Open access • Proceedings Article • DOI:10.1109/IPDPS.2013.28

Data-Driven Versus Topology-driven Irregular Computations on GPUs

— Source link

Rupesh Nasre, Martin Burtscher, Keshav Pingali

Institutions: University of Texas at Austin, Texas State University

Published on: 20 May 2013 - International Parallel and Distributed Processing Symposium

Topics: Graph (abstract data type), Data structure and Operator (computer programming)

Related papers:

- Scalable GPU graph traversal
- Accelerating CUDA graph algorithms at maximum warp
- Morph algorithms on GPUs
- A quantitative study of irregular programs on GPUs
- The tao of parallelism in algorithms

Share this paper:

View more about this paper here: https://typeset.io/papers/data-driven-versus-topology-driven-irregular-computations-on-2kgkbr0w62

Data-driven versus Topology-driven Irregular Computations on GPUs

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

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

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

***Abstract*—Irregular algorithms are algorithms with complex main data structures such as directed and undirected graphs, trees, etc. A useful abstraction for many irregular algorithms is its *operator formulation* in which the algorithm is viewed as the iterated application of an operator to certain nodes, called *active nodes*, in the graph. Each operator application, called an *activity*, usually touches only a small part of the overall graph, so non-overlapping activities can be performed in parallel. In *topology-driven implementations*, all nodes are assumed to be active so the operator is applied everywhere in the graph even if there is no work to do at some nodes. In contrast, in *data-driven implementations* the operator is applied only to nodes at which there might be work to do. Multicore implementations of irregular algorithms are usually data-driven because current multicores only support small numbers of threads and work-efficiency is important. Conversely, many irregular GPU implementations use a topology-driven approach because work inefficiency can be counterbalanced by the large number of GPU threads.**

In this paper, we study data-driven and topology-driven implementations of six important graph algorithms on GPUs. Our goal is to understand the tradeoffs between these implementations and how to optimize them. We find that data-driven versions are generally faster and scale better despite the cost of maintaining a worklist. However, topology-driven versions can be superior when certain algorithmic properties are exploited to optimize the implementation. These results led us to devise hybrid approaches that combine the two techniques and outperform both of them.

Keywords: irregular algorithms, data-driven, topology-driven, algorithmic properties, GPGPU

I. INTRODUCTION

GPUs are very effective at exploiting parallelism in *regular*, data-parallel algorithms, and there is a broad knowledge base on the efficient GPU parallelization of these algorithms [7]. However, many problem domains employ algorithms that build, traverse, and update *irregular* data structures such as trees, graphs, and priority queues. Examples of these problem domains include *n*-body simulation [2], data mining [25], Boolean satisfiability [4], optimization theory [9], social networks [14], compilers [1], discrete-event simulation [21], and meshing [8]. We know much less about how to produce efficient implementations of irregular algorithms on GPUs.

Recently, several case studies of irregular algorithm implementations on GPUs have appeared in the literature. For example, Harish *et al.* have implemented breadth-first search (BFS) and single-source shortest-paths (SSSP) algorithms [11], [12], Burtscher *et al.* have implemented the Barnes-Hut *n*-body (BH) algorithm [6], and Mendez-Lojo *et al.* have implemented Andersen-style points-to analysis [19] on GPUs. A common feature of these implementations is that all the nodes of the underlying graph are examined at each super-step of the algorithm even if there is no work to be done at some of the nodes. In contrast, the recent BFS implementation of Merrill *et al.* [20] maintains a global worklist of nodes where computation needs to be performed, so it avoids visiting nodes at which there is no work to be done.

A useful abstraction for understanding these differences is the *operator formulation* of algorithms [24], which is a data-centric abstraction for parallel algorithms. In this abstraction, the algorithm is viewed as the iterated application of an *operator* to the *active nodes* in the graph. Each such *activity* usually only touches a small fraction of the graph, allowing non-overlapping activities to execute in parallel. Implementations are classified as either *topology-driven* or *data-driven*. In topology-driven implementations, all nodes are assumed to be active so the operator is applied everywhere even if there is no work to do at some nodes; the aforementioned BH and SSSP implementations are examples. Data-driven implementations, in contrast, apply the operator only to nodes at which there might be work to do, as in Merrill *et al.*'s BFS implementation.

In our experience, topology-driven implementations on GPUs are comparatively easy to write and maintain: one only needs to worry about a single unit of work when implementing a kernel, and the kernel is invoked with as many threads as there are work items. However, the work-efficiency of topology-driven implementations can be a concern since many threads end up either performing redundant work or finding that they do not need to do any work in the current super-step. Data-driven implementations often need a worklist that tracks nodes that need processing because nodes may become active as a result of processing other active nodes. Apart from the overhead of maintaining this worklist, one needs to worry about synchronization between thousands of GPU threads accessing the worklist and the performance of worklist updates, as these may require costly atomic operations. Of course, topology-driven and data-driven approaches need not be mutually exclusive and one may be able to combine them in a beneficial manner for some problems. Therefore, a systematic evaluation of the two approaches is critical to determine the best implementation strategy for irregular GPU algorithms.

This paper makes the following contributions towards this goal.

- We perform qualitative and quantitative comparisons of data-driven and topology-driven irregular GPU codes.
- For each approach, we propose several optimizations to exploit the GPU's computation capabilities and avoid its limitations. Specifically, for data-driven approaches, we propose and/or evaluate hierarchical worklists, work chunking, work donating and variable kernel configuration. For topology-driven approaches, we propose and/or evaluate kernel unrolling, exploiting shared memory and improved memory layouts.
- We propose two hybrid schemes that combine the benefits of data-driven and topology-driven approaches. Specifically, we combine the two schemes temporally and spatially in an attempt to outperform either of the techniques.
- We compare the two approaches using six irregular programs and find that an optimized data-driven implementation is usually superior. For a topology-driven implementation to execute faster, it is essential to exploit additional algorithmic properties. We evaluate the proposed optimizations individually as well as in combination. We obtained significant performance improvements with atomic-free worklist updates, variable kernel configuration, kernel unrolling and intra-block work donation. We also find that a spatial hybrid degrades performance whereas a temporal hybrid can improve performance.

The rest of this paper is organized as follows. Section II reviews modern GPU hardware as well as the CUDA programming model and defines data-driven and topology-driven implementations. We also describe the six irregular applications studied in this paper. Section III discusses the data-driven approach as well as applicable optimizations. Section IV discusses the topology-driven approach and applicable optimizations. Section V proposes two ways to combine the two approaches to achieve hybrid implementations. Section VI evaluates the optimized data-driven and topology-driven versions of our irregular applications and the effect of individual optimizations. Section VII compares these results with prior work in the literature. Section VIII presents our conclusions.

II. BACKGROUND

This section introduces data-driven and topology-driven implementations of irregular algorithms and describes the irregular applications studied in the paper. Although we focus on Fermi GPUs in this paper, these concepts should be applicable to other similar architectures.

