We compare the result with a naive parallel implementation without patched update schemes (nbFIM and patchNB). In this naive implementation, the nodal computations in the reinitialization and the elemental computations in the evolution are distributed amongst threads and performed in parallel. These computations are not grouped according to patches.

The plots in Figure 6.9 show the strong scaling comparison between the multithreaded CPU implementations with the proposed schemes (Patched) and the

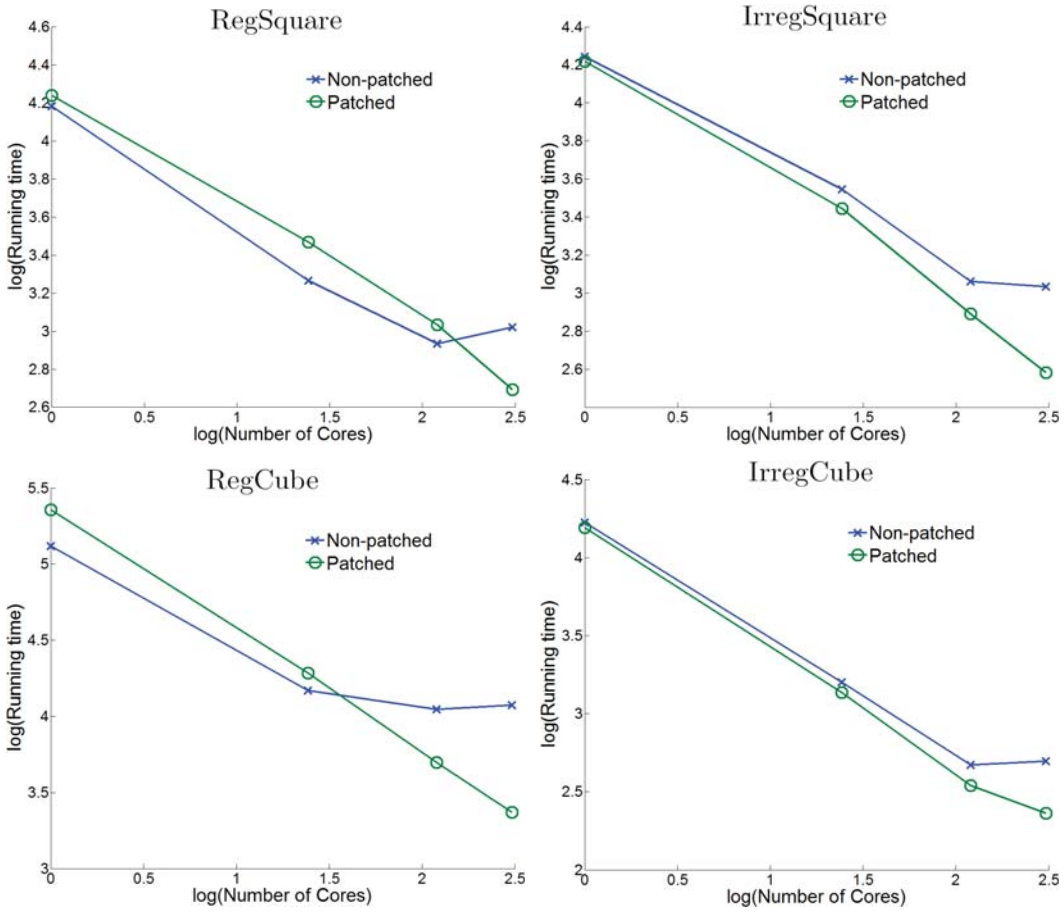


Figure 6.9. Performance comparison between nonpatched CPU implementation and patched implementation.

naive parallel implementation (Nonpatched). We perform this test with two 2D triangular meshes (RegSquare and IrregSquare) and two 3D tetrahedral meshes (RegCube and IrregCube). As shown from the plots, our proposed multithreaded implementation scales up to 12 cores and achieves up to 7× speedup with 12 cores against the serial implementation (with 1 core). By contrast, the nonpatched implementation scales poorly when running with more than four cores, and it does not scale when running with more than eight cores. This supports our claim that with the patched update schemes, each thread accesses data mainly from a single patch (except for boundaries), and in this way, the implementation enforces data locality and achieves better cache performance. In addition, the results show that the proposed implementation scales better on the tetrahedral meshes than on the triangular meshes. This is because for tetrahedral meshes, the number of active

nodes inside each patch is larger in the reinitialization step. The data required by these active node updates are very likely in cache already since each patch is assigned to a thread. Similarly, in 3D cases, the narrowband contains more elements in the evolution step, and each patch within the narrowband has more elements, which leads to more cache hit. Also, for the tetrahedral meshes, the computation is more complicated, and hence the computational density is higher.

