Atomic-free Irregular Computations on GPUs

Rupesh Nasre
The University of Texas
Austin, USA
nasre@ices.utexas.edu

Martin Burtscher
Texas State University
San Marcos, USA
burtscher@txstate.edu

Keshav Pingali
The University of Texas
Austin, USA
pingali@cs.utexas.edu

ABSTRACT

Atomic instructions are a key ingredient of codes that operate on irregular data structures like trees and graphs. It is well known that atomics can be expensive, especially on massively parallel GPUs, and are often on the critical path of a program. In this paper, we present two high-level methods to eliminate atomics in irregular programs. The first method advocates synchronous processing using barriers. We illustrate how to exploit synchronous processing to avoid atomics even when the threads' memory accesses conflict with each other. The second method is based on exploiting algebraic properties of algorithms to elide atomics. Specifically, we focus on three key properties: monotonicity, idempotency and associativity, and show how each of them enables an atomic-free implementation. We illustrate the generality of the two methods by applying them to five irregular graph applications: breadth-first search, single-source shortest paths computation, Delaunay mesh refinement, pointer analysis and survey propagation, and show that both methods provide substantial speedup in each case on different GPUs.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Languages, Performance

Keywords

graph algorithms, irregular algorithms, atomic-free, GPGPU

1. INTRODUCTION

GPUs have been used successfully to accelerate applications in many problem domains ranging from graphics to molecular dynamics simulations and climate modeling. GPUs perform particularly well when data access patterns are regular and predictable, as is the case in dense linear algebra computations. Not surprisingly, a lot is known about how to implement such regular algorithms efficiently on GPUs. In contrast, much less is known about efficient GPU implementations of *irregular* algorithms that make unpredictable, data-dependent accesses to complex data structures such as large graphs. Fine-grain synchronization is usually necessary to exploit parallelism in such algorithms, but this requires the use of atomic operations like `atomicCAS` on shared memory locations, which are an order of magnitude more expensive than regular memory accesses.

For performance reasons, it is often beneficial to avoid the use of atomic instructions whenever feasible. This is particularly true when the contention on a shared resource is high. Consider, for instance, a shared worklist from which threads extract work items and onto which they push new work items. Under heavy contention, the worklist quickly becomes a performance bottleneck. This effect is exacerbated on massively multi-threaded processors such as GPUs.

An alternative synchronization mechanism is the barrier. A global barrier is a primitive that guarantees that all threads of a kernel reach a specific point in the code before any thread may progress beyond that point. Thus, barriers make part of the processing synchronous, which may enable an atomic-free implementation. In effect, barriers partition the processing into phases. The synchronization requirements of phase-based processing are determined by the type of processing performed within a phase. As long as a phase contains homogeneous thread operations, there may be no need for further synchronization. For instance, in the above worklist example, if work extraction and insertion are separated into two phases, the work-extraction phase becomes read-only, which requires no synchronization, and the work-insertion phase becomes write-only, which can be efficiently implemented using a barrier-based prefix sum. Thus, both phases can be implemented without atomics.

In certain cases, which we found to be abundant in graph algorithms, fine-grain synchronization can be eliminated by exploiting algebraic properties of the computation. Consider the example of computing single-source shortest paths (SSSP) in a directed graph. The core operation in SSSP is an edge-relaxation step, which, for an edge (`src`, `dst`), checks if the current distance of `dst` is greater than the distance via `src` by adding the distance of `src` to the edge weight. In general, due to multiple incoming edges, a node may act as `dst` for two or more edges, *e.g.*, ($\texttt{src}_1$, `dst`) and ($\texttt{src}_2$, `dst`). Threads performing an edge-relaxation operation on the two edges may update the distance of `dst` in

GPGPU'13, March 16, 2013, Houston, Texas, USA.

parallel and, therefore, this operation requires synchronization (in the form of an `atomicMin` instruction). However, we observe that in SSSP the distance updates are monotonic, *i.e.*, the distance of a node only decreases. We show that we can take advantage of this monotonicity to avoid atomics altogether.

This paper makes the following contributions.

- We present two methods to elide atomic instructions from a program. The first method is more general and employs barrier-based processing to avoid conflicts. The second method exploits algebraic properties of algorithms to avoid atomics entirely.
- We show that the two methods are applicable to several real-world irregular programs. In particular, we demonstrate how to avoid atomics in shortest paths computations, breadth first search, Delaunay mesh refinement, points-to analysis, and survey propagation.
- We evaluate the effect of the two proposed methods on the above-mentioned programs and show that the application of our methods results in considerable performance improvements on Fermi- and Kepler-based GPUs. For instance, using barrier-based processing speeds up Delaunay mesh refinement by 29% on Fermi, and exploiting algebraic properties accelerates survey propagation by up to 43% on Kepler.

The rest of this paper is organized as follows. Section 2 provides an overview of irregular algorithms and introduces our applications. Section 3 describes how barrier-based processing can avoid atomics. Section 4 explains how to exploit algebraic properties to elide atomics. Section 5 briefly mentions other well-known techniques to eliminate atomics. Section 6 assesses the effectiveness of our methods using five irregular algorithms. Section 7 discusses related work. Section 8 summarizes the main results.

2. BACKGROUND

We first define the notion of an irregular algorithm and then introduce the irregular applications we study.

2.1 Irregular Algorithms

In regular code, control flow and memory references are not data dependent. Matrix-vector multiplication is a good example. Based only on the input *size* and the data-structure starting addresses, but without knowing any *values* of the input data, we can determine the dynamic behavior of the program on an in-order processor, *i.e.*, the memory reference stream and the conditional branch decisions. This sort of regularity can be exploited to improve memory coalescing and minimize thread divergence and synchronization.

In irregular code, the input values to the program determine the runtime behavior, which therefore cannot be statically predicted and may be different for different inputs. Irregular code usually arises from the use of complex data structures such as trees and graphs. As a rule, irregular algorithms are more difficult to parallelize and more challenging to map to GPUs than regular algorithms. For example, in graph applications, the memory-access patterns are usually data dependent since the connectivity of the graph and the values on nodes and edges may determine which nodes and edges are touched by a given computation. This information is usually not known at compile time and may change dynamically even after the input graph is available. This leads to uncoalesced memory accesses. Similarly, the control flow is usually irregular because branch decisions may be different for nodes having different numbers of neighbors or labels, leading to thread divergence and load imbalance.

In most regular applications, threads operate on disjoint regions of memory that are statically known or can be statically determined. This eliminates the need for locks or atomics when accessing the data. In contrast, the threads of irregular applications often operate on potentially overlapping memory regions because their reads and writes are input dependent and statically unknown. Hence, dynamic mechanisms such as atomics or locks are typically required to deal with conflicting accesses in irregular applications.

2.2 Applications

In this paper, we study the following five irregular programs from the LonestarGPU benchmark suite (available at `http://iss.ices.utexas.edu/?p=projects/galois`).