A. Regular versus irregular algorithms

The terms *regular* and *irregular* stem from the compiler literature (where the terms *structured* and *unstructured* are also used). 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. In irregular code, the input values to the program determine the program's runtime behavior, which therefore cannot be statically predicted and may be different for different inputs. For example, in a binary search tree implementation, the values and the order in which they are processed affect the shape of the search tree and the order in which it is built.

In most problem domains, irregular algorithms arise from the use of complex data structures such as trees and graphs. In general, irregular algorithms are more difficult to parallelize and more challenging to map to GPUs than regular programs. For example, in graph applications, 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, leading to uncoalesced memory accesses and bank conflicts. Similarly, the control flow is usually irregular because branch decisions differ for nodes with different degrees (neighbor counts) or labels, leading to thread divergence and load imbalance.

B. Data-driven versus topology-driven approaches

Data-driven implementations visit nodes only if there may be work that needs to be performed there. Since applying the operator to an active node may cause other nodes to become active, data-driven implementations are organized around a worklist; threads obtain work by pulling active nodes from the worklist, and they may add new active nodes to the worklist after processing an active node. To ensure that concurrently executing threads do not interfere with each other, it is necessary to insert appropriate synchronization. For example, in SSSP (see below for description), synchronization is needed at the nodes to ensure that updates happen atomically. Synchronization is also needed at the worklist to ensure that active nodes are added and removed correctly. Data-driven implementations on multicores usually implement the worklist in a physically distributed manner to reduce contention [23].

Implementing an efficient worklist-based approach on a GPU is non-trivial since it involves managing a central worklist that is shared by thousands of threads. There are two aspects of the worklist that an efficient algorithm needs to handle: pulling active nodes from the worklist and putting new active nodes onto the worklist. Getting active nodes from the worklist can be implemented by partitioning the worklist between threads. Each thread then operates on the nodes in the partition assigned to it, without synchronizing with other threads. Putting new active nodes onto the worklist can be done by atomically incrementing the length of the worklist and then writing an active node's identifier to that location in the worklist. However, this way of writing to the worklist is inefficient and does not scale on GPUs. A better way is for each thread to keep track of the number of new work items it wishes to add to the worklist, and to update the length of the worklist once by that number instead of

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)
BH	9	1 M – 10 M bodies, 1 time step
DMR	4	1 M – 10 M triangles
MST	8	RMAT12 (4 K nodes, 28 K edges), NY (0.2 M nodes, 0.7 M edges), GRID20 (1 M nodes, 2 M edges)
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 I: LonestarGPU applications and their input characteristics (B/M = benchmark, #K = number of kernels)

incrementing it many times. After this, the thread can write node identifiers to the reserved range in the worklist without requiring further communication with other threads. An even better implementation can be obtained by recognizing that set/multiset union is a reduction operation, so we can take advantage of commutativity and associativity to implement this operation in a hierarchical manner as is done with other reduction operations like integer addition. The threads in a thread block can compute the total amount of work to be added by all threads in the thread block using a prefix sum (in shared memory) and then use a single atomic instruction per thread block to reserve space on the worklist. This hierarchical approach has been shown to be quite effective in achieving a scalable implementation of breadth-first search [20].

In a topology-driven approach, all graph nodes are considered to be active nodes, so every node is processed in each super-step. This approach often results in a work-inefficient implementation since there may not be any computation that needs to be performed at many nodes. This limitation prevents the widespread adaptation of topology-driven approaches in sequential and multi-core systems for large sparse graphs. Despite performing unnecessary work, a topology-driven approach has certain useful characteristics when implemented on a GPU. First, if the input graph is static, a fixed number of nodes can be assigned to each thread. Second, a topology-driven approach eliminates the need for an explicit worklist and the corresponding centralization bottleneck, enabling the algorithm in principle to scale better. Third, a fixed work allotment enables better load balancing and allows the sorting of nodes (*e.g.*, by out degrees) in a pre-processing step, which can reduce thread divergence. Finally, a topology-driven approach is usually easier to implement on a GPU.

C. Applications

We study the following six irregular programs from the LonestarGPU benchmark suite (available online at `http://iss.ices.utexas.edu`). Their properties (number of static kernels and inputs) are listed in Table I.

- Breadth-First Search (BFS): This graph problem is a key kernel in many important applications such as mesh partitioning. The BFS problem is to label each graph node with the node's minimum level (or number of hops) from a designated start node, which is at level zero.
- Barnes Hut (BH): This *n*-body algorithm simulates the effect of gravity on a star cluster [2], [6]. In each time step, the code hierarchically decomposes the space around the stars into successively smaller volumes, called cells, and computes the center of gravity of each cell to enable the quick approximation of the forces. The hierarchical decomposition is recorded in an octree. We investigate the kernel that computes the centers of mass of all octants.
- Delaunay Mesh Refinement (DMR): This is a mesh-refinement algorithm from computational geometry [8], [17]. 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 retriangulating the cavity around each bad triangle.
- Minimum Spanning Tree (MST): Boruvka's MST algorithm computes a minimal spanning tree through successive application of minimum weight edge contractions on an input graph. This process is repeated until the graph has just a single node, assuming the input is a connected graph. If the input graph is not connected, the algorithm computes a minimum spanning forest.
- Survey Propagation (SP): Survey Propagation is a heuristic SAT-solver based on Bayesian inference [4]. 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 [9]. In the topology-driven version, all edges are relaxed in each super-step (similar to Bellman-Ford's algorithm), whereas in the data-driven version, only the active nodes, *i.e.*, the nodes whose distances have recently changed, are processed.

The behavior of these applications is quite diverse, which proves useful to test different aspects of data-driven and topology-driven implementations. For instance, SP is compute-bound whereas BFS and SSSP are memory-bound. The amount of parallelism in BFS, BH, DMR, MST, SSSP and varies considerably during execution. DMR is a morph algorithm [24], *i.e.*, it modifies the underlying graph.

III. Data-driven approach

In this section, we discuss the implementation and optimization of data-driven algorithms on GPUs.

A. Base implementation

Data-driven implementations of graph algorithms use worklists to track active elements. The algorithm starts by populating the worklist with an initial set of graph elements; for example, in the single-source shortest-paths problem, the worklist is

```
main():
  read input                                 // CPU
  transfer input                             // CPU → GPU
  initialize_kernel()                        // GPU
  initialize_worklist(in)                    // GPU
  clear out worklist                         // GPU

  while in worklist not empty {
      process_kernel(in, out, ...)           // GPU
      transfer out worklist size             // CPU ← GPU
      clear in worklist                      // GPU
      swap(in, out)                          // CPU
  }
  transfer results                           // CPU ← GPU
```

Fig. 1: Pseudo code of data-driven implementation

initialized with the source node. A thread extracts an element (or a set of elements) from the worklist, performs algorithm-specific processing on it, identifies new active elements based on the processing, and inserts those new elements into the worklist. This process is repeated until the worklist becomes empty. In DMR, for example, the worklist maintains the set of all bad triangles in the mesh. A thread removes a triangle from the worklist and processes it by retriangulating the surrounding cavity, which may create additional bad triangles. The new bad triangles are added to the worklist. The refinement process halts when there are no more bad triangles to process.