For the patched update scheme, the patch size is a factor that may influence the overall performance as it affects how the data are loaded into the cache. With larger patch size, each patch has more active nodes in the reinitialization step, and thus there are more elements inside the larger narrowband for update during the evolution step. These nodes and elements are updated by a single thread, and after a thread updates the first node or element in the patch, the data needed for the following updates are very likely in cache already. However, large patch size may lead to load balancing issue as the workloads of the threads, each updating a corresponding patch, can be very different. Also, if the patch is too large to fit into the cache, the number of cache misses will increase. Table 6.1 shows how the patch size affects the performance of our patched multithreaded CPU implementation. It can be seen from the table that there is a sweet spot for the patch size, which achieves the best balance between the cache performance and load balancing. In our parallel system, this sweet spot is around 64 for all the test meshes, and the CPU results reported in the following subsection (Section 6.4.2) are all with patch size 64.

Table 6.1. Running times (in seconds) to show patch size influence on performance. Bold numbers denote the sweet spot for the patch size.

	size 32	size 64	size 128	size 256
RegSquare	15.40	**14.78**	15.35	18.78
IrregSquare	14.89	**13.22**	15.42	19.22
Sphere	18.15	**15.02**	15.66	17.86
Brain	157.75	142.39	**140.03**	141.89
RegCube	33.08	29.07	**29.05**	30.11
IrregCube	12.89	**10.63**	11.24	11.78

6.4.2 GPU Performance Results

To demonstrate the performance of our proposed schemes on SIMD parallel architectures, we have implemented and tested on an NVIDIA Fermi GPU using the NVIDIA CUDA API [68]. The NVIDIA GeForce GTX 580 graphics card has 1.5 GBytes of global memory and 16 streaming multiprocessors (SM), where each SM consists of 32 SIMD computing cores that run at 1.544 GHz. Each computing core has a configurable 16 or 48 KBytes of on-chip shared memory, for quick access to local data. All codes are compiled with NVCC 4.2. Computation on the GPU entails running a kernel function with a batch process of a large group of fixed size thread blocks, which maps well to the our patched update scheme that employs patch-based update methods, where a single patch is assigned to a CUDA thread block. In this section, we compare our GPU implementation with the multithreaded CPU implementation with patched update scheme.

As described in Section 6.2, the narrowband scheme requires to recompute the distance transform to the zero levelset every few time-steps, and the number of time-steps performed between reinitialization is related to the narrowband width. This width greatly affects the performance of our implementation. When the narrowband width is large, the reinitialization step requires more time to converge, and each evolution step needs to update more nodes that are inside the narrowband. However, with a larger narrowband width, the program needs to perform fewer reinitialization to reach the user-specified total number of time-steps. Table 6.2 shows how performance is related to the narrowband width for the IrregSquare and IrregCube meshes. As seen from the table, there is a narrowband width sweet spot for both CPU and GPU performance that achieves the best balance between the narrowband width and reinitialization frequency. For the CPU, this sweet spot is around five, and for the GPU, it is approximately ten for the IrregSquare mesh. We obtain similar ideal narrowband width for all other triangular meshes. As described in Section 6.3, the reinitialization maintains active patch list instead of active node list, and this makes our GPU reinitialization efficient for larger narrowband width. When the narrowband width is smaller than the patch size, those nodes with values larger than narrowband width are not updated in the

Table 6.2. Running time (in seconds) to show narrowband width influence on performance.

		2	3	5	10	20
IrregSquare	CPU	14.90	14.88	13.22	13.33	15.57
	GPU	3.13	3.14	2.75	2.12	2.46
IrregCube	CPU	13.81	12.90	15.33	55.22	—
	GPU	6.13	5.75	4.94	4.48	7.01

evolution step, and hence their distance values need not to be computed in the reinitialization step. This explains why our GPU implementation prefers a larger narrowband width relative to CPU implementation. For the tetrahedral mesh, although the GPU performance sweet spot is the same, five is no longer the optimal narrowband width for the CPU. For 3D tetrahedral meshes, the cost of the reinitialization is dramatically increased, and smaller narrowband width leads to fewer *local_solver* computations. Although with smaller narrowband width, the frequency of reinitialization is increased, performance improvement from fewer local solver call per iteration outweighs the increased reinitialization frequency. In the following testing results, CPU running times are measured with bandwidth five for triangular meshes and three for tetrahedral meshes, respectively, while GPU running times are measured with narrowband width of ten for both triangle and tetrahedral meshes.

Table 6.3 shows the performance comparison for the time-stepping stage of the levelset equation solver. We present the performance comparison for the reinitialization, evolution and the total running times separately to demonstrate the performance of the proposed schemes for each step. For the reinitialization with the nbFIM scheme, our proposed GPU implementation performs up to 10× faster than the multithreaded CPU implementation running on a 12-core system. For the evolution, the GPU implementation achieves up to 44× speedup over the same 12-core system. The total running times shown here include CPU-GPU data transfer and time-step computation described in Section 6.3.4. In addition, from this table and the tables in Section 6.4.1, we can see that in CPU implementations, the reinitialization step takes a small portion of the total running time while it takes

Table 6.3. Running times (in seconds) for the reinitialization, evolution, and total, respectively. The numbers in the parentheses are the speedups compared against the CPU.