- Breadth-First Search (BFS): This graph problem is a key kernel in many important applications such as mesh partitioning [8]. The BFS problem is to label each graph node with the node's minimum level (or number of hops) from a designated start node.
- Delaunay Mesh Refinement (DMR): This is a mesh-refinement algorithm from computational geometry [6, 14]. It works on a triangulated input mesh in which some triangles do not conform to certain quality constraints. The algorithm iteratively transforms such 'bad' triangles into 'good' triangles by recreating the neighborhood (called cavity) around each bad triangle.
- Points-To Analysis (PTA): Andersen's flow-insensitive, context-insensitive points-to analysis is used in compilers like GCC and LLVM [16]. It employs a fixed-point algorithm that operates on a dynamically growing constraint graph in which directed edges are added depending upon the input points-to constraints.
- Survey Propagation (SP): Survey Propagation is a heuristic SAT-solver based on Bayesian inference [5]. The algorithm represents the Boolean formula as a factor graph, *i.e.*, a bipartite graph with variables on one side and clauses on the other. The general strategy of SP is to iteratively update each variable with the likelihood that it should be assigned a truth value of *true* or *false*.
- Single-Source Shortest Paths (SSSP): This is another classic graph problem. It computes the shortest path of each node from a designated source node in a directed graph [7]. We use the Bellman-Ford algorithm.

3. BARRIER-BASED PROCESSING

A global barrier is a synchronization primitive that guarantees that all threads from all thread blocks belonging to a kernel reach a specific point in the code before any thread may progress beyond that point. CUDA supports a barrier at the thread-block level (`syncthreads`). However, a global barrier (across thread blocks) needs to be emulated in software. There are multiple ways in which a global barrier can be implemented (*e.g.*, see Baskaran *et al.* [3]), and it can be done without using atomics [19, 24].

We advocate using barrier-based processing when resource contention is high. The use of barriers transforms large amounts of asynchronous processing into smaller synchronous

```
 1: shared donationbox[...], totalwork
 2:
 3: // determine whether I should donate
 4: for all work_items assigned to me do
 5:    mywork += estimated work per item
 6: atomicAdd(totalwork, mywork)
 7: — barrier —
 8:
 9: // donate
10: averagework = totalwork / n_threads
11: if mywork > averagework + threshold then
12:    push excess work into donationbox
13: — barrier —
14:
15: // process assigned work
16: for all work items assigned to me do
17:    process item
18:
19: // empty donation box
20: while donationbox is not empty do
21:    item = pop work from donationbox
22:    process item
```

Figure 1: Pseudo code of work donation

phases. While asynchronicity is an essential ingredient of high-performance parallel codes, we argue that limiting asynchronicity to a phase can lead to simpler synchronization requirements that can improve overall performance.

Specifically, in irregular algorithms, threads operate on shared resources like a graph or a global worklist. Operations on such data structures need to be synchronized using atomic instructions and locks implemented thereon. Depending upon the algorithm, a shared access may be to a disjoint or an overlapping region and may require different mechanisms to arrive at an atomic-free implementation.

3.1 Disjoint Accesses

Consider the example of work donation. Threads that are assigned more work than the average thread may donate their excess work to ensure better load balance. One way to implement work donation is by creating a donation box that is shared across all threads. The donor threads concurrently push excess work items into the donation box. Threads that finish their assigned work early extract work items from the donation box and process them while slower threads are still processing their originally assigned work. This leads to better load balance, reducing the critical path length. Figure 1 shows pseudo code for work donation.

Let us focus on the pushing of excess work into the donation box on Line 12. This can easily be implemented by maintaining a buffer with an index specifying the current size of the buffer. Each donor thread atomically increases the index to make space for pushing its items (using an `atomicAdd`) as shown on Line 6. Atomic accesses are required because threads conflict on reading and updating the value of the shared index. However, once the index is updated by a thread, the thread can carry out the actual donation concurrently with other threads, even when all threads push work into the same donation box. Thus, the purpose of the atomic in this case is to obtain an exclusive slot (disjoint access) per thread for performing the donation. Assuming the donation box never overflows, no thread ever aborts. However, $O(n)$ serial atomic operations need to be executed, where n is the number of threads that want to donate work.

```
 1: shared array[n_threads]
 2: array[threadID] = my_n_items
 3: — barrier —
 4:
 5: for (s = n_threads / 2; s > 0; s = s / 2) do
 6:    if threadID + s < n_threads then
 7:       element = array[threadID + s]
 8:    — barrier —
 9:    if threadID + s < n_threads then
10:       array[threadID] += element
11:    — barrier —
12:
13: // compute the start index for each thread
14: startindex = array[threadID] - my_n_items
```

Figure 2: Prefix-sum computation

Barriers can help avoid these atomics. In the above work-donation scenario, threads can alternatively perform a standard prefix-sum computation to find the range in the donation box into which they can concurrently push their items. The prefix sum is computed in phases, operating on a shared array initialized with the number of elements each thread wishes to donate. The computation is cleverly divided among the threads such that each array element is written by only one thread in every iteration and there are no data races. The step-size for accessing array elements is initialized to half the number of threads and is halved in each phase until it reaches zero, which indicates the end of the computation. Thus, only $O(log\ n)$ barriers are executed, where n is the number of threads involved. When all threads belong to the same thread block, the barrier can be implemented using the fast `__syncthreads()` intrinsic, which is directly supported by the GPU hardware. Pseudo code of such a prefix-sum computation is given in Figure 2. The `for` loop computes the prefix sum over the shared array, which finds the end index for each thread. By subtracting the number of items to be pushed, as shown in the last line of the code, each thread obtains the start index of the donation box where it should push its data items.

Note that emptying the donation box (Line 21 in Figure 1) requires atomic accesses, but typically only a small number of threads does this work. Barrier-based processing can be utilized in other cases with disjoint accesses, *e.g.*, by separating reading from and writing to a worklist into separate phases; a similar mechanism can be used to insert elements into a worklist [17].

3.2 Overlapping Accesses

In disjoint accesses, each thread operates on a nonoverlapping set of memory locations. Therefore, threads can concurrently perform tasks on the same data structure (*e.g.*, a donation box). However, in applications where accesses from different threads overlap, only the non-overlapping set of accesses may proceed; the conflicting threads generally must wait or abort and try again later.

This can happen in several situations. Consider, for instance, Delaunay mesh refinement, which operates on a triangulated input mesh. Each thread is assigned a 'bad' triangle, which does not conform to certain geometric constraints (*e.g.*, the constraint could be that none of the angles of the triangle should be less than 30 degrees). The task of a thread is to find a set of neighboring triangles, called the cavity, around the bad triangle and transform (refine) it into a new

set of triangles, thus eliminating the bad triangle. Note that a thread must obtain exclusive ownership of not only the bad triangle but also the cavity to execute the refinement task. However, the cavities of different bad triangles may overlap, in which case only one of them can be refined at a time.

For exclusive ownership, each thread needs to acquire fine-grained logical locks over all triangles in the cavity. A coarse-grain lock at the cavity level is infeasible since a cavity's shape depends upon the geometry of the mesh, which changes dynamically during the refinement procedure. Mesh partitioning may allow lock coarsening at the partition granularity, but care must be taken when cavities span partitions. In our input meshes, the number of bad triangles is initially close to 50%, leading to many inter-partition cavities. Therefore, instead of mesh partitioning, we focus on the uniform scheme of fine-grained locking over a cavity's triangles, which is also used in other implementations [4, 14].

Deadlock may occur when two threads operating on overlapping cavities have acquired exclusive locks on some common triangles and wait for the other common triangles to be released. This situation can be avoided in multiple ways; for instance, by locking triangles in a particular order, say by their IDs. Thread starvation is also possible, but as long as wait-freedom can be proven, *i.e.*, at least one thread makes progress, algorithm termination is guaranteed.

