Morph Algorithms on GPUs

Rupesh Nasre[1] Martin Burtscher[2] Keshav Pingali[1,3]

[1]Inst. for Computational Engineering and Sciences, University of Texas at Austin, USA
[2] Dept. of Computer Science, Texas State University, San Marcos, USA
[3] Dept. of Computer Science, University of Texas at Austin, USA
nasre@ices.utexas.edu, burtscher@txstate.edu, pingali@cs.utexas.edu

Abstract

There is growing interest in using GPUs to accelerate graph algorithms such as breadth-first search, computing page-ranks, and finding shortest paths. However, these algorithms do not modify the graph structure, so their implementation is relatively easy compared to general graph algorithms like mesh generation and refinement, which *morph* the underlying graph in non-trivial ways by adding and removing nodes and edges. We know relatively little about how to implement morph algorithms efficiently on GPUs.

In this paper, we present and study four morph algorithms: (i) a computational geometry algorithm called Delaunay Mesh Refinement (DMR), (ii) an approximate SAT solver called Survey Propagation (SP), (iii) a compiler analysis called Points-to Analysis (PTA), and (iv) Boruvka's Minimum Spanning Tree algorithm (MST). Each of these algorithms modifies the graph data structure in different ways and thus poses interesting challenges.

We overcome these challenges using algorithmic and GPU-specific optimizations. We propose efficient techniques to perform concurrent subgraph addition, subgraph deletion, conflict detection and several optimizations to improve the scalability of morph algorithms. For an input mesh with 10 million triangles, our DMR code achieves an 80× speedup over the highly optimized serial *Triangle* program and a 2.3× speedup over a multicore implementation running with 48 threads. Our SP code is 3× faster than a multicore implementation with 48 threads on an input with 1 million literals. The PTA implementation is able to analyze six SPEC 2000 benchmark programs in just 74 milliseconds, achieving a geometric mean speedup of 9.3× over a 48-thread multicore version. Our MST code is slower than a multicore version with 48 threads for sparse graphs but significantly faster for denser graphs.

This work provides several insights into how other morph algorithms can be efficiently implemented on GPUs.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Languages, Performance

Keywords Morph Algorithms, Graph Algorithms, Irregular Programs, GPU, CUDA, Delaunay Mesh Refinement, Survey Propagation, Minimum Spanning Tree, Boruvka, Points-to Analysis

1. Introduction

Graphics Processing Units (GPUs) have been shown to be very useful in many application domains outside of graphics. GPU hardware is designed to process blocks of pixels at high speed and with wide parallelism, so it is well suited for executing regular algorithms that operate on dense vectors and matrices. We understand much less about how to use GPUs effectively to execute *irregular* algorithms that use dynamic data structures like graphs and trees. Harish and Narayanan [10] pioneered this field with their CUDA implementations of algorithms such as breadth-first search and single-source shortest paths. BFS has recently received much attention in the GPU community [9, 12, 17, 20]. Barnat *et al.* [3] implemented a GPU algorithm for finding strongly-connected components in directed graphs and showed that it achieves significant speedup with respect to Tarjan's sequential algorithm. Other irregular algorithms that have been successfully parallelized for GPUs are *n*-body simulations and dataflow analyses [5, 18, 25].

An important characteristic of most of the irregular GPU algorithms implemented to date is that they are graph analysis algorithms that do not modify the structure of the underlying graph [3, 10, 12, 17, 20]. When they do modify the graph structure, the modifications can be predicted statically and appropriate data structures can be pre-allocated [5, 25]. However, there are many important graph algorithms in which edges or nodes are dynamically added to or removed from the graph in an unpredictable fashion, such as mesh refinement [7] and compiler optimization [1]. Recently, Mendez-Lojo *et al.* described a GPU implementation of Andersen-style points-to analysis. In this algorithm, the number of edges strictly increases during the computation. However, we are unaware of a high-performance GPU implementation of an irregular graph algorithm that adds and removes nodes and edges. In TAO analysis [24] – an algorithmic classification for irregular codes – these are called *morph* algorithms. Implementation of a morph algorithm on a GPU is challenging because it is unclear how to support dynamically changing graphs while still achieving good performance.

In this paper, we describe efficient GPU implementations of four morph algorithms: (i) Delaunay Mesh Refinement (DMR) [7], which takes a triangulated mesh as input and modifies it in place by adding and removing triangles to produce a triangulated mesh satisfying certain geometric constraints; (ii) Survey Propagation (SP) [4], an approximate SAT solver that takes a k-SAT formula as input, constructs a bipartite factor graph over its literals and constraints, propagates probabilities along its edges, and occasionally deletes a node when its associated probability is close enough to 0 or 1; (iii) Points-to Analysis (PTA) [1], which takes a set of points-to constraints derived from a C program, creates a constraint graph whose nodes correspond to program variables and whose directed edges correspond to the flow of points-to information, and iteratively computes a fixed-point solution by propagating the points-to information and adding the corresponding edges; and (iv) Boruvka's Minimum Spanning Tree algorithm (MST), which operates

PPoPP'13, February 23–27, 2013, Shenzhen, China.
Copyright © 2013 ACM 978-1-4503-1922/13/02…$15.00

on an undirected input graph and relies on the process of minimum edge contraction, which involves merging of the adjacency lists of the edge's endpoints, until an MST is formed. Each of these algorithms poses different and varying levels of challenges for a massively parallel architecture like a GPU, as we discuss next.

- In DMR, triangles are added and deleted on the fly, thus requiring careful attention to synchronization. Another challenge is memory allocation as the number of triangles added or deleted cannot be predicted *a priori*. Allocating storage statically may result in over-allocation whereas dynamic allocation incurs runtime overhead. Further, the amount of work tends to be different in different parts of the mesh, which may lead to work imbalance. During refinement, new bad triangles may be created, which can lead to an unpredictable work distribution.
- In SP, the situation is simpler as the number of nodes only decreases. Deletion of a node involves removal of a node and its edges. However, implementing subgraph deletion in a highly concurrent setting is costly due to synchronization overhead.
- Although the number of nodes is fixed in PTA (it is equal to the number of variables in the input program), the number of edges in the constraint graph increases monotonically and unpredictably. Therefore, we cannot use a statically allocated storage scheme to represent the dynamically growing graph.
- In MST, the adjacency lists of two nodes need to be merged during each edge contraction. While edge merging can be done explicitly in small graphs, it is too costly for large graphs, especially in the later iterations of the algorithm. Edge merging can also skew the work distribution.

We address these challenges by proposing several mechanisms that are novel in the context of irregular algorithms; many of them are applicable to the implementation of other morph and non-morph algorithms on GPUs. We contrast our approach with alternative ways of addressing these challenges. The contributions of this paper are as follows.