Reinitialization						
	RegSquare	IrregSquare	Sphere	Brain	RegCube	IrregCube
GPU(regular)	0.27(9.7×)	—	—	—	3.03(7.3×)	—
GPU(METIS)	0.65(4.0×)	1.25(3.5×)	1.20(3.8×)	4.91(3.4×)	5.73(3.9×)	3.22(1.4×)
CPU	2.62	4.38	4.53	16.88	22.22	4.53
Evolution						
GPU(regular)	0.26(44×)	—	—	—	1.02(6.4×)	—
GPU(METIS)	0.28(41×)	0.61(14×)	0.46(17×)	2.32(13×)	2.10(3.1×)	1.26(6.3×)
CPU	11.54	8.46	7.90	30.04	6.54	7.93
Total						
GPU(regular)	0.69(21×)	—	—	—	4.33(6.7)	—
GPU(METIS)	1.09(14×)	2.02(6.5×)	1.73(7.6×)	7.48(6.5×)	7.83(3.7×)	4.48(2.9×)
CPU	14.78	13.22	13.13	48.39	29.07	12.90

a large portion in the GPU implementation. This is due to the active-patch scheme for the eikonal solver for GPU as described in Section 6.3, and as a result, the GPU is doing more work than in the CPU implementation for the reinitialization. In addition, the meshes we choose to use in our tests have different maximum valence. Among the triangular meshes, we achieve the greatest GPU over CPU speedup on the RegSquare mesh, because this mesh is regular and has smallest maximum valence. On the other hand, we observe the worst speedup on the IrregSquare and Brain mesh that have larger maximum valences. What is more, we can note from the table that the performance for the 3D tetrahedral meshes is generally worse for both implementations than that for triangular meshes. This is due to the much higher node valence of the tetrahedral meshes and much more complex computations especially in the reinitialization step. We also found that for tetrahedral meshes, the kernel functions for the value update in the reinitialization and evolution steps require more registers than available in hardware, so some local storage is spilled into local memory space that has much higher latency than registers. However, overall, our proposed method suits GPU architecture very well, and our GPU implementation achieves large performance speedup comparing to optimized parallel CPU implementation regardless of mesh complexity.

As mentioned in Section 6.3, lock-free algorithm is usually achieved with atomic

operations. However, on GPU, the atomic operations are expensive. Currently NVIDIA GPU does not support native double precision atomic addition and atomic minimum. We have to rely on atomic compare and swap (atomicCAS) operation to implement the atomic addition and atomic minimum as suggested by [68]. Table 6.4 shows the effectiveness of our lock-free scheme on GPU. This table compares the running times of the GPU implementations with hybrid gathering and atomic operations to solve contention, respectively. This table shows that the implementation with hybrid gathering gives much better performance than the version with atomic operations.

Finally, Figure 6.10 shows how the CPU and GPU implementation scale with different mesh sizes. We start from a coarse mesh with 2048 triangles and subdivide it by connecting the midpoints of each triangle. We do this four times and obtain five meshes with different mesh sizes. The largest mesh is the RegSquare mesh. It can be seen from the plot that as the mesh size increases, the performance gap widens between the CPU implementation and the GPU implementation. The number of computations per time-step increases with mesh size, which makes the GPU operations more efficient. At lower mesh sizes, the performance difference is not as great due to the low computational density per kernel call.

6.5 Conclusions

This work proposes the nbFIM and patchNB schemes to efficiently solve the levelset equation on parallel systems. The proposed schemes combine narrowband scheme and domain decomposition to reduce the computation workload and enforce data locality. Also, combined with our proposed hybrid gathering scheme and novel data structure, these schemes suit the GPU architecture very well and

Table 6.4. Running times (in seconds) for the GPU implementations with hybrid gathering and atomic operations. HG denotes hybrid gathering.

	RegSquare	IrregSquare	Sphere	Brain	RegCube	IrregCube
atomic	2.71	6.03	6.03	14.32	17.24	12.25
HG	0.69	2.12	1.73	6.48	7.83	4.48
speedup	3.9×	2.8×	3.5×	2.2×	2.2×	2.7×