In principle, the lock-based mutual exclusion approach that is typically used in multi-core CPU versions can be directly applied to GPUs. Current GPUs support atomic primitives with which locks are easy to implement. However, multi-core code ported to GPUs is unlikely to perform well in practice due to the massive multi-threading on the GPU. Again, barrier-based processing performs better.

Exclusive ownership of the triangles in a cavity can be obtained through three barrier-separated phases [19]. The first phase is the 'race' phase wherein each thread marks its cavity triangles with its thread ID. The second phase is the 'priority-race' phase wherein each thread checks its triangles' markings to see if it owns all of them. If a needed triangle is marked with a higher ID, the current thread aborts. If a triangle is marked with a lower ID, the current thread changes the marking to its own ID. The third phase is the 'check' phase wherein each thread simply checks if all its markings are intact. If it owns all the triangles, it owns the cavity and proceeds with the refinement; otherwise, it aborts. The aborting threads try again in the next round depending upon whether their bad triangle still exists in the mesh. This three-phase race-and-resolve scheme ensures exclusivity and probabilistically avoids livelocks [19]. This scheme is best suited for overlapping accesses. For disjoint accesses, one can use the mechanisms presented in Section 3.1.

The race-and-resolve scheme as originally proposed supports exclusive access to *all* work items. We extend it to also support scenarios when *any* of the work items can be owned. In other words, the scheme was originally proposed in an *AND* context in which a thread owns either all or none of the work items. We extend it to support *OR* contexts where a thread may only receive a subset of the work items it races for. Figures 3 and 4 show the operational semantics of the original and the extended scheme. It is modeled as a primitive `race-and-resolve`, which takes three inputs: (i) `data` is a set of items a thread wants to operate on, (ii) `flag` ∈ {`AND, OR`} indicates if a thread wishes to lock all data

```
1: for item ∈ data do
2:    mark[item] = my-thread-id;
3: — barrier —
4: for item ∈ data do
5:    if mark[item] > my-thread-id then
6:       abort = true;
7:    else if mark[item] < my-thread-id then
8:       mark[item] = my-thread-id;
9: — barrier —
10: for item ∈ data do
11:    if mark[item] ≠ my-thread-id then
12:       abort = true;
13: if !abort then
14:    callback(data);
```

Figure 3: Original race-and-resolve: AND case

```
1: for item ∈ data do
2:    mark[item] = my-thread-id;
3: — barrier —
4: for item ∈ data do
5:    if mark[item] == my-thread-id then
6:       callback(item);
```

Figure 4: Extended race-and-resolve: OR case

items together or only some of them, and (iii) `callback` is a function that is called once the thread owns the data items.

The extended race-and-resolve scheme is useful, for example, to ensure unique elements in a worklist. Consider the case of SSSP where different threads may push the same graph node into the worklist, leading to work duplication. For work efficiency and better synchronization, one may wish to avoid such duplicates. This can be readily accomplished using our extended race-and-resolve scheme as shown in the pseudo code of Figure 5.

A potential problem with using global barriers is deadlock. To ensure that all threads from all thread-blocks can reach the barrier, all thread-blocks need to be resident (*i.e.* simultaneously running) on the GPU. This demands that the number of thread-blocks be equal to or a small multiple of the number of SMs, depending on the resource needs. Therefore, the graph processing may need to be split across a reduced number of threads, and each thread will have to process more than one graph element. Whereas this can complicate the implementation a little, the benefit of avoiding locks is typically well worth the effort. As an alternative, one may choose to implement a global barrier without reducing the number of thread-blocks by returning control to the CPU and re-launching the GPU kernel.

3.3 Benign Overlaps

In certain barrier-based implementations, one may lift the synchronization requirement even in the presence of overlapping accesses. Consider, for instance, the breadth-first search (BFS) computation. It can be implemented in a completely asynchronous manner, such that the graph nodes are processed in any order. However, a better way of implementing BFS is by processing nodes level-by-level, where the source node is at level 0. The nodes reachable from the source by a direct edge are at level 1. The nodes reachable from the level 1 nodes by a direct edge are at level 2, and so on. The process terminates when all nodes have their levels marked. Processing of two consecutive levels is separated by barriers. For instance, consider the graph shown in Figure 6

```
myworkitems = ...
race-and-resolve(myworkitems, OR, callback);

callback(workitem) {
  worklist.push(workitem);
}
```

Figure 5: Unique worklist elements using the extended race-and-resolve function

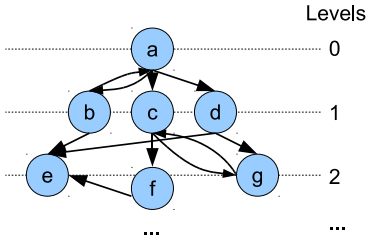

Figure 6: Benign overlaps in BFS

with the BFS levels indicated on the right. Node `a` is the designated source at level 0. Nodes `b`, `c`, and `d` are at level 1 because they are adjacent to the source node `a`. Nodes `e`, `f`, and `g` are at level 2 because they are adjacent to level 1 nodes (and not adjacent to a level 0 node).

Level-by-level BFS processing ensures that work is done in an efficient order. However, depending upon the graph connectivity, a single node's level may be updated by more than one thread. For instance, if a node `dst` is reachable by direct edges from two source nodes $\texttt{src}_1$ and $\texttt{src}_2$, the threads operating on $\texttt{src}_1$ and $\texttt{src}_2$ may try to update `dst`'s level in parallel, leading to overlapping accesses. This scenario occurs in Figure 6 where node `e` is adjacent to nodes `b` and `d` (as well as `f`). The threads operating on nodes `b` and `d` may try to update node `e`'s level simultaneously. This would, in general, require synchronization (either atomics or race-and-resolve). However, since the processing is level-by-level, both threads attempt to update node `e`'s level to the same value. This means the conflict is benign. It does not matter which thread succeeds in updating the level of node `e`; the level will invariably be updated correctly. Hence, due to the algorithmic property of barrier-based level-by-level BFS, synchronization is not required. Note that in Figure 6, node `e` is also adjacent to node `f`, but the level-by-level processing precludes node `e`'s distance being updated by the thread operating on node `f`.

4. EXPLOITING ALGEBRAIC PROPERTIES

Several regular and irregular computations bear special properties that can be exploited to reduce or avoid the use of atomic instructions. We focus on three algebraic properties in the context of irregular algorithms: monotonicity, idempotency, and associativity. Each of these properties provides us with a different opportunity to avoid atomic instructions.

4.1 Monotonicity

We call a computation monotonic if the values produced by repeated application of the computation either only increase or only decrease, but not both. For instance, in the single-source shortest paths (SSSP) computation, the distance of a node never increases; hence the SSSP computation is monotonic. Similarly, the points-to information computed by a flow-insensitive, context-insensitive pointer analysis never decreases; hence PTA is also monotonic.

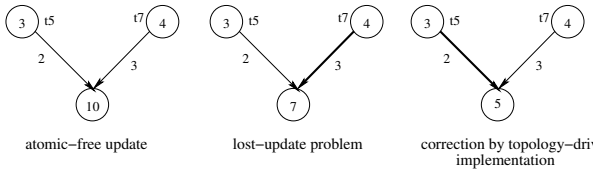

Figure 7: The lost-update problem and its solution

In our experience, monotonicity is a key algebraic property to avoid synchronization. We illustrate it on the SSSP computation. In SSSP, multiple threads may simultaneously update the distance of the same node, necessitating synchronization (in the form of `atomicMin` instructions). In the absence of atomics, however, concurrent thread execution may result in lost updates. For example, a node with a current distance of 10 may be updated by two threads, one updating it to 7 and another to 5 (see Figure 7). However, due to the data race, the node's distance may first be updated to 5 and then to 7. Thus, the node may end up with a final distance of 7, which is incorrect. This is called the *lost update* problem, and it happens because checking whether to update the value (compare) and the actual value update (write) are two separate operations. Such an implementation may compute an incorrect solution to SSSP.

