2014 IEEE 28th International Parallel & Distributed Processing Symposium

Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths

Andrew Davidson
University of California, Davis
aaldavidson@ucdavis.edu

Sean Baxter
NVIDIA Research
sbaxter@nvidia.com

Michael Garland
NVIDIA Research
mgarland@nvidia.com

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

***Abstract*—Finding the shortest paths from a single source to all other vertices is a fundamental method used in a variety of higher-level graph algorithms. We present three parallel-friendly and work-efficient methods to solve this Single-Source Shortest Paths (SSSP) problem: *Workfront Sweep*, *Near-Far* and *Bucketing*. These methods choose different approaches to balance the tradeoff between saving work and organizational overhead.**

In practice, all of these methods do much less work than traditional Bellman-Ford methods, while adding only a modest amount of extra work over serial methods. These methods are designed to have a sufficient parallel workload to fill modern massively-parallel machines, and select reorganizational schemes that map well to these architectures. We show that in general our *Near-Far* method has the highest performance on modern GPUs, outperforming other parallel methods.

We also explore a variety of parallel load-balanced graph traversal strategies and apply them towards our SSSP solver. Our work-saving methods always outperform a traditional GPU Bellman-Ford implementation, achieving rates up to 14x higher on low-degree graphs and 340x higher on scale-free graphs. We also see significant speedups (20–60x) when compared against a serial implementation on graphs with adequately high degree.

***Keywords*-GPU computing, graph traversal, single-source shortest paths, sparse graphs**

I. Introduction

In a graph, finding the shortest path from a single source node to all connected nodes is a well-known and long-studied problem with many practical applications on a wide spectrum of graph types (e.g., road networks, 3D models, AI, and social networks) [1], [2]. Known as "single-source shortest path" (SSSP), this graph primitive will be one of three core application kernels in the new Graph 500 benchmark [3].

The input to SSSP is a source node in a graph with v *vertices* and e directed *edges* with non-negative weights. We treat undirected edges as two directed edges, pointing in both directions. Figure 1(a) shows an example graph with a source and edge weights, and Figure 1(c) shows the resulting final distances to all nodes. As an example, consider a road network, where vertices are cities, (undirected) edges are roads between cities, and weights are road distances. Beginning with a home city (the source), SSSP calculates the shortest distances between the home city and each other city. The number of edges (roads) connected to each vertex (city) is called the *degree* of the vertex; the number of hops in the longest shortest path (the largest number of roads required to travel between any two cities) is called the *diameter* of the graph.

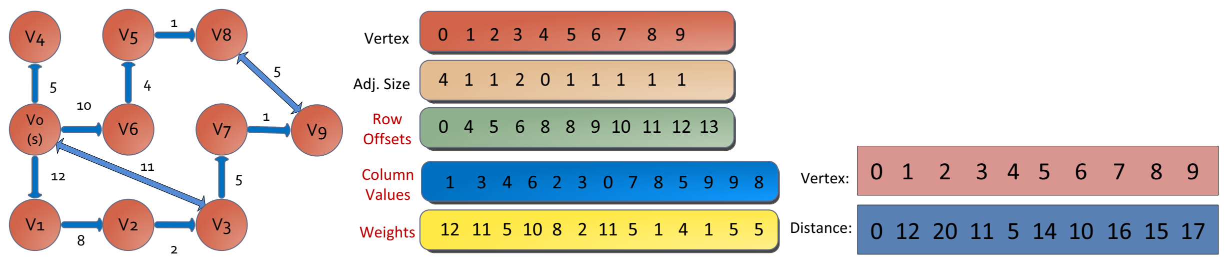
The traditional serial approach to SSSP is Dijkstra's method (Section II-A), which utilizes a priority queue where one vertex is processed at a time. While this method is an efficient ($O(v \log v + e)$) serial algorithm, it is poorly suited for a parallel architecture such as a GPU that requires thousands or more parallel threads to fully occupy the machine. To expose more parallelism, we consider algorithms that can process more vertices at the same time. The Bellman-Ford algorithm (Section II-B) is one such method. It repeatedly processes all edge connections, updating vertices continuously until final distances converge. Though Bellman-Ford is classically a serial algorithm, it is well-suited to parallel execution across vertices, and unlike Dijkstra, also works on graphs with negative-length cycles, which we do not consider in this paper. However, these features come with a higher cost: $O(ev)$ work.

These two classic algorithms span a parallel vs. efficiency spectrum. Neither is ideal: Dijkstra exposes no parallelism across vertices, while Bellman-Ford is expensive. This paper explores the space between these two endpoints, studying algorithms that both exploit parallelism and maintain efficiency. We develop and describe three such methods that are targeted for fine-grained, massively parallel machines such as GPUs. We analyze these three methods in the context of the tradeoff between parallelism and efficiency on a variety of graphs. With our implementations, we show that in practice, we deliver good parallel efficiency while adding only a modest amount of extra work: the overall work complexity is usually much closer to Dijkstra than Bellman-Ford. On all graphs, we demonstrate significant speedups over the previous GPU state of the art, LonestarGPU's Bellman-Ford implementation [4], and on dense graphs that are more amenable for parallel SSSP computation, speedups over a variety of CPU and GPU implementations.

II. Related Work

A. Dijkstra's Algorithm

Dijkstra's algorithm [5] is the most efficient sequential algorithm on directed graphs. The algorithm maintains a priority queue of vertices prioritized by its shortest discovered


V4
V5
V8
V0 (s)
V6
V7
V9
V1
V2
V3
1
5
5
4
10
1
11
12
5
8
2
Vertex 0 1 2 3 4 5 6 7 8 9
Adj. Size 4 1 1 2 0 1 1 1 1 1
Row Offsets 0 4 5 6 8 8 9 10 11 12 13
Column Values 1 3 4 6 2 3 0 7 8 5 9 9 8
Weights 12 11 5 10 8 2 11 5 1 4 1 5 5
Vertex: 0 1 2 3 4 5 6 7 8 9
Distance: 0 12 20 11 5 14 10 16 15 17

(a) Sample Graph (b) Resulting CSR Format (c) Shortest Paths result

Figure 1. An example weighted graph in CSR format with final distances calculated by SSSP. Figure 1(b) shows how a vertex array with adjacency sizes can be represented by each vertex's row offsets and edge pointers (column values). The bottom three arrays marked with red text are the only arrays actually stored (total size: $2e + v$).

distance. It considers the top vertex on the queue and all edges leaving that vertex. Vertices reached by these edges are added back into the priority queue if they haven't been reached before, or the distance value is updated if a shorter path has been discovered.