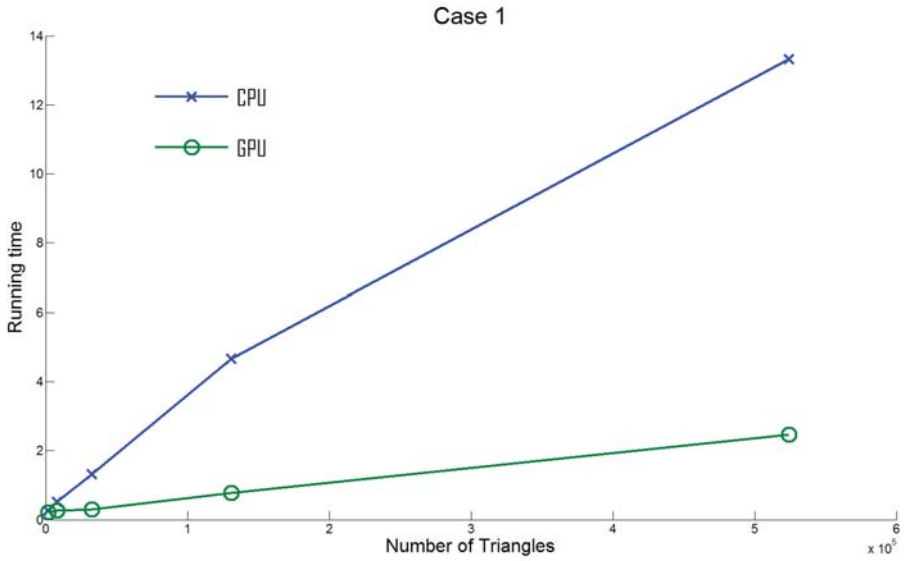


Figure 6.10. Performance comparison between CPU and GPU implementations for different problem sizes.

achieve great performance in a wide range of numerical experiments. The hybrid gathering scheme avoids contention without using atomic operations that are costly on GPUs, and this scheme can be applied to more general problems where data dependency is dictated by a graph structure, and one can choose multiple parallelism strategies corresponding to different graph components to solve these problems. We will explore such problems in our future work. In addition, for many real scientific and engineer applications, a single node computer does not have enough storage for the data or computing power to perform computations efficiently, and thus we will work on extending of the levelset equation solver to multiple GPUs or GPU clusters.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This dissertation describes efficient PDE solvers on unstructured body-fitting meshes for massively-parallel SIMD processors such as GPUs. Unstructured meshes pose challenges for the SIMD architecture, and these challenges are largely unaddressed in the literature. This dissertation has introduced techniques to overcome these challenges and obtained impressive performance gain, comparing to the state-of-the-art CPU or GPU implementations.

This dissertation introduces two general strategies, the domain decomposition and the hybrid gathering, for designing efficient PDE solvers. Based on these strategies, we propose novel algorithms and data structures to efficiently solve two types of PDEs: hyperbolic and elliptic equations. Specifically, this dissertation introduces efficient solvers for the eikonal equation, the Helmholtz equation and the levelset equation. Parabolic PDEs can be also be solved as with our elliptic PDE solver with implicit temporal discretization.

This dissertation focuses on single-GPU solution of PDEs. However, in many practical science and engineering applications, the problem sizes may be too large to fit into a single GPU memory, or the computations are so complicated that running on a single GPU is too slow. Therefore, it would be very useful to develop out-of-core strategy to handle large data on a single GPU. Another way to deal with large problems is to use multiple GPUs or GPU clusters. The domain decomposition strategy used in the PDE solvers proposed in this dissertation has good potential to perform efficiently on multi-GPUs or GPU clusters by mapping multiple subdomains to a GPU.

APPENDIX

PUBLICATIONS

A.1 Papers

- "Fast Iterative Method for Eikonal Equations on Triangular Surface",
  Zhisong Fu, Won-Ki Jeong, Yongsheng Pan, Robert M. Kirby, Ross T. Whitaker,
  SIAM Journal of Scientific Computing, Volume 33 Issue 5, September 2011, Pages 2468-2488.

- "Architecting the Finite Element Method Pipeline for the GPU",
  Zhisong Fu, T. James Lewis, Robert M. Kirby, Ross T. Whitaker,
  accepted upon revision to *Journal of Computational and Applied Mathematics*.

- "A Fast Iterative Method for Solving the Eikonal Equation on Tetrahedral Domains",
  Zhisong Fu, Robert M. Kirby, Ross T. Whitaker,
  in press, *SIAM Journal of Scientific Computing*.

- "Fast Parallel Solver for the Levelset Equations on Unstructured Domains",
  Zhisong Fu, Sergey Yakovlev, Robert M. Kirby, Ross T. Whitaker,
  under review for *Concurrency and Computation: Practice and Experience*.

- "Exploiting Batch Processing on Streaming Architectures to Solve 2D Elliptic Finite Element Problems: A Discontinuous Galerkin Case Study",
  James King, Sergey Yakovlev, Zhisong Fu, Robert M. Kirby, Spencer J. Sherwin,
  accepted upon revision for *Journal of Scientific Computing*.

- "Proper Ordered Meshing of Complex Shapes and Optimal Graph Cuts Applied to Atrial-Wall Segmentation from DE-MRI",

G. Veni, Zhisong Fu, S.P. Awate, R.T. Whitaker,
IEEE Proceedings of ISBI 2013, pp. (accepted). 2013.

- "Globally Optimal, Bayesian Segmentation of Atrium Wall using Graph Cuts on 3D Meshes",
  G. Veni, S. Awate, Zhisong Fu, R.T. Whittaker,
  Proceedings of the International Conference on Information Processing in Medical Imaging (IPMI), Lecture Notes in Computer Science (LNCS), pp. (accepted). 2013.