However, if this atomic-free algorithm is implemented in a topology-driven manner [21], it is guaranteed that the lost update will be reconsidered in the next iteration because a topology-driven algorithm processes all nodes in each iteration. (In contrast, a data-driven algorithm processes only the modified nodes in each iteration using a worklist.) Figure 8 shows the pseudo code of data-driven and topology-driven processing. The data-driven version operates on a worklist and processes work items until the worklist is empty. Note that only the newly created work items are added to the worklist, making it work-efficient. A topology-driven approach, in contrast, processes all possible work items (*e.g.*, all graph nodes) as long as at least one of them is active (*i.e.*, changes the underlying data). This kind of processing usually makes a topology-driven approach work-inefficient, but it may be suited for GPU-based processing due to the large number of available threads (*cf.* Nasre *et al.* [18]).

Hence, in a topology-driven atomic-free SSSP implementation, even if a node attains a larger distance in the current iteration due to a lost update, the node will be active in the next iteration and the distance is guaranteed to eventually be reduced to the true minimum. In the above example where the current node's distance is 10, it may be updated to 7 by thread $\texttt{t}_7$, overwriting the value of 5 that was just written by thread $\texttt{t}_5$ in iteration `i`. In the next iteration `i + 1`, thread $\texttt{t}_7$ does not update the distance as it is already set to 7. However, thread $\texttt{t}_5$ will reduce it to the correct value of 5. This approach generalizes to multiple threads, and it can be proven that at least one thread updates data, thus ensuring progress. This guarantees that a distance will eventually attain the minimum in at most $\texttt{t} - 1$ iterations, where `t` is the number of threads racing to update the distance.

```
// Data driven
worklist.init();
while !worklist.empty() do
  e = worklist.pop();
  newelems = process(e);
  worklist.pushall(newelems);
```

```
// Topology driven
repeat
  changed = false;
  for all graph elements e do
    if e is active then
      process(e);
      changed = true;
    end if
until !changed
```

Figure 8: Data-driven versus topology-driven

```
// with atomics
dsrc = dist[src];
for all edges <src,dst,wt> do
  ddst = dist[dst];
  if ddst > dsrc + wt then
    atomicMin(&dist[dst],
    dsrc + wt);
```

```
// atomic-free
dsrc = dist[src];
for all edges <src,dst,wt> do
  ddst = dist[dst];
  if ddst > dsrc + wt then
    dist[dst] = dsrc + wt;
```

Figure 9: SSSP operator with and without atomics

Thus, a topology-driven implementation can be made atomic-free if the underlying computation is monotonic. Figure 9 shows the SSSP computation for a node `src` that involves relaxing all its outgoing edges. When monotonicity is not exploited, we are forced to use `atomicMin` to ensure correct computation. However, by making use of the monotonicity property, coupled with a topology-driven approach, we can remove the synchronization and update the distance directly.

Apart from SSSP, monotonicity is exhibited by breadth-first search (BFS) and survey propagation (SP). In BFS, the level number of a node never increases. In SP, the propagation of surveys involves multiplication of the probabilities of several literals (the probability indicates how close to a truth value a literal is). Since each probability is a real number in the range 0..1, multiplication by a probability never increases the value, even if the probability of a literal changes arbitrarily. We exploit this fact to avoid atomics in SP when storing the results of these multiplications.

The work efficiency of an atomic-free implementation can be poor due to *stale reads* since a thread may read and propagate an outdated value. For instance, in SSSP, it can take several iterations until the actual minimum distance of a node becomes available. This results in wasted work, leading to poor work efficiency. On a multi-core system, a topology-driven approach is not only work inefficient but may also result in inferior performance. However, on a massively parallel GPU, significant performance can be achieved at the cost of some work inefficiency by removing the bottleneck of a centralized worklist (due to the topology-driven implementation) and the cost of using atomics (due to monotonicity).

Monotonic size In some algorithms, instead of the *value* being computed, the *size* of the computed information increases monotonically. Flow-insensitive, context-insensitive inclusion-based pointer analysis [1] is an example. In this algorithm, more and more points-to information is computed as the analysis progresses, but no points-to information is ever *deleted*, leading to a points-to solution that monotonically increases in size.

Consider the example shown in Figure 10 where the program consists of the points-to constraints shown in the left-most column. Each column on the right shows the corresponding points-to information computed by the constraints

	Points-to information		
Program	Iteration 0	Iteration 1	Iteration 2
a = &x	a → x	a → x	a → x
q = &b	q → b	q → b	q → b
*q = c			b → x
c = a		c → x	c → x

Figure 10: PTA exhibits monotonicity

in each iteration of the analysis. Note that the points-to information only increases due to flow-insensitive analysis.

When implemented on a GPU, PTA takes points-to constraints as input (*e.g.*, `p = &q`) and outputs a set of points-to facts (*e.g.*, `p` → `q`). The set of points-to facts computed at the end of the analysis constitutes the points-to solution of the input program. Since the analysis is performed on the GPU, the final solution needs to be copied to the CPU. Since computing a points-to fact and copying it are conflicting tasks, there is a need for synchronization.

Usually, the synchronization is implemented as an implicit barrier – when the GPU threads complete the processing, the analysis kernel terminates and the CPU commences the copying of the points-to solution. However, the CPU can initiate the copying as soon as some (although not all) points-to information is available. To avoid races, shared access to the points-to information must be synchronized between the CPU and the GPU. Current NVIDIA GPUs do not support inter-device atomic instructions, which poses a challenge to this asynchronous copying of points-to information.

Monotonicity comes to the rescue once again. Since the points-to information increases monotonically in size, it is okay to copy stale points-to information from the GPU to the CPU. Any missed information will be copied in a future iteration and, due to the monotonicity, the CPU is guaranteed to see a consistent copy of the information. This allows us to avoid synchronization for accessing the points-to information – the GPU can safely write to it while the CPU asynchronously copies it. In our set of programs, the copying time per iteration is always less than the per-iteration analysis time. Therefore, the copying time can be hidden except for the small amount of points-to information computed in the last iteration. This asynchronous copying provides considerable performance benefits in PTA.

4.2 Idempotency

The second algebraic property we exploit to avoid synchronization is idempotency. A computation is idempotent if its repeated application is equivalent to a single application. Formally, $\forall x : f(f(x)) = f(x)$. For instance, all our topology-based computations of irregular algorithms make use of idempotency to detect termination – an extra round of processing is performed in the end and if no changes are detected, then the fixed-point solution has been reached. Since the computation is idempotent, this extra round does not change the computed solution.

We now discuss how idempotent computations enable atomic-free implementations. Consider the example of a worklist in a data-driven SSSP implementation. In the absence of any race-and-resolve mechanism (described in Section 3.2), a graph node may occur multiple times in the worklist. Therefore, when the worklist items are assigned to threads, two threads may end up operating on the same node and may relax the same set of edges.