Using Fibonacci heaps, or relaxed heaps, the sequential runtime for this method is bounded by $O(v \log v + e)$ [6]. These heap methods reduce the cost of setting vertex values at the cost of added code complexity [2]. However, Fibonacci heaps have large constant factors in practice, so using simpler binary heaps can be more efficient. We test against two implementations of Dijkstra's algorithm: (1) a Boost Graph Library [7] implementation using a relaxed heap, and (2) our own implementation using a simpler binary heap.

Dijkstra's method exposes no parallelism between vertices; the only parallelism available is between edges leaving the vertex at the top of the queue. It yields an efficient serial implementation, but is poorly suited for parallel architectures like GPUs that require large numbers of parallel threads for efficient execution.

PHAST: The work we present in this paper begins with a graph with no preprocessing, but recent work on parallel hardware-accelerated shortest path trees (PHAST) [8] uses preprocessing techniques using contraction hierarchies [9] to expose more parallelism in SSSP calculation. PHAST's preprocessing step precomputes distances to highly ranked high-degree vertices using a Dijkstra-like traversal. At runtime, PHAST can then perform multiple searches in parallel from these highly ranked vertices. In practice, contraction hierarchies (and therefore PHAST) work well on low-degree, high-diameter graphs such as road networks. However, for highly connected graphs (such as power-law graphs), the contraction hierarchy preprocessing step is expensive, as the number of shortcuts explodes as the dimensionality increases, while lower-degree graphs add extra parallelism without the large amounts of extra work.

B. Bellman-Ford

Rather than operate on only the vertex with the lowest weight, Bellman-Ford operates on all vertices independently. Each vertex maintains its distance to the source. On each Bellman-Ford iteration, a vertex checks each adjacent vertex, updating its own distance to the source if it finds any shorter path. This operation is repeated until the distances converge. In the worst case, this operation repeats $v - 1$ times, performing $O(ev)$ work, but for low-diameter graphs, implementations typically converge with fewer iterations.

Given how easy it is to parallelize, Bellman-Ford is a popular SSSP method for any parallel architecture, with the LonestarGPU graph library [4], [10] representing the current state of the art on GPUs. Their implementation parallelizes over vertices, with one thread per vertex that updates distance labels for all adjacent vertices. In order to avoid race conditions, an atomic min is used for each of these updates.

C. Delta-Stepping

The Delta (Δ)-stepping method for SSSP, proposed by Meyer and Sanders [11], extracts parallelism by relaxing the one-vertex-at-a-time constraint imposed by a traditional serial Dijkstra's algorithm. Rather than considering one vertex at a time from a priority queue like Dijkstra's method, delta-stepping groups vertices into buckets and processes all vertices within a bucket simultaneously and in parallel.

In delta-stepping, all vertices are grouped into buckets depending on the distance of the vertex from the source. Vertices that lie in a specific distance range share the same *bucket*. Delta-stepping then processes vertices from the smallest bucket in parallel. Edges emanating out of a vertex are grouped into two categories: *light* edges and *heavy* edges. For light edges, the updated distance of the destination vertex lies within the same bucket as the source vertex. Destination vertices attached to heavy edges lie outside this range and inside another bucket.

Delta-stepping first considers light edges in a bucket and processes them until the entire bucket is emptied. Then it turns to the heavy edges. Heavy edges that result in a smaller delta for a particular vertex require moving that vertex to a closer bucket (similar to relabeling in Dijkstra's method).

Efficient delta-stepping methods have been successfully implemented on coarse multi-threaded machines and vector PRAM machines. We are aware of one previous GPU

Algorithm	Wk. Complexity	Type	Parallelism
Dijkstra	$O(v \log v + e)$	General	Serial
Bellman-Ford	$O(ve)$	High-Degree	Parallel
Delta Step	$O(v \log v + e)$	General	Coarse Parallel
PHAST	$O(v \log v + e)$	Low-Degree	Preprocessing Parallel

Table I
SUMMARY OF PRIOR WORK IN SSSP ALGORITHMS.

implementation of delta-stepping by Baggio [12]. Baggio tests his implementation on a fixed 2D grid (1024x1024 size) with possible edges emanating to neighbors on the grid. Utilizing an NVIDIA 8600 GT, their implementation remains inefficient; they achieve rates 8–17x times slower than Dijkstra's method.

There are three main characteristics that make delta-stepping difficult to implement efficiently on a GPU-like machine. First, delta-stepping's bucket implementation requires dynamic arrays that can be quickly resized in parallel. Dynamic arrays are poorly suited to the current programming model of GPUs, and implementing a custom memory management system for dynamic arrays (utilizing heaps) would be difficult and inefficient. Second, fine-grained renaming and moving vertices between buckets is difficult to parallelize, likely requiring atomics and thus losing concurrency. Finally, efficient GPU implementations require exploiting the GPU's three-layer memory hierarchy (global DRAM, per-block shared memory, and per-thread registers), but such a memory hierarchy is absent from the traditional delta-stepping formulation.

III. GPU PRELIMINARIES

We briefly introduce the salient characteristics of GPUs and efficient GPU programs. GPUs operate on a large number of parallel threads organized into a computation hierarchy. These threads are programmed with a single-instruction, multiple-threads (SIMT) programming model in which threads are grouped into small, divergent-free groups called *warps* (on our GPU, warps have 32 threads). These warps are then grouped into *blocks* whose threads can communicate through a pool of on-chip shared memory. The GPU also features a memory hierarchy, with per-thread registers, per-block shared memory, and global DRAM accessible to all threads; memory accesses in global memory are an order of magnitude faster when neighboring threads read or write neighboring data ("coalescing") when compared to random accesses.

Therefore, in order to get high performance on GPU graph algorithms, we prioritize methods that (a) contain enough parallelism to fill the machine, (b) effectively use the memory hierarchy, (c) avoid thread divergence within a warp, and (d) limit scattered reads and writes. Designing an SSSP method that fulfills all of these qualities is a significant challenge. Bellman-Ford on large graphs, for example, only fulfills the first. We also prioritize the use of high-performance parallel algorithmic GPU primitives (e.g., scan, reduce, vector search) when applicable.

IV. SSSP ALGORITHMS

Most graph algorithms, including SSSP, require three components to solve their graph problem. These three components are:

1) A data structure to store vertices
2) A graph traversal method that visits selected vertices
3) A methodology for organizing which vertices to process next.

As an example, let us consider Dijkstra's method. Here, the data structure is a managed priority queue. All elements within the priority queue will be considered at some point. Since this method is serial, the graph traversal method is straightforward: the serial thread considers each adjacent vertex in order. The key to Dijkstra's efficiency is the methodology for organizing vertices within the priority queue. It ensures that the next vertex to be processed at each step is the vertex with the lowest labeled distance.

The three components for a parallel Bellman-Ford implementation are also straightforward. The data structure is a set that contains all vertices; the graph traversal method maps each thread to a single adjacency list; and on each iteration, every vertex is processed until the graph converges.