- Many morph algorithms have to deal with neighborhood conflicts. For efficient execution, the corresponding implementation needs to be cautious [19]. We propose a GPU-friendly mechanism for conflict detection and resolution.
- Addition and deletion of arbitrary subgraphs can be implemented in several ways. We present a qualitative comparison of different mechanisms and provide guidelines for choosing the best implementation.
- Neighborhood conflicts lead to aborted work, resulting in wasted parallelism. We propose an adaptive scheme for changing the kernel configuration to reduce the abort ratio, thus leading to improved work efficiency.
- Information can be propagated in a directed graph either in a *push* or a *pull* manner, *i.e.,* from a node to its outgoing neighbors or from the incoming neighbors to the node, respectively. We compare the two approaches and observe that a *pull* model usually results in reduced synchronization.
- The amount of parallelism present in morph algorithms often changes considerably during their execution. A naive work distribution can lead to load imbalance. We propose better ways to distribute tasks across threads, which improves system utilization and, in turn, performance.
- To avoid the bottleneck of a centralized worklist and to propagate information faster in a graph, we propose using local worklists, which can be efficiently implemented in shared memory.
- Our GPU codes, written in CUDA, outperform existing state-of-the-art multicore implementations. Our DMR code is faster than a 48-core CPU version of the same algorithm [16], achieving up to 2.3× speedup. With respect to the highly optimized and widely-used sequential CPU implementation of the Triangle program [28], our implementation achieves up to 80× speedup. Our SP code is 3× and the PTA code is 9.3× faster than their multicore counterparts. Our GPU implementation of MST is slower for sparse graphs (like road networks) but significantly faster for denser graphs (like RMAT).

The rest of this paper is organized as follows. We introduce Delaunay Mesh Refinement, Survey Propagation, Points-to Analysis and Boruvka's Minimum Spanning Tree algorithm and discuss the sources of parallelism in these algorithms in Sections 2 through 5. We describe our graph representation and its memory layout in Section 6. We discuss efficient implementations of morph algorithms in Section 7, where we explain generic techniques for subgraph addition and deletion, conflict detection and resolution, improving work efficiency, and using local worklists. The experimental evaluation comparing the performance of the GPU, multicore and sequential implementations is discussed in Section 8. We compare and contrast with related work in Section 9 and conclude in Section 10.

2. Delaunay Mesh Refinement

Mesh generation and refinement are important components of applications in many areas such as the numerical solution of partial differential equations and graphics.

DMR Algorithm The goal of mesh generation is to represent a surface or a volume as a tessellation composed of simple shapes like triangles, tetrahedra, *etc*. Although many types of meshes are in use, Delaunay meshes are particularly important since they have a number of desirable mathematical properties [7]. The Delaunay triangulation for a set of points in the plane is the triangulation such that none of the points lie inside the circumcircle of any triangle.

In practice, the Delaunay property alone is not sufficient, and it is necessary to impose quality constraints governing the shape and size of the triangles. For a given Delaunay mesh, this is accomplished by iterative mesh refinement, which successively fixes *bad* triangles (triangles that do not satisfy the quality constraints) by adding new points to the mesh and re-triangulating the affected areas. Figure 1 illustrates this process; the shaded triangle is assumed to be bad. To fix it, a new point is added at the circumcenter of this triangle. Adding this point may invalidate the empty circumcircle property for some neighboring triangles. Therefore, the affected triangles around the bad triangle are determined. This region is called the *cavity* of the bad triangle. The cavity is re-triangulated, as shown in Figure 1(c) (in this figure, all triangles lie in the cavity of the shaded bad triangle). Re-triangulating a cavity may generate new bad triangles, but the iterative refinement process will ultimately terminate and produce a guaranteed-quality mesh. Different orders of processing bad elements lead to different meshes, but all of them satisfy the quality constraints [7].

Amorphous data-parallelism in DMR The natural unit of work for parallel execution is the processing of a bad triangle. Because a cavity is usually just a small neighborhood of triangles around the bad triangle (typically 4 to 10 triangles), two bad triangles that are far apart in the mesh typically have cavities that do not overlap. Furthermore, the entire refinement process (expansion, retriangulation, and graph updating) for the two triangles is completely independent. Thus, the two triangles can be processed in parallel. This approach obviously extends to more than two triangles. However, if the cavities of two triangles do overlap, the triangles can

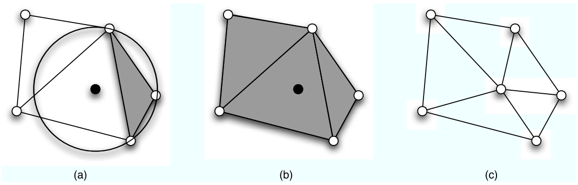


Figure 1. Refinement: Fixing a bad triangle by modifying its cavity. (a) A bad triangle (shaded) and its circumcircle indicating the overlapping triangles that form the cavity. (b) Cavity of the bad triangle and new point. (c) Refined cavity formed using the new point.

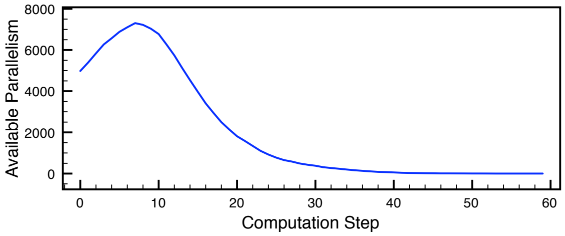


Figure 2. Parallelism profile of Delaunay Mesh Refinement

be processed in either order but only one of them can be processed at a time. Whether or not two bad triangles have overlapping cavities depends entirely on the structure of the mesh, which changes throughout the execution of the algorithm. Figure 2 shows a parallelism profile for DMR produced by the ParaMeter tool [15]. The profile was obtained by running DMR on a randomly generated input mesh consisting of 100K triangles, half of which are initially bad. The amount of parallelism changes significantly during the execution of the algorithm, and this has an impact on how we implement DMR as we discuss in Section 7.4. Initially, there are about 5,000 bad triangles that can be processed in parallel. This number increases as the computation progresses, peaking at over 7,000 triangles, after which point the available parallelism drops slowly. The amount of parallelism is higher for larger inputs.

GPU Implementation The DMR algorithm we use on the GPU is slightly different from the multicore algorithm described above. The pseudo code of our implementation is shown in Figure 3. The comments indicate if the relevant code is executed on the CPU or the GPU, or whether it is a data transfer between the two devices.

The main program starts with the host code (on the CPU) reading an input mesh from a file. The mesh is initially stored in the CPU's memory and has to be transferred to the GPU via an explicit call to the `cudaMemcpy()` function. Then the GPU performs a few initialization steps. These include resetting the deletion flag of each triangle, identifying if a triangle is bad, and computing the neighborhood of each bad triangle. Next, the host code repeatedly invokes the refinement kernel, which re-triangulates the mesh. In each invocation, a thread operates on a bad undeleted triangle, creates a cavity around that triangle, re-triangulates the cavity, marks the triangles in the old cavity as deleted, and adds the new triangles to the mesh. If any of the new triangles are bad, the thread sets a flag *changed* to inform the host that another kernel invocation is necessary. Thus, the host code only stops invoking the GPU kernel once there are no more bad triangles to process. At this stage, the refined mesh is transferred back to the CPU using another call to `cudaMemcpy()`. We emphasize that all of the refinement code executes on the GPU. The CPU is responsible only for reading the input mesh, transferring it to the GPU, repeatedly invoking the GPU kernel, and transferring the refined mesh back.

```
main():
  read input mesh                        // CPU
  transfer initial mesh                  // CPU → GPU
  initialize_kernel()                    // GPU
  do {
      refine_kernel()                    // GPU
      transfer changed                   // GPU → CPU
  } while changed
  transfer refined mesh                  // GPU → CPU

refine_kernel():
  foreach triangle t in my worklist {
    if t is bad and not deleted {
      create and expand cavity around t
      mark all triangles in the cavity with my thread id
      __global_sync()
      check and re-mark with priority     // Section 7.3
      __global_sync()
      if all cavity-triangles are marked with my thread id {
        create new cavity by retriangulating
        delete triangles in old cavity from mesh
        add triangles from new cavity to mesh
        if any new triangle is bad {
          changed = true
        }
      } else {                           // back-off
        changed = true
} } }
```

