Deploying Graph Algorithms on GPUs: an Adaptive Solution

Da Li
Dept. of Electrical and Computer Engineering
University of Missouri - Columbia
dlx7f@mail.missouri.edu

Michela Becchi
Dept. of Electrical and Computer Engineering
University of Missouri - Columbia
becchim@missouri.edu

***Abstract*—Thanks to their massive computational power and their SIMT computational model, Graphics Processing Units (GPUs) have been successfully used to accelerate a wide variety of regular applications (linear algebra, stencil computations, image processing and bioinformatics algorithms, among others). However, many established and emerging problems are based on irregular data structures, such as graphs. Examples can be drawn from different application domains: networking, social networking, machine learning, electrical circuit modeling, discrete event simulation, compilers, and computational sciences. It has been shown that irregular applications based on large graphs do exhibit runtime parallelism; moreover, the amount of available parallelism tends to increase with the size of the datasets.**

In this work, we explore an implementation space for deploying a variety of graph algorithms on GPUs. We show that the dynamic nature of the parallelism that can be extracted from graph algorithms makes it impossible to find an optimal solution. We propose a runtime system able to dynamically transition between different implementations with minimal overhead, and investigate heuristic decisions applicable across algorithms and datasets. Our evaluation is performed on two graph algorithms: breadth-first search and single-source shortest paths. We believe that our proposed mechanisms can be extended and applied to other graph algorithms that exhibit similar computational patterns.

I. INTRODUCTION

Graphs are a powerful representation used in many practical applications (e.g., networking, social networking, online marketing, webpage search, citation networks, among others). Recent work [1-3] has shown that, even if graph algorithms are intrinsically irregular, they exhibit a high amount of runtime parallelism, which is data dependent. In the past decades, the size of real world datasets has rapidly increased, thus exposing higher amount of parallelism.

Because of the practical relevance of these data structures, several efforts have proposed parallel graph libraries (e.g. Parallel BGL [4], ParGraph [5] and STAPL [6]) as well as programming models and runtime systems (e.g. Galois system [1-3]) to enable the effective deployment of graph algorithms on multi-core CPUs. More recently, there has been a rich body of work targeting the deployment of graph algorithms on GPUs [7-13], traditionally used to accelerate regular applications [14, 15]. Most of these proposals aim to provide highly optimized implementations of specific graph algorithms.

It has been shown that graphs used in real-world applications [16-18] exhibit significant topological differences. The topology of the graphs dictates the amount of parallelism that can be extracted at runtime, thus affecting the performance of specific GPU implementations. This heterogeneity makes it difficult to design a GPU implementation of a graph algorithm that is optimal on a large variety of datasets. In this work, we argue for an adaptive solution that takes into account the topological characteristics of the dataset to dynamically select the most suitable alternative among a set of available GPU implementations.

Our contributions can be summarized as follows.

- We explore an implementation space for graph algorithms on GPU. The considered space is characterized by the following dimensions: (i) kind of algorithm (i.e., ordered vs. unordered), (ii) mapping granularity (i.e., thread-mapping vs. block-mapping), and (iii) working set implementation (i.e., bitmask vs. work queue). Our analysis shows that there is no optimal solution across graph problems and datasets.
- We propose an adaptive runtime that dynamically selects the most suitable implementation among the ones resulting from the aforementioned exploration space. We design data structures that lead to minimal overhead when switching between implementations at runtime. We devise heuristics that guide the decisions of our adaptive runtime.
- We evaluate the described static and dynamic solutions on a variety of real world datasets, and show that our dynamic solution outperforms the best static one (up to a factor of 2X) on most datasets, and is more robust to the irregularities typical of real world graphs.

Our evaluation is limited to two graph problems: breadth-first search (BFS) and single-source shortest paths (SSSP). However, we believe that our analysis can be extended to many other graph algorithms, which can be expressed as a sequence of iterative steps, each step processing a (ordered or unordered) set of elements (i.e., nodes or edges).

II. RELATED WORK

Recent work [4-6] has proposed parallel graph libraries for multi-core processors and distributed systems. Parallel BGL [4] and ParGraph [5] are extensible parallel libraries. Parallel BGL provides a generic interface to graph structures, a set of core algorithm patterns (breadth-first, depth-first and uniform-cost search), and a set of *concrete* graph algorithms (shortest paths, minimum spanning tree, and connected components algorithms, among others). STAPL [6] is an extension to the ANSI C++ Standard Template Library that provides a collection of thread-safe distributed data structures, including graphs. Differently from Parallel BGL,

which is static and makes parallelization decisions at compile time, STAPL has a runtime system that dynamically selects among different algorithm implementations of the same library routine. Our proposal targets GPUs. Similarly to Parallel BGL, we provide to the user a graph API including some algorithm patterns that can be reused in the context of more complex applications. However, like STAPL, our library is coupled with a runtime system that can dynamically adapt to the characteristics of the data and to the underlying hardware.

The Galois system [1-3] offers a programming model and a runtime system to dynamically extract parallelism from irregular applications, and is based on the following consideration: the parallelism present in irregular algorithms is mostly data dependent, and cannot be discovered by static analysis at compile time; however, such parallelism can be exploited at runtime. Specifically, the authors use optimistic (or speculative) parallelization to dynamically expose parallelism. The Galois system consists of two main components: a programming model and a runtime system. The former includes an ordered and an unordered set iterator, that allow constructing *do-across* and *do-all* loops, and can be used by the programmer to represent irregular algorithms in a sequential fashion. Using this programming model, graph algorithms are represented iteratively: in each iteration, a simple or complex operator is applied to the neighborhood of an active element (e.g., a node or an edge). The runtime system schedules iterations to different execution threads concurrently, enables their speculative execution, and recovers from potentially unsafe accesses to shared memory due to speculation. In this work, we view graph algorithms such as BFS and SSSP iteratively similarly to what done in the Galois's programming model. However, we do not consider speculative execution.

There has been a rich body of work on the design of parallel algorithms to solve various graph problems (e.g., breadth-first search [19-21], shortest paths [22, 23], minimum spanning trees [24-26], connected components [27-30]). In this work, we consider a serial implementation of BFS and SSSP, and use the parallelism inherent in the working set to efficiently deploy these algorithms on GPUs.

A few proposals [7-13] have accelerated graph algorithms on GPU. Harish and Narayanan [7] were the first to perform this operation; their proposed implementations, however, are pretty basic and ineffective on sparse graphs used in practice. Better results have been reported through subsequent efforts, which focused on specific algorithms (breadth-first search [8, 9, 13], inclusion-based points-to analysis [10], strongly connected components [11]). The optimizations introduced by these proposals are somehow orthogonal to this work, and can be integrated with it. Because of their higher generality, the proposals closest to ours are those by Hong et al [12, 13]. [12] proposes a virtual warp-centric programming model to allow datasets with different characteristics to more efficiently use the GPU hardware. This idea can be integrated with our work. [13] considers an adaptive solution that alternates CPU and GPU execution. We, on the other hand, focus on the automatic selection of different GPU solutions and on the conditions that make this beneficial.

III. MOTIVATION

In this section, we present some motivating facts that show the potential of GPUs as accelerators of graph algorithms and give an intuition of why a dynamic solution can be preferable to a static one. First, we characterize some graph datasets used in real-world applications. Second, we introduce the main architectural features of modern GPUs, and discuss their suitability to the deployment of graph algorithms.