The methods we develop also contain three components. First, we maintain a work queue that divides all vertices into one or more buckets, where the vertices within a bucket can be processed in parallel within an iteration. Second, we demonstrate a load-balanced method to efficiently traverse the vertices within a bucket. The key difference between our three methods is in the third component: how we organize the vertices to be processed next.

A. Graph Data Structures

We select the Compressed Sparse Row (CSR) graph format to store our graphs. For a more in-depth look at possible graph formats on the GPU, Bell and Garland [13] discuss possible implementations, advantages, and disadvantages of several formats for sparse matrices, which also apply in our case to graphs. First, CSR is space-efficient, requiring one entry for every vertex and one entry for every edge. Second, CSR works well for general rather than specialized matrices, and is commonly used by other libraries allowing for data reuse. Finally, it is easy to strip out adjacency lists and find offsets using scan, a common parallel primitive, on the row offsets array. Figure 1(b) shows the CSR representation for an example graph.

B. Graph Traversal

Our graph traversal methods input a subset of vertices and visit all edges in their per-vertex adjacency lists. Because these adjacency lists can vary greatly in size, traversing these lists in parallel and in a load-balanced way is essential when developing a work-efficient graph algorithm. A natural way to parallelize this traversal is to assign one independent thread per list (as in straightforward GPU implementations of Bellman-Ford). However, since each edge list has a variable length, such a method would be inherently load-imbalanced. More complex strategies that evenly split work within adjacency lists across threads will achieve better load balance at the cost of extra work to split these work items between threads.

Rather than independently processing per-thread adjacency lists, we can address load imbalance by cooperating between threads. The following graph traversal methods use this implementation strategy to efficiently traverse multiple lists in parallel. Merrill et al. [14] discuss this problem in the context of a parallel breadth-first search (BFS) algorithm. They propose a three-pronged approach to traverse a set of adjacency lists that we discuss in further detail below.

Though both our SSSP methods and Merrill's BFS require efficient graph traversal, they differ in several important ways. In BFS, any race condition between edges visiting a vertex is benign. However, SSSP maintains distances rather than depths, so atomics are needed to resolve race conditions. Secondly, in BFS, nodes are never visited in a later iteration. In SSSP, if a new distance is updated, this vertex will be reinserted into the work queue. Finally, BFS only requires flags for visited and unvisited vertices, while SSSP requires keeping track of edge weights, distances from a source node, and the distance of the destination node. Due to these difference we feel it important to explore a number of graph traversal strategies, and measure there performance for processing an SSSP work queue. These traversal methods include *Group Blocking*, Merrill's *CTA+Warp+Scan* method, and a *Edge Partitioned* method. All of these methods input the same data structure, a workfront of vertices, and output a set of updated adjacent vertices.

Cooperative Blocks: Our first traversal strategy assigns a block of threads (a *cooperative thread array* (CTA)) to process a set of n vertices in parallel. Each thread loads the edge list offset for an assigned vertex into shared memory. This offset list is similar in structure to the row offsets found in Figure 1(b), but organized to fit in shared memory.

Threads cooperatively strip edges off the created edge list and process each edge independently. In order to determine the originating vertex, a binary search is initially performed in the n offset list in shared memory; the destination vertex can then be determined by subtracting the value from the edge offset list. When a thread moves to a new work item, it checks to see if a new vertex lookup is needed. For large adjacency lists these searches are rarely needed, while smaller lists require more frequent updates.

Since threads cooperate to solve a set of vertices, work within a block is load-balanced. However, different blocks of vertices may have more work than others, leading to a possible interblock load imbalance. For many graphs, this is usually not an issue since taking a CTA sized subset of the graph will often be a representative average of all vertices. For graphs of sufficient size (many blocks), the GPU's ability to quickly context-switch between blocks also contributes to good overall load balance.

CTA+Warp+Scan: Since per-vertex adjacency lists can be arbitrarily small or large, specializing strategies for certain types of adjacency lists can both reduce the amount of overhead within each kernel and better utilize the GPU. The CTA+Warp+Scan method separates adjacency lists into three categories based on size: (a) larger than a CTA (the number of threads per block); (b) larger than a warp (32 threads) but smaller than a CTA; and (c) smaller than a warp.

First, we assign a group of vertices to a block. Within that block, each thread owns one vertex, and determines which type of adjacency list it contains. Threads assigned an edge list larger than a CTA get first priority and arbitrate for control of the entire block (each thread writes to a shared memory location, and the winner is the last to write). Now all threads within the CTA cooperate with this winner to process that vertex. We then rearbitrate between threads for control until all CTA-category vertices are processed. Now all unprocessed vertices have size less than a CTA. Next, threads that own vertices with adjacency lists larger than a warp perform a similar operation. Each active thread now arbitrates within its warp for control, and the winner processes its edge list with help from all other threads in the same warp. Finally, in the Scan portion of this traversal method, threads owning lists smaller than a warp iteratively update their vertex ID and edge offset into a shared memory pool. Threads stall waiting for the largest list to be loaded in a set. Then each thread grabs an edge (work item) to be processed. The process is continued until all smaller adjacency lists are processed.

The benefit of this approach is specialization for each type of adjacency list, and correspondingly higher throughput. However, since we are separating each work-group into three separate stages within our kernel, we lose parallelism within each type. Also, since each thread in the *Scan* portion of the algorithm must communicate its whole adjacency list to the rest of the CTA, other threads stall while waiting for all of these items to be loaded.

Load-Balanced Partitioning: Instead of grouping an equal number of *vertices* to be traversed by a block, our load-balanced partitioning traversal attempts to organize groups of *edges* of equal lengths. To perform this division, since adjacency lists have arbitrary size, we must find the intersection of each block's edge-list start and end points within the work queue. We can do this easily with an efficient sorted search [15], which maps edge start and end points with our scanned edge offset queue. Like group blocking, when we strip edges to be processed, if we encounter a change in a vertex, we must do an update. Using this method,

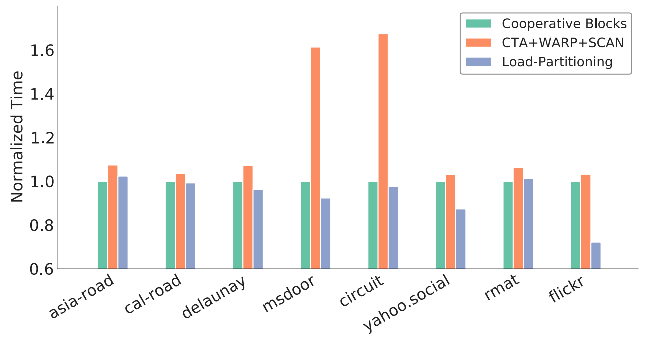


Figure 3. Performance of our graph traversal strategies on a variety of graph types (summarized in Table II).

we ensure perfect load-balance between blocks. Since each thread within the block is always finding and consuming work, we are also load-balanced *within* a block.