Figure 3. Pseudo code of our DMR implementation

Our approach is *topology-driven* [24], *i.e.,* threads process both good and bad triangles. In contrast to a *data-driven* approach [24], which processes only the bad triangles, a topology-driven approach may perform more work and threads may not have useful work to do at every step. However, a data-driven approach requires maintenance of a worklist that is accessed by all threads. A naive implementation of such a worklist severely limits performance because work elements must be added and removed atomically. Therefore, we use a topology-driven approach and add optimizations to improve work efficiency (*cf.* Section 7.4). Topology-driven approaches have also been used in GPU implementations of other algorithms [18].

3. Survey Propagation

Survey Propagation is a heuristic SAT solver based on Bayesian inference [4]. The algorithm represents a Boolean SAT formula as a *factor graph*, which is a bipartite graph with literals on one side and clauses on the other. An undirected edge connects a literal to a clause if the literal participates in the clause. The edge is given a value of -1 if the literal in the clause is negated, and +1 otherwise. The general strategy of SP is to iteratively update each literal with the likelihood that it should be assigned a truth value of *true* or *false*. The amorphous data-parallelism in this algorithm arises from the literals and clauses that need to be processed. Although there are no ordering constraints on processing the elements, different orders may lead to different behavior. An example 3-SAT formula and the corresponding factor graph are given in Figure 4.

SP has been shown to be quite effective for hard SAT instances [4]. For $K = 3$, *i.e.,* when the number of literals per clause is three, a SAT instance becomes hard when the clause-to-literal ratio is close to 4.2. We focus on hard SAT problems in this work.

SP Algorithm The SP algorithm proceeds as follows. Each phase of the algorithm first iterates over the clauses and the literals of the formula updating 'surveys' until all updates are below some small epsilon. Then, the surveys are processed to find the most biased literals, which are fixed to the appropriate value. The fixed literals are then removed from the graph. If only trivial surveys remain or

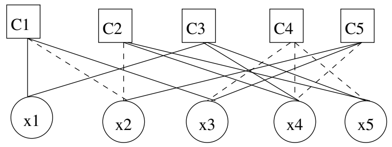


Figure 4. Factor graph of $(x_1 + \overline{x_2} + x_3)(\overline{x_2} + x_4 + x_5)(x_1 + x_4 + x_5)(\overline{x_3} + \overline{x_4} + \overline{x_5})(x_2 + x_3 + \overline{x_4})$. Dotted edges are negated.

the number of literals is small enough, the problem is passed on to a simpler solver. Otherwise, the algorithm starts over with the reduced graph as input. It performs as many such phases as are necessary until no more literals can be fixed. If there is no progress after some number of iterations, the algorithm gives up.

Amorphous data-parallelism in SP The active nodes are the literals in the factor graph. Initially every literal node is active. If the surveys on some edges change during processing, the nodes' neighbors' neighbors (*i.e.,* other literals in common clauses) become active. Thus, the neighborhood of each active node encompasses the neighbors and neighbors' neighbors.

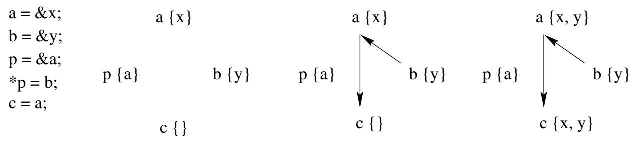
Each iteration of SP touches a single node's neighborhood in the factor graph. Iterations conflict with one another if their neighborhoods overlap. The structure of the graph is mostly constant, except that nodes are removed occasionally when their values become stable (*i.e.,* biased towards 0 or 1). Thus, the available parallelism reflects the connectivity of the graph and remains roughly constant, dropping infrequently as nodes are removed from the graph. SP is a heuristic SAT solver, and it behaves non-deterministically.

GPU Implementation Our implementation of SP first generates a random SAT formula based on the input number of clauses and literals. After initialization, the surveys are propagated through the edges in the factor graph until the surveys stabilize or until a fixed number of iterations has been performed. Next, the literals are updated to reflect their bias towards their neighbors (the clauses) for satisfiability. If the kernel updates the bias of any literals, a process of decimation kicks in, which removes any literals whose bias is close to *true* or *false*. Once the biased literals have been removed, the reduced factor graph is processed again, continuing with the current state of the surveys and biases.

4. Points-to Analysis

Points-to analysis is a key static compiler analysis. We implement a variant of the flow-insensitive, context-insensitive, inclusion-based points-to analysis [1]. It works on the points-to constraints obtained from the input program's statements and computes a fixed-point points-to solution based on the constraints.

PTA Algorithm PTA operates on a constraint graph in which nodes correspond to pointers and each directed edge between two nodes represents a subset relationship between the points-to sets of the corresponding two pointers. In other words, the points-to information *flows* from the source to the target node along an edge. There are four kinds of constraints: address-of (`p = &q`), copy (`p = q`), load (`p = *q`) and store (`*p = q`). The address-of constraints determine the initial points-to information in the constraint graph and the other three types of constraints add edges. The points-to information is then propagated along these edges until convergence. Due to the accumulation of new points-to information at some nodes, the load and the store constraints add more edges to the constraint graph and create additional opportunities for propagation. This process continues until a fixed-point is reached. The final points-to information is then available at the pointer nodes. An example set of constraints and the state of the constraint graph in different stages of the algorithm are shown in Figure 5.



```
a = &x;
b = &y;
p = &a;
*p = b;
c = a;
```

Figure 5. Points-to constraints and states of the constraint graph

Amorphous data-parallelism in PTA PTA has been shown to exhibit a moderate amount of parallelism [18, 26]. Its source of parallelism is the optimistic propagation of points-to information along edges that do not have common nodes. Although the amorphous data-parallelism in PTA is input dependent, it usually increases gradually at first, remains high for some time and then slowly decreases (similar to DMR). Although conflicts can be handled using atomics, we employ a novel pull-based approach that avoids synchronization (*cf.* Section 6.4).

GPU Implementation Our PTA implementation first extracts points-to constraints from an input C/C++ program and transfers them to the GPU. The initialization kernel initializes each pointer with a 'null' points-to set. Processing of each constraint happens in two phases. In the first phase, the constraints are evaluated to determine their source and destination. In the second phase, the points-to information is actually updated/propagated. First, the address-of constraints are processed in this two-phase manner. Then, the remaining three kinds of constraints are also processed in a two-phase manner. In the first phase, the constraints add edges to the graph. In the second phase, the points-to information is propagated along these edges. This process repeats until a fixed-point is reached. Finally, the points-to information is copied to the CPU.