A. Characterization of Graph Datasets

Graphs are a powerful representation used in many practical applications, where the relationships among the nodes in some network are relevant. Some examples drawn from different application domains are: the road network, the web link network and the social network. The road network is typically extracted from GPS maps and used to calculate the optimal route (or shortest path) between two endpoints. The web link network contains links between web pages, and its connectivity is typically used by search algorithms to rank the results of queries. The social network contains relationships between individuals, and is used to compute a variety of connectivity properties (in applications like Facebook, for instance, such relationships are used to suggest new friends).

In this paper, we use graphs from the 9th and the 10th DIMACS implementation challenges [16, 17] and from the Stanford Large Data Collection [18]. In particular, we consider datasets used in different application domains: the Colorado road network [16], a paper co-citation network (from the CiteSeer library) [17], a p2p networking network [18], the Amazon co-purchase network [18], the Google webpage link network [18] and a SNS network (from Live-Journal) [18]. All but the road network and the paper co-citation network are directed graphs. Table 1 shows a characterization of these datasets, in terms of total number of nodes, total number of edges and node degree (that is, number of edges per node). We observe the following facts.

- The graph size varies considerably across the datasets: from the small *p2p* network (with about 36.6 K nodes and 183.8 K edges) to the large *SNS* network (with about 4.3 M nodes and 34.5 M edges).
- The average node outdegree also varies considerably: from 2.4 in the *CO-road* network, to 73.9 in the *CiteSeer* network. Four networks (*CiteSeer*, *p2p*, *Google*

Table 1: Dataset characterization.

Network	# Nodes	# Edges	Node Outdegree		
			min	max	avg
CO-road	435,666	~1 M	1	8	2.4
CiteSeer	434,102	~16 M	1	1,188	73.9
p2p	36,692	~0.18 M	0	1,383	10.0
Amazon	396,803	~1.7M	0	10	8.4
Google	739,454	~2.5 M	0	456	6.9
SNS	4,308,452	~34.5 M	0	20,293	16.0

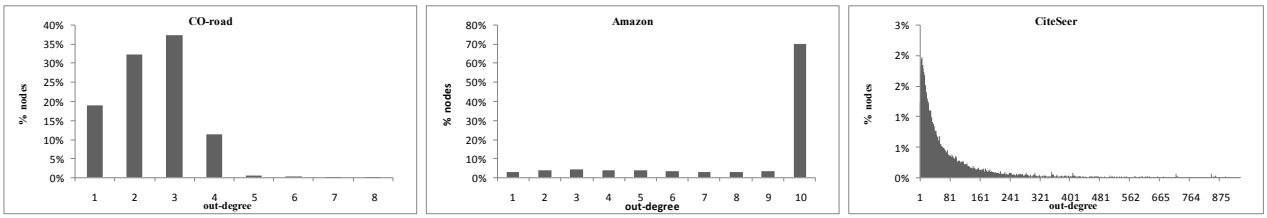

Figure 1: Outdegree distributions of *CO-road*, *Amazon* and *CiteSeer* networks.

and *SNS*) exhibit a considerable outdegree variance, leading to large outdegree values. The other networks (*CO-road* and *Amazon*) have a more regular structure.

Figure 1 shows the outdegree distribution of the *CO-road*, the *Amazon* and the *CiteSeer* network. As can be seen, these networks exhibit different characteristics. The *CO-road* graph is pretty sparse: most of its nodes have an outdegree from 1 to 4, and the maximum outdegree is 8. This is because most towns are usually directly connected to a handful of other towns, whereas few bigger cities serving as transportation hubs have as many as 7-8 intercity roads. The *Amazon* network is very regular: 70% of the nodes have 10 outgoing edges, and the remaining nodes have an outdegree uniformly distributed between 1 and 9. The *CiteSeer* network is far less regular: about 90% of the nodes have less than 200 outgoing edges. On the other hand, the outdegree range is very wide for the remaining nodes (up to 1,188). The outdegree distribution of the *p2p*, the *Google* and the *SNS* networks is similar to that of the *CiteSeer* graph.

This fact has a practical significance. Most graph algorithms proceed iteratively. In each iteration, they visit the local neighborhood of a working set consisting of nodes or edges, remove elements from the set and add new elements to it. Intuitively, large outdegree lead to large working sets, and thus to potentially high amounts of parallelism. However, unbalanced outdegree distributions can cause work imbalances during graph traversals. An adaptive solution may therefore better support a wide variety of graphs, including those with irregular topologies.

B. Imbalanced Work of Graph Algorithms

In this work, we focus on breadth-first search (BFS) and single-source shortest path (SSSP), two fundamental graph problems. BFS computes the depth of each node *n*, that is, the minimum number of nodes visited when moving from a given source node to *n*. SSSP computes the minimum cost paths from a given source node to any other node in the graph. These problems are solved through an iterative graph traversal. Initially, the working set consists of the source node. In each traversal step the local neighborhood of the working set is processed. The traversal terminates when the working set becomes empty. During execution, two kinds of work imbalance can take place.

- *Inter-iteration work imbalance* - The size of the working set typically changes from iteration to iteration. For example, Figure 2 shows how the size of the working set varies during the execution of SSSP on three datasets (*CO-road*, *Amazon* and *SNS*). As can be seen, the work is generally limited at initial stages, when the traversal is restricted to the neighborhood of the source node. When enough nodes have been processed, the working set starts growing and keeps growing until a large fraction of the nodes have been visited. At that point, the working set starts shrinking. The working set size and the convergence speed depend on the specific algorithm and on the characteristics of the dataset. For instance, on the datasets of Figure 2, BFS has working set sizes from 2 to 20 times smaller than those reported by SSSP.
- *Intra-iteration work imbalance* - Different nodes can have different outdegrees. As a consequence, each node in the working set can be potentially associated with a different amount of work. This fact affects the performance of the GPU design. For example, if a node-to-thread mapping is adopted on a graph with an irregular topology, thread divergence may arise during execution, and the performance will be limited by the node with the largest outdegree.

C. Architectural pros & cons of GPUs

GPUs are known for the massive hardware parallelism that they offer. NVIDIA GPUs consist of several SIMT processors, called Streaming Multiprocessors (SMs), each containing a set of in-order CUDA cores. In the Fermi architecture, each SM comprises either 32 or 48 cores (depending on the compute capability of the device). The CUDA programming model [31] facilitates writing parallel algorithms for GPUs. In CUDA, the computation is organized in a hierarchical fashion: threads are grouped into

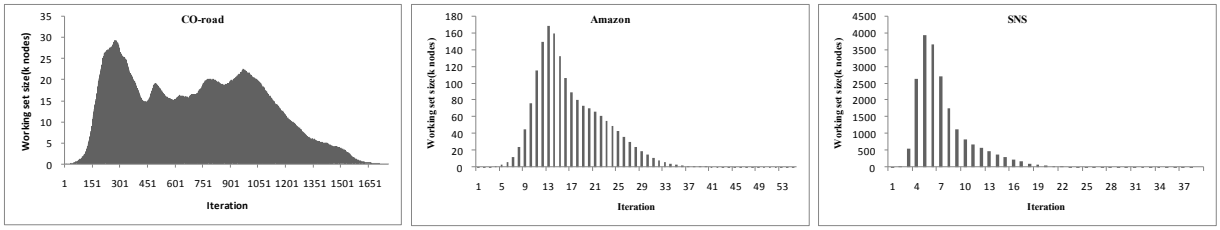

Figure 2: Unordered SSSP – size of the working set during the execution (*CO-road*, *Amazon* and *SNS* networks).