Figure 3 shows the relative performance of all traversal methods on a variety of graph types. For very low-degree graphs, all of these traversal methods have similar performance. However, CTA+Warp+Scan performs poorly on medium-degree graphs, and Partitioning performs very well on high-degree dense graphs. Due to its ability to handle pathological cases, and its overall edge in performance on our tested datasets, we utilize the Partitioning method for our graph traversals.

C. Work Organization

Finally, we turn to the problem of selecting the vertices to process. Here we trade off parallelism and work: the more effort we put into managing an ordering of vertices to process, the more work we save. However, this comes at the cost of overhead to manage this ordering, extra iterations (step-efficiency) to solve the entire problem, and a reduction in parallelism.

We develop three different variants to organize and select what work to process: Workfront Sweep, Near Set and Far Pile (Near-Far), and Bucketing. For all of these implementations, the initial setup is the same. We initialize the source distance to zero, and all other distances to infinite.

Workfront Sweep: The simplest strategy to improve on Bellman-Ford is to prune the list of active vertices on each iteration to avoid reprocessing a vertex whose computation has not changed from the previous iteration. We begin with a vertex queue containing only the source. On each iteration, we visit all associated edges in parallel for each vertex in the queue and update the distances of the vertices attached to those edges. Since these edges are being processed in parallel in a non-deterministic order, we must either save updated distances and process them later or instead use atomics (an atomic min) to ensure proper behavior. On any iteration, if a vertex is marked as updated, it is inserted into the vertex queue for processing in the next iteration. We continue to process this queue until it becomes empty.

Since we do not know the order in which distances will be compared using atomics, it is possible (and for highly connected graphs very likely) to add duplicate vertices into the queue. In terms of correctness, these reinsertions are benign, but will affect performance due to redundant computation. After each workfront expansion iteration, we can remove duplicates in parallel with one simple pass. Each vertex attempts to write its workfront index to a lookup table indexed by vertex id. Then each workfront item reads back from the same lookup table. If the item read is equal to the workfront index, then it is declared the owner of that vertex, and remains valid. Otherwise that index is marked invalid and is compacted away before beginning the next iteration.

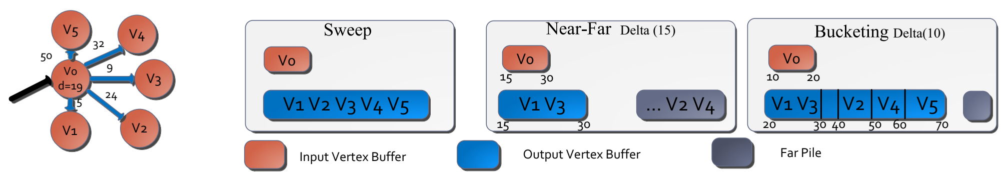
This queue management system is similar to Shun and Blelloch's Ligra [16], designed for multi-core CPUs. Though they do the same amount of work, the parallel distribution of these two methods differs greatly. Ligra utilizes a different graph representation and traversal strategy better suited for non-bulk-synchronous programming. For each valid vertex it finds, it automatically splits work within the adjacency list (utilizing a parallel for), rather than our bulk-synchronous management of vertices, splitting of adjacency lists, and traversal of edges.

In Workfront Sweep, the amount of extra work to organize the queue at each iteration is minimal, but leads to substantial savings in edges touched. Our next two methods add additional organizational overhead to prioritize which vertices to process first, further increasing the work efficiency.

Near-Far Pile: Workfront sweep treats all vertices within the work queue with equal priority. Rather than processing all vertices in the work queue, we can instead select a subset of vertices to process based on some scoring heuristic. A simple and effective heuristic is to select a splitting distance and then process only those vertices less than that distance. After all elements below this distance are processed, we update this split point by adding some incremental weight. We will define this incremental weight as delta (Δ). At any iteration, the vertices being processed will always fall into a range between $i\Delta$ and $(i+1)\Delta$; since we do not consider negative weights in our graphs, no visited vertex will be updated with a distance less than $i\Delta$.

Our Near-Far Pile method splits the work queue of vertices into two sets: one set (the *Near Set*) with distances less than $i\Delta$, to be processed next, and the second with distances outside that range (the *Far Pile*), deferred for later processing.

We first process work in the near set. We traverse all edges from the vertices in the near set and split the resulting vertices that have been updated into two piles. We append elements outside of our range to the end of the far pile, and begin the next iteration only processing the near pile. Once we run out of elements to process in the near set, we check the far pile for valid elements, compact all duplicates, and run another split with an updated range ($(i+1)\Delta$). In all



(a) Sample Graph (b) Bucketing Strategy Overview

Figure 2. Priority queue strategies for each of our three variants.

cases, our split primitives can be performed quite cheaply with a simple scan and modified compaction routine.

The key to this method's efficiency is that in many cases, unprocessed work in the Far Pile can be discarded as closer vertices are processed, therefore minimizing the number of times we must touch data in the Far Pile. When we perform our split we merely append data to the end of the pile, requiring no extra data movement. Thus the Far Pile data is only touched when we run out of work queue items. There are two costs to adding this functionality. First we are reducing the amount of available parallelism within the work queue. Second, we add the overhead of a split on every iteration.

Selecting a reasonable delta that separates our near pile from our far pile is important in the tradeoff between parallelism vs. work-saved. If we select a delta range that is too small, we serialize the algorithm into something much closer to Dijkstra's algorithm. Selecting too large of a delta transforms this method into our workfront sweep method, but with extra overhead and the additional work of updating bucket information for each element in our near pile. Meyer and Sanders [11] show that a value of $\Delta = \Theta(1/d)$, where d is the degree, gives a good tradeoff between work-efficiency and parallelism. We select a similar heuristic, $\Delta = cw/d$, where d is the average degree in the graph, w is the average edge weight, and c is the warp width (32 on our GPUs).

When we run out of items in our near set, we must first update a new split value then retrieve items from the far pile that are below this split point. We first check to see if any vertex in the far pile has already been updated and processed in the near pile (by checking its distance). We mark those vertices as invalid and compact away invalid and duplicate vertices as we split the far pile into a new near set and far pile. Figure 4 shows how the size of the data structures in Workfront Sweep and Near-Far change as we iterate over the graph.

Bucketing with Far Pile: The original CPU-based delta-stepping work did not use two but instead many finer-grained buckets, considering the smallest bucket on each iteration. Their implementation does not map efficiently onto a GPU: rather than using delta-stepping's dynamic bucketing scheme to push and pop vertices on and off buckets, we instead design an implementation better suited for bulk-synchronous programming.