5. Boruvka's Minimum Spanning Tree Algorithm

Boruvka's algorithm computes a minimum spanning tree of an edge-weighted undirected graph (without self-loops) through successive applications of *edge contraction*. Edge contraction is a general operation on a graph in which an edge is chosen and the two end points of that edge are fused to form a new node that is connected to all nodes incident on either of the two original nodes.

MST Algorithm Boruvka's algorithm is an *unordered* algorithm [24]: if an edge (a, b) is the lightest edge connecting its end point a to its end point b, it can be contracted. This is in contrast to Kruskal's algorithm, which is an *ordered* algorithm in which edge contractions are performed in increasing weight order. For efficiency, our implementation of edge contraction does not literally merge the incident edges on the two nodes being fused; instead, we maintain groups of endpoints that form a partition over nodes. Only edges whose endpoints lie in distinct components are eligible for contraction. The number of partitions (which we call components) decreases and the partition sizes grow as edges are contracted. The algorithm stops when there is only one component left. The contracted edges form the MST of the original graph.

Amorphous data-parallelism in MST Initially, there is a lot of parallelism in Boruvka's minimum spanning tree algorithm as half the nodes can perform independent edge contractions. After each edge contraction, the graph becomes denser with fewer nodes, lowering the available parallelism. This is why many parallel MST implementations begin with Boruvka's algorithm but switch algorithms as the graph becomes dense.

GPU Implementation Initially, each node forms its own component in our MST implementation. Then, it repeatedly performs pseudo edge contractions. The process is divided into multiple kernels. The first kernel identifies the minimum edge of each node whose other endpoint is in another component. The second kernel isolates the minimum inter-component edge for each component. Since the sequence of the identified edges can form cycles over components (due to common edge weights), the third kernel finds the partner component with which each component needs to be merged to break these cycles. This is done by choosing the component with minimum ID as a cycle representative. All components in a cycle are then merged with the cycle representative in the fourth kernel. The process repeats until there is a single component. The inter-component minimum-weight edges are the MST.

6. Graph Representation on the GPU

We store the graphs in compressed sparse row (CSR) format. Thus, all edges are stored contiguously with the edges of a node stored together. Each graph node records a start offset into the edge array. This representation makes it easy to find a node's outgoing neighbors and improves locality since the edges of a node are stored together. By default, we store outgoing edges, but the same representation can be used to store incoming edges (*e.g.*, PTA). This representation assumes directed edges. For undirected graphs (*e.g.*, MST and SP), we store each edge twice, once for each direction.

6.1 Memory Layout Optimization

In many graph algorithms, computation 'flows' from a node to its neighbors. For instance, in DMR, the refinement process works by identifying a cavity around a bad triangle. Therefore, neighboring graph elements that are logically close to each other should also be close to each other in memory to improve spatial locality. We optimize the memory layout in this way by performing a scan over the nodes that swaps indices of neighboring nodes in the graph with those of neighboring nodes in memory. This optimization is similar in spirit to the data reordering approach by Zhang *et al.* [33].

6.2 Graph Specialization for DMR

We focus on 2D DMR, which operates on a triangulated mesh and modifies it in place to produce a refined triangulated mesh. Each triangle is represented by a set of three coordinates and each coordinate is simply an ordered pair `<x, y>`. The triangle vertices are stored in two associative arrays for the `x` and `y` coordinates, and the `n` triangles are stored in an `n`×3 matrix, where the three entries per line designate indices into the `x` and `y` coordinate arrays.

This simple representation is, in theory, sufficient for performing mesh refinement. However, for performance reasons, it is essential to also keep connectivity information with each triangle to accelerate the formation of a cavity, which requires the determination of the neighbors of a given triangle.

One possible representation for the neighbor information is a Boolean adjacency matrix in which an entry `[i, j]` is true if triangles `i` and `j` are adjacent to each other in the mesh. However, since meshes are very sparse, an adjacency matrix representation would waste a lot of space. As is well known, a more compact way of storing sparse graphs is using adjacency lists or sparse bit vectors. However, since we are dealing with a triangulated mesh, it is easy to see that each triangle can have at most three neighbors. Therefore, the neighborhood information of the `n` triangles can be represented by an `n`×3 matrix. Boundary triangles may have only one or two neighbors. We further record which edge is common between a triangle and its neighbor. Additionally, we maintain a flag with each triangle to denote if it is bad. Note that all of this information needs to be computed when a new triangle is added to the mesh during refinement.

6.3 Graph Specialization for SP

SP is based on a factor graph with two kinds of nodes: one for literals and one for clauses. We split the graph nodes into two arrays and store the clauses separately from the literals. This simplifies the processing since one kernel only works on clause nodes whereas two other kernels only work on literal nodes. Note that the edges connect both kinds of nodes.

Since the factor graph is undirected, surveys flow freely from clause nodes to literal nodes and vice versa. Therefore, it is essential to be able to easily find the neighbors of a type, given a node of the other type. To make neighbor enumeration efficient, we store mappings from each clause to all the literals it contains and from each literal to all the clauses it appears in. Fortunately, each clause has a small limit on the number of literals it can contain, which is the value of K in the K-SAT formula. As in DMR, this allows accessing literals in a clause using a direct offset calculation instead of first having to read the starting offset. Note, however, that this optimization is only possible for the clause-to-literal mapping. Since a literal may appear in an unpredictable number of clauses, the literal-to-clause mapping uses the standard CSR format.

6.4 Graph Specialization for PTA

In its iterative processing, PTA operates in two synchronized phases: one to add new directed edges between nodes and another to propagate information along the edges. There are two ways to manage propagation: *push*-based and *pull*-based. In a push-based method, a node propagates points-to information from itself to its outgoing neighbors, whereas in a pull-based method, a node pulls points-to information to itself from its incoming neighbors. The advantage of a pull-based approach is that, since only one thread is processing each node, no synchronization is needed to update the points-to information. Although some other thread may be reading the points-to information, due to the monotonicity of flow-insensitive points-to analysis (*i.e.,* new edges may be added but no edges are deleted), it is okay to read potentially stale points-to information as long as the up-to-date information is eventually also read [26]. In contrast, in a push-based approach, multiple threads may simultaneously propagate information to the same node and, in general, need to use synchronization.

To implement a pull-based approach, each node keeps a list of its incoming neighbors. Unfortunately, since the number of edges is not fixed, we cannot rely on a single static list of incoming edges but need to maintain a separate list for each node to allow for dynamic growth (*cf.* Section 7.1).

6.5 Graph Specialization for MST

The MST algorithm repeatedly merges components (disjoint sets of nodes) until a single component remains. Conceptually, these components are similar to the clauses in SP. Thus, our MST implementation also maintains a many-to-one (nodes-to-component) mapping and a one-to-many (component-to-nodes) mapping. The key differences are that a literal may be part of multiple clauses in SP whereas a node is part of only a single component in MST, which simplifies the implementation. However, in SP, the number of literals in a clause is bounded by K (a small number) whereas in MST the size of a component can change in each iteration. These changes require dynamic updates to the two mappings, which complicate the MST implementation.