In general, this scenario is avoided using synchronization. For example, each thread can atomically mark its assigned nodes. If the marking succeeds, the thread owns the node and processes it. If the marking does not succeed, some other thread owns and processes the node. An alternative way of solving this issue is using the race-and-resolve mechanism. However, in SSSP, the relax-edge kernel performs an idempotent operation. Therefore, even if multiple threads operate on the same graph node and relax the same set of edges, the effect is as if the operation is performed only once. Hence, we can afford to implement the worklist as a multiset (avoiding the cost of converting it to a duplicate-free set) and still avoid synchronization between threads. This is possible solely due to the idempotency of the computation.

The example of spurious data-races in BFS from Section 3.3 may also be viewed as an artifact of idempotent computation – since data updates from multiple threads are idempotent (same value), synchronization across those threads is unnecessary.

Another application with idempotent processing is points-to analysis. Recall from Section 4.1 that the CPU can asynchronously copy points-to information while the GPU continues the analysis. This copying is typically done by merging the newly computed points-to information of each node with its previously copied information. Merging of points-to sets is an idempotent operation. Therefore, it is okay to copy the same points-to fact multiple times from the GPU to the CPU. In the absence of idempotency (*e.g.*, had the data structure been a list instead of a set), one would be forced to keep track of the last element added and copy only the new information following the last element. This would necessitate synchronization in the form of an atomic compare-and-swap instruction (`atomicCAS`).

Discussion. One may think that implementing a worklist as a multiset may go on propagating the duplicate items in the worklist over several iterations. However, note that although multiple threads operate on a node, only one would succeed in updating a node's distance. This is because a worklist-based SSSP relies on atomic instructions to update a node's distance. Therefore, although duplicates continue to get added to the worklist, they do not get propagated.

Exploiting idempotency usually involves redundant work, making the computation work-inefficient. However, it often enables avoiding synchronization as we discussed above. Therefore, idempotency must be exploited by balancing the cost of redundant computation against the synchronization cost. On a multi-core system running with at most a few hundred threads, the redundant work can quickly become a considerable factor. However, on a GPU running with hundreds of thousands of threads, computation is often cheap whereas the synchronization cost is high. Therefore, exploiting idempotency is especially suited to GPUs.

4.3 Associativity

The third algebraic property we exploit for avoiding atomics is associativity. A computation is associative if two applications of the computation can be reordered without changing the result. Formally, $f(a, f(b, c)) = f(f(a, b), c)$. For instance, hierarchical reductions and prefix sums exploit the associativity of the computation. Similarly, the merging step in parallel merge-sort is associative, *i.e.*, irrespective of the order in which individual sorted lists are merged, the end result is the same (assuming unique elements).

We now explain how associativity can help avoid atomics. Consider, once again, points-to analysis. In PTA, points-to information is propagated along pointers represented as nodes in a constraint graph. When a node has multiple incoming edges, points-to information may be propagated to it by multiple threads in parallel. A naïve way to implement this propagation is to have each thread atomically increase an offset at the destination node by the number of new points-to facts it wants to add. However, because merging of points-to sets is an associative operation, one can eliminate the use of atomic instructions by computing a prefix sum of the number of points-to facts each thread wants to add. This avoids atomics and may improve performance.

5. OTHER METHODS

There are other well-known ways to avoid atomics in irregular computations. These methods are often based on the single-writer policy (also called the owner-computes rule), wherein the work distribution to threads is performed in such a way that threads can operate without conflicts.

5.1 Graph Partitioning

Graph partitioning divides the underlying graph into sets of nodes and each partition is assigned to a thread. This ensures that if a thread only operates on the nodes in its partition, it does not need to synchronize with other threads. Regular, matrix-based codes often exhibit this behavior naturally, which can be exploited through blocking and tiling. In contrast, a node-ID-based partitioning is often ineffective on graphs due to their input-dependent connectivity.

Consider, for instance, Delaunay mesh refinement (DMR). Each bad triangle in the mesh is assigned to a thread, which forms a cavity around the bad triangle. Since refining a cavity requires exclusive access to all the involved triangles, it is beneficial to perform the work distribution in such a manner that each cavity falls entirely into one partition. In general, this is impossible to achieve since cavities may overlap arbitrarily. The goal of graph partitioning, then, is to minimize the overlap so that the maximum number of cavities may be processed in each iteration. Such a partitioning needs to take into account a node's connectivity for assigning nearby nodes to the same partition.

Unfortunately, such a layout-based graph partitioning requires traversing the graph, which quickly becomes a performance bottleneck. The partitioning itself can be parallelized (*e.g.*, with ParMetis [13]), but the partitioning time is still much larger than the total running times of our algorithms. Therefore, we do not evaluate partitioning in this paper. However, when the partitioning cost is not a concern, layout-based graph partitioning can help reduce conflicts.

In cases where the processing can be restricted to a single item (vis-a-vis multiple triangles as in DMR), it is possible to partition the graph across threads based on the node identifiers. The processing can then be modified to ensure a single-writer policy. This leads to pull-based (as opposed to push-based) implementations of graph algorithms.

For instance, SSSP's operator can be restricted to a single item, *i.e.*, a thread may operate on a single node. Typically, the operator performs edge relaxations on all outgoing edges of a node, potentially modifying distances of multiple (neighboring) nodes. This is a push-based approach. But the SSSP computation can be altered such that the operator

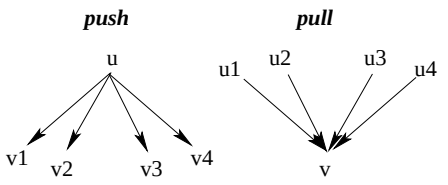

Figure 11: Push- versus pull-based processing

reads the distances of the incoming neighbors when updating the distance of a node. This is a pull-based approach. The two approaches are shown in Figure 11. A pull-based approach may be beneficial in certain cases as it enables avoiding atomics by exploiting the single-writer policy. In a pull-based approach, a node's distance is updated only by the single thread operating on that node. In contrast, in a push-based approach, a node's distance may be updated by multiple threads operating on the incoming neighbors of that node. Therefore, a push-based approach requires synchronization for updating the distance of a node, whereas it is possible to eliminate synchronization in a pull-based approach (as long as the read-write races do not affect the result). This approach has been used to accelerate PTA [16].

5.2 Scatter Gather

Consider once more the example of threads that wish to insert elements into a global worklist. As described earlier, this can be implemented in an atomic-free fashion using a barrier-based prefix-sum computation, *cf.* Section 3. In certain cases, the cost of the global barrier, which is an expensive operation, can be reduced by pushing elements in a scatter-gather like manner as we discuss next.

If the maximum number of elements a thread may push is bounded by a small constant, it may be possible to push elements directly in parallel without performing a prefix-sum computation. Several practical algorithms bear this property. For instance, many road networks have a small maximum degree, which can be utilized as bounds in SSSP, BFS, *etc.* Similarly, in DMR, each cavity comprises no more than 12 triangles for all our meshes, which can be used as this small constant. Given such a constant, work items can be pushed onto the worklist as follows. Each thread is assigned a c-element wide range in the worklist, where c is the maximum number of elements to be pushed by a thread. No two ranges overlap. Threads push their work items in parallel into their range, marking any holes (places in the worklist range with no elements) with a special value. At this stage, the worklist contains all the work items to be processed, but they are scattered. A post-processing step then removes the holes from the worklist, thereby compacting (or gathering) the set of active work items. This scatter-gather processing does not require atomics. Note, however, that a barrier is needed between the two phases. In some cases, a gather operation may not be required at all. For instance, one may keep the holes in the worklist and simply process all the work items, ignoring the holes during processing.