- "A Comparison of Delaunay-based Meshing Algorithms for Electrophysical Cardiac Simulation",
  Joshua A. Levine, Zhisong Fu, Darrell Swenson, Rob S. MacLeod. and Ross T. Whitaker,
  VPH: Virtual Physiological Human, 2010.

- "The Effect of Non-conformal Finite Element Boundaries on Electrical Monodomain and Bidomain Simulations",
  Darrell Swenson, Joshua A. Levine, Zhisong Fu, Jess Tate, Robert S. MacLeod,
  Computing in Cardiology, 37:–. 2010.

- "A Relaxation Method for Surface-Conforming Prisms",
  Ross T. Whitaker, Robert M. Kirby, Zhisong Fu,
  17th International Meshing Roundtable (research note), 2008.

REFERENCES

[1] CUDA Occupancy Calculator. `http:///developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[2] *Partial Differential Equations: An Introduction*. Wiley, 2007.

[3] Adalsteinsson, D., and Sethian, J. A. A fast level set method for propagating interfaces. *Journal of Computational Physics 118* (1995), 269–277.

[4] Adalsteinsson, D., and Sethian, J. A. A fast level set method for propagating interfaces. *J. Comput. Phys. 118*, 2 (May 1995), 269–277.

[5] Adalsteinsson, D., and Sethian, J. A. Transport and diffusion of material quantities on propagating interfaces via level set. *Journal of Computational Physics 185* (2003), 271–288.

[6] Adams, M., and Demmel, J. Parallel multigrid solver for 3D unstructured finite element problems. In *Proceedings of Supercomputing'99 (CD-ROM)* (Portland, OR, Nov. 1999), ACM SIGARCH and IEEE.

[7] Alon, N., Babai, L., and Itai, A. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms 7* (1986), 567–583.

[8] Baker, A. H., Falgout, R. D., Kolev, T. V., and Yang, U. M. Multigrid smoothers for ultraparallel computing. *SIAM J. Scientific Computing 33*, 5 (2011), 2864–2887.

[9] Barth, T. J., and Sethian, J. A. Numerical schemes for the hamilton-jacobi and level set equations on triangulated domains, 1997.

[10] Barth, T. J., and Sethian, J. A. Numerical schemes for the Hamilton-Jacobi and level set equations on triangulated domains. *Journal of Computational Physics 145* (1998), 1–40.

[11] Bell, N., Dalton, S., and Olson, L. N. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* (2011). in review.

[12] Bell, N., and Garland, M. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[13] Bertsekas, D. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, 1995.

[14] Board, O. A. R., Shreiner, D., and et al. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2*. Addison Wesley, 2005.

[15] Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics 22*, 3 (July 2003), 917–924.

[16] Bornemann, F., and Rasch, C. Finite-element discretization of static Hamilton–Jacobi equations based on a local variational principle. *Computing and Visualization in Science 9*, 2 (July 2006).

[17] Brandt, A., and Livne, O. *Multigrid Techniques: 1984 Guide With Applications to Fluid Dynamics, Revised Edition (Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics, 2011.

[18] Bridson, R. E. *COMPUTATIONAL ASPECTS OF DYNAMIC SURFACES*. PhD thesis, stanford university.

[19] Buatois, L., Caumon, G., and Levy, B. Concurrent number cruncher - a gpu implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems* (to appear).

[20] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 777–786.

[21] Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Langou, J., Luszczek, P., and Tomov, S. Exploiting mixed precision floating point hardware in scientific computations. In *High Performance Computing (HPC) and Grids in Action*, L. Grandinetti, Ed., vol. 16 of *Advances in Parallel Computing*. IOS Press, Amsterdam, Mar. 2008, pp. 19–36.

[22] Cantwell, C., Sherwin, S., Kirby, R., and Kelly, P. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids 43* (2011), 23–28.

[23] Cecil, T. C., Osher, S. J., and Qian, J. Simplex free adaptive tree fast sweeping and evolution methods for solving level set equations in arbitrary dimension. *Journal of Computational Physics 213* (2006), 458–473.

[24] Cecka, C., Lew, A. J., and Darve, E. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering 85* (2011), 640–669.

[25] Cockburn, B., Qian, J., Reitich, F., and Wang, J. An accurate spectral/discontinuous finite-element formulation of a phase-space-based level set approach to geometrical optics. *Journal of Computational Physics 208* (2005), 175–195.

[26] Colli-Franzone, P., and Guerri, L. Spreading of excitation in 3-D models of the anisotropic cardiac tissue I. validation of the eikonal model. *Mathematical Biosciences 113*, 2 (1993), 145–209.