7. Accelerating Morph Algorithms on GPUs

Many morph algorithms involve common operations and depend upon efficient strategies to implement these operations. Such strategies often work across a range of algorithms and can be tailored to special needs. We now discuss several techniques for different scenarios we encountered while implementing our morph algorithms.

7.1 Subgraph Addition

Addition of a subgraph is a basic operation in graph algorithms that increases the number of graph elements. For instance, in DMR, the number of triangles in the mesh increases, in PTA, the number of edges increases, and in MST, the number of nodes in a component increases. Subgraph addition can be performed in various ways.

Pre-allocation If the maximum size of the final graph is bound by a reasonably small factor of the original graph size, one may simply choose to pre-allocate the maximum amount of memory required. For instance, a PTA implementation may choose to pre-allocate memory for all the address-taken variables in the input constraints for each pointer to avoid allocating on the fly. Pre-allocation often results in a simpler implementation and higher performance. However, due to wasted space, the implementation may quickly run out of memory for larger inputs. MST is a special case: the components are graphs without edges (*i.e.,* sets) and, although the component sizes grow, the sum of the nodes in all components is constant. Therefore, the newly formed components can be handled by reshuffling the nodes in an array.

Host-Only Another way of identifying additional memory requirements is by pre-calculation, where the host code performs some computation to determine the amount of memory that will be needed in the next kernel. It then either allocates the additional storage (using `cudaMalloc()`) or re-allocates the existing plus additional storage. For example, in each iteration of DMR, our host code pre-calculates the maximum number of new triangles that can be added by the next kernel invocation depending upon the current number of bad triangles in the mesh and performs the corresponding memory reallocation. Although reallocation can be costly due to memory copying, by choosing an appropriate over-allocation factor, the number of reallocations can be greatly reduced. Similarly, in PTA, the host code can compute the additional number of points-to facts to be propagated in the next iteration and allocate additional storage accordingly.

Kernel-Host An alternative to host-side pre-calculation is to calculate the future memory requirement in the kernel itself and inform the host about it. The host can then either allocate additional storage or perform a re-allocation. The advantage of Kernel-Host over Host-Only is that the computation of the amount of memory needed may be able to piggyback on the main kernel processing.

Kernel-Only The most complicated way of dealing with subgraph addition is to allocate memory in the kernel itself. CUDA 2.x devices support `malloc`s in kernel code with semantics similar to CPU heaps. Thus, a `malloc`ed region can span kernel invocations, can be pointed to by pointers from different threads, *etc*. We employ this strategy in PTA to allocate storage for the incoming edges of a node. Each node maintains a linked list of chunks of incoming neighbors. Each chunk contains several nodes. The best chunk size is input dependent and, in our experiments, varies between 512 and 4096. Chunking reduces the frequency of memory allocation at the cost of some internal fragmentation. To enable efficient lookups, we sort the nodes in the chunks by ID.

Clearly, the best way to perform subgraph addition is application dependent. Pre-allocation usually performs better but can suffer from over-allocation. The host-only approach is suitable when it is relatively easy for the host to compute the additional memory requirement, which may not be the case because the data often only reside on the device. If the calculation of the additional memory requires a non-trivial computation based on the current state of the graph, a kernel-host approach is probably preferable.

The host-only and kernel-host strategies should be used when memory allocation is global with respect to the graph, as is the case for DMR. The kernel-only approach is well-suited when memory allocation is local to a part of the graph. For instance, PTA may require additional memory for individual pointer nodes.

7.2 Subgraph Deletion

Marking If a morph algorithm operates mostly on a fixed graph where nodes and edges are only deleted occasionally, it may be best to simply mark the corresponding nodes and edges as deleted. We employ this strategy in SP since the decimation process is called infrequently. The marking approach is simple to implement, reduces synchronization bugs, and usually performs well as long as only a small fraction of the entire graph is deleted.

Explicit Deletion When an application performs both deletions and additions of subgraphs, simple marking may rapidly increase the unused space. In such codes, explicit freeing of the deleted memory may be necessary (*e.g.*, using `free` or `cudaFree`) to ensure that the deleted memory can be reused for graph additions. This approach is particularly suitable for local deletions. Otherwise, the implementation may have to compact the storage to obtain a contiguous data layout, resulting in high overheads.

Recycle An implementation may also choose to manage its own memory. It can then reuse the deleted memory of a subgraph for storing a newly created subgraph. This strategy works well if the memory layout is such that the new data fit within the memory of the old data. We use this strategy in DMR. Memory recycling offers a useful tradeoff between memory-compaction overhead and the cost of allocating additional storage.

There is no single best subgraph addition and deletion strategy; the choice should be guided by the application and the scale of the underlying problem. However, the above classification of these well-known techniques can help a morph-algorithm implementer select an appropriate strategy.

7.3 Probabilistic 3-Phase Conflict Resolution

Activities that require disjoint neighborhoods necessitate conflict resolution mechanisms across threads. One way to deal with conflicts is using mutual exclusion (implemented with atomic instructions), but this approach is ill-suited for GPUs due to the large number of threads. Therefore, we have developed an efficient 3-phase strategy to detect and resolve conflicts (*cf.* Figure 3). We explain it in the context of DMR, but it is generally applicable.

We exploit the parallelism in DMR by optimistically assuming that cavities around bad triangles do not overlap. When this assumption is valid, multiple bad triangles can be processed and refined in parallel. However, this strategy requires a mechanism to check for conflicts between threads and for resolving them.

Conflict checking A naive way to check for conflicts (*i.e.,* cavity overlap) is to use the two-phase procedure *race-and-check*. In the *race* phase, each thread marks the triangles in the cavity that it wants to refine, and in the *check* phase, the thread checks if its markings still exist. If all markings of thread t_1 exist, t_1 can go ahead and refine the cavity. Otherwise, some other thread t_2 has succeeded in marking (part of) the cavity and t_1 must back off. For correct execution, it is essential that the *race* phase of all threads finishes before the *check* phase of any thread starts. This requires a global barrier between the two phases.

Barrier implementation Current GPU architectures do not directly support global barriers in hardware, so barriers need to be implemented in user code. One way to implement a global barrier inside a kernel is using atomic operations on a global variable. Each thread atomically decrements the variable, initially set to the number of threads, and then spins on the variable until it reaches zero. Unfortunately, this method is particularly inefficient on GPUs because of the large number of threads, because atomic operations are significantly slower than non-atomic accesses [29], and because of the high memory bandwidth requirement of the spinning code.

A better way of implementing a global barrier is to use a hierarchical approach in which threads inside a block synchronize using the `__syncthreads()` primitive and block representatives (*e.g.*, threads 0) synchronize using atomic operations on a global variable. The hierarchical barrier significantly improves performance over the naive barrier implementation.

Xiao and Feng developed an even more efficient global barrier that does not require atomic operations [31]. However, their code was developed for pre-Fermi GPUs, which do not have caches, so writes to global variables directly update the global memory. Fermi GPUs have incoherent L1 caches and a coherent L2 cache, which may optimize global writes by not writing them through to global memory. Therefore, Xiao and Feng's barrier code needs to be augmented with explicit `__threadfence()` calls after writes to global memory to make the writes visible to all threads.