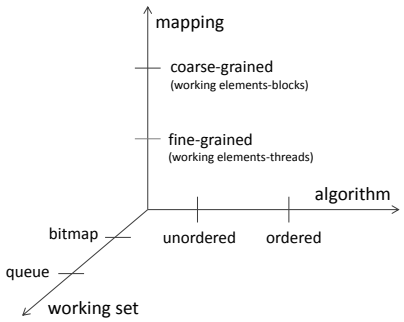

Figure 3: Exploration space.

thread-blocks; at runtime, each thread is mapped onto a core and each thread-block is mapped onto a SM. In CUDA 4, as many as 64K*64K*64K blocks with at most 1,024 threads each are allowed. This parallelism can clearly be advantageous for graph applications that operate on large datasets consisting of millions of nodes and edges.

Another characteristic of the GPU architecture is its memory hierarchy. GPUs are equipped with a relatively large off-chip, high-latency, read-write global memory; a smaller low-latency, read-only constant memory (which is off-chip but cached); and a limited on-chip, low-latency, read-write shared memory. The global memory can be accessed via 32-, 64- or 128-byte transactions and has a high access bandwidth (up to 144 GB/sec). Multiple memory accesses to contiguous memory locations are automatically coalesced into a single memory transaction, thus saving memory bandwidth. The graph algorithms in consideration are not computation intensive, but – especially when running on large datasets - can be memory bound. In fact, when processing hundreds of nodes in parallel, it is necessary to access their neighbors in an efficient way. The GPU high memory bandwidth can be beneficial for these memory intensive applications.

Two GPU architectural features are particularly problematic when deploying graph algorithms on GPUs. First, SMs are SIMT-processor. During execution, threads are grouped into 32-element SIMT units, called warps. In every clock cycle, threads belonging to the same warp must execute the same instruction. Branches are allowed through the use of hardware masking. In the presence of branch divergence within a warp, both paths of the control flow operation are in principle executed by all CUDA cores. Therefore, the presence of branch divergence within a warp leads to core underutilization. Unfortunately, the irregular nature of graph algorithms leads to relatively frequent branch operations. Second, to fully utilize its high memory bandwidth, the GPU requires regular memory access patterns. In fact, contiguous memory accesses can be coalesced into few memory transactions when accessing global memory, and allow avoiding bank conflicts when accessing shared memory. However, the memory access patterns within graph algorithms are often irregular and hard to predict. Even if this effect can be limited by representing graphs with ad-hoc data structures (e.g. adjacency matrices in compressed sparse row form), unpredictable and irregular memory accesses cannot be fully avoided.

```
1.  Graph g;
2.  WorkingSet ws;
3.  g.root.level = 0;
4.  ws.add(g.root);
5.  foreach(Node n:ws){
6.     int level = n.level + 1;
7.     for (Node m: g.neighbors(n)){
8'.       if (m.level == INF){     //ORDERED VERSION
8".       if (m.level > level){    //UNORDERED VERSION
9.           m.level = level;
10.          ws.add(m);
11. }  }  }
```

Figure 4: Ordered and unordered BFS algorithms. In the ordered version the working set *ws* is ordered by *level*. Instructions 8' and 8" belong to the ordered and unordered version, respectively.

IV. Exploration Space

In this section, we present and discuss a possible exploration space for implementing graph algorithms on GPU. Our study focuses on the BFS and SSSP problems. However, we believe that our analysis can be extended to other amorphous graph algorithms with similar computational patterns. We consider a 3-dimensional exploration space (Figure 3), which is built according to the following questions.

- Is the working set used by the algorithm ordered or unordered?
- What is the granularity of the mapping of the work to the GPU hardware? Two obvious alternatives consist of mapping each element of the working set to a thread (fine-grained mapping) or to a thread-block (coarse-grained mapping).
- How is the working set implemented? We will consider a bitmap-based and a queue-based implementation.

A. *Ordered vs. unordered algorithms*

In this paper, we consider the distinction between unordered and ordered graph algorithms introduced by Hassaan et al [3]. The basic idea is the following. Most graph algorithms operate iteratively over a working set consisting of nodes or edges; in unordered graph algorithms, the elements can be extracted from the working set and processed in any order; conversely, in ordered algorithms, an ordering relation over the working set imposes a constraint on the processing sequence of the elements in it.

Figure 4 shows the pseudo-code of an ordered and an unordered BFS algorithm, which compute the level (or depth) of the nodes in a graph. The two algorithms differ in the nature of the working set (ordered vs. unordered, respectively) and in instruction 8. The ordered version processes each node exactly once, and adds it to the working set the first time it is visited (that is, when its level is undefined). The unordered algorithm may add the same node to the working set multiple times, as long as its level decreases when the node is visited. The ordered version clearly terminates when all nodes have been processed. Since the node level is a monotonically decreasing function, the unordered version is also guaranteed to terminate.

Figure 5 shows the pseudo-code of an ordered and an unordered SSSP algorithm (Dijkstra and Bellman-Ford [32], respectively). In the ordered algorithm, the working set is ordered by distance, and the distance of each node is updated only once. In the unordered version, such attribute may be updated multiple times (as long as its value decreases). The ordered algorithm terminates when all node distances have been set. Since the distance is a monotonically decreasing function, the unordered algorithm is also guaranteed to terminate. Note that, in the ordered version, the same node can appear multiple times in the working set with different weight values. However, the ordered nature of the working set ensures that the node distance is updated only once with the minimum weight value.

In general, ordered algorithms are more work efficient than their unordered counterparts (in that they process each element a minimum number of times), but take more iterations to converge. However, unordered algorithms may exhibit higher degrees of parallelism. In fact, unordered algorithms can process all the nodes in the working set at the same time, whereas ordered algorithms can process in parallel only elements that are equivalent in terms of the underlying order relation (in other words, at every iteration, ordered algorithms effectively process only a subset of the working set). Intuitively, ordered algorithms are better candidates for serial implementation, whereas unordered ones can be more suitable for parallel implementation.

```
1. Graph g;            -- ORDERED VERSION
2. WorkingSet ows;
3. for (Node m: g.neighbors(g.root)){
4.   ows.add(<m, g.edgeWeight(g.root,m)>);
5. }
6. foreach(Pair <n, d>: ows)
7.   if (n.distance == INF)){
8.     n.distance = d;
9.     for (Node m: g.neighbors(n)){
10.      if (m.distance == INF){
11.        ows.add(<m, d + g.edgeWeight(n,m)>);
12.} } } }
```

```
1. Graph g;            -- UNORDERED VERSION
2. WorkingSet ws;
3. ws.add(g.root);
4. foreach(Node n: ws){
5.   for (Node m: g.neighbors(n)){
6.     int d = n.distance + g.edgeWeight(n,m);
7.     if (d < m.distance){
8.       m.distance = d;
9.       ws.add(m);
10.} } }
```

Figure 5: Ordered and unordered SSSP.

B. *Coarse- vs. fine-grained mapping*

The second dimension of exploration has to do with the mapping granularity of the work to the GPU hardware. As mentioned, a graph algorithm can be expressed as a sequence of iterations over a working set. In each iteration, a subset of elements are extracted from the working set, they are processed (e.g., their depth level or their distance from a source node is computed), their neighborhood is queried, and possibly new elements are added to the working set for the next iteration. The per-element work consists on the node processing and on the visit to its neighborhood. In each iteration, the elements extracted from the working set can be processed in parallel. One question must be addressed: how to map the work of each node onto the GPU?

Two basic mapping strategies can be devised: fine-grained (or thread-based) and coarse-grained (or block-based) mapping. In thread-based mapping, each element in the working set is mapped to a GPU-thread; each thread processes such element and visits its neighbors. In block-based mapping, each element in the working set is mapped to a thread-block. Different threads within the same block handle different neighbors of the element. Block-based mapping exhibits two levels of parallelism: (i) active elements are processed in parallel by different blocks, and (ii) neighbors are visited in parallel by different threads.