[27] Demmel, J. W. *Applied Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[28] Dziekonski, A., Lamecki, A., and Mrozowski, M. Gpu acceleration of multilevel solvers for analysis of microwave components with finite element method. *Microwave and Wireless Components Letters, IEEE 21*, 1 (2011), 1–3.

[29] Dziekonski, A., Lamecki, A., and Mrozowski, M. Tuning a hybrid gpu-cpu v-cycle multilevel preconditioner for solving large real and complex systems of fem equations. *Antennas and Wireless Propagation Letters, IEEE 10* (2011), 619–622.

[30] Dziekonski, A., Sypek, P., Lamecki, A., and Mrozowski, M. Accuracy, memory, and speed strategies in gpu-based finite-element matrix-generation. *Antennas and Wireless Propagation Letters, IEEE 11* (2012), 1346–1349.

[31] Dziekonski, A., Sypek, P., Lamecki, A., and Mrozowski, M. Finite element matrix generation on a gpu. *Progress In Electromagnetics Research 128* (2012), 249–265.

[32] Dziekonski, A., Sypek, P., Lamecki, A., and Mrozowski, M. Generation of large finite-element matrices on multiple graphics processors. *International Journal for Numerical Methods in Engineering 94*, 2 (2013), 204–220.

[33] Emans, M., and van der Meer, A. Mixed-precision AMG as linear equation solver for definite systems. *Procedia CS 1*, 1 (2010), 175–183.

[34] Falcone, M. A numerical approach to the infinite horizon problem of deterministic control theory. *Appl. Math. Optim. 15* (1987), 1–13.

[35] Falcone, M. The minimum time problem and its applications to front propagation. In *Motion by Mean Curvature and Related Topics* (Berlin, 1994), de Gruyter, pp. 70–88.

[36] Fortmeier, O., and Bücker, H. M. A parallel strategy for a level set simulation of droplets moving in a liquid medium. In *Proceedings of the 9th international conference on High performance computing for computational science* (Berlin, Heidelberg, 2011), VECPAR'10, Springer-Verlag, pp. 200–209.

[37] Fu, Z., Jeong, W.-K., Pan, Y., Kirby, R. M., and Whitaker, R. T. A fast iterative method for solving the eikonal equation on triangulated surfaces. *SIAM Journal of Scientific Computing 33*, 5 (Oct. 2011), 2468–2488.

[38] Fu, Z., Kirby, R. M., and Whitaker, R. T. A fast iterative method for solving the eikonal equation on tetrahedral domains. *SIAM Journal of Scientific Computing* (2013). Accepted upon revision.

[39] Fu, Z., Lewis, T. J., Kirby, R. M., and Whitaker, R. T. Architecting the finite element method pipeline for the gpu. *accepted upon revision to Journal of Computational and Applied Mathematics*. Accepted upon revision.

[40] Fu, Z., Sergey, Y., Kirby, R. M., and Whitaker, R. T. Fast parallel solver for the levelset equations on unstructured domains. *under review for Concurrency and Computation: Practice and Experience*. Accepted upon revision.

[41] Geveler, M., Ribbrock, D., Goeddeke, D., Zajac, P., and Turek, S. Towards a complete fem-based simulation toolkit on gpus: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses. *Computers & Fluids* (2012).

[42] Giles, M. B., Mudalige, G. R., Sharif, Z., Markall, G., and Kelly, P. H. Performance analysis of the op2 framework on many-core architectures. *SIGMETRICS Perform. Eval. Rev. 38*, 4 (Mar. 2011), 9–15.

[43] Greivenkamp, J. E. *Field Guide to Geometrical Optics*. SPIE Publications, 2003.

[44] Haase, G., Liebmann, M., Douglas, C. C., and Plank, G. A parallel algebraic multigrid solver on graphics processing units. In *HPCA (China)* (2009), W. Zhang, Z. Chen, C. C. Douglas, and W. Tong, Eds., vol. 5938 of *Lecture Notes in Computer Science*, Springer, pp. 38–47.

[45] Henson, V. E., and Yang, U. M. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics: Transactions of IMACS 41*, 1 (Apr. 2002), 155–177.

[46] Herrmann, M. A domain decomposition parallelization of the fast marching method. *Center for Turbulence Research Annual Research Briefs* (2003), 213–225.

[47] Holm, D. D. *Geometric Mechanics: Part I: Dynamics and Symmetry (2^{nd} edition)*. Imperial College London Press, London, UK, 2011.

[48] Houston, B., Nielsen, M. B., Batty, C., Nilsson, O., and Museth, K. Hierarchical rle level set: A compact and versatile deformable surface representation. *ACM Trans. Graph. 25*, 1 (Jan. 2006), 151–175.

[49] Hughes, T. J. R. *The finite element method: linear static and dynamic finite element analysis*. Prentice-Hall, 1987.

[50] Jeong, W.-K., Beyer, J., Hadwiger, M., Vazquez, A., Pfister, H., and Whitaker, R. T. Scalable and interactive segmentation and visualization of neural processes in em datasets. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (Nov. 2009), 1505–1514.