Avoiding live-lock Assuming conflict detection is implemented using a marking scheme coupled with a global barrier, the detected conflicts still need to be resolved. This can easily be achieved by backing off conflicting transactions. Thus, if the cavities created by two or more threads overlap, all affected threads back off, which guarantees conflict-free refinement but may result in live-lock.

One way to address live-lock is by reducing the number of threads, thereby reducing the probability of conflicts. However, avoiding live-lock cannot be guaranteed unless only one thread is running. Thus, when live-lock is detected, *i.e.,* when no bad triangle is refined during a kernel invocation, the next iteration can be invoked with just a single thread to process the current bad triangles serially. The following kernel invocation can resume parallel execution. However, running a single-threaded GPU kernel to avoid conflicts is an inefficient way of addressing this issue.

A better solution is to avoid live-lock with high probability by prioritizing the threads, *e.g.*, using their thread IDs. A thread with a higher ID gets priority over another thread with a lower ID. This approach changes the two-phase *race-and-check* procedure into *race-and-prioritycheck*. The *race* phase works as before, *i.e.,* each thread marks the triangles in its cavity with its thread ID. In the *prioritycheck* phase, thread t_i checks the marking t_m on each triangle in its cavity and uses the following priority check.

1. if $t_i == t_m$ then t_i owns the triangle and can process the cavity if the remaining triangles in the cavity are also owned by t_i.
2. if $t_i < t_m$ then t_m has priority and t_i backs off.
3. if $t_i > t_m$ then t_i has priority and t_i changes the marking on the triangle to its thread ID.

Note that it is not necessary for a thread to remove its markings when it backs off. Whereas the above modification greatly alleviates the probability of live-lock, it introduces another. Two threads may process overlapping cavities simultaneously due to a race in the *prioritycheck* phase. The following scenario illustrates the issue. Two cavities that share a triangle are processed by threads t_i and t_j. Let $t_i > t_j$. In the *race* phase, both threads mark the common triangle. Let the lower priority thread t_j write its ID last. Thus, the common triangle is marked by t_j at the end of the *race* phase. After the global synchronization, both threads check if they own all the triangles in their cavities. Let thread t_j check the marking of the common triangle first. It finds the triangle marked with its ID. Assuming that it also owns all the remaining triangles, t_j is ready to process its cavity. Now the higher priority thread t_i starts its *prioritycheck* phase and finds that the common triangle has been marked by a lower priority thread. Therefore, t_i changes the marking to its own ID. Assuming that it owns the remaining triangles, t_i is also ready to process its cavity. Thus, both threads start processing their cavities even though they overlap with one another.

The race condition occurs due to unsynchronized reads and writes of IDs in the *prioritycheck* phase. It can be avoided by adding a third phase. Thus, the two-phase *race-and-prioritycheck* mechanism becomes the three-phase *race-prioritycheck-check* procedure. In the third phase, each thread checks if its markings from the previous two phases still exist. If a thread owns all the triangles in its cavity, it is ready to process the cavity, otherwise it backs off. Because the *check* phase is read-only, the race condition mentioned above does not arise, thus guaranteeing correct conflict detection. As long as overlaps involve only two cavities, this approach is also guaranteed to avoid live-lock. However, with three or more overlapping cavities, it is possible that all conflicting threads abort due to a race in step 3 of the *prioritycheck* phase. To handle this rare instance, one thread may be allowed to continue either in the same kernel invocation or as a separate kernel launch.

7.4 Adaptive Parallelism

In some morph algorithms, the degree of parallelism changes considerably during execution. For example, Figure 2 shows that the amount of parallelism in DMR increases at first and then gradually decreases. Boruvka's MST algorithm exhibits high parallelism initially, but the parallelism drops quickly. To be able to track the amount of parallelism at different stages of an algorithm, we employ an adaptive scheme rather than fixed kernel configurations.

For DMR and PTA, we double the number of threads per block in every iteration (starting from an initial value of 64 and 128, respectively) for the first three iterations. This improves the work efficiency as well as the overall performance (by 14% and 17%). In SP, the number of threads per block is fixed at 1024 because the graph size mostly remains constant.

The number of thread blocks is set once per program run for all kernel invocations. It is proportional to the input size. Depending upon the algorithm and the input, we use a value between 3×SM and 50×SM, where SM is the number of streaming multiprocessors in the GPU. Note that these are manually tuned parameters.

7.5 Local Worklists

For large inputs, performance increases when each thread processes more than one graph element per iteration (bad triangles in DMR, pointer nodes in PTA, clause nodes in SP, and components in MST). However, due to the large number of threads, it is inefficient to obtain these graph elements from a centralized work queue. Hence, we use a local work queue per thread.

In local queues, work items can be dequeued and newly generated work enqueued without synchronization, which improves performance. Local work queues can be implemented by partitioning the graph elements into equal-sized chunks and assigning each chunk to a thread. It is often possible to store the local work queues in the fast shared memory of GPUs. Due to the memory layout optimization discussed in Section 6.1, neighboring graph elements tend to be near one another in memory. Thus, when assigning a range of consecutive triangles to a thread in DMR, the cavity of a bad triangle has a good chance of falling entirely into this range, which reduces conflicts. Intuitively, the combination of the memory layout optimization and the local work queues forms a pseudo-partitioning of the graph that helps reduce conflicts and boosts performance.

7.6 Reducing Thread Divergence

To minimize thread divergence in DMR, we try to ensure that all threads in a warp perform roughly the same amount of work by

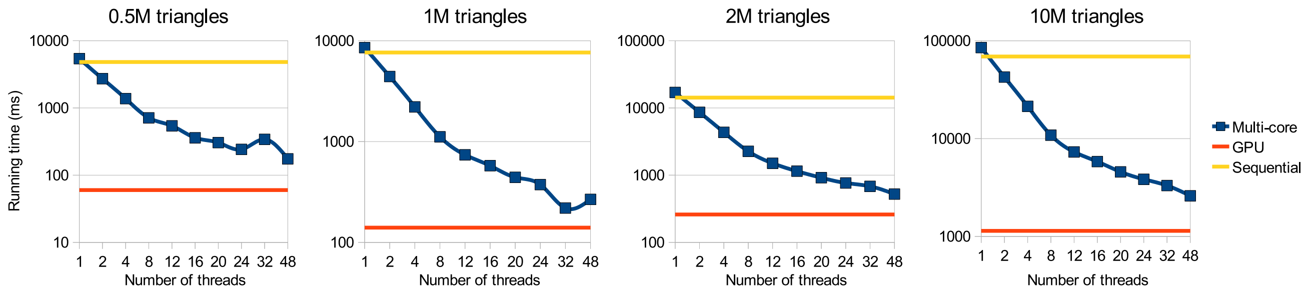


Figure 6. DMR runtime of the GPU, sequential CPU (Triangle), and multicore CPU (Galois) codes for different inputs

Triangles		Speedup	
Total $\times 10^6$	Bad $\times 10^6$	Galois-48	GPU
0.5	0.26	27.6	80.5
1	0.48	28.6	54.6
2	0.95	27.2	54.8
10	4.75	26.5	60.6

Figure 7. Speedup obtained using multicore (Galois) and GPU implementations of DMR over the sequential version (Triangle)

moving the bad triangles to one side of the triangle array and the good triangles to the other side. This way, the threads in each warp (except one) will either all process bad triangles or not process any triangles. Note that this distribution is not perfect since the amount of work depends upon the cavity size. Nevertheless, the sorting assigns roughly equal amounts of work to each warp-thread. We perform sorting at the thread-block level in each iteration. In PTA, we similarly move all pointer nodes with enabled incoming edges to one side of the array. An edge is enabled if the points-to information of its source has changed, thus requiring its destination to be processed. Since each clause node is connected to a fixed number of literal nodes in SP, the work distribution is largely uniform even without sorting. There is also no sorting in MST.