These mapping strategies have advantages and disadvantages. Fine-grained mapping is suitable for graphs with a regular topology (that is, low outdegree variance across the nodes) and in case of large working sets. In fact, a large outdegree distribution may cause work imbalances across threads during the neighborhood visit, thus leading to thread divergence. In addition, small working sets may lead to idle cores. The two-level parallelism of coarse-grained mapping is naturally suited to the GPU hardware. However, in the presence of nodes with small outdegree (i.e., less than 32 neighbors) this mapping strategy will keep some cores idle, thereby underutilizing the hardware. In addition, the GPU has a limited number of SMs. Therefore, block-based mapping is more suitable for dense graphs (i.e., graphs with high average outdegree) and for small working sets. Since the amount of per-node computation varies from application to application, block-based mapping may also be preferable when BFS and SSSP are building blocks of complex applications, and more work is associated to each element of the working set. In the pseudo-code in Figures 4 and 5, for example, if we exclude the neighborhood visit, the processing associated to each node is limited to setting the level/distance value. In more generic situations where additional work must be performed, block-based mapping brings an extra level of parallelism and allows distributing the work within the thread-block.

Both strategies allow a one-to-one and a many-to-one element-to-thread/block assignment. In the case of thread-based mapping, for example, each thread can be assigned either a single, or multiple elements of the working set. This choice affects the configuration of the kernel launches. In this work, we adopt a one-to-one mapping, which maximizes the number of threads (or thread-blocks) instantiated at every kernel call. We note that the thread- and block-based mappings are not the only options, and intermediate solutions can be devised. For example, when performing the neighborhood visit, nodes with a high outdegree can be split across multiple threads or thread–blocks. In this work, we limit ourselves to the two basic mapping strategies, and do not perform this form of load balancing.

C. *Working set: bitmap vs. queue*

The third dimension of exploration has to do with the working set representation. Previous work has adopted two

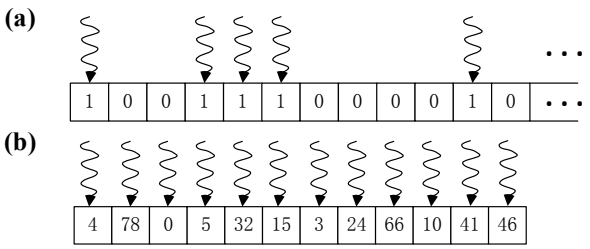

Figure 6: Working set: (a) bitmap vs. (b) queue.

representations: bitmap-based [7] and queue-based [9] working sets (Figure 6). The former consists of a 1-dimension array of bits, each indicating whether the corresponding element is in the working set and must be processed in the current iteration. The latter consists of a queue containing the identifiers of the elements to be processed in the current iteration. Bitmaps are generally used in combination with thread-based mapping [7].

Both representations have advantages and disadvantages. The bitmap solution is simple to implement, update and access. In fact, it requires only a one-dimensional array of bits, and can be accessed with minimal synchronization. However, bitmaps are inefficient when sparse, that is, when the working set is small. This is especially true if the number of threads launched is equal to the number of elements in the graph (that is, nodes in case of BFS and SSSP). In this case, most threads will be idle, leading to high GPU underutilization. This inefficiency is avoided when representing the working set as a queue, which contains only elements that must be effectively processed, and can be efficiently accessed by contiguous threads. However, a queue is more difficult to implement and especially to update, since it requires more synchronization mechanisms.

V. IMPLEMENTATION

In this section, we discuss our implementation in details.

A. Data Structures

On both CPU and GPU, we store the graph in *compressed sparse row (CSR)* form, which is an efficient encoding scheme also adopted in previous work [7]. CSR represents the nodes and the edges in the graph through two one-dimensional arrays, the *node vector* and the *edge vector*. The i^{th} entry of the node vector contains an index to the edge vector. Specifically, such index points to the start of an adjacency list containing the neighbors of node *i*. The entries in the edge vector store node identifiers, which in turn can be used to index the node vector. The size of the node vector is equal to the number of nodes in the graph (+1); that of the edge vector is equal to the number of edges.

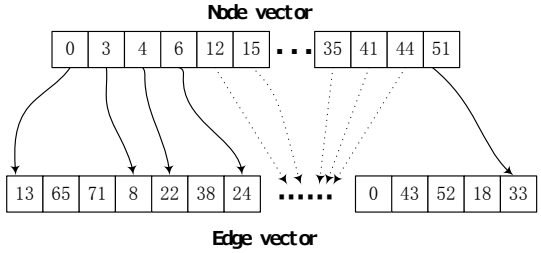

Figure 7: Compressed sparse row graph representation.

An example is shown in Figure 7, which assumes that the nodes are numbered starting from 0. The neighbors of nodes 2, for example, can be retrieved by first querying the node vector, and extracting the element at position 2 and its successor (that is, values 4 and 6, respectively). These values represent the starting and ending index of the neighbors of node 2 within the edge vector. As can be seen, the elements stored in the edge vector from position 4 to position 6 (excluded) are 22 and 38. The node vector requires an extra element to point to the end of the edge vector.

Besides the node and the edge vectors, BFS and SSSP require additional data structures, which we also represent in array form to allow for easy and efficient implementation. In particular, BFS and SSSP need an array to store the level and the distance information, respectively, which is node-specific. In addition, SSSP requires an array to record the edges' weights. Finally, some of our implementations require a node-specific *update* variable to indicate whether a node needs to be updated in the current iteration.

B. Parallel Kernels

Figure 8 shows the CPU implementation framework of BFS and SSSP. In the first three steps (lines 1-3), the data structures are created and initialized on both CPU and GPU, and the required data transfers are performed. The loop in lines 4-7 represents the graph traversal, which terminates when the working set becomes empty. Each loop iteration consists essentially of the invocation of two GPU kernels: *CUDA_computation* and *CUDA_workset_gen*. The former processes the elements in the working set, computes their level/distance value, and visits their neighborhood, adding the elements that should be processed in the next iteration to an *update* vector. Since multiple active nodes can have common neighbors, modifications to the *update* vector are performed through atomic operations (thus introducing some serializations). The *CUDA_workset_gen* kernel generates the working set by transforming the *update* vector to bitmap or queue form. The computation and the working set generation are split into two kernels because CUDA does not offer primitives for global synchronization inside kernels.

The *CUDA_computation* kernel can be implemented according to all possible combinations of the alternatives in the exploration space of Figure 3. The ordered/unordered property determines which elements to extract from the working set and how to process them (Figure 4 and 5). The mapping strategy and the working set implementation affect how the work is distributed among threads and thread-blocks. The pseudo-code in Figure 9 summarizes the body of the two

Framework of BFS and SSSP

```
1: Create data structures on CPU and GPU
2: Initialize working set on CPU
3: Transfer working set and support data from CPU to GPU
4: while working set is not empty do
5:     Invoke CUDA_computation kernel
6:     Invoke CUDA_workingset_generation kernel
7:end while
```

Figure 8: Generic CPU pseudo-code for BFS and SSSP.

```
CUDA_computation kernel
1':  id = getThreadId()                          // thread mapping
1":  id = getBlockId()                           // block mapping
2':  if (id<nodeNumber && bitmap[id ])           // bitmap
2":  if (id<queue length)                        // queue
3:       process current node                    //compute level or distance
4':      current thread visits all neighbors     // generate update vector
4":      each thread in block visits a neighbor  // generate update vector
5:   end if
CUDA_workset_gen kernel
1:   id = getThreadId()
2:   if (id<nodeNumber && update[id])
3':      generate bitmap working set             // bitmap
3":      generate queue working set              // queue
4:   end if
```

Figure 9: Pseudo-code of kernel functions. In the *CUDA_computation* kernel, instructions 1' and 4' correspond to thread-based mapping, whereas 1" and 4" correspond to block-based mapping; instructions 2' and 2'' correspond to bitmap- and queue-based working set, respectively.

kernels using different mappings and working set representations. The 1st and 2nd lines of the pseudo-code partition the work among threads or thread-blocks. The 4th line represents the neighborhood visit. In the case of thread-based mapping, a single thread visits all the neighbors of the current node; in the case of block-based mapping, each thread visits a single neighbor. In the *CUDA_workset_gen* kernel, each thread processes one element in the *update vector* and, if necessary, adds it to the working set. The queue-based implementation requires atomic operations to avoid race conditions while adding nodes to the queue.