In our implementation, we partition active elements into A buckets. At each iteration, reorganizing active elements into buckets is expensive, so we use relatively few buckets; $A = 10$ delivers the best performance in our implementation. We also use a far pile for vertices that fall beyond the last bucket in a similar way to the far pile in the previous method. Using more buckets limits the number of times we must process the far pile. Given that the number of elements in intermediate buckets is generally much fewer than the number of elements in the far pile, reducing the number of visits to the far pile should be advantageous.

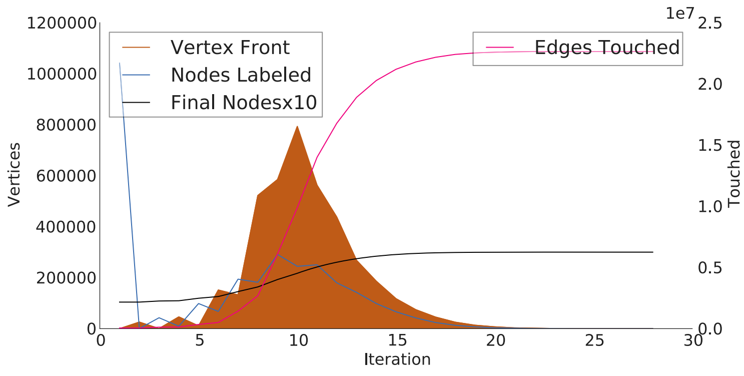
The ideal primitive to reorganize elements into buckets is a multisplit that splits on distance. Such a primitive has not been efficiently implemented on GPUs to date, so we instead use Thrust's radix sort [17] to sort into buckets by distance. We then process the smallest bucket. The resulting output set is split again into two sets: those inside our bucket range, and those outside. We sort the bucketable output set (vertices within our bucketable range) and perform a merge back into our bucket array. We append the other output set to the far pile. Once all buckets are processed, we compact the far pile, split the set into A new buckets and the far pile, and repeat the process.

This method most closely resembles delta stepping, and the amount of work (measured in edges touched) is less with this bucketing strategy than our other two methods. However, the reduction in work comes with much higher overhead per iteration.

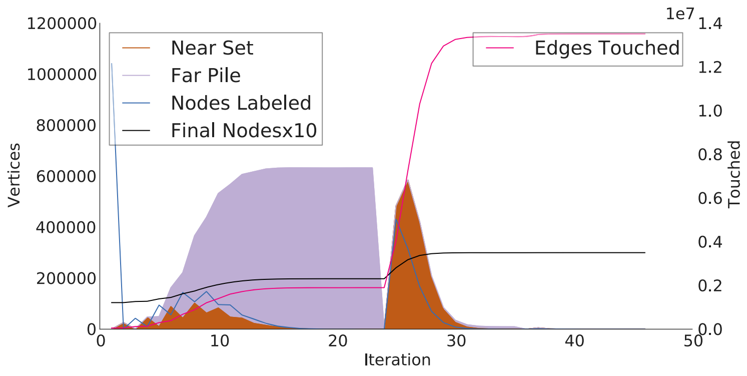
V. Timing Results

We test our implementations against eight graphs grouped into three types: (1) road networks, (2) meshes, and (3) scale-free graphs. Road networks tend to have high diameter and low degree, while scale-free graphs show the opposite behavior with relatively high degree and low diameter. Meshes lie somewhere in the middle of these extremes.

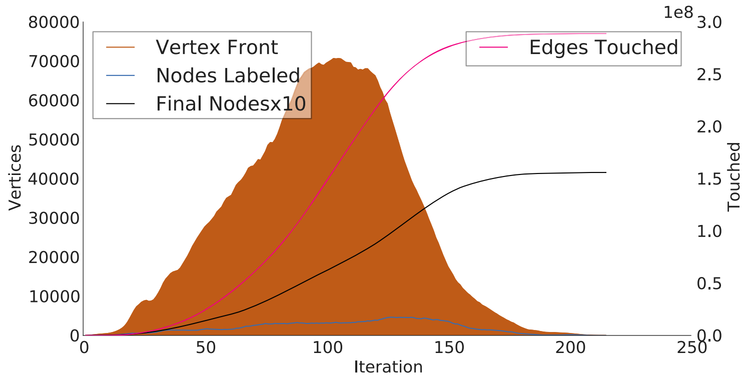
We have collected these datasets from a variety of online repositories [3], [18], [19] and summarize them in Table II. We compare our implementations against two serial Dijkstra implementations, one from the highly composable and easily usable Boost graph library utilizing relaxed heaps [7] and a second implementation that we developed using binary heaps. We also compare against Lonestar's Bellman-Ford implementation, and for some larger graphs where Lonestar's Bellman-Ford returns errors or crashes, against our own



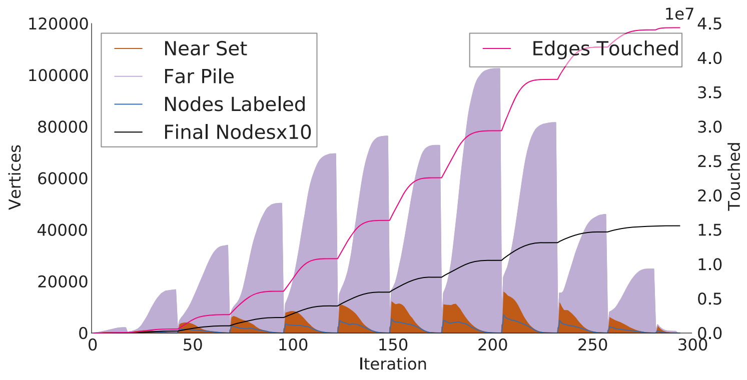
rmat Workfront Sweep: Utilizing the *Workfront Sweep* algorithm on a scale-free graph results in a early large spike in vertices considered. Due to the high connectivity, all shortest paths are found in a relatively short number of iterations. However, the number of vertices that must be considered during the peak are quite high (up to 800K), and the total number of edges touched is close to 1.2M.



rmat Near-Far: Using *Near-Far* instead leads to more iterations, but most candidate vertices are placed into the *Far Pile* early on. When work in the near set runs out, elements are loaded from the *Far Pile* and placed into the *Near* set. The amount of work saved (measured in edges touched) over *Workfront Sweep* is nearly 2x. Though it requires more iterations to solve, utilizing this method results in a modest 1.3x performance boost over *Workfront Sweep*.



msdoor Workfront Sweep: This dataset of a 3D model has less dense connectivity than the scale-free graph. In this case it takes more iterations (larger diameter) to compute the shortest paths problem. The amount of extra work being done by this method is quite high (around 300M edges touched for 4.15M vertices).



msdoor Near-Far: Switching to the *Near-Far* method greatly reduces the amount of work done. Instead of touching around 300M edges (as in *Workfront Sweep*), the analysis shows we only touch around 45M edges. On average we consider fewer than 20K vertices at every iteration, at the cost of more iterations. Though the *Far Pile* tends to get large, most of these candidates will be thrown out when everything in the *Near* set is considered. *Near-Far* solves the SSSP problem 1.74x faster than *Workfront Sweep* due to the amount of work saved.