A general outline of a data-driven implementation is presented in Figure 1. The comments indicate whether the computation is performed on the host, on the GPU, or involves a data transfer between the two devices. After reading the input graph on the host and transferring it to the GPU, the code initializes the global data structures such as the graph (*e.g.*, in SSSP, all nodes' distances are initialized to ∞ and the source node's distance is initialized to zero). The kernel `initialize_worklist` populates the input worklist. In SSSP, only the source node is included in the initial worklist. In MST, all graph nodes are in the initial worklist. Additionally, the output worklist is cleared. The while loop then repeatedly calls `process_kernel` until the input worklist is empty. The input and the output worklists are swapped (using pointers) and the accumulated work items in the output worklist become the input items for the next iteration. Finally, the computed result is transferred to the CPU for further processing.

Pseudo-code for `process_kernel` is presented in Figure 2. Each thread first calculates the range of worklist items it must process. This calculation, performed by `myworkitems`, can be based on the current size of the worklist and the number of threads with which the kernel is invoked. For each of the items retrieved from the worklist (outer `for` loop), the thread performs algorithm-specific processing (call to `process`) and generates a new set of items to be pushed onto the output worklist. These items are added to the end of the worklist (*e.g.*, using atomic instructions). Note that no global synchronization is necessary for extracting items from the input worklist because it is only read.

```
process_kernel(in, out, ...):
  myworkitems(&start, &end, thread_id)

  for index = start to end {
    item = in[index]
    newitems = process(item)
    for each newitem in newitems
      push newitem onto out
  }
```

Fig. 2: Pseudo code of the processing kernel

B. Hierarchical worklist

Although a two-worklist mechanism reduces synchronization, it still suffers from the cost of accessing slow global memory. Worklists need to be allocated in global memory because their sizes can easily surpass the amount of fast on-chip software-managed cache. It is, however, imperative to exploit the GPU memory hierarchy to improve worklist access time. This is achieved by implementing a hierarchical worklist in which threads use the limited shared memory whenever there is space and push elements onto the global worklist otherwise. Additional benefits can be reaped by storing a few graph elements in the registers assigned to each thread. Note that current high-end Fermis have more total register capacity than combined on-chip cache capacity.

Exploiting on-chip shared memory is an essential optimization for GPU implementations of most algorithms. In case of data-driven approaches, using a hierarchical worklist often results in significant performance improvement. By partitioning the shared memory across threads, both reads and writes to the per-thread worklists can be atomic free.

C. Work chunking

Accessing shared worklists requires atomic instructions to add and remove elements. Using two worklists can reduce but not eliminate the use of atomics. However, on a massively parallel architecture like a GPU, concurrent atomics are costly. An alternative is to batch element addition by work chunking.

In work chunking, threads delay writing to the worklist until they know how many elements they want to add. Based on this knowledge, the threads issue a single atomic instruction (`atomicAdd`) to increment the worklist size by that number of elements and then insert its elements into the obtained range. This can enhance implementation efficiency if concurrent updates to the worklist are a performance bottleneck. However, we did not obtain a significant performance gain from work chunking, as we discuss in Section VI.

D. Atomic-free worklist update

Further efficiency can be gained by using the prefix computation. Each thread records in an array at index i the number of elements it wants to add to the worklist, where i is the thread ID. A hierarchical scan operation is then performed, which computes, in $log\ n$ steps (where n is the number of threads), the sum of all array elements up to the ith element for all i and places the sums back into the array. After computing this prefix

```
donate_kernel():
  shared donationbox[...], totalwork = 0;

  // determine whether I should donate
  for all work_items assigned to me
    mywork += estimated work per item
  atomicAdd(totalwork, mywork)
  barrier();

  // donate
  averagework = totalwork / threads_per_block
  if mywork > averagework + threshold
    push excess work into donationbox
  barrier();

  // normal processing
  for all work items assigned to me
    process item

  // empty donation box
  while donationbox is not empty
    item = pop work from donationbox
    process item
```

Fig. 3: Pseudo code of work donation

sum, array entry $a[i]$ can be used as the index into the worklist where thread $(i+1)$ should start writing its elements. Thread 0 starts at index 0. A thread without any elements to add gets the same index as the previous thread. This way, all threads can push work onto the worklist in parallel without requiring atomic instructions. Note that (global) barriers are needed after the threads record their number of elements, after each step of the prefix sum, and before the first write to the worklist; in effect, fine-grain synchronization with atomics is replaced by coarse-grain synchronization with barriers. This approach has been employed in data-driven BFS by Merrill *et al.* [20] and we found it to be quite effective in our experiments (see Section VI).

E. Work donating

Load imbalance is a common problem in irregular algorithms. For instance, in DMR a cavity may consist of between 3 and 12 triangles depending upon the input mesh. Hence, the cavity sizes cannot be statically predicted, necessitating dynamic schemes for balancing the load. Even when threads can easily be assigned the same number of nodes, load imbalance may occur because different threads have to process a different numbers of edges (*e.g.*, in BFS and SSSP).

There are two popular mechanisms to handle load imbalance: work stealing [3] and work donating [13]. Work donating has a better memory footprint on GPUs in comparison to work stealing [26]. Hence, we focus on work donating.

In work donating, a thread that has more work than others donates some of its work items to other threads. There are numerous ways to implement work donation on GPUs. We implement it at the thread-block level, that is, threads donate work to other threads within the same block. Inter-block donation is not performed as it involves either expensive global barriers or global donation-box synchronization. Implementing it at the block level allows us to use fast shared memory as a donation box.

Figure 3 presents our work-donation algorithm executed by each thread in a thread block. It operates in three phases, separated by fast hardware-supported barriers (`syncthreads`). It defines a donation box in shared memory as well as a variable `totalwork` to track the total amount of work assigned to all threads in the block.

In the first phase, each thread performs a pass over its work items and estimates the amount of work to be done. For instance, in SSSP, a thread may compute the sum of the out-degrees of all nodes assigned to it. The block-level shared variable `totalwork` is updated atomically by each thread to compute the total work to be performed by the block.

In the second phase, a thread that is assigned an above-average amount of work donates work into the donation box. Using the total work indicated by `totalwork` and the size of the thread block, the average amount of work per thread is computed. If the average is lower than the amount of work to be performed by a thread beyond a threshold, the thread pushes excess work into the donation box using atomic instructions.

In the third phase, each thread performs its normal processing (*e.g.*, edge relaxation in SSSP) and then helps emptying the donation box if necessary.