The ordered and unordered implementations of BFS are very similar. The ordered SSSP has the added complexity of finding the minimum distance value in the working set (*findmin*), operation which is not required by the unordered version of SSSP. A CPU implementation of ordered SSSP usually uses a *heap* data structure to ensure fast insertions in the working set and accesses to it. We implemented the *findmin* operation on GPU by parallel reduction (which is faster than maintaining a heap on CPU).

C. Working Set on GPU

The bitmap representation of the working set was first adopted in [7] in combination with thread-based mapping. The advantage of this representation is its simplicity: since each node has an own entry in the bitmap and is handled by a different thread (or thread-block), no synchronization is required when accessing the working set or generating it from the *update* vector.

The queue representation has an added complexity. As explained above, the *CUDA_workset_gen* kernel requires atomic operations to convert the *update* vector into queue form. In this work, we adopt the basic implementation described in [33]. Specifically, each thread uses an atomic operation only to get a unique insertion index within the queue. Thus, threads get indexes sequentially, but insert nodes into the queue in parallel. Once the queue-based working set is created, accesses to it within the *CUDA_computation* kernel are coalesced, and do not require any additional synchronization mechanism.

There are several ways to improve the performances of work queues on GPU. Luo et al [8] propose a hierarchical queue implemented using both shared and global memory. The use of shared memory provides faster accesses and lighter synchronizations. To avoid serialization while generating indexes into the queue, Merril et al [9] replace atomic operations with prefix scans. These optimizations are orthogonal to this work, but can certainly be applied to our reference implementations.

VI. Adaptive Runtime

As discussed in the previous sections, the heterogeneity of the graphs used in practical applications makes it impossible to determine a single GPU implementation which is optimal across all datasets and algorithms. This fact is supported by the experimental evaluation that we will present in Section VII. To tackle this problem, we design an adaptive runtime that allows dynamically selecting the GPU implementation that better suits the characteristics of the dataset. Specifically, our runtime can perform coarse- and fine-grained decisions. First, given a graph and an algorithm (e.g. BFS or SSSP), it can select the best GPU implementation according to the graph topology and the underlying GPU hardware. Second, while processing a graph, our runtime can dynamically switch between different implementations of the same algorithm in different phases of the traversal.

A. Overview

The structure of our adaptive framework is shown in Figure 10. We expose to the user an API consisting of an abstract graph data type. Such API provides primitives to define and instantiate graphs, as well as functions to run the SSSP and BFS algorithms on them. At the low level, we have different GPU implementations for both SSSP and BFS, as defined by the exploration space described in Section IV. Between the graph API and the algorithm implementation layers, we have a runtime layer. Such runtime consists of two components: a *graph inspector* and a *decision maker*. The former inspects relevant characteristics of the graph (e.g., number of nodes, number of edges, minimum, maximum, average node degree), and monitors significant runtime attributes (e.g., working set size). The latter dynamically selects the most suitable implementation based on the values of these attributes and on the hardware characteristics of the underlying GPU.

In our experimental evaluation (Section VII), we found that ***unordered implementations of both BFS and SSSP generally perform better than their ordered counterparts***. This observation is coherent with [3]; this result is due not only to the larger amount of parallelism available in unordered graph algorithms, but also to the overhead due to applying the order relationship to the working set in case ordered versions. Therefore, our adaptive framework uses ***only unordered versions of SSSP and BFS***, and makes decisions in two dimensions: mapping method and working set implementation. This leads to 4 combinations: (1) thread mapping + bitmap, (2) thread mapping + queue, (3) block mapping + bitmap, and (4) block mapping + queue. As discussed below, the selection mechanism takes the

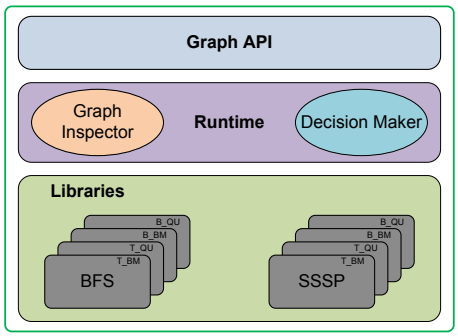

Figure 10: Overview of our adaptive framework.

utilization of the GPU hardware into account: specifically, we consider the fraction of cores and SMs effectively used, as well as the amount of thread divergence introduced.

B. Selection of the Mapping Method

The first decision to be made by our runtime system is whether to use thread- or block-based mapping. This decision is based on two considerations: core/SM utilization and amount of thread divergence introduced.

Core/SM utilization - As mentioned in Section III.C, in CUDA thread-blocks are mapped onto SMs and threads are mapped onto cores. In Fermi GPUs (used in this work), each SM consists of 32 or 48 cores. In each graph traversal step, only nodes belonging to the working set need to be processed. The *working set size* is an indicator of the amount of coarse-grained parallelism available within a graph traversal step. Thus, small working sets (for which thread-based mapping is unable to fully utilize the available GPU cores) make block-based mapping preferable. Thread-based mapping becomes a viable option when the working set size approximates the number of cores available on the GPU.

In case of large working sets, both thread- and block-based mapping are viable options, and an additional selection criterion is required. To this end, we consider the *average outdegree* of the nodes in the graph. In case of block-based mapping, the neighborhood visit is cooperatively performed by the threads within the block (that is, each thread will process one or more neighbors). The minimum practical size for a thread-block corresponds to the warp size (i.e., 32 threads). If block-based mapping is used, an average outdegree well below the warp size causes cores within a SM to be unutilized. Therefore, a small average outdegree makes a thread-based mapping preferable to block-based mapping.

Thread divergence – Using the average outdegree to discriminate between thread- and block-based mapping helps also with another consideration: in case of thread-based mapping, better performances are achieved if the amount of warp divergence is limited. Since, in case of thread-based mapping, every thread processes all the neighbors of an active node, the performances of each warp will be limited by the node with the largest outdegree. In particular, large outdegree variance may cause warp divergence. As can be observed in Table 1, graphs with high average outdegree tend to exhibit uneven outdegree distributions. By using thread-based mapping only when the average outdegree is low, we limit the amount of thread divergence which would originate by unbalanced outdegree distributions.

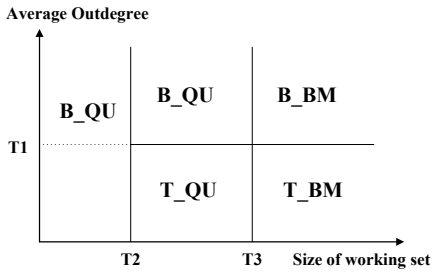

Figure 11: Decision space.