[51] Jeong, W.-K., Fletcher, P. T., Tao, R., and Whitaker, R. Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware Hamilton–Jacobi solver. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (Nov./Dec. 2007), 1480–1487.

[52] Jeong, W.-K., and Whitaker, R. T. A fast iterative method for eikonal equations. *SIAM Journal of Scientific Computing 30*, 5 (2008), 2512–2534.

[53] Joshi, A., Shattuck, D., Damasio, H., and Leahy, R. Geodesic curvature flow on surfaces for automatic sulcal delineation. In *Biomedical Imaging (ISBI), 2012 9th IEEE International Symposium on* (may 2012), pp. 430 –433.

[54] Karniadakis, G., and Sherwin, S. J. *Spectral/hp element methods for CFD*. Numerical mathematics and scientific computation. Oxford University Press, 1999.

[55] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1998), 359–392.

[56] Keener, J. P. An eikonal equation for action potential propagation in myocardium. *J. Math. Biol. 29* (1991), 629–651.

[57] Kimmel, R., and Sethian, J. A. Computing geodesic paths on manifolds. In *Proc. Natl. Acad. Sci. USA* (1998), vol. 95, pp. 8431–8435.

[58] Kiss, I., Gyimothy, S., Badics, Z., and Pavo, J. Parallel realization of the element-by-element fem technique by cuda. *Magnetics, IEEE Transactions on 48*, 2 (2012), 507–510.

[59] Klöckner, A., Warburton, T., Bridge, J., and Hesthaven, J. S. Nodal discontinuous galerkin methods on graphics processors. *J. Comput. Physics 228*, 21 (2009), 7863–7882.

[60] Komatitsch, D., Michéa, D., and Erlebacher, G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *J. Parallel Distrib. Comput 69*, 5 (2009), 451–460.

[61] Lefohn, A., Cates, J., and Whitaker, R. Interactive, gpu-based level sets for 3d brain tumor segmentation. In *in Medical Image Computing and Computer Assisted Intervention* (2003), In: MICCAI, pp. 564–572.

[62] Li, R., and Saad, Y. Gpu-accelerated preconditioned iterative linear solvers. Tech. rep., Technical report, University of Minnesota, 2010.

[63] LLNL. Hypre library.

[64] Markall, G., Slemmer, A., Ham, D., Kelly, P., Cantwell, C., and Sherwin, S. Finite element assembly strategies on multi-core and many-core architectures. *Int. J. Numer. Meth. Fluids 71* (2013), 80–97.

[65] McCormick, S. F., Ed. *Multigrid Methods*. SIAM Publications, 1987.

[66] Meuer, H., Strohmaier, E., Dongarra, J., and Simon, H. Top500 supercomputer sites. http://www.top500.org/.

[67] NVIDIA. http://https://developer.nvidia.com/thrust.

[68] NVIDIA. Cuda programming guide. http://www.nvidia.com/object/cuda.html.

[69] NVIDIA. Tesla c2050 / c2070 gpu computing processor. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf.

[70] NVIDIA. Cusp library.

[71] NVIDIA. Nvidias next generation cuda compute architecture: Fermi.

[72] Osher, S., Cheng, L.-T., Kang, M., Shim, H., and Tsai, Y.-H. Geometric optics in a phase-space-based level set and Eulerian framework. *Journal of Computational Physics 179* (2002), 622–648.

[73] Osher, S., and Sethian, J. A. Fronts propagating with curvature dependent speed: Algorithms based on hamilton-jacobi formulations. *JOURNAL OF COMPUTATIONAL PHYSICS 79*, 1 (1988), 12–49.

[74] Otani, N. F. Computer modeling in cardiac electrophysiology. *Journal of Computational Physics 161* (2010), 21–34.

[75] Pharr, M., and Fernando, R., Eds. *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley, 2005.

[76] Pichon, E., and Tannenbaum, A. Curve segmentation using directional information, relation to pattern detection. In *ICIP (2)* (2005), pp. 794–797.

[77] Polymenakos, L. C., Bertsekas, D. P., and Tsitsiklis, J. N. Implementation of efficient algorithms for globally optimal trajectories, Jan. 27 1998.

[78] Potse, M., Dube, B., Richer, J., Vinet, A., and Gulrajani, R. A comparison of monodomain and bidomain reaction-diffusion models for action potential propagation in the human heart. *Biomedical Engineering, IEEE Transactions on 53*, 12 (2006), 2425–2435.

[79] Qian, J., Zhang, Y., and Zhao, H. Fast sweeping methods for eikonal equations on triangulated meshes. *SIAM Journal on Numerical Analysis 45*, 1 (2007), 83–107.

[80] Qin, F., Luo, Y., Olsen, K. B., Cai, W., and Schuster, G. T. Finite-difference solution of the eikonal equation along expanding wavefronts. *Geophysics 57*, 3 (2009), 478.