Note that a barrier between the normal processing and the reading from the donation box is not only unnecessary but also hurts performance. Without the barrier, one warp may start emptying the donation box while another may still be performing the normal processing. Thus, all the threads that finish their processing early participate in taking up the excess load and reducing the overall computation time. Care must be taken not to push too much work into the donation box because adding work to and removing it from the donation box (which is essentially a worklist) involves expensive atomic operations whereas reading from the global input worklist as done in the normal processing is atomic free.

F. Variable kernel configuration

At any point during computation, the worklist size reflects the amount of work to be done. This size often changes considerably during the execution of an irregular program. Unlike in a topology-driven approach, where the worklist size remains constant, a data-driven approach may necessitate adapting the kernel configuration to the current amount of work, since it is natural to assign more threads when the worklist size is large and reduce the number of threads when the worklist shrinks. This adaptive behavior can be achieved by variable kernel configuration in which the number of threads and the number of blocks per grid are changed dynamically depending upon the worklist size. Varying the kernel configuration is possible whenever a GPU kernel is invoked repeatedly from the host. Variable kernel configuration not only improves the work efficiency of the GPU threads but often also improves performance. We demonstrate its effect in Section VI.

```
main():
  read input                          // CPU
  transfer input                      // CPU → GPU
  initialize_kernel()                 // GPU
  do {
      transfer false to changed       // CPU → GPU
      process_kernel()                // GPU
      transfer changed                // CPU ← GPU
  } while changed
  transfer results                    // CPU ← GPU
```

Fig. 4: Pseudo code of topology-driven implementation

IV. Topology-driven approach

In this section, we discuss the implementation and optimization of topology-driven algorithms on GPUs.

A. Base implementation

The base implementation of a topology-driven algorithm is outlined in Figure 4. The difference in comparison to a data-driven algorithm (*cf.* Figure 1) is in the processing of the do-while loop. In the topology-driven algorithm, a single flag `changed` is used to indicate whether another iteration of processing is necessary. The flag is set by a processing thread whenever it updates the global data. For instance, in SSSP, the flag is set by a thread when it updates a node's distance to a lower value, indicating that the shorter distance should be further propagated in the graph. This flag is reset at the start of each iteration and checked by the CPU after running `process_kernel`. If the flag is set, at least one thread modified the global data and another iteration is invoked. Instead, if the flag is still reset, no thread modified the global data, indicating that a fixed-point has been reached and no further iterations are needed. The cost of transferring this flag between the CPU and the GPU is typically negligible compared to the total processing cost.

An interesting aspect of topology-driven BFS and SSSP is that the computation can avoid atomic updates of the global distances. This is due to the monotonicity of the distance updates, *i.e.*, the distance of a node never increases. Without atomics, the computation may result in lost updates, that is, the smaller distance of a node may be overwritten by a larger distance from another thread due to a data race. However, the topology-driven approach ensures that all nodes will be active in the next iteration and the distance will eventually be set to the lower value again, guaranteeing forward progress towards the fixed point. Avoiding atomics by exploiting this algorithmic property for BFS and SSSP significantly improves their performance compared to the data-driven approach.

B. Kernel unrolling

In many graph applications, information flows from one node to another via edges. The speed of information flow determines how fast the fixed-point solution is obtained. Kernel unrolling is a way to quickly propagate computed information across the graph. It is implemented by processing a node and its neighborhood together in a single invocation. For instance, in case of kernel-unrolled SSSP, each thread computes the shortest distances of a node's neighbors and continues to update the shortest distances of the neighbors' neighbors. The order in which the neighboring nodes are processed matters and we found that processing a subgraph in a breadth-first manner helps in quick propagation of updated distances. This can be implemented using a local worklist per thread.

Due to the memory layout optimization discussed in Section IV-D, neighboring nodes are stored close to one another in memory. This improves spatial locality when processing a subgraph and improves performance (*cf.* Section VI-C).

C. Exploiting shared memory

Depending on the unroll factor and the number of threads per block, the local worklist needed for kernel unrolling (see previous subsection) can be stored in the fast, on-chip shared memory. The lower the number of threads, the larger the amount of available shared memory per thread. In our setup, with 512 threads per block, each local worklist can accommodate 24 integers, which is sufficient for an unroll factor of two to six, depending on the program and the input.

Interestingly, in a topology-driven implementation, overflow of the local worklist due to too high an unroll factor does not result in incorrect execution. This is because in the next iteration, all nodes, including the nodes that could not be added to the local worklist, are again active and will be processed.

D. Optimized memory layout

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 by performing a scan over the nodes and swapping 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.* [27].

V. Hybrid approach

Data-driven and topology-driven implementations can sometimes be combined. In this section, we discuss two high-level methods to create a hybrid implementation based on temporal and spatial hybridization.

A. Temporal hybrid

A temporal hybrid uses different implementations in different computation steps. For instance, in a data-driven implementation of SSSP, the worklist size is usually small during the initial and final computation steps but large in the middle. Therefore, a temporal hybrid would execute the middle iterations of SSSP in a topology-driven way. This is beneficial because in the middle iterations, the number of active nodes is large and the cost of worklist updates can be high.

Alternatively, in applications like MST, the amount of parallelism is high in the initial iterations and drops gradually as the

computation progresses. A temporal hybrid may execute MST in a topology-driven manner for the first few iterations and execute the remaining iterations using a data-driven approach when the graph becomes small.

Switching from a data-driven to a topology-driven approach is easy: the processing can simply discard the worklist and start processing all graph elements. However, the reverse, *i.e.*, switching from a topology-driven to a data-driven approach, involves extra processing: we need to create the input worklist. This is achieved by modifying the topology-driven computation to push modified elements onto a worklist instead of (or apart from) processing them. Alternatively, an implementation may employ a separate kernel to populate the worklist. Although worklist creation involves extra processing over a pure topology-driven implementation, this extra cost is incurred only once per switch and, in our experience, results in negligible overhead. Setting judicious thresholds on when to switch can offer great benefits but may require tuning.

B. Spatial hybrid

An alternative to temporal hybrids is to combine data-driven and topology-driven approaches in a spatial manner. In a spatial hybrid, multiple graph elements are grouped into partitions and each partition has a representative element. The data-driven part of the computation involves pushing representative elements onto the worklist and the topology-driven part of the computation involves processing all elements in the partition corresponding to the representatives popped from the worklist. Note that a non-representative graph element is never pushed onto the worklist and even if only a single graph element requires processing, all graph elements in its partition will be processed. The optimal partition size depends on the input and may have to be tuned. A partition size of unity yields a pure data-driven approach whereas a partition size of infinity (encompassing all graph elements) results in a pure topology-driven approach. We illustrate the effectiveness of hybrid approaches in Section VI-D.