8. Experimental Evaluation

We evaluate the performance of our CUDA implementations on a 1.15 GHz NVIDIA Tesla C2070 GPU with 14 SMs (448 processing elements, *i.e.,* CUDA cores) and 6 GB of main memory. This Fermi GPU has a 64 KB L1 cache per SM. We dedicate 48 KB to shared memory (user-managed cache) and 16 KB to the hardware-managed cache. The SMs share an L2 cache of 768 KB. We compiled the code with *nvcc* v4.1 RC2 using the *-arch=sm_20* flag.

To execute the CPU codes, we used a Nehalem-based Intel Xeon E7540 running Ubuntu 10 with 8 hex-core 2 GHz processors. The 48 CPU cores share 128 GB of main memory. Each core has two 32 KB L1 caches and a 256 KB L2 cache. Each processor has a 18 MB L3 cache that is shared among its six cores.

8.1 Delaunay Mesh Refinement

We compare the GPU DMR performance to two CPU implementations: the sequential Triangle program [28] and the multicore Galois version 2.1.4 [16].

The input meshes are randomly generated. We use the quality constraint that no triangle can have an angle of less than 30 degrees. The mesh sizes range from 0.5 million to 10 million triangles, and roughly half of the initial triangles are bad.

The relative performance of the three implementations for four different inputs is depicted in Figure 6. The x-axes show the number of threads used in the multicore version and the y-axes give the running time in milliseconds on a logarithmic scale. The two horizontal flat lines correspond to the sequential and the GPU runtime. The reported GPU runtime is for the execution of the `do-while` loop in Figure 3. Similarly, we only report the mesh-refinement runtime for the sequential and multicore codes.

	Optimization	Time (ms)
1	Topology-driven with mesh-partitioning	68,000
2	3-phase marking	10,000
3	2 + Atomic-free global barrier	6,360
4	3 + Optimized memory layout	5,380
5	4 + Adaptive parallelism	2,200
6	5 + Reduced thread-divergence	2,020
7	6 + Single-precision arithmetic	1,020
8	7 + On-demand memory allocation	1,140

Figure 8. Effect of optimizations on the running time of DMR using an input mesh with 10 million triangles

It is evident that our GPU implementation is significantly faster than the serial and multicore codes are. It takes the GPU 1.14 seconds to refine the largest mesh consisting of 10 million triangles, out of which 4.7 million triangles are initially bad.

The speedups of the multicore and GPU implementations over the sequential implementation are shown in Figure 7. The multicore version running with 48 threads achieves a speedup between 26.5 and 28.6 on our data sets. The GPU version is 2× to 4× faster and achieves a speedup between 54.6 and 80.5 over the sequential code. Figure 8 shows the effect of individual optimizations on DMR.

8.2 Survey Propagation

Figure 9 shows the SP runtimes for $K = 3$ and a clause-to-literal ratio of 4.2 (*i.e.,* hard SAT problems). The number of literals ranges from 1 million to 4 million. The GPU code is almost 3× faster than the 48-thread Galois code.

Both codes implement the same algorithm with one exception. The GPU code caches computations along the edges to avoid some repeated graph traversals. The importance of this optimization is more pronounced for larger K, as Figure 9 shows. Note that the hard SAT ratios differ (we obtained them from Mertens *et al.* [21]). The multicore version does not scale with the problem size and requires hours to complete. In contrast, the GPU version scales linearly and successfully completes within a few minutes.

8.3 Points-to Analysis

The PTA runtimes on six SPEC 2000 inputs are shown in Figure 10. The CPU codes perform optimizations like online cycle elimination and topological sort that are not included in our GPU code. Yet, the GPU implementation is 1.9 to 34.7 times faster. Major performance gains are obtained by separating the phases for constraint evalua-

M	N	K	Time (s)	
$\times 10^6$	$\times 10^6$		Galois-48	GPU
4.2	1	3	108	35
8.4	2	3	230	73
12.6	3	3	336	117
16.8	4	3	445	157
4.2	1	3	108	35
9.9	1	4	3,033	85
21.1	1	5	40,832	178
43.4	1	6	*OOT*	368

Figure 9. Performance of Survey Propagation

tion and propagation and by transforming the code from a push- to a pull-based approach to avoid costly synchronization.

8.4 Boruvka's Minimum Spanning Tree

We evaluate the GPU and multicore Galois MST codes on a range of graphs as shown in Figure 11. The graphs other than the road networks are randomly generated. The Galois version 2.1.4 implements edge contraction by explicitly merging adjacency lists. The GPU code implements it using components as discussed above. Note that we run the multicore MST version with up to 48 threads and report the runtime for the best-performing number of threads.

For road networks and grids, the multicore code achieves up to 7.0 million edges/sec and generally outperforms our GPU code, which only reaches up to 2.5 million edges/sec. For the RMAT and random graphs, however, the multicore rate drops to 0.014 million edges/sec, whereas the GPU achieves 0.85 million edges/sec, greatly outperforming the CPUs. This behavior is due to differing graph densities that affect the edge-contraction performance. Road networks and grids are relatively sparse, *i.e.,* have small degrees, whereas the RMAT and random graphs are denser. The cost of merging adjacency lists in the Galois version is directly proportional to the node degrees. Therefore, denser graphs are processed more slowly. Moreover, the cost increases for later iterations as the graph becomes smaller and denser. In contrast, the GPU implementation maintains the original adjacency list for each node and performs edge traversals per node. The cost of merging increases with the number of nodes rather than with the number of edges because each component only contains nodes, and component merging requires node merging rather than edge merging.

Based on this observation, we modified the Galois implementation (in version 2.1.5) to also use a component-based approach. Additionally, the new multicore code incorporates a fast union-find data structure that maintains groups of nodes, keeps the graph unmodified, and employs a bulk-synchronous executor. The resulting CPU code is much faster and, in fact, outperforms the GPU code. It would be interesting to see whether including these optimizations in the GPU code would result in a similar performance boost.

9. Related Work

There are many implementations of parallel graph algorithms on a variety of architectures, including distributed-memory supercomputers [32], shared-memory supercomputers [2], and multicore machines [16]. DMR in particular has been extensively studied in sequential [8], multicore [6] and distributed-memory settings [14].