6. EXPERIMENTAL EVALUATION

We perform our experiments on a Fermi- and a Kepler-based GPU. The Fermi-based GPU is a 1.45 GHz Quadro 6000 with 6 GB of main memory and 448 CUDA cores distributed over 14 SMs. The Kepler-based GPU is a 0.7 GHz Tesla K20 with 5 GB of main memory and 2496 CUDA cores distributed over 13 SMXs. Both GPUs have 64 kB of fast memory per SM that is split between the L1 data cache and the shared memory. We compiled the CUDA programs with *nvcc* v5.0 using the *-O3 -arch=sm_20* flags on the Fermi and the *-O3 -arch=sm_35* flags on the Kepler.

B/M	#K	Inputs
BFS	2	RMAT22 (4 M nodes, 32 M edges), RANDOM23 (8 M nodes, 32 M edges), USA road network (23 M nodes, 58 M edges)
DMR	4	1 M, 2 M, 3 M triangles with ∼50% bad
PTA	40	vim (172,188 pointers, 246,944 constraints), pine (406,990 pointers, 612,928 constraints), tshark (642,333 pointers, 1,555,840 constraints)
SP	3	1 M – 5 M literals, clauses/lit.=4.2, 3 lit./clause
SSSP	2	RMAT22 (4 M nodes, 32 M edges), RANDOM23 (8 M nodes, 32 M edges), USA road network (23 M nodes, 58 M edges)

Table 1: Applications and their input characteristics. B/M = benchmark, #K = number of kernels

Our benchmark programs stem from the LonestarGPU suite and their inputs are listed in Table 1. We chose three types of graphs for BFS and SSSP, three randomly generated meshes for DMR, three open source programs for PTA, and five randomly generated 3-SAT formulae for SP. Survey propagation was proposed to deal especially with SAT formulae that take particularly long to assess [5]. 3-SAT problems are known to be hard (time-consuming) to solve when the ratio of the number of clauses to the number of literals is around 4.2. In our evaluation, we chose the number of clauses and literals to generate such hard-SAT instances.

6.1 Effect of Barrier-based Processing

Figure 12 shows the effect of barrier-based worklist processing in SSSP on the Fermi and Kepler GPUs. In the base version, threads push the newly updated nodes onto the worklist using atomic instructions, whereas in the atomic-free version threads perform a prefix sum for adding items to the worklist (*cf.* Section 3.1). We observe that atomic-free SSSP performs consistently better than the base version with atomics. The performance improvement is more pronounced for the USA road network on Fermi and for the RANDOM graph on Kepler. On average, the atomic-free version requires 16% and 8% less time than the base version for computing the shortest paths on the two GPUs.

The considerable performance difference between Fermi and Kepler for the RMAT graph is due to an interplay of architectural differences and input-graph characteristics. The RMAT graph is denser and has a higher out-degree per node. Therefore, more (sequential) work per thread needs to be done in terms of uncoalesced memory accesses. Our Kepler has a narrower bus (320 bits) in comparison to our Fermi (384 bits). Therefore, application performance is more susceptible to (irregular) memory accesses on Kepler, of which there are many in memory-bound kernels like SSSP. We suspect some inputs like RMAT to exacerbate this situation, which is why it takes longer to execute on the Kepler.

Figure 13 shows the effect of barrier-based locking in DMR on Fermi and Kepler GPUs. In the base version, threads use atomic instructions to ensure exclusive ownership of a cavity's triangles, whereas in the atomic-free version threads perform a race-and-resolve operation (*cf.* Section 3.2). We

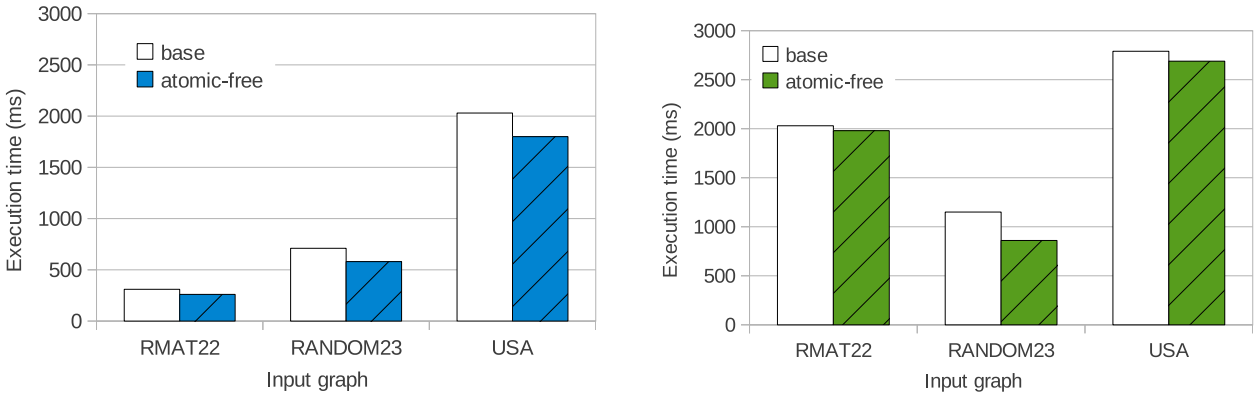

Figure 12: Effect of barrier-based processing on SSSP for different input graphs (Fermi and Kepler)

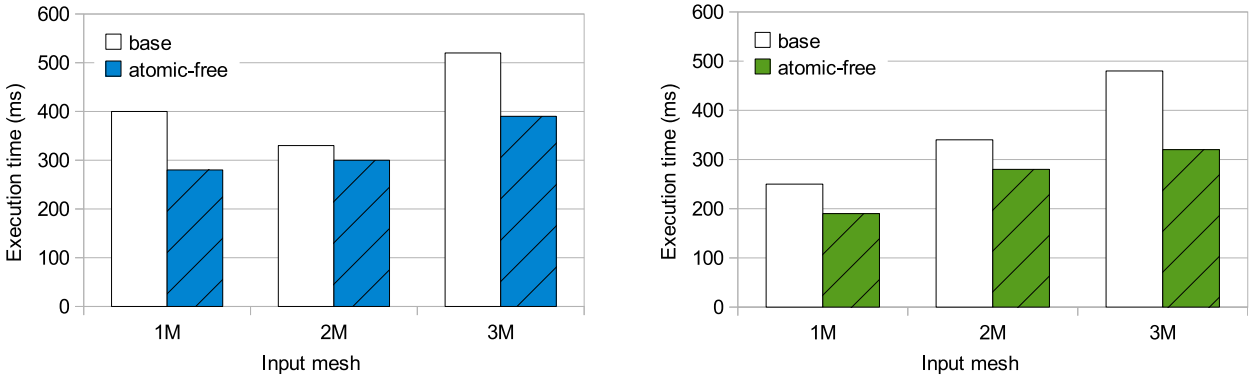

Figure 13: Effect of barrier-based processing on DMR for different input meshes (Fermi and Kepler)

observe that atomic-free DMR performs consistently better than the base version with atomics.