Figure 4. As we traverse the graph, the sizes of the workfront (Workfront Sweep) and near set and far pile (Near-Far) grow and shrink. On both a mesh (msdoor) and a scale-free graph (rmat), the size of the near set is much smaller than the workfront. Also, every time we compact the far pile, we discard a large percentage of its vertices due to duplication or invalidation.

Bellman-Ford implementation. For our parallel implementations, we run our benchmarks on an NVIDIA GTX 680; for our serial implementations (Dijkstra), a 2-core 3.2 GHz Intel i5-650 CPU. Finally we also compare our Workfront Sweep method versus Ligra's Bellman-Ford reported results on a 10-CPU 40-core 2.4 GHz Intel E7-8870 Xeon workstation.

We show the timing results for each method on each dataset in Figure 5(a). These times are normalized to our binary-heap serial Dijkstra implementation, which in all cases was faster than Boost's. In Figure 5(b), we show the throughput performance in edges touched per second for each of these methods.

All of our work-saving methods yield speedups versus the traditional Bellman-Ford method, and versus Dijkstra on non-road-network graphs. On {road-network, meshes, scale-free graphs}, our Workfront sweep and Near-Far methods outperform Bellman-Ford in runtime by factors of {1.3–14, 3.9–19.85, 34.7–343}, and Dijkstra by {0.24–0.6, 2–29.4, 22–64}. Shun and Blelloch solve SSSP on a 17M-vertex, 90M-edge rmat graph with 40 CPU cores, achieving a throughput of 130 MTEPS [16]; their SSSP implementation, unlike ours, handles negative weights. Similarly Nguyen et al. report slight speedups on a similar rmat graph achieving a maximum throughput of 141 MTEPS [21] using their library for graph algorithms. We solve an rmat matrix of similar size and parameters utilizing Near-Far on a single GPU with a throughput of 350 MTEPS (2.4–2.7x increase in performance). This increase in performance illustrates the potential benefits of a bulk-synchronous implementation of SSSP and the benefits of Near-Far as a GPU-friendly work-saving strategy.

Our speedup is primarily due to our effective work-saving strategies and intelligent graph traversal. Though we would expect Bellman-Ford to behave reasonably well on low-diameter graphs, Bellman-Ford is in practice much slower on these graphs. This is mostly due to heavy load imbalance in the size of the adjacency lists. For example, flickr has an average edge degree of 11, but one vertex in this dataset is connected to over 3 million other vertices. In this case, having one thread assigned to traverse that list is wildly inefficient.

For high-diameter road networks, without preprocessing, our implementations are unable to outperform an efficient serial implementation. This is expected behavior; the lack of parallelism in processing these networks means that a good serial implementation *should* outperform a good parallel one. Also, because computing SSSP on these graphs requires many steps, our implementation suffers from the high aggregate cost of per-iteration overhead.

While our Bucketing method is faster than Dijkstra and Bellman-Ford on higher-degree graphs, it is slower than our other two strategies. Since bucketing requires a sort on the incoming vertices, and a merge into the workfront, it requires by far the most overhead at every iteration. We note an interesting dilemma when we change delta. If we decrease delta, we save work overall, but at the cost of more iterations in which we must perform an expensive reorganization of the buckets. If we instead increase delta to reduce iteration count, the increase in work rises to an amount similar to workfront expansion, removing any benefit bucketing would offer. Therefore, bucketing is the least appealing of the methods we propose in the absence of faster primitives than sort and merge with their high overhead.

Method Name	Work Saved vs. B-F	Parallelism	Iterations vs. B-F
Workfront	9.6x	87,998	1.05x
Near-Far Pile	260x	17,138	1.66x
Bucketing	353x	8,135	2.27x

Table III
SUMMARY IN REDUCTION OF WORK, AVERAGE PARALLELISM AND INCREASE IN ITERATION (MULTIPLIER) FOR EACH OF OUR METHODS VERSUS TRADITIONAL BELLMAN-FORD.

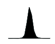
VI. ANALYSIS

The effectiveness of our work-saving methods and the amount of parallelism we are able to extract affect the *step efficiency* of our methods (Figure 6(b)). Bellman-Ford will of course have the best step efficiency, because all nodes are processed at every iteration, allowing for the fastest propagation from the source to other nodes. Dijkstra has the worst step efficiency since it can only process one vertex at any given iteration. We aim to build methods that have enough parallelism to fully occupy the GPU, work efficiency much closer to Dijkstra's method than Bellman-Ford, and finally step efficiency much closer to Bellman-Ford.

Figure 6(a) measures the parallelism-to-work-efficiency tradeoff for each method on each dataset. For high-diameter graphs, Bellman-Ford does anywhere from 100–1000x more work than Dijkstra's method. Even for lower-diameter scale-free graphs, Bellman-Ford does over an order of magnitude of additional work. Our simplest work-saving method (Workfront Sweep) is able to reduce this amount of unnecessary work by around 10x, at the cost of some parallelism and the overhead of managing a vertex queue. However, the amount of parallelism is still more than sufficient for large graphs to fill modern GPUs.

Our Near-Far method further reduces the amount of work done (anywhere from 2–10x over Workfront Sweep), still has enough parallelism (for all but our road-network graphs), and doesn't increase the iteration count by a substantial amount. Table III summarizes these tradeoffs. Figure 7 illustrates the work vs. parallelism tradeoff when modifying delta. Too small of a delta leads to a lack of parallelism, while too large of a delta leads to more unnecessary work. Our heuristically selected delta is shown by the vertical line.