Irregular programs have only recently been parallelized for GPUs. Harish and Narayanan [10] describe CUDA implementations of graph algorithms such as BFS and single-source shortest paths computation. BFS has received significant attention [9, 12, 17, 20]. In these algorithms, the structure of the graph remains unchanged, simplifying the implementation. Hong *et al.* [11] propose a warp-centric approach for implementing BFS, which is related to our solution for distributing work among threads. Vineet *et al.* [30] and Nobari *et al.* [23] propose computing the minimum spanning tree and forest, respectively, on GPUs. Both implementations use statically allocated memory.

			Time (ms)		
Benchmark	Vars	Cons	Serial	Galois-48	GPU
186.crafty	6,126	6,768	595	86	44.4
164.gzip	1,595	1,773	456	73	7.1
256.bzip2	1,147	1,081	396	94	2.7
181.mcf	1,230	1,509	382	59	8.7
183.equake	1,317	1,279	436	49	3.3
179.art	586	603	485	72	7.4

Figure 10. Performance of Points-to Analysis

There are relatively few GPU implementations of *morph* algorithms. Burtscher and Pingali [5] describe an implementation of an *n*-body simulation (Barnes Hut algorithm) based on unbalanced octrees. Whereas tree building is a morph operation, the vast majority of the runtime is spent in the force calculation, which does not modify the octree. Prabhu *et al.* [25] describe a GPU implementation of a 0-CFA analysis and Mendez-Lojo *et al.* [18] implemented an Andersen-style points-to analysis on the GPU. In these two algorithms, the number of nodes in the graph is invariant, and the number of edges grows monotonically until a fixed-point is reached. Our work deals with more general morph algorithms.

Our conflict-detection scheme has some similarity to the coloring heuristic of Jones *et al.* [13]. However, their distributed memory algorithm involves no aborting threads; hence, there is no issue with live-lock. Delaunay triangulation (not refinement) has been studied by several researchers [27]. A refinement algorithm based on edge-flipping has been proposed by Navarro *et al.* [22]. Although it is a morph algorithm, it does not exhibit the same challenges because the number of nodes and edges in the mesh do not change during execution. Instead, edges are flipped to obtain a better triangulation.

To the best of our knowledge, ours is the first paper on general morph algorithms for GPUs that describes efficient arbitrary subgraph addition and deletion and provides general techniques for implementing other morph algorithms.

10. Conclusions

Using a GPU to accelerate morph algorithms is very challenging. In this paper, we describe various aspects of efficiently implementing such algorithms on GPUs and discuss techniques for subgraph addition and deletion, conflict detection and resolution, and improving load distribution and work efficiency. We illustrate our techniques on efficient GPU implementations of four graph algorithms that modify the underlying graph in different ways: Delaunay Mesh Refinement, Survey Propagation, Points-to Analysis, and Boruvka's MST algorithm. We evaluate the effectiveness of our implementations on a range of inputs and show that our techniques lead to large speedups compared to multicore and sequential versions of these algorithms. We hope that this work proves useful for other GPU implementations of general morph algorithms.

Acknowledgments

This work was supported by NSF grants 0833162, 1062335, 1111766, 1141022, 1216701, 1217231 and 1218568 as well as gifts from NVIDIA, IBM, Intel and Qualcomm Corporations.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen,

Graph	N	M	Time (s)		
	$\times 10^6$	$\times 10^6$	Galois v2.1.4	Galois v2.1.5	GPU
USA	23.9	57.7	8.2	3.0	35.8
W	6.3	15.1	2.3	0.8	9.5
RMAT20	1.0	8.3	1,393.6	0.4	26.8
Random4-20	1.0	4.0	281.9	0.4	4.7
grid-2d-24	16.8	33.6	14.3	5.0	71.8
grid-2d-20	1.0	2.0	0.7	0.2	0.9

Figure 11. Performance of Boruvka's MST Algorithm. Time shown for Galois is the best time obtained with up to 48 threads.

May 1994. (DIKU report 94/19).

[2] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[3] J. Barnat, P. Bauch, L. Brim, and M. Češka. Computing Strongly Connected Components in Parallel on CUDA. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*, pages 541–552. IEEE Computer Society, 2011.

[4] A. Braunstein, M. Mèzard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, 2005.

[5] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. Morgan Kaufmann, 2011.

[6] Andrey N. Chernikov and Nikos P. Chrisochoides. Three-dimensional delaunay refinement for multi-core processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 214–224, New York, NY, USA, 2008. ACM.

[7] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proc. Symp. on Computational Geometry (SCG)*, 1993.

[8] Panagiotis A. Foteinos, Andrey N. Chernikov, and Nikos P. Chrisochoides. Fully generalized two-dimensional constrained delaunay mesh refinement. *SIAM J. Sci. Comput.*, 32(5):2659–2686, 2010.

[9] Abdullah Gharaibeh, Lauro Beltro Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *The 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[10] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC'07: Proceedings of the 14th international conference on High performance computing*, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[12] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *20th International Conference on Parallel Architectures and Compilation Techniques*, PACT'11, 2011.

[13] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, May 1993.

[14] Andriy Kot, Andrey Chernikov, and Nikos Chrisochoides. Effective out-of-core parallel delaunay mesh refinement using off-the-shelf software. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 125–125, Washington, DC, USA, 2006. IEEE Computer Society.

[15] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *Proc. Symp. on Principles and practice of parallel programming (PPoPP)*, pages 3–14, New York, NY, USA, 2009.

[16] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (PLDI)*, 42(6):211–222, 2007.

[17] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[18] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM.

[19] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practiceof Parallel Programming (PPoPP'10)*, pages 3–14, January 2010.

[20] Duane G. Merrill, Michael Garland, and Andrew S. Grimshaw. Scalable gpu graph traversal. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'12, 2012.

[21] Stephan Mertens, Marc Mézard, and Riccardo Zecchina. Threshold values of random k-sat from the cavity method. *Random Struct. Algorithms*, 28(3):340–373, May 2006.

[22] Cristobal A. Navarro, Nancy Hitschfeld-Kahler, and Eliana Scheihing. A parallel gpu-based algorithm for delaunay edge-flips. In *The 27th European Workshop on Computational Geometry*, EuroCG '11, 2011.

[23] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 205–214, New York, NY, USA, 2012. ACM.

[24] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[25] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. Eigencfa: accelerating flow analysis with gpus. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 511–522, New York, NY, USA, 2011. ACM.

[26] Sandeep Putta and Rupesh Nasre. Parallel replication-based points-to analysis. In *Proceedings of the 21st international conference on Compiler Construction*, CC'12, pages 61–80, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. Computing 2d constrained delaunay triangulation using the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 39–46, New York, NY, USA, 2012. ACM.

[28] Jonathan Richard Shewchuk. Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1996.

[29] Jeff A. Stuart and John D. Owens. Efficient synchronization primitives for gpus. *CoRR*, abs/1110.4623, 2011.

[30] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 167–171, New York, NY, USA, 2009. ACM.

[31] Shucai Xiao and Wu chun Feng. Inter-block gpu communication via fast barrier synchronization. In *IPDPS*, pages 1–12. IEEE, 2010.

[32] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society.

[33] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 369–380, New York, NY, USA, 2011. ACM.