[81] Rawlinson, N., and Sambridge, M. The fast marching method: an effective tool for tomographics imaging and tracking multiple phases in complex layered media. *Exploration Geophysics 36* (2005), 341–350.

[82] Rawlinson, R., and Sambridge, M. Wave front evolution in strongly heterogeneous layered media using the fast marching method. *Geophys. J. Internat. 156* (2004), 631–647.

[83] Rodríguez-Navarro, J., and Sánchez, A. S. Non structured meshes for cloth GPU simulation using FEM, 2006.

[84] Rouy, E., and Tourin, A. A viscosity solutions approach to shape-from-shading. *SIAM Journal of Numerical Analysis 29* (June 1992), 867–884.

[85] Saad, Y. *Iterative methods for sparse linear systems*, 2 ed. SIAM, 2003.

[86] Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore gpus. NVIDIA Technical Report NVR-2008-001, NVIDIA Corporation, Sept. 2008.

[87] Sethian, J. Evolution, implementation, and application of level set and fast marching methods for advancing fronts. *Journal of Computational Physics 169* (2001), 503–555.

[88] Sethian, J. A. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences of the United States of America 93*, 4 (1996), 1591–1595.

[89] Sethian, J. A. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999.

[90] Sethian, J. A., and Vladimirsky, A. Ordered upwind methods for static Hamilton–Jacobi equations: Theory and algorithms. *SIAM Journal on Numerical Analysis 41*, 1 (Feb. 2003), 325–363.

[91] Sheriff, R., and Geldart, L. *Exploration Seismology*. Cambridge University Press, 1995.

[92] Sifri, O., Sheffer, A., and Gotsman, C. Geodesic-based surface remeshing. In *IMR* (2003), pp. 189–199.

[93] Smith, B., Bjorstad, P., and Gropp, W. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, 1996.

[94] Spira, A., and Kimmel, R. An efficient solution to the eikonal equation on parametric manifolds. *Interfaces Free Bound. 6* (2004), 315–327.

[95] Srinark, T., and Kambhamettu, C. A novel method for 3D surface mesh segmentation. In *Computer Graphics and Imaging* (2003), IASTED/ACTA Press, pp. 212–217.

[96] Strain, J. Tree methods for moving interfaces. *Journal of Computational Physics 151*, 2 (1999), 616 – 648.

[97] Swenson, D., Levine, J., Fu, Z., Tate, J., and MacLeod, R. The effect of non-conformal finite element boundaries on electrical monodomain and bidomain simulations. In *Computing in Cardiology, 2010* (2010), pp. 97–100.

[98] Tan, L., and Zabaras, N. A level set simulation of dendritic solidification of multi-component alloys. *J. Comput. Phys. 221*, 1 (Jan. 2007), 9–40.

[99] Trefethen, L. N., and III, D. B. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[100] Tsai, Y., Cheng, L., Osher, S., and Zhao, H. Fast sweeping algorithms for a class of Hamilton–Jacobi equations. *SIAM Journal on Numerical Analysis 41* (2003).

[101] Tugurlan, M. C. *Fast Marching Methods-Parallel Implementation and Analysis*. PhD thesis, Louisiana State University, 2008.

[102] Tuminaro, R. S., and Tong, C. Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In *in SuperComputing 2000 Proceedings* (2000).

[103] Vanek, P., Mandel, J., and Brezina, M. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing 56*, 3 (1996), 179–196.

[104] Vos, P. E. J., Sherwin, S. J., and Kirby, R. M. From *h* to *p* efficiently: Implementing finite and spectral *hp* element methods to achieve optimal performance for low and high order discretisations. *Journal of Computational Physics 229* (2010).

[105] Wang, L., Hu, X., Cohen, J., and Xu, J. A parallel auxiliary grid algebraic multigrid method for graphic processing units. *SIAM Journal on Scientific Computing 35*, 3 (2013), C263–C283.

[106] Whitaker, R. T. A level-set approach to 3d reconstruction from range data. *International Journal of Computer Vision 29* (1998), 203–231.

[107] Wikipedia. Supercomputer. http://en.wikipedia.org/wiki/Supercomputer.

[108] Wu, C., and Tai, X. A level set formulation of geodesic curvature flow on simplicial surfaces. *Visualization and Computer Graphics, IEEE Transactions on 16*, 4 (july-aug. 2010), 647 –662.

[109] Zhang, H., van Kaick, O., and Dyer, R. Spectral methods for mesh processing and analysis. In *STAR Proceedings of Eurographics 2007* (Prague, 2007), D. Schmalstieg and J. Bittner, Eds., Eurographics Association, pp. 1–22.

[110] Zhao, H. A fast sweeping method for eikonal equations. *Mathematics of Computation 74*, 250 (Apr. 2005), 603–627.

[111] Zhao, H. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics 25*, 4 (2007), 421–429.