VI. Experimental Evaluation

We first present a performance comparison of the data-driven versus topology-driven approaches and then evaluate the effect of various optimizations. Finally, we analyze the result of various hybrid implementations.

We perform our experiments on a 1.45 GHz NVIDIA Quadro 6000 GPU with 6 GB of main memory and 448 cores distributed over 14 SMs. This Fermi-based GPU has 64 kB of fast memory per SM that is split between the L1 data cache and the shared memory. All SMs share an L2 cache of 768 kB. We compiled the CUDA programs with *nvcc* v4.2 and the *-O3 -arch=sm_20* flags. Our benchmark programs and their inputs are listed in Table I.

A. Overall performance

Figures 5 and 6 show the relative performance of the optimized topology-driven and data-driven variants for our suite of irregular programs. Each program is run on multiple inputs whose characteristics are presented in Table I. The best result obtained using the optimizations discussed for each approach is reported in the figure. Note that both variants are faster than highly optimized multi-core versions of the same algorithm from the Galois benchmark suite [17] running on a high-end Xeon with 48 threads, except for MST whose performance is comparable to 16 threads.

From Figures 5 and 6, it is evident that the topology-driven approach turns out to be superior to its data-driven counterpart for BFS, BH, MST and SSSP. For BFS and SSSP, it is due to avoidance of atomic instructions in the topology-driven version. The data-driven versions have to use atomic instructions (`atomicMin`) for updating the node distances. Although the overhead is compensated, in part, by processing only the active nodes in the data-driven approach, topology-driven versions benefit from not having to use atomics.

To understand these results better, we plot the number of edges relaxed in SSSP as a percentage of the total number of edges when computing the shortest paths for the USA road network. The plot is shown in Figure 7 with the computation step (indicating time) on the x-axis and the percentage of edges relaxed on the y-axis. It is obvious that the optimized topology-driven version succeeds in processing several edges in the first few iterations. This is not always a sign of progress because the non-atomic updates may result in lost updates, but the performance numbers show that topology-driven SSSP reaches the fixed-point faster. In contrast, data-driven SSSP starts with a small worklist size that eventually grows and then gradually drops. The maximum number of edges relaxed by the data-driven version in any iteration is less than 10%.

The performance margin between the two variants is highest for MST (note the log scale in Figure 5). In fact, the results are for relatively small inputs because for larger inputs the data-driven version times out. The data-driven version is so slow because of duplicate entries in the worklist. MST works by merging nodes into components until a single component remains (when the graph is connected). As the algorithm progresses, the number of components decreases and the size of each remaining component increases. Once a node changes its component, it adds its new component to the worklist for processing. Thus, each component gets pushed onto the worklist by all the nodes it contains. This leads to costly atomic updates to the component's data by multiple threads. Searching the worklist first to avoid pushing duplicates onto it would likely also be slow.

Neither version of the BH code requires atomics because the bottom-up octree traversal is implemented in a pull-based fashion, *i.e.*, each node is updated by one thread using information from the node's children. Thus, there is a single writer per node, and a memory fence followed by the setting of a ready flag suffices to safely propagate information up the tree. The performance benefit of the topology-driven approach is due to the fact that it frequently manages to update children and their parents in the same iteration (if the parents happen to be processed later than the children). In contrast, the data-driven approach uses a worklist per tree level and requires as

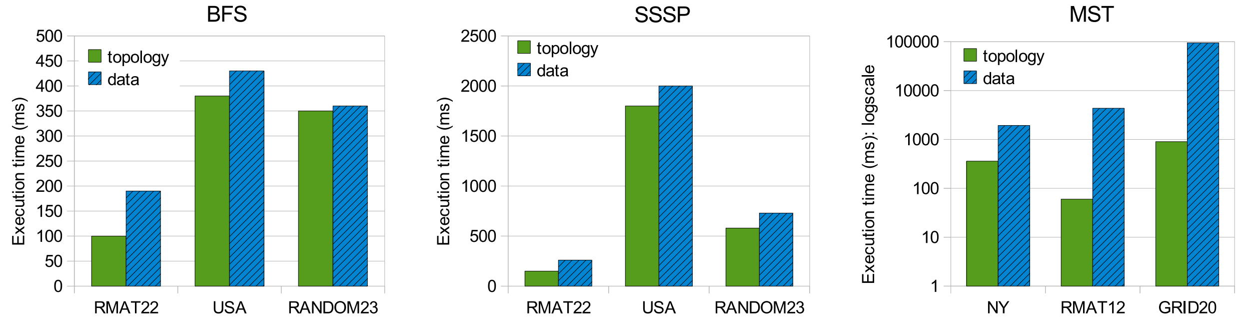

Fig. 5: Topology-driven versus data-driven BFS, SSSP and MST

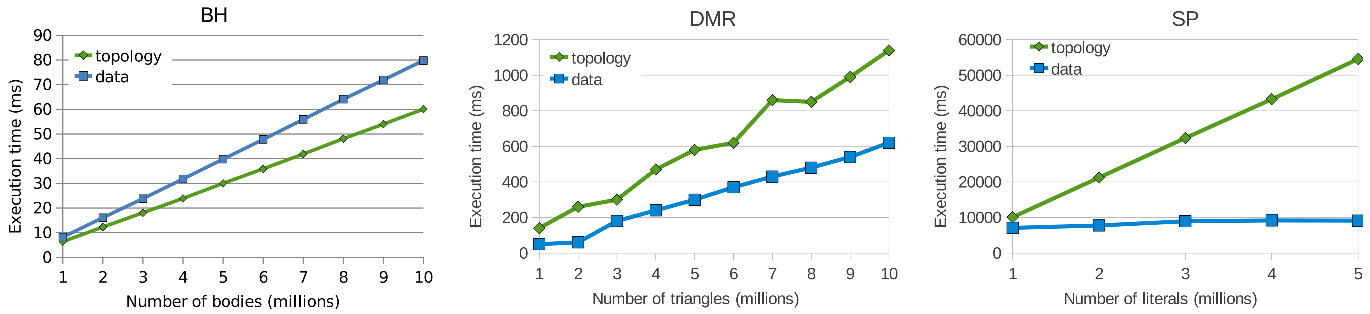

Fig. 6: Topology-driven versus data-driven BH, DMR and SP

many iterations as there are levels, which is roughly 15% more than the number of iterations of the topology-driven approach.

On DMR and SP, the data-driven approach consistently beats the topology-driven approach and scales better, as shown in Figure 6. Note that in DMR, even the topology-driven approach uses atomics for adding points and triangles to the mesh during cavity refinement. Moreover, the variable kernel configuration boosts the work-efficiency and the performance of the data-driven approach (*cf.* Section III-F).

The large performance difference for SP is not only due to processing of only active nodes but also due to faster convergence. In the data-driven approach, the surveys are propagated only from the active clauses, which helps in quick biasing of literals. The topology-driven approach takes advantage of the monotonic computation to avoid atomics when updating the surveys. However, in our experience, topology-driven SP results in the propagation of too many outdated surveys, leading to slow convergence.