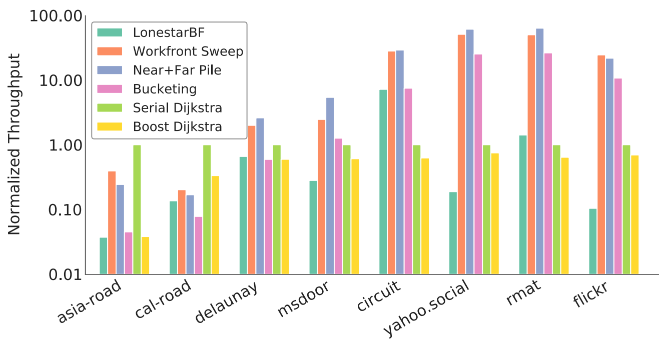
Road-network datasets tend to have a low parallelism-to-work-efficiency tradeoff. Though Near-Far is nearly as

Graph Name	Description	Vertices	Edges	Avg. Degree	Diameter	Graph Type	Weight Type	BFS Front
road-asia [19]	Road network in Asia	11M	25M	2.27	38576	Road Network	Distances	
road-cal [19]	California road network	1.9M	4.6M	2.47	2575	Road Network	Distances	
delaunay [3]	Three-dimensional mesh	1M	6.3M	6	380	Regular Mesh	Uniform Random	
msdoor [18]	Three-dimensional mesh	4.15M	20.65M	4.98	167	Regular Mesh	Uniform Random	
circuit [18]	Circuit simulation problem	3.5M	19M	5.43	135	Freescale	DC Analysis	
yahoo-social* [20]	Links between users and friends	1.2M	4M	3.14	16	Social Graph	Uniform Random	
rmat [3]	Synthetically created graph	3M	20M	6.66	15	Power Law	Uniform Random	
flickr* [18]	Social graph	820K	9.8M	11.95	12	Power Law	Uniform Random	

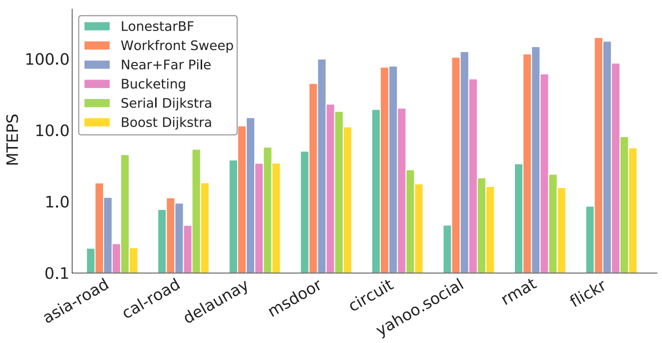
* Indicates this dataset either crashed or failed to produce the correct answer on the Lonestar GPU Bellman-Ford implementation. Therefore we utilized our own Bellman-Ford implementation to compare against these datasets.

Table II

DATASETS FOR BENCHMARKING, INCLUDING BOTH REAL AND SYNTHETIC DATASETS, SORTED BY DECREASING DIAMETER.

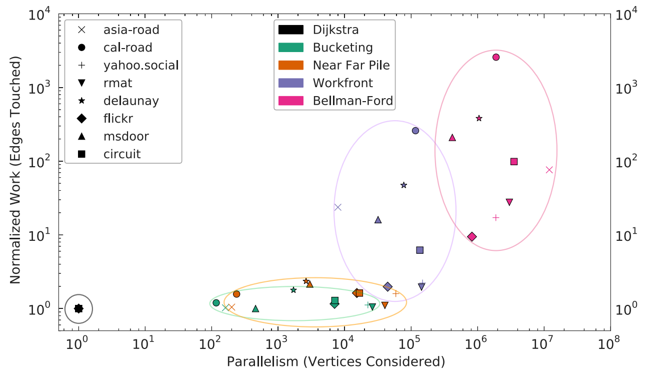


(a) NORMALIZED RUNTIME PERFORMANCE

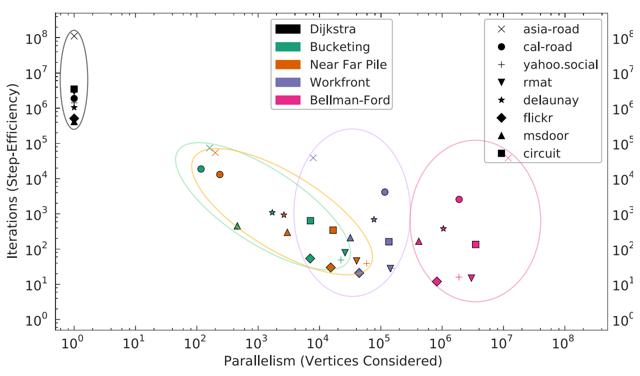


(b) EDGE THROUGHPUT PERFORMANCE

Figure 5. We benchmark our three SSSP variants against Dijkstra's method (our implementation and in Boost) and Bellman-Ford. We show both normalized throughput to serial Dijkstra (left, higher is faster) and raw throughput measured in millions of edges touched per second (MTEPS). For a fair comparison, the number of edges per problem is defined as the number of edges required by a Dijkstra traversal.

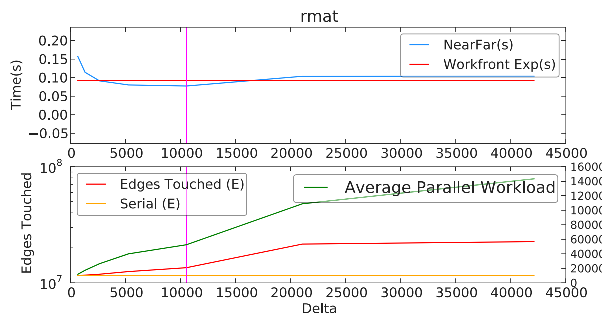


(a) WORK DONE VS. PARALLELISM

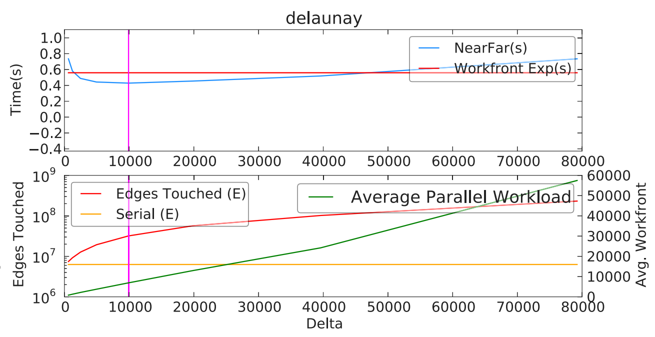


(b) ITERATIONS VS. WORKFRONT

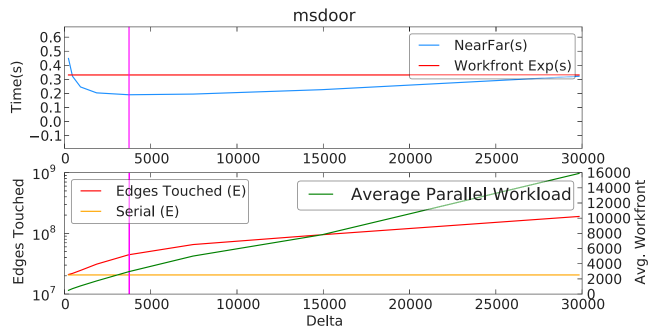
Figure 6. Four dimensions affect performance for our methods: (1) work efficiency; (2) parallelism; (3) iteration count; and (4) overhead per iteration. We show the tradeoffs that our methods take within this domain on each of our datasets. The left graph shows the tradeoff between degree of parallelism (x-axis) and work normalized to the serial Dijkstra's algorithm (y-axis). Each oval indicates the range of these tradeoffs over all datasets for one method. In general, the best parallel algorithms are at the bottom right, exhibiting a high degree of parallelism with little additional work over the serial case, and the methods we describe in this paper, particularly Near-Far, exploit ample parallelism with little additional work over the serial case. The right graph shows the tradeoff between degree of parallelism (x-axis) and iteration count (y-axis).