Interestingly, DMR runs faster on a larger mesh (with 2 million triangles) than a smaller mesh (with 1 million triangles) on the Fermi. This effect is mainly due to the geometry of the randomly-generated input mesh, which affects the non-deterministic DMR execution. However, we note that the Kepler does not exhibit this anomaly.

To confirm that the change in performance is, in fact, due to the different ways of synchronizing (with and without atomics), we plot the percentage of triangles refined in each iteration of DMR in Figure 14. The profile is obtained on the Fermi GPU when running the two DMR versions on the input mesh with 1 million initial triangles. We observe that the two superimposed line graphs follow almost the same execution pattern. The differences are minor and are primarily due to the non-deterministic execution, which affects the order in which triangles are processed. This plot indicates that the performance difference between the base version and the atomic-free version shown in Figure 13 is mainly due to the cost of the atomics.

In case of atomic-free DMR, the usage of a global barrier poses the constraint that the number of blocks is limited by the number of blocks that can simultaneously be running in the GPU, requiring a persistent-thread-based implementation. There is no such requirement for the base case using atomics. However, using a global barrier does not significantly reduce the optimization opportunities in DMR.

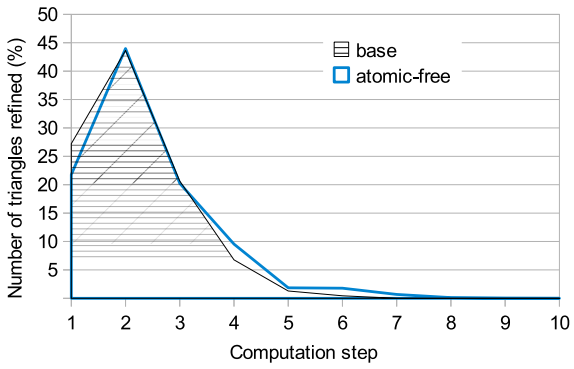

Figure 14: Processing of triangles in DMR (Fermi)

6.2 Effect of Exploiting Algebraic Properties

Figure 15 shows the effect of exploiting monotonicity in SP on Fermi and Kepler GPUs. In the base version, threads use atomic instructions to update the product of the probabilities at each clause node, whereas in the atomic-free version threads update it directly (*cf.* Section 4.1). We observe that the benefit of avoiding atomics grows with the size of the

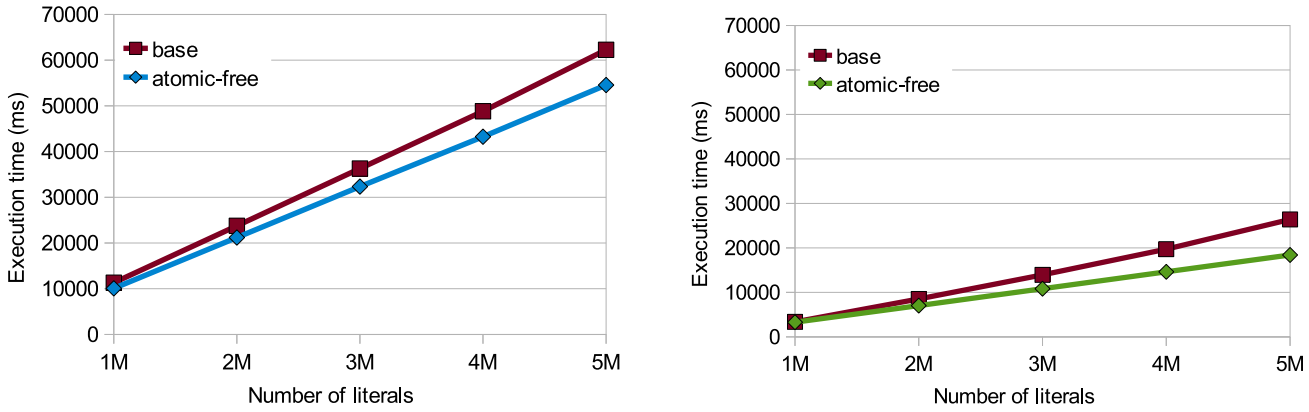

Figure 15: Effect of exploiting algebraic properties on SP for different 3-SAT inputs (Fermi and Kepler)

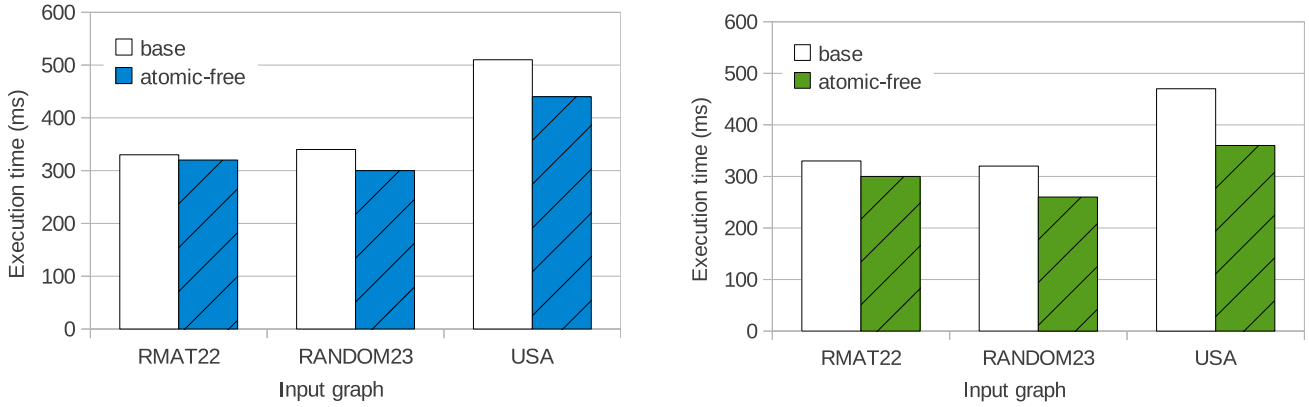

Figure 16: Effect of exploiting algebraic properties on BFS for different input graphs (Fermi and Kepler)

input SAT formula. In particular, by increasing the number of literals from 1 to 5 million, the performance between the two SP versions varies from 14% to 43% on the Kepler.

We also note that SP executes faster on Kepler than on Fermi. This is mainly due to more registers being available per thread block on Kepler (64K) compared to Fermi (32K). The additional registers considerably help compute-intensive kernels like SP, resulting in more than a 2× speedup.

Figure 16 shows the effect of exploiting idempotency in BFS on Fermi and Kepler GPUs. In the base version, additional processing is performed to keep only the unique items in the worklist, whereas in the atomic-free version no such step is included and multiple threads may operate on the same node (*cf.* Section 4.2). We observe that avoiding atomics results in consistently better performance, with the maximum impact on the USA road network. On average, atomic-free BFS executes 11% and 22% faster on the Fermi and Kepler, respectively, compared to the base versions.

We also note that level-by-level BFS is neither memory-bound nor compute-intensive. Hence, its relative performance on the two GPUs is quite similar.

Figure 17 shows the percentage of nodes finalized in each iteration of BFS on the USA road network. The plot is obtained on the Fermi and is the same for both the base and the atomic-free versions. The processing follows a profile similar to a normal distribution, with a small number of nodes finalized in the initial iterations, exponential growth in the middle, and dropping off as the threads start running out of work. The plot is also indicative of the amount of parallelism exhibited by BFS on road networks.