B. Effect of optimizations on data-driven implementations

Figure 8 shows the effect of various optimizations on the data-driven implementation of SSSP (*cf.* Section III). The plot indicates execution times relative to the baseline implementation, which does not implement any of the optimizations discussed. The performance benefits differ considerably across inputs. For instance, implementing a hierarchical worklist improves SSSP's performance on the RMAT graph by only 2%, whereas for the road network the improvement is 18% (first bars). This arises from the smaller average node degree

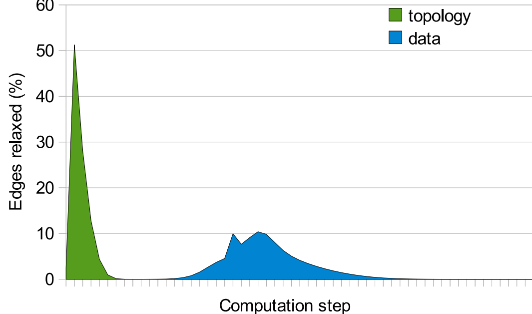

Fig. 7: Work done by data-driven and topology-driven SSSP

in the road network, which makes it possible to hold most of the worklist in shared memory. In contrast, for the RMAT and RANDOM graphs with large average degrees, the shared memory quickly fills up and new work items are spilled to global memory.

Work chunking does not improve performance (second bars) because the cost of batched atomics is still considerable. When atomics are avoided altogether (third bars), the performance improves by 15% on an average. The benefit of an atomic-free implementation is higher for higher-degree graphs.

Donating work improves performance by 10% on an average (fourth bars). As expected, the benefit is higher for more skewed distributions such as in the road network. This indicates that load balancing is important for irregular workloads. Figure 10 shows the amount of work donated by the

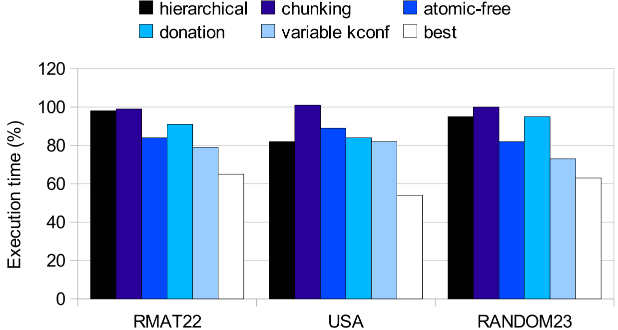

Fig. 8: Effect of data-driven optimizations on SSSP

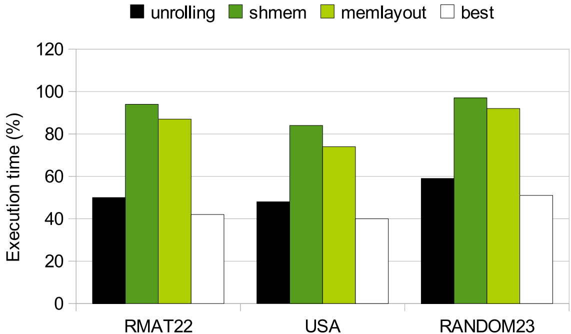

Fig. 9: Effect of topology-driven optimizations on SSSP

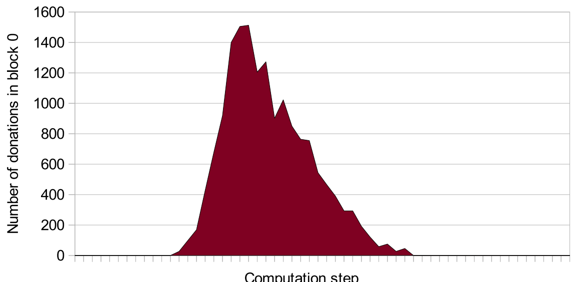

Fig. 10: Amount of work donated in block 0 for SSSP

threads in block 0 in each iteration of SSSP on the USA road network. The plot illustrates that work donation occurs in the middle of the computation where the load imbalance is the highest. The amount of donation quickly rises to 1500 donations and gradually drops to zero. We expect better load-balancing techniques to further improve performance.

Finally, variable kernel configuration offers the largest benefit (fifth bars). This is because using only the required number of threads drastically reduces contention over shared data. It also has the advantage of improving the work efficiency. The plot also shows that the best performance is obtained by using a combination of these optimizations (sixth bars).

C. *Effect of optimizations on topology-driven implementations*

Figure 9 presents the effect of various optimizations on topology-driven implementations of SSSP (*cf.* Section IV). The plot shows execution times of several optimized versions relative to the baseline implementation, which does not implement any of the optimizations discussed. Recall that the baseline version performs atomic-free distance updates.

Kernel unrolling is very effective for the topology-driven approach (first bars). This is the case because unrolling mitigates the effect of lost updates. Even with atomics, kernel unrolling helps propagate information faster in the graph by avoiding the wait time for the next iteration.

Using on-chip shared memory to store the local worklists avoids expensive global memory accesses (second bars). On average, this improves performance by 10%. Similar to the benefit of hierarchical worklists (*cf.* Section VI-B), shared memory offers higher benefits when the local worklist sizes are small and do not often spill over to global memory (as for the road network that has a lower out-degree).

Employing a better memory layout to store nearby nodes nearby in memory helps improve performance by almost 15% (third bars). The performance improvement is mainly due to improved spatial locality and is higher for the relatively sparse road network. Note that improving the layout for some nodes may adversely affect locality for other nodes, based on connectivity. This is relatively more common for dense graphs leading to reduced performance benefits, although a locality-friendly layout generally improves performance.

Finally, irregularity in the data access pattern prohibits effective coalescing. However, when feasible, coalescing helps more with the relatively uniform topology-driven approach than with the data-driven approach where the nodes assigned to a thread likely differ in each iteration. The plot also shows the best performance obtained using a combination of these optimizations (fourth bars).

D. *Effect of hybrid implementations*

Figures 11 and 12 show the effect of temporal and spatial hybrid implementations, respectively, of BFS, SSSP and DMR. For reference, the execution times of data-driven and topology-driven SSSP are shown as horizontal lines. The presented runtimes are relative to the time taken by their data-driven variant. The results are for the USA road-network input in case of BFS and SSSP, and for an input mesh with 10 million triangles in case of DMR.