C. Selection of the Working Set Representation

The second decision to be made by our runtime is the working set representation, i.e., bitmap vs. queue. As discussed in Section V.C, a queue implementation involves a larger number of synchronizations. In particular, the creation of the queue requires a number of atomic operations equal to the queue length; such atomic operations introduce serialization among threads, thus degrading performance. This suggests that bitmaps are preferable to queues in the presence of large *working sets*. This criterion is also coherent with another consideration. When using a bitmap representation, there will be a one-to-one mapping between threads (for thread-based mapping) or blocks (for block-based mapping) and nodes. Small working sets can cause many threads/blocks to be invoked without performing any real work, leading to core/SM underutilization. Specifically, in case of a bitmap representation and a graph with |N| nodes, a working set of size |WS| leads to a fraction of wasted threads/blocks equal to 1-|WS|/|N|. In conclusion, small working sets will be implemented using a queue, and large ones using a bitmap.

D. Decision Space

The decision space resulting from the previous considerations is illustrated in Figure 11. We represent the size of the working set along the x-axis, and the average outdegree of the graph along the y-axis. This decision space is broken into 5 regions by three threshold values: T_1, T_2 and T_3. In particular, T_1 and T_2 correspond to the considerations made for the selection of the mapping method, and T_3 to the one for the choice of the working set representation.

The areas in the decision space represent different implementations. To the left of T_2, the implementation will always be B_QU (block-mapping + queue). Between T_2 and T_3, the working set is implemented with a queue while the mapping strategy depends on the average node outdegree (see T_1). To the right of T_3, the working set is represented as a bitmap, and the mapping still depends on the average outdegree (T_1). T_1, T_2 and T_3 are experimentally tuned, as discussed in Section VII.

E. Runtime Overhead

To understand the overhead introduced by our runtime, we must consider its two components: the decision maker and

Table 2: Speed up of BFS (GPU implementation over serial CPU baseline).

	O_T_BM	O_T_QU	O_B_BM	O_B_QU	U_T_BM	U_T_QU	U_B_BM	U_B_QU
CO-road	0.81	1.12	0.04	1.15	0.94	1.50	0.04	1.49
CiteSeer	24.39	15.63	12.35	49.22	24.68	15.04	12.48	48.94
p2p	3.79	3.34	0.93	3.22	3.66	3.44	0.95	3.37
Amazon	13.59	11.12	2.05	10.62	13.94	10.60	2.07	11.52
Google	20.76	18.82	2.94	18.90	21.57	18.03	2.96	20.36
SNS	20.39	16.33	8.43	24.02	24.04	18.00	8.64	24.30

the graph inspector. The former has extremely low overhead since its logic (summarized by Figure 11) is straightforward. In order to allow the decisions described above, our graph inspector must monitor the working set size and the average outdegree of the nodes within the working set. This information can be collected at runtime by running a separate kernel (parallel scan can allow a more efficient computation of the average outdegree). This overhead is much greater than that of the decision maker. In our implementation, we reduce this overhead in two ways: (i) by considering the average outdegree of the whole graph (which is a value computed only once when reading the graph) rather than the one of the current working set, and (ii) by sampling (that is, by not performing measurements in every traversal step). These design decisions represent a trade-off between execution efficiency and runtime overhead. The selection of the sampling rate and its effect on performances will be discussed in Section VII.

VII. Experimental Evaluation

We present an experimental evaluation on the datasets of Table 1. We first evaluate the static implementations corresponding to Figure 3. We then study how to tune the parameters of our adaptive runtime. We finally compare the performance achieved through our runtime with those achieved through the static solutions.

Our testing platform consists of an Intel Core i7 CPU (running CentOS 5.5) and an Nvidia Tesla C2070 GPU, which contains 14 32-core SMs. We use *gcc* 4.1.2 and *nvcc* 4.0 compilers, both with –O3 optimizations. Our results include CPU processing, GPU processing and CPU-GPU transfer times. We do not measure the time spent loading graph data from the hard drive.

A. Performance of Static Implementations

Tables 2 and 3 summarize the performances of the BFS and SSSP implementations covering the exploration space in Figure 3. In particular, the tables report the speed up of each GPU implementation over a serial CPU implementation. All the GPU solutions are named with three fields separated by underscore. The first field indicates whether the implementation is ordered (O) or unordered (U); the second field distinguishes thread-based (T) and block-based (B) mapping; the third field indicates the representation of the working set: bitmap (BM) vs. queue (QU). For instance, "O_B_BM" indicates that the implementation is ordered, uses block-based mapping and a bitmapped working set. For each dataset, grey cells show the best performance achieved.

The tables show the results reported using the best kernel configurations, which have been obtained by using the "CUDA Occupancy Calculator" and conducting experiments under different settings. When using thread-based mapping, we found that the best results can be achieved with 192 threads per block. When using block-based mapping, the optimal number of threads per block is the multiple of 32 closest to the average node outdegree in the graph.

For BFS, we can make the following observations. First, when using the same mapping strategy and working set representation, ordered and unordered algorithms achieve very similar performance. In ordered BFS, the nodes are processed level by level and each node is accessed exactly once. In unordered BFS, in principle each node may be updated multiple times. However, since in every iteration we process the entire working set, our unordered GPU implementation also proceeds level by level, unless the working set is initialized through some depth-based traversal. We experimentally verified that limited amount of initialization (e.g., depth-based traversal in 3-5 directions) does not substantially affect the results. Second, the GPU implementation does not outperform its CPU counterpart on all datasets. In fact, the GPU performance is poor for the *CO-road* network, whose average outdegree is only 2.6 (see

Table 3: Speed up of SSSP (GPU implementation over serial CPU baseline – Dijkstra's algorithm).

	O_T_BM	O_T_QU	O_B_BM	O_B_QU	U_T_BM	U_T_QU	U_B_BM	U_B_QU
CO-road	0.02	0.01	0.0012	0.01	1.88	1.76	0.35	2.11
CiteSeer	136.23	112.77	11.81	139.53	126.47	118.91	483.24	867.91
p2p	1.29	1.16	0.22	1.22	135.65	127.38	49.17	131.88
Amazon	3.29	3.03	0.26	3.12	95.10	58.93	37.71	99.83
Google	2.27	2.16	0.18	2.19	96.57	58.12	32.94	89.32
SNS	25.37	24.33	1.87	24.81	174.82	140.45	136.08	276.23

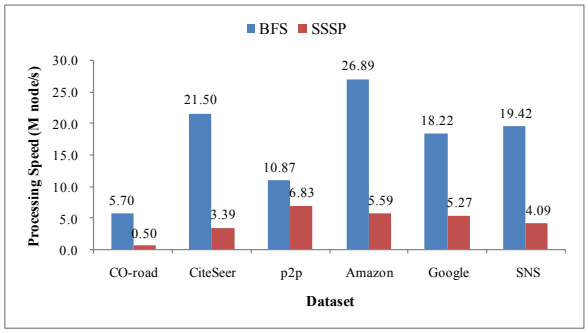

Figure 12: Processing speed of best implementation.

Table 1), and whose diameter is relatively large (more than 1000 levels). Third, the best GPU implementation varies from dataset to dataset. For instance, the *CO-road* and *CiteSeer* networks favor U_B_QU, while the *Amazon* and *p2p* networks achieve best performance with U_T_BM.

For SSSP, we can observe the following facts. First, unordered algorithms are significantly faster than their ordered version. This has two motivations: (1) unordered SSSP exhibits more parallelism than ordered SSSP, and (2) ordered SSSP suffers from the cost of implementing the *findmin* operation. Second, the ordered SSSP on GPU can achieve considerable speedup over its serial CPU version. In every iteration, nodes at the same distance can be processed in parallel. In addition, the parallel reduction on GPU is a good alternative to executing the *findmin* operation on CPU. Third, by concurrently processing the elements in each node's neighborhood, block-based mapping leads to high speedups on graphs with large average outdegrees (e.g., *CiteSeer* and *SNS*). Finally, we can again observe that the best implementation strictly depends on the dataset; for example U_B_BM performs very well on *CiteSeer*, but exhibits the worst performance on the other 5 datasets.