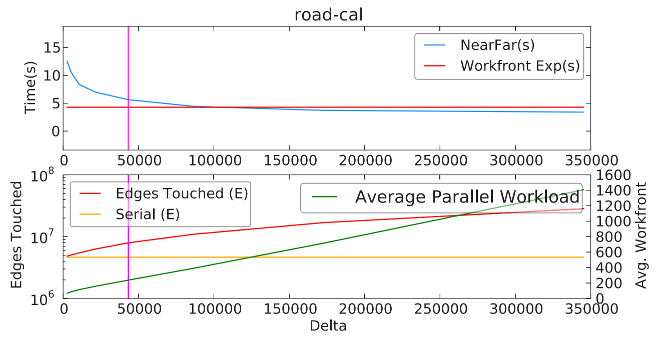
(a) RMAT MEASUREMENTS



(b) DELAUNAY MESH MEASUREMENTS



(c) MSDOOR MEASUREMENTS



(d) USA-CAL ROAD MEASUREMENTS

Figure 7. We show more detailed analysis on a subset of our datasets as we change delta. Choosing a small delta leads to a lower amount of work, but serializes the algorithm. Choosing a large delta transforms this method into Workfront sweep, but with unnecessary overhead. Finding the sweet spot between these two extremes leads to a method with both adequate parallelism and significantly lower work than Workfront sweep. The vertical line in the graphs indicates the delta selected by our simple heuristic.

work-efficient as a serial implementation, the amount of parallelism at every iteration is extremely low (100–200 vertices). This amount of parallelism is too low to saturate the GPU, and therefore is unable to outperform an efficient serial implementation. However, for any graph in which the vertex front will grow to a substantial size, Near-Far does a good job of reducing unnecessary work while maintaining enough parallelism to saturate the machine.

Though our bucketing method saves work when compared to the Near-Far method, this amount is not significantly higher than the improvements from Workfront Sweep to Near-Far. Together with the higher iteration count, lower parallelism, and higher overhead per iteration, we see lower performance from this method compared to our other implementations.

VII. CONCLUSION AND DISCUSSION

In this paper we have demonstrated the effectiveness of work-saving SSSP methods on GPUs for a variety of datasets. The fine-grained primitives we have used to reorganize data have proved effective enough to achieve impressive speedups. Furthermore, our methods find a good balance between parallelism, step-efficiency and work-efficiency.

First, we would like to discuss some of the lessons leaned from our lowest-performing method, bucketing. Bucketing is inspired by the successful delta-stepping method for coarse-grained parallel CPUs [11], but when reorganizing data each iteration on the more fine-grained GPUs, we see that the overhead of this reorganization is significant: on average, with our bucketing implementation, the reorganizational overhead takes 82% of the runtime. Thus it is critical to choose reorganization methods that leverage high-performance, fine-grained, bulk-synchronous primitives rather than high-overhead dynamic buckets. The missing primitive on GPUs is a high-performance *multisplit* that separates primitives based on key value (bucket id); in our implementation, we instead use a sort; in the absence of a more efficient multisplit, we recommend utilizing our Near-Far work-saving strategy for most graphs.

One fruitful future direction of work is a dynamic method that switches between Workfront Sweep and Near-Far. Since at the beginning of problems there is often a lack of parallelism, we would begin with a Workfront Sweep method until the vertex queue becomes large enough to saturate the GPU. Now that there is sufficient parallelism, we would switch to our Near-Far Pile technique where work efficiency is more important. Finally, after most of the graph has been explored, and we return to a lack of sufficient parallelism, we would switch back again to Workfront Sweep to finish off the problem. This hybrid method would share some similarities with BFS traversal work by Hou et al. [22]. In their work, they begin with a BFS approach, while there is limited parallelism. As the frontier expands and their is adequate parallelism, they switch to a DFS for better efficiency.

Another possible improvement targets road network graphs in particular. The largest performance bottleneck for these low-degree graphs was a lack of available parallelism. Though our methods require no pre-processing, we could combine our work-saving method with a PHAST-like pre-processing step to identify more vertices from which to start our SSSP sweeps. These vertices would then be loaded into our vertex front with accurate distance labels, boosting the amount of available parallelism in our algorithm.

Acknowledgments

Thanks to Julian Shun for thoughtful comments on the paper. We would like to acknowledge Yangzihao Wang for feedback in algorithmic development, and early drafts of this paper. We appreciate the funding support of the Department of Defense's XDATA program, the National Science Foundation (grant # CCF-1017399), Sandia LDRD award # 13-0144, and UC Lab Fees Research Program Award # 12-LR-238449.

References

[1] A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Reach for A*: Efficient point-to-point shortest path algorithms." in *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, Jan. 2006, pp. 129–143.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, Sep. 2001.

[3] "The Graph 500 list," http://www.graph500.org/, Jul. 2013.

[4] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, Nov. 2012, pp. 141–151.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[6] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[7] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Dec. 2001.

[8] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "PHAST: Hardware-accelerated shortest path trees," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 940–952, Sep. 2010.

[9] R. Geisberger, P. Sanders, and D. Schultes, "Better approximation of betweenness centrality," in *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, Jan. 2008, pp. 90–100.

[10] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 65–76.

[11] U. Meyer and P. Sanders, "Δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, Oct. 2003, 1998 European Symposium on Algorithms.

[12] D. L. Baggio, "GPGPU based image segmentation livewire algorithm implementation," Master's thesis, Technological Institute of Aeronautics, São José dos Campos, Brazil, 2007.

[13] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, Nov. 2009, pp. 18:1–18:11.

[14] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, Feb. 2012, pp. 117–128.

[15] S. Baxter, "Modern GPU library," http://www.moderngpu.com/, 2013.

[16] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, Feb. 2013, pp. 135–146.

[17] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU Computing Gems*, Oct. 2011, vol. 2, ch. 26, pp. 359–371.

[18] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[19] "DIMACS implementation challenge - shortest paths," http://www.dis.uniroma1.it/challenge9/download.shtml, Jul. 2013.

[20] "Yahoo labs dataset selections," http://webscope.sandbox.yahoo.com/, Jul. 2013.

[21] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Nov. 2013, pp. 456–471.

[22] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha, "Memory-scalable GPU spatial hierarchy construction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 4, pp. 466–474, Apr. 2011.