The x-axis of Figure 11 indicates the threshold worklist size beyond which the execution switches to the topology-driven approach (*cf.* Section V-A). The threshold is the fraction of the total number of nodes in the input graph. For instance, for the USA road network with 23 million nodes, we vary the worklist size from 10% (2.3 million elements) to 120% (27.6 million elements). Note that the worklist size can exceed 100% because the same node may be pushed onto the worklist multiple times. For this reason, a 100% size does not imply that every node is on the worklist. From the plot, we observe that the temporal hybrid attains an execution time that is better than either the data-driven or the topology-driven approach for SSSP and DMR. When the worklist percentage is between 40

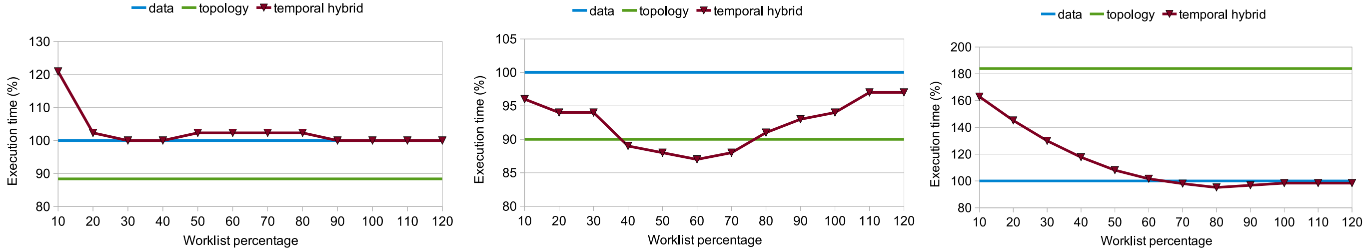

Fig. 11: Effect of temporal hybrid on BFS, SSSP and DMR

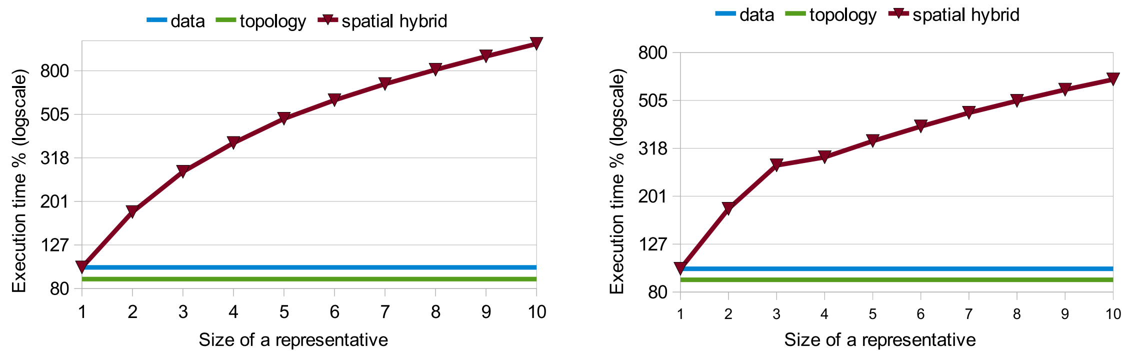

Fig. 12: Effect of spatial hybrid on BFS and SSSP

and 70, hybrid SSSP performs better than the solo approaches, peaking at around 60%. Hybrid DMR performs better than the solo approaches when the worklist percentage is between 70 and 90. As the worklist percentage increases, the execution time approaches that of the data-driven approach. This occurs because the number of edges relaxed grows in the initial iterations, stays high for some time and then drops. Thus, using a data-driven approach in the beginning and end and a topology-driven approach in the middle helps these algorithms.

In contrast, hybrid BFS performs comparable to the data-driven version. This is mainly due to the relatively low execution time of BFS and, in turn, a large fraction of time being spent in creating the worklist when switching from a topology-driven version to a data-driven version. In fact, for a worklist percentage of 10, the cost is higher than either of the solo approaches. However, above 20% its performance is similar to that of the data-driven version.

For the spatial hybrid (*cf.* Section V-B), we increased the representative size from 1 to 10. From the performance plot in Figure 12, we can see that choosing a representative size greater than 1 is bad for both BFS and SSSP. In fact, the performance becomes worse with increasing representative size (note the log scale of the y-axis). Due to irregularity in the data accesses, static partitioning based on node ID in the spatial hybrid approach groups unrelated nodes. All such nodes are processed even if only one is active. This incurs a large penalty for a relatively sparse computation. We also experimented with hashing to insert unique elements into the worklist. However, the cost of accessing the global hash-table was large enough to nullify the benefit of avoiding duplicates. We conclude that, without further optimizations, such a spatial hybrid approach may not be well suited for data-driven irregular algorithms.

E. Data-structure specific hybrid

Since every node is updated exactly once in the BH summarization, this kernel can be implemented using a special kind of hybrid between a data-driven and a topology-driven approach. In particular, all nodes are active like in a topology-driven approach but the threads wait until all children of the current node are ready akin to a data-driven approach. This way, only one kernel call is needed to complete the entire tree traversal. However, potential deadlock must be avoided [6].

As the first data point (with zero pre-passes) in Figure 13 reveals, this hybrid performs substantially worse than the pure topology-driven approach because threads often wait rather than processing other, ready nodes. This shortcoming can be remedied by further (temporal) hybridization, in particular by first running a few iterations of the topology-driven kernel, which we refer to as 'pre-passes' over the data structure.

Figure 13 shows the benefit of this approach, which now outperforms the non-hybrid approaches. One pre-pass works best, probably because most of the work in this traversal of a high-fan-out octree is completed in the first iteration.

Since the final iteration waits for any unready children, there is no need for barriers, either explicit or implied by kernel calls, to separate the pre-passes from each other or from the final iteration. Hence, we can include the pre-pass code in the kernel that performs the final iteration and allow the warps to run in an unsynchronized way. This approach is called 'combined pre-passes' in Figure 13 and results in the

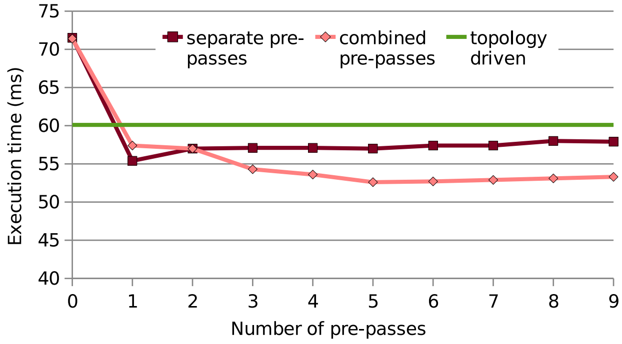

Fig. 13: Effect of pre-pass on BH with 10 million bodies

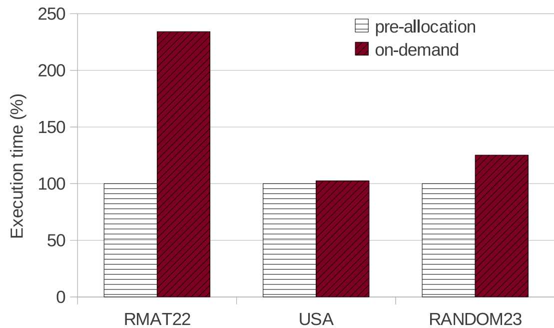

Fig. 14: Effect on SSSP of on-demand worklist re-allocation

fastest implementation of the BH summarization operation. With five combined pre-passes, it performs 14% better than the topology-driven approach.