Figure 12 shows the processing speed (in millions nodes per second) of the best GPU implementation of BFS and SSSP across the considered datasets. BFS achieves better performance than SSSP due to its faster convergence. Our experiments show similar results to previous work [8] and prove that GPU can be successfully used to run graph algorithms on large datasets. Since our results show that the best solution depends on the characteristics of the underlying dataset, we now evaluate the use of our adaptive runtime.

B. *Parameter Tuning for our Adaptive Runtime.*

Before evaluating the performance of our adaptive runtime, we study how to tune its parameters. In particular, we start with T_1, T_2, and T_3, the thresholds used by the decision maker and illustrated in Figure 11.

Recall that T_1 is related to the average outdegree of the graph, and allows discriminating between thread- and block-based mapping when the size of the working set would allow both alternatives. Since each thread-block must at least contain one warp (i.e., 32 threads), if the average outdegree is less than 32, block-based mapping will underutilize the hardware resources. Thus, we set T_1 to 32.

T_2 indicates the size of the working set below which block-based mapping should be always preferred to thread-based mapping. Its value is related to the kernel configuration and the number of SMs on the GPU. As mentioned in the previous section, we experimentally verified that good configurations for thread-based mapping are characterized by 192 threads per block. The GPU used in our experiments has 14 SMs. When the size of the working set is less than 192*14 = 2,688 nodes, thread-based mapping will leave some SMs idle, thus underutilizing the GPU. To confirm this analysis, we measure the kernel execution time of T_QU and B_QU across all iterations of BFS and SSSP. Our results show that B_QU outperforms T_QU for working set sizes smaller than ~3000. Therefore, we set T_2 to 2,688.

T_3 indicates the size of the working set above which a bitmap representation is preferable to a queue. In Figure 13 we report the results of experiments conducted to study how the performance changes with T_3. We recall that the ratio between the size of the working set and the number of nodes in the graph indicates, in case of a bitmap representation, the fraction of threads/blocks instantiated that will effectively perform some work. Therefore, in the x-axis we show the

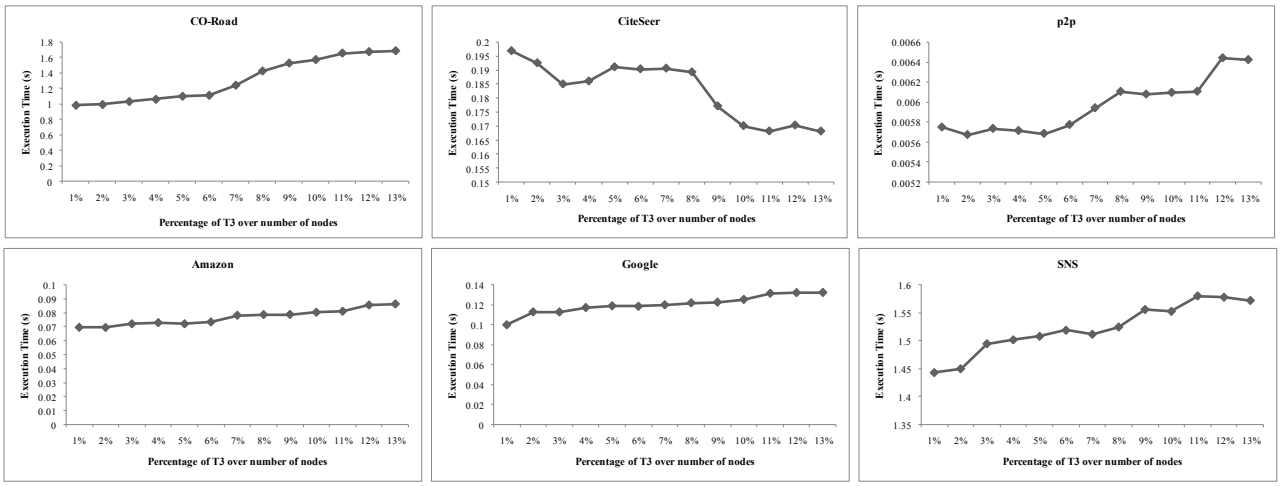

Figure 13: Performance under different T_3 settings (SSSP)

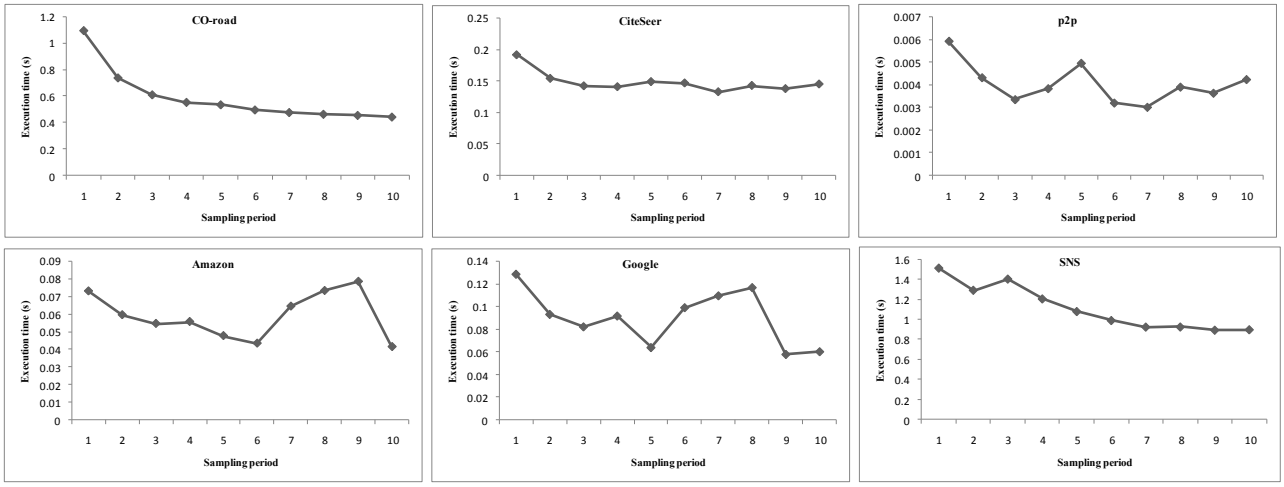

Figure 14: Performance under different sampling rates (SSSP).

percentage ratio of T_3 over the number of nodes in the graph. As can be seen, for all datasets but *CiteSeer*, the execution time increases with T_3. This can be easily explained as follows: in the presence of large working sets, the queue generation incurs higher overheads due to atomic operations. When the ratio T_3/#*node* is less than 6%, the execution time increases very slowly. However, when it exceeds 7% or 8%, the execution time increases rapidly (*CO-road*, *p2p* and *SNS*). Although this trend is not exactly the same for all datasets, we set the value of T_3 to 6%. It is worth explaining why, in the case of *CiteSeer*, the execution time decreases even when the ratio T_3/#*node* reaches 13%. The *CiteSeer* dataset is characterized by a high average outdegree, which leads to higher parallelism. In case of a queue implementation, the amount of work performed within a thread-block amortizes the overhead due to the atomic operations performed when generating the queue, making a queue preferable to a bitmap.