Figure 18 shows the effect of monotonicity on PTA. The plot depicts the savings due to overlapping communication and computation of points-to information. The left set of bars shows the additional GPU-to-CPU (d2h) communication time in milliseconds in the non-overlapped processing. The right set of bars indicates the additional non-hidden GPU-to-CPU memory transfer amount in megabytes. These overheads are avoided by exploiting monotonicity in copying the points-to information to the CPU (*cf.* Section 4.1).

In summary, avoiding atomic instructions in irregular algorithms using barrier-based processing and exploiting algebraic properties offers considerable performance benefits.

7. RELATED WORK

There are many implementations of parallel graph algorithms on a variety of architectures, including distributed-memory supercomputers [25], shared-memory supercomputers [2], and multi-core SMP machines [14].

GPUs have also been used for the acceleration of irregular programs. Harish and Narayanan [10] describe implemen-

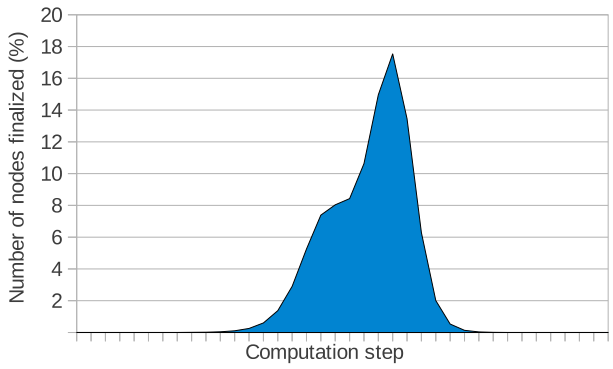

Figure 17: Processing of nodes in BFS (Fermi)

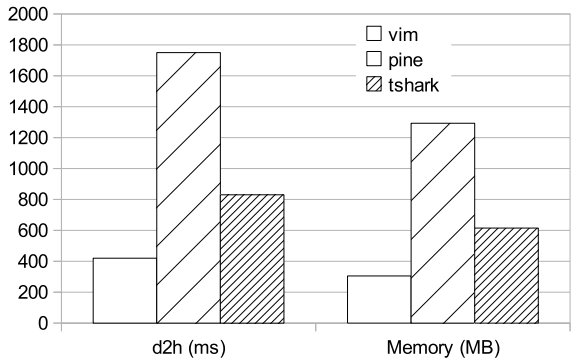

Figure 18: Savings by exploiting monotonicity of PTA (Fermi)

tations of important graph algorithms such as BFS, Single-Source Shortest Paths, Minimum Spanning Tree, *etc.* Vineet *et al.* [23] and Nobari *et al.* [20] propose computing the minimum spanning tree and forest, respectively, on GPUs. All of these approaches rely on atomic instructions for updating shared data, and none of them exploit algebraic properties.

Martín *et al.* [15] propose a GPU implementation of Dijkstra's algorithm to compute the shortest paths. Their algorithm is synchronous and is based on computing a frontier. Similar to others, they use `atomicMin` instructions to update the node distances.

Hong *et al.* [11] propose a warp-centric approach for the parallelization of BFS. Our approach is orthogonal to their work and the two techniques can likely be combined to achieve improved performance.

Merill *et al.* [17] present a worklist-based work-efficient BFS. Theirs and the previous approaches to BFS rely on level-synchronous updates of nodes. This avoids the use of atomic instructions since the data-races inside a level are benign. It is a special case of the approach we propose in Section 3.3, which is more general and encompasses other interesting cases that are common in irregular computations.

Garland [9] proposes sparse matrix computations on GPUs and discusses how matrix multiplication can be used for computing the shortest paths in a sparse graph.

Nasre *et al.* [19] discuss morph algorithms on GPUs. Their focus is on supporting graph operations like node insertion and deletion and not about the removal of atomics. We use their barrier-based race-and-resolve scheme for overlapping accesses and extend it to support OR contexts for owning *any* of the work items of interest (*cf.* Section 3.2).

Putta and Nasre [22] exploit algebraic properties to improve parallelization. They take advantage of monotonicity and the unordered nature of flow-insensitive points-to analysis to design a replication-based algorithm. Their work involves creating multiple copies of shared data and targets multi-core CPUs. Our work deals with a single data copy, targets GPUs, and focuses on avoiding atomics.

Previous research on GPU implementations of irregular algorithms has focused on optimizing a specific feature of irregular programs for GPU execution. This includes work on removing dynamic irregularities from irregular applications [26] and on optimizing CPU-GPU transfers for dynamically managed data [12]. G-Streamline is a software-based runtime approach to eliminate control-flow and memory-access irregularities from GPU programs [26]. DyManD is an automatic runtime system for managing recursive data structures (like trees) on GPUs [12]. These approaches are orthogonal to our work.

This paper is the first proposal targeting atomic-free implementation of irregular algorithms on GPUs.

8. CONCLUSIONS

Accelerating irregular applications is a challenging task, in particular on GPUs. A critical optimization in addressing this challenge is to minimize the use of atomic instructions. Towards this goal, we present key methods to avoid atomic primitives. Specifically, we focused on eliminating atomics using barrier-based synchronization and by exploiting algebraic properties of computations. For each method, we discuss several real-world scenarios and the associated techniques for eliminating atomic operations. Using Fermi and Kepler GPUs, we illustrate the efficacy of our methods by applying them to five well-known graph algorithms that represent a range of complexity and pose different sets of challenges. The techniques presented in this paper offer substantial performance benefits on these irregular applications. For example, using barrier-based processing improves the running time of Delaunay mesh refinement by 29% on Fermi, and exploiting algebraic properties improves the running time of survey propagation by 43% on Kepler. We believe that other irregular codes will likely also obtain good speedups when incorporating the discussed techniques.

Acknowledgments

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

9. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] D. A. Bader and K. 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] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 181–192, New York, NY, USA, 2012. ACM.

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

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

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to algorithms, McGraw Hill, 2001.

[8] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Comput.*, 26(12):1555–1581, Nov. 2000.

[9] M. Garland. Sparse matrix computations on manycore GPU's. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 2–6, New York, NY, USA, 2008. ACM.

[10] P. 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] S. Hong, S. K. Kim, T. Oguntebi, and K. 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] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 165–174, NY, USA, 2012. ACM.

[13] G. Karypis and V. Kumar. Multilevel K-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not.*, 42(6):211–222, 2007.

[15] P. J. Martín, R. Torres, and A. Gavilanes. CUDA Solutions for the SSSP Problem. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 904–913, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] M. Mendez-Lojo, M. Burtscher, and K. 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.

[17] D. G. Merrill, M. Garland, and A. S. Grimshaw. Scalable GPU Graph Traversal. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'12, 2012.

[18] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the 27th IEEE International Parallel & Distributed Processing Symposium*, IPDPS '13, 2013.

[19] R. Nasre, M. Burtscher, and K. Pingali. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[20] S. Nobari, T.-T. Cao, P. Karras, and S. 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.

[21] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. 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.

[22] S. Putta and R. Nasre. Parallel Replication-Based Points-To Analysis. In M. F. P. O'Boyle, editor, *CC*, volume 7210 of *Lecture Notes in Computer Science*, pages 61–80. Springer, 2012.

[23] V. Vineet, P. Harish, S. 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.

[24] S. Xiao and W. chun Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, pages 1–12. IEEE, 2010.

[25] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. 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.

[26] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. 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*, pages 369–380, New York, NY, USA, 2011. ACM.