F. Cost of growing the worklist on demand

So far, we have assumed that the worklist is large enough to accommodate all work items pushed onto it in an iteration. This is not a serious limitation because Fermi-based GPUs are single-user devices. However, looking to the future, it might be useful not to pre-allocate the worklist but to grow it gradually. We implement this approach by allocating a reasonable amount of memory for the initial worklist and then doubling the space whenever it fills up. Since the kernel is invoked by the host iteratively, we can use CUDA host functions (`cudaMalloc` and `cudaMemcpy`) for the on-demand worklist re-allocation. Each such re-allocation requires the copying of the old work-list to the new worklist.

Worklist overflow is detected by a thread when it tries to push a new work item beyond the current output worklist capacity. On overflow, the thread stops processing, sets the overflow flag and terminates. Any other thread noticing the overflow also terminates. Note that a thread that does not encounter an overflow continues processing. Thus, the kernel performs as much computation as possible despite the over-flow. When all threads have finished (either due to overflow or due to reaching the end of normal processing), the host finds the overflow flag set, re-allocates the output worklist as well as the input worklist, does not swap the input and output worklists, and restarts the latest iteration with the overflow flag reset. This time, threads again perform normal processing and push data items to the output worklist.

The main cost of on-demand re-allocation is copying work items from the old worklists to the new worklists. Figure 14 shows the relative performance of SSSP with pre-allocated memory and on-demand memory re-allocation. For on-demand memory re-allocation, the initial worklist size is heuristically set to one fifth of the number of graph nodes. We observe that the cost of copying increases with graph density.

VII. Related Work

To the best of our knowledge, this is the first work that performs a comprehensive comparison of data-driven versus topology-driven implementations of irregular GPU programs.

Most of the prior work on irregular algorithms on GPUs uses a topology-driven approach. Examples include graph algorithms [11], [12], points-to analysis [19], *n*-body simu-lation [6] and morph algorithms [22]. Very few GPU im-plementations are based on data-driven approaches. Tzeng *et al.* [26] propose several strategies for managing irregular tasks. They use queuing techniques for adding work to the worklist, advocate warp-centric kernel implementations and uberkernels to avoid kernel-switching overhead, as well as work-stealing and donating. Based on their finding that work-donating performs better than work-stealing, we evaluate only work donation in our experiments.

The concept of persistent threads [10] relies on a data-driven implementation. The idea is to invoke kernels with only as many blocks as can remain persistent in the SMs and continue operating on a worklist in a do-while loop. This avoids the need of returning to the host and the cost of repeated kernel in-vocations. However, as the authors point out, persistent threads are beneficial in only certain situations. First, the workload should be small with few memory accesses, and second, the worklist size should have a constrained growth pattern. None of our irregular graph-based applications satisfy these criteria: each application involves several memory accesses, workloads are large and the worklist size grows rapidly and substantially. Therefore, persistent threads are not typically used for irregular benchmarks. We use it only in the BH kernel, which is a special case because it operates on a tree and the nodes assigned to each thread are partially ordered by tree level.

Merrill *et al.* [20] implemented a data-driven version of BFS. They employ prefix sums to populate the worklist in an atomic-free manner. We use and evaluate their technique as one of our data-driven optimizations.

Bruant [5] simulated cellular automata on GPUs in a data-driven manner. He found that removing duplicates from a large worklist is very slow if the `thrust` library is used, and that it performs better when it is done on the CPU despite the cost of copying data back and forth between the CPU and GPU.

Previous research on GPU implementations of irregular algorithms generally focuses on optimizing a specific feature of irregular programs or on tuning a specific application

for GPU execution. The first category includes work on removing dynamic irregularities from applications [27] and on optimizing CPU-GPU transfers for dynamically managed data [16]. G-Streamline is a software-based runtime approach to eliminate control-flow and memory-access irregularities from GPU programs [27]. DyManD is an automatic runtime system for managing recursive data structures (like trees) on GPUs [16]. The second category includes GPU-specific optimized implementations of breadth-first search [15], [20], single-source shortest paths [18], points-to analysis [19] and n-body simulation [6].

VIII. Conclusions

We perform qualitative and quantitative comparisons of data-driven and topology-driven implementations of six irregular algorithms on GPUs. We found that, in general, a data-driven implementation achieves better performance because it is more work-efficient, but in cases where an algorithmic property can be exploited for optimizing a kernel, a topology-driven implementation may outperform its data-driven counterpart. For both implementations, we propose several optimizations that yield significant performance improvements, including atomic-free worklist updates, variable kernel configuration, kernel unrolling and intra-block work-donation. We also combined the two implementation strategies to devise two hybrid schemes. We found that a spatial hybrid degrades performance whereas a temporal hybrid can perform better than the individual approaches. Automating the choice of the best implementation strategy for a given application seems difficult, and we leave it for future work.

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, Intel, and Qualcomm Corporations.

References

[1] A. Aho, R. Sethi, , and J. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.

[2] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4), December 1986.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

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

[5] Bruant, Hugues. Parallel simulation of cellular automata, CMU 15-418 (Spring 2012) Final Project Report. http://www.andrew.cmu.edu/user/hbruant/15418/finalreport.html.

[6] 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.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributing Computing*, 68:1370–1380, October 2008.

[8] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Symposium on Computational Geometry (SCG)*, 1993.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms, McGraw Hill, 2001.

[10] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Innovative Parallel Computing*, page 14, may 2012.

[11] 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.

[12] Pawan Harish, Vibhav Vineet, and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report Technical Report Number IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, 2009.

[13] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 2008. ACM.

[14] Kirsten Hildrum and Philip S. Yu. Focused Community Discovery. In *International Conference on Data Mining*, 2005.

[15] 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*, pages 267–276, New York, NY, USA, 2011. ACM.

[16] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 165–174, New York, NY, USA, 2012. ACM.

[17] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *SIGPLAN Notices (Proceedings of PLDI)*, 42(6):211–222, 2007.

[18] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes. CUDA Solutions for the SSSP Problem. In *Proceedings of the 9th International Conference on Computational Science: Part I*, pages 904–913, Berlin, Heidelberg, 2009. Springer-Verlag.

[19] 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*, pages 107–116, New York, NY, USA, 2012. ACM.

[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*, 2012.

[21] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[22] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[23] Donald Nguyen and Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS '11: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[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*, pages 12–25, New York, NY, USA, 2011. ACM.

[25] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar, editors. *Introduction to Data Mining*. Pearson Addison Wesley, 2005.

[26] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 29–37, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[27] 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*, pages 369–380, New York, NY, USA, 2011. ACM.