The graph inspector introduces a runtime overhead while monitoring the working set size. Such overhead can be reduced by performing this measurement task periodically, rather than at every iteration. Figure 14 shows the performance under different sampling rates. Due to lack of space and given the similarity between BFS and SSSP, we show only the SSSP results. We can observe that the performance generally benefits from a decreased sampling rate. However, the changes are not smooth and vary across datasets. In general, datasets characterized by a longer convergence time (e.g. *CO-road*, *CiteSeer*, and *SNS*) experience a slow and steady performance improvement as the sampling rate decreases. On the other hand, graphs that take only a few iterations to converge (e.g., *Google*, *p2p*) are more sensitive to changes in the sampling rate. To make a trade-off, we set the sampling rate of our adaptive runtime to 6. With this setting, we observed a runtime overhead varying from 10.6% (on *CiteSeer*) to 13% (on *p2p*).

C. Overall Performance of Adaptive Runtime

Figure 15 shows the speedup comparison between our adaptive runtime, the worst and the best static solution. In every case, the best static implementation is taken as baseline. Only unordered implementations are considered (also for the worst case). The values reported on the bars are the speedup numbers of the worst static and of the adaptive solution over the best static implementation. Since our goal is to argue that an adaptive system can capture dynamic parallelism and not to develop a highly tuned code, in our study we use basic kernels similar to those by Harish and Narayanan [7]. However, the performance achieved by our dynamic solution is in the same order of magnitude as that achieved by Merrill et al [9] (for example, our dynamic BFS computes 0.74 billion edges/sec on the *CiteSeer* network).

We can observe the following. First, the performance of the worst static solution can be as low as 3% (and as high as 52%) of that of the best static solution. Second, despite its overhead, our adaptive runtime achieves better performance than the best static implementation on most of these real world graphs. Specifically, the speedup over the best static solution ranges from 1.43 to 2.02. Although on *CiteSeer*, *Google* (SSSP) and *SNS* (SSSP) the adaptive runtime has no advantage compared to the best static solution, it still achieves 95-99% of its performance (and avoids the penalty associated with possibly choosing a bad implementation).

VIII. Conclusion and Future Work

In this paper we explored different ways to implement graph algorithms on GPU. Our characterization of datasets used in real world applications motivate us to design an adaptive runtime system that dynamically selects the most suitable GPU implementation of a given graph algorithm based on the topology of the input dataset and on other parameters monitored at runtime. Our experiments show that our framework can achieve higher performance than a static solution and is resilient to the irregularity and heterogeneity of real world graphs.

In the future, we plan to extend our framework to support more graph algorithms (e.g. minimum spanning tree and minimum cut) and multiple GPUs. In addition, we plan to integrate other mechanisms (e.g. speculative execution, dynamic parallelism) that can enable and facilitate the effective deployment of graph algorithms on GPUs.

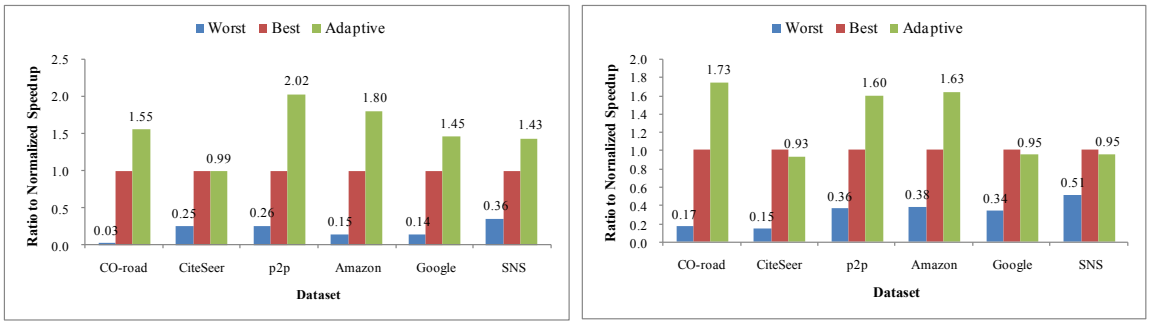

Figure 15: Performance of our adaptive runtime on BFS (to the left) and SSSP (to the right) – baseline: best static solution.

Furthermore, we plan to validate our study on large, synthetic datasets (e.g., Graph500).

ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This work has been supported by NSF award CNS-1216756 and by equipment donations from Nvidia Corporation.

REFERENCES

[1] M. Kulkarni, K. Pingali, B. Walter, *et al*, "Optimistic parallelism requres abstractions," in Proc. of PLDI 2007.

[2] M. Kulkarni, M. Burtscher, *et al*, "How much parallelism is there in irregular applications," in Proc. of PPoPP 2009.

[3] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in Proc. of PPoPP 2011.

[4] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations," In Proc. of Parallel Object-Oriented Scientific Computing, 2005.

[5] F. Hielscher and P. Gottschling. (2004). *ParGraph*. http://pargraph.sourceforge.net/

[6] P. An, A. Jula, S. Rus, *et al*, "STAPL: an adaptive, generic parallel C++ library," in Proc. of LCPC 2003.

[7] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in Proc. of HiPC 2007.

[8] L. Luo, M. Wong, and W.-M. Hwu, "An Effective GPU Implementation of Breadth-first Search," in Proc. of DAC 2010.

[9] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in Proc. of PPoPP 2012.

[10] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU Implementation of Inclusion-based Points-to Analysis," in Proc. of PPoPP 2012.

[11] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Computing Strongly Connected Components in Parallel on CUDA," in Proc. of IPDPS 2011.

[12] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," in Proc. of PPoPP 2011.

[13] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," in Proc. of PACT 2011.

[14] S. Che, M. Boyer, J. Meng, *et al*, "Rodinia: A benchmark suite for heterogeneous computing," in Proc. of IISWC 2009.

[15] V. W. Lee, C. Kim, J. Chhugani, *et al*, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in Proc. of ISCA 2010.

[16] *9th DIMACS Implementation Challenge*. www.dis.uniroma1.it/challenge9

[17] *10th DIMACS Implementation Challenge*. www.cc.gatech.edu/dimacs10/index.shtml

[18] *Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data

[19] D. A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in Proc. of ICPP 2006.

[20] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in Proc of SPAA 2010.

[21] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable Graph Exploration on Multicore Processors," in Proc. of SC 2010.

[22] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A Parallelization of Dijkstra's Shortest Path Algorithm," in Proc. of MFCS 1998.

[23] G. Vaira and O. Kurasova, "Parallel Bidirectional Dijkstra's Shortest Path Algorithm," in Proc. of DB&IS 2010, 2011.

[24] D. B. Johnson and P. Metaxas, "A parallel algorithm for computing minimum spanning trees," in Proc. of SPAA 1992.

[25] F. Dehne and S. Gotz, "Practical Parallel Algorithms for Minimum Spanning Trees," in Proc. of SRDS 1998.

[26] D. A. Bader and G. Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," *J. Parallel Distrib. Comput.,* vol. 66, pp. 1366-1378, 2006.

[27] V. Koubek and J. Krsnakova, "Parallel algorithms for connected components in a graph," in Fundamentals of Computation Theory, 1985.

[28] H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph," in Proc. of FOCS 1986.

[29] I. William Mclendon, B. Hendrickson, *et al*, "Finding strongly connected components in parallel in particle transport sweeps," in Proc. of SPAA 2001.

[30] W. Schudy, "Finding strongly connected components in parallel using $O(\log^2 n)$ reachability queries," in Proc. of SPAA 2008.

[31] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue,* 2008.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*: MIT Press, 2001.

[33] J. Balfour. (Aug 5). *CUDA Threads and Atomics*. mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf