A Fast Work-Efficient SSSP Algorithm for GPUs

Kai Wang
Department of Computer Science
The University of Texas at Austin
USA
kaiwang@cs.utexas.edu

Don Fussell
Department of Computer Science
The University of Texas at Austin
USA
fussell@cs.utexas.edu

Calvin Lin
Department of Computer Science
The University of Texas at Austin
USA
lin@cs.utexas.edu

Abstract

This paper presents a new Single Source Shortest Path (SSSP) algorithm for GPUs. Our key advancement is an improved work scheduler, which is central to the performance of SSSP algorithms. Previous GPU solutions for SSSP use simple work schedulers that can be implemented efficiently on GPUs but that produce low quality schedules. Such solutions yield poor work efficiency and can underutilize the hardware due to a lack of parallelism. Our solution introduces a more sophisticated work scheduler—based on a novel highly parallel approximate priority queue—that produces high quality schedules while being efficiently implementable on GPUs.

To evaluate our solution, we use 226 graph inputs from the Lonestar 4.0 benchmark suite and the SuiteSparse Matrix Collection, and we find that our solution outperforms the previous state-of-the-art solution by an average of 2.9×, showing that an efficient work scheduling mechanism can be implemented on GPUs without sacrificing schedule quality.

While this paper focuses on the SSSP problem, it has broader implications for the use of GPUs, illustrating that seemingly ill-suited data structures, such as priority queues, *can* be efficiently implemented for GPUs if we use the proper software structure.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**; **Massively parallel algorithms**; **Concurrent algorithms**.

Keywords: SSSP, Worklists, GPUs

1 Introduction

The Single-Source Shortest Path (SSSP) problem, in which the goal is to find the shortest path from a given source vertex to all vertices, is important for two reasons. First, it is widely used with many applications. Second, it is algorithmically interesting because the worklist that is central to SSSP algorithms presents a conundrum for GPU programmers: GPUs operate best on data parallel computations that exhibit regularity in both data access and control flow, yet worklists would appear to have neither of these properties. Thus, while the best performing SSSP algorithms currently run on GPUs, it has been difficult to realize the advantages of worklist processing on GPUs, suggesting that further improvements in GPU performance on problems that traditionally use worklists are possible.

The importance of the worklist to SSSP algorithms can be seen by noting the difference between the two classic SSSP algorithms. Dijkstra's algorithm [9] uses an ordered worklist, i.e., a priority queue, yielding a solution that is optimal in work efficiency but that admits little parallelism. By contrast, the Bellman-Ford algorithm's [2] use of an unordered worklist provides maximum parallelism at the cost of redundant work. For parallel platforms, the delta-stepping algorithm from Meyers, et al. [15] provides a compromise between these two extremes: The idea is to use a coarse-grained priority queue that places work items into one of multiple buckets, with vertices in the same bucket sharing the same priority. At its finest granularity, the algorithm collapses to Dijkstra's algorithm. At its coarsest granularity, the algorithm collapses to Bellman-Ford. For CPU-based multicores, the best granularity resides somewhere in the middle.

To date, the best SSSP solution for GPUs [4] uses the Near-Far algorithm [7], an adaptation of delta-stepping that makes several GPU-friendly simplifications to the worklist. First, it uses just two buckets, known as Near and Far, which can be implemented using pre-allocated arrays, thereby avoiding the need to perform dynamic memory management. Second, it uses the Bulk Synchronous Parallel (BSP) model in which an algorithm proceeds in a series of supersteps separated by barrier synchronization. This model allows each bucket to be double buffered, with writes made to one buffer and reads made to a second buffer, greatly simplifying synchronization.

Unfortunately, Near-Far has three deficiencies. First, because it uses just two buckets, Near-Far provides an extremely coarse-grained approximation of a priority queue, leading to poor work efficiency. Second, its use of barrier synchronization and double buffering severely limits parallelism, particularly for high-diameter graphs. Third, the granularity

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441605

of the priority queue—that is, the range of priorities represented by each bucket, known as the *Delta* value—is chosen using a simple offline method that does not adequately capture important characteristics of the input graph and its relation to available parallelism.

In this paper, we present ADDS (Asynchronous Dynamic Delta-Stepping), a novel formulation of delta-stepping that introduces an efficient worklist for GPUs to address the three deficiencies of Near-Far:

- It uses multiple buckets, which improves work efficiency.
- Instead of using the BSP model, it operates asynchronously, which avoids barrier synchronization and increases parallelism.
- It uses a dynamically selected Delta value that is chosen based on runtime information.

One key to our solution is the introduction of a Manager Thread Block (MTB), which plays a role in all three aspects: (1) It executes a new custom dynamic memory allocator, which supports the use of multiple buckets; (2) it coordinates the accesses of multiple worker threads so that they do not conflict, thus providing the functionality of multiple readers and multiple writers (MRMW) with the implementation efficiency of a solution that uses a single reader and multiple writers (SRMW); (3) it gathers dynamic information that is used to periodically select a good Delta value. In short, we have identified the bottlenecks in a GPU implementation of delta-stepping, and we have found ways to transform these bottlenecks into data parallel computations.

This paper makes the following contributions:

- We present the ADDS algorithm, a formulation of delta-stepping for GPUs that addresses three major limitations of the Near-Far algorithm. The key advancement is a sophisticated coarse-grained worklist that runs efficiently on GPUs.
- We extensively evaluate our new algorithm on a set of 226 graphs from the Lonestar 4.0 benchmark suite [11] and SuiteSparse Matrix Collection [8]. On an NVIDIA RTX 2080 ti GPU, we find that ADDS outperforms the best Near-Far implementation by an average of 2.9×.
- More broadly, we demonstrate that while previous work has had to choose between algorithmic efficiency and fitness for GPUs, we have shown that we can achieve both. Thus, we have shown that GPU programmers *can* use sophisticated data structures, such as coarse-grained priority queues, if they design their data structures carefully.

This paper is organized as follows. Section 2 first places our work in the context of prior work, and Section 3 then provides background information that is useful for understanding our solution. We describe in Section 4 three important design considerations for implementing work schedulers on GPUs, before presenting our new algorithm in Section 5. We empirically evaluate our solution in Section 6 and then conclude in Section 7.

2 Related Work

Dijkstra's algorithm has been parallelized [6, 14, 19] by processing vertices with the same priority (the same distance from the source node) in parallel. However, such algorithms typically admit much less parallelism than is available on GPUs.

The Bellman-Ford algorithm is much more straightforward to parallelize, since it does not require a priority queue. Many GPU implementations have been proposed [4, 5, 10, 13, 21], but the processing of vertices in arbitrary order leads to redundant work.

Meyer et al's [15] delta-stepping algorithm was designed from a theoretical perspective and has since been adapted to GPUs [1, 3, 7, 22, 23] by simplifying the design of the worklist in exchange for inferior work schedules, which reduces work efficiency and limits parallelism. Our ADDS algorithm is a GPU adaptation of delta-stepping that uses a more sophisticated work scheduler to improve work efficiency and parallelism.

3 Background

This section (1) briefly describes the general structure of SSSP algorithms, (2) explains how the implementation of its worklist impacts work efficiency, and (3) briefly explains the delta-stepping algorithm.

SSSP algorithms compute the shortest distance from a source vertex to every other vertex in a directed graph with non-negative edge weights. All SSSP algorithms associate with each vertex the current shortest distance to that vertex from the source; initially set to ∞, this current shortest distance is refined until the shortest distance is found. When a vertex' shortest distance is updated, the vertex' neighboring nodes are processed to propagate updated information across the graph. A *worklist* is used to store the IDs of outstanding vertices to be processed.

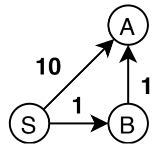


Figure 1. Sample graph with source node S.

3.1 Work Scheduling

To understand the importance of a good work schedule, we first define the **work efficiency** of an SSSP algorithm to be the inverse of the total number of vertices processed.

We can then consider the input graph shown in Figure 1, where an unordered worklist can lead to redundant work, while an ordered worklist, ie, a priority queue, does not. In particular, if we process vertices in priority order based on increasing distance from the source S, we would process B first and then A, so we would visit each vertex once. If on the other hand we processed A before B, then the current shortest distance to A will have to be updated after the shortest distance to B becomes known, so A will be visited twice. For high diameter graphs, we find that Dijkstra's algorithm, which uses a priority queue, can be 1000× more efficient than Bellman-Ford, which uses an unordered queue. As the diameter of the graph decreases, the significance of ordering diminishes, with power law graphs seeing moderate to no benefit from ordering. For example, a priority queue improves the work efficiency by only 2× for the *rmat22* graph.

3.2 Delta-Stepping

The delta-stepping algorithm [15] uses a coarse-grained priority queue that admits more parallelism than Dijkstra's while providing better work efficiency than Bellman-Ford. This coarse-grained priority queue consists of multiple buckets, where all vertices in a given bucket are given the same priority (see Figure 2), even though their priority values may differ by up to a constant called Δ. Essentially, this multiple-bucket data structure sorts the outstanding vertices at a coarse granularity. To schedule work, vertices in the first bucket can be processed in parallel, and when the bucket becomes empty, the next bucket is processed, enforcing a coarse-grained priority order.

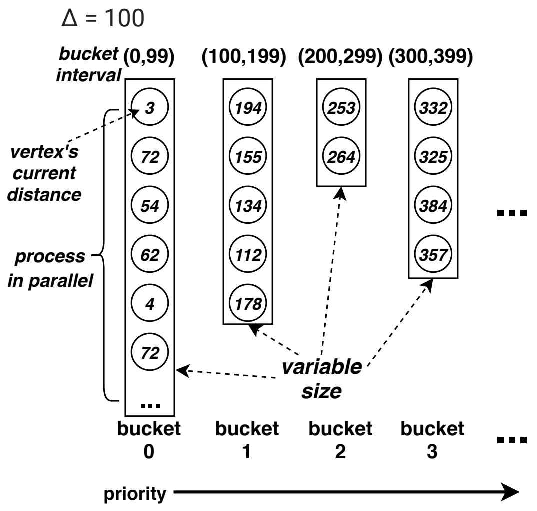


Figure 2. Delta-Stepping's Work Scheduling Data Structure.

4 Design Considerations

This section describes three important design considerations for implementing delta-scheduling on GPUs.

We first observe that the three SSSP algorithms that we have discussed—Dijkstra's, Bellman-Ford, and delta-stepping—differ primarily in the way that they schedule work. In the context of GPUs, work scheduling is particularly important and challenging because it profoundly impacts performance in two ways. First, the data structure itself must be sufficiently scalable to support tens of thousands of hardware threads. Second, the resulting schedule needs to provide sufficient parallelism to utilize the abundant hardware threads, while at the same time managing the competing concern of maintaining good work efficiency, i.e., avoiding redundant work.

4.1 Design Consideration 1: Memory Management

The first consideration is memory management. Fixed-sized arrays avoid the cost of memory management, but they waste memory because b buckets consume $b \times |E|$ memory, where $|E|$ is the number of graph edges. Thus, as b grows, fixed-sized arrays become impractical for large graphs. Near-Far sets $b = 2$, yielding an extremely coarse-grained priority queue that severely degrades work efficiency.

Thus, we seek a worklist data structure that can efficiently grow and shrink the sizes of individual buckets as the program executes, without limiting the number of buckets to some small constant.

4.2 Design Consideration 2: Synchronization

The second issue is synchronization. If multiple threads can read from and write to the same buckets, then we have MWMR access, which in general is not scalable on GPUs. Thus, many GPU SSSP algorithms, such as Near-Far, employ double-buffering, so that in any iteration of the algorithm, threads read from one buffer and write to another, removing the need for reader-writer synchronization. However, double buffering reduces concurrency, since newly generated work items can only be read in the next iteration, even if idle threads are available to perform the read. Double buffering is particularly harmful for high diameter graphs, where the execution is forced into many tiny iterations; e.g. for the the road.USA graph, the average work count per iteration is only 800, while a RTX 2080 GPU has 68K hardware threads. Double-buffering also doubles the memory requirements of the relevant buffers.

Thus, the challenge is to provide parallel access to buckets without resorting to synchronous double-buffering.

4.3 Design Consideration 3: Granularity

The third design consideration involves the granularity of the priority queue, which is guided by the value of the Δ parameter. A smaller Δ value produces finer-grained buckets that improve work efficiency but reduce parallelism. The optimal choice depends on characteristics of both the input graph and the underlying hardware, since it depends on the weights and connectivity of the graph and the hardware's available parallelism.

Near-Far uses a simple heuristic [7] that often picks a value that is far from optimal. The value is chosen statically

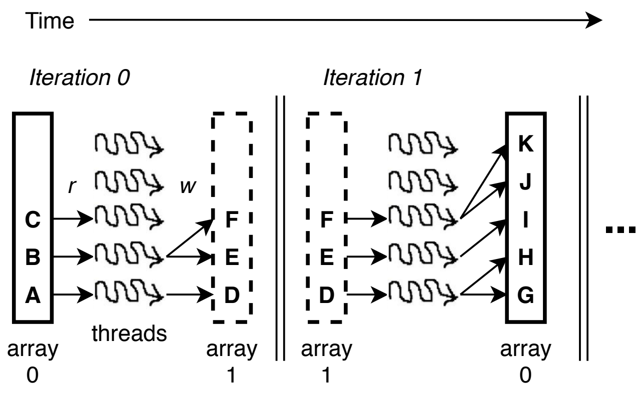


Figure 3. Implementing a List Using Double-Buffering.

based on the average weight (W) and the average degree (D) of the graph: Δ=$C\times(W/D)$, where C is a constant for all graphs.

To show the limitations of this heuristic, we identify the optimal value of C for each input graph. Figure 4 illustrates two points. First, the choice of Δ has a significant impact on performance. Second, the optimal values of C for the two input graphs are far away from each other; so it is impossible to pick a constant C that is optimal for all graphs.

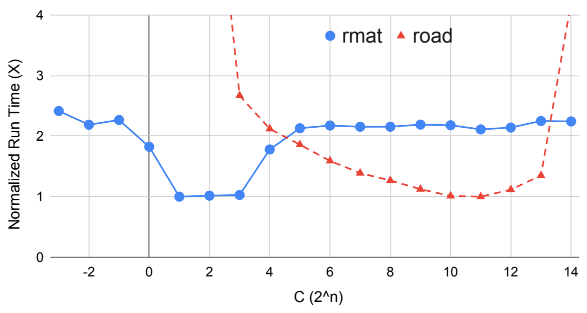


Figure 4. Execution Time Against the Constant C for Two Graphs. Execution time is normalized to the minimum in the series; labels of the x-axis are powers of 2.

Thus, we conclude that we need to incorporate runtime information to select the best value of Δ.

5 Our Solution

Our solution is a GPU adaptation of delta-stepping that introduces a new work scheduling mechanism that substantially improves work efficiency and parallelism.

There are three keys to our solution. First, we develop a highly parallel approximate priority queue with dynamically-sized buckets and customized memory management, which allows our solution to use many buckets instead of just two;

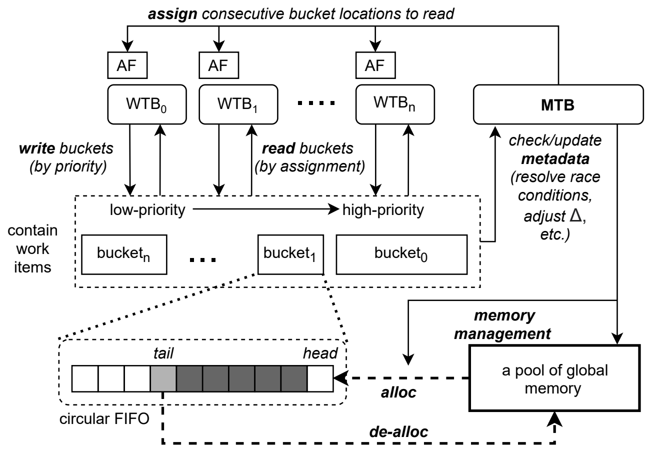


Figure 5. Overview of Our Solution

this finer-grained priority queue improves work efficiency. Second, instead of using double buffering, our bucket data structure efficiently supports asynchronous concurrent access by large numbers of worker threads, which significantly improves parallelism. Third, our scheduler gathers run-time information to dynamically identify more suitable values of Δ.

In the rest of this section, we first discuss the key features of our *decoupled* approach that allows thousands of worker threads to access a common bucket without ruinous contention for shared data. We then explain how ADDS manages work allocation across multiple buckets. Finally, we describe our dynamic scheme for determining the appropriate Δ step size to balance parallelism and work efficiency.

5.1 Basic Operation

Our main data structure is an ordered circular work queue of **buckets**, whose sizes are determined dynamically to avoid wasting memory as the distribution of priorities changes over time. As we describe in Section 5.3, this work queue is supported by a fast custom memory manager that exploits the FIFO nature of the buckets.

The key to our solution is a delegation strategy in which worker thread blocks **(WTBs)** process vertices and assign *work items* to buckets according to their priority, but they do not read items directly from buckets. Instead, a single manager thread block **(MTB)** reads the buckets and assigns work items to worker threads. Thus, while the overall work queue has MRMW functionality, the metadata managed by the MTB ensures that the many WTBs do not write to the same memory locations. And the synchronization typically associated with MRMW access to the work queue is replaced by a simpler SRMW synchronization to the work queue metadata.

Our use of delegation allows the MTB to read many data items in parallel with a single read operation. In particular, an MTB thread operates on N-word *segments* instead of individual work items, and even greater efficiency is achieved because all MTB threads in a warp perform read operations in that same single memory access. Thus, instead of having all threads read and write directly to buckets in an MRMW fashion, our solution amortizes the overhead of metadata access across a large number of read operations. Moreover, since our solution is SRMW, it does not need to resolve conflicts among multiple readers (e.g. collisions on the same bucket location), which simplifies the design.

Once available work items are known, the MTB assigns this work to WTBs using a dedicated assignment flag (AF) for each WTB (see Figure 5). In addition to indicating whether the WTB is currently idle, an AF also contains the location and size of the array of work items that is assigned to the WTB when it is not idle. Each idle WTB polls its respective AF in scratchpad memory and thus receives work from the MTB without contention with other WTBs.

5.2 SRMW Queue Access Management

We now describe details of our queue management scheme.

Logically, each bucket is an array managed as a circular FIFO queue. WTB threads can add work directly to a bucket; hence there are multiple writers. Work is added to these buckets in a lightweight manner by maintaining in the bucket's metadata a reservation pointer **(resv_ptr)** that is accessed atomically by the WTBs. The resv_ptr is atomically incremented, and each accessing WTB is returned a unique index into the bucket array where it can place new work items without contention. Likewise, the MTB maintains a read pointer **(read_ptr)** to indicate the location in the queue at which it should start reading work items.

To avoid a race condition, we must ensure that as the MTB performs queue management and work distribution, it does not attempt to access work from locations that have not yet had work items written to them. As mentioned above, each MTB thread operates on an N-word segment of a bucket. Each segment is associated with a write completed counter **(WCC),** which is initially 0. When a WTB thread writes a work item into the location at its unique bucket index, it executes a memory fence to ensure that the item is fully written and then atomically increments the WCC for the segment into which it is writing.

To read work off the queue, the MTB checks each segment's WCC. If its value is N, then that entire segment contains assignable work. If the WCC value is less than N, then the beginning index of the segment is added to the WCC, and to guarantee that the value of resv_ptr is not stale, the result is compared to resv_ptr after performing a memory fence operation. If these values are equal, then the segment up to resv_ptr is full, and all locations from read_ptr to resv_ptr are known to be fully written. If they are not equal, then no data from the current segment can be considered written since nothing is known about the order in which WTB threads write their locations, so the final known written location is the end of the previous segment. Thus, the MTB computes the bounds of a set of known written locations from read_ptr to the determined upper bound, and the read_ptr is updated to the new bound.

5.3 Memory Management

Because the distribution of work item priorities varies over time, we need an efficient dynamic memory management scheme that allows the bucket sizes to change over time. Recall that each bucket is treated as an array with a 32 bit index implementing a circular queue of work items. To allow a bucket to grow and shrink dynamically, memory for a bucket is allocated in blocks of 64K 32 bit words. An array of pointers to allocated blocks is maintained for each bucket. The high order 16 bits of each 32 bit index are treated as an index into the pointer array, and the lower order 16 bits are an offset into the particular block. Thus, compared to a simple static array, a data item access requires an additional level of indirection. This overhead can be substantial when accessing global memory, but it can be ameliorated by keeping direct-mapped translation caches for each WTB and for the MTB in scratchpad, where the high order 16 bits of an index are treated as a tag for the cached block at that index.

Our system performs its own memory management for buckets, using a large block of pre-allocated GPU memory and keeping as much metadata in scratchpad memory as possible. Because the memory blocks are always part of a FIFO queue, they are read and written in a monotonically increasing order, so management is much simpler than for a general purpose memory allocator. All memory management is performed by the MTB, freeing WTBs from dealing with this task.

5.4 Managing the Ordered Work Queue

Our work queue consists of a fixed number of 32 buckets, managed as a circular priority queue with bucket priorities that increase from the tail (initially index 0) to the head (initially index 31).

When the highest priority bucket becomes empty, the MTB increments the head and tail pointers so that the empty bucket can be reused. To determine when these pointers can be incremented, our solution maintains a completed work counter **(CWC)** for each block. When a WTB completes k work items that have been assigned to it, the WTB atomically increments the block's CWC by k. When the CWC matches the resv_ptr for the block, again following a memory fence to be sure the value of resv_ptr is not stale, all work is done and the pointers may be incremented.

It might seem possible to simply detect cases where the head bucket has no work item left, but with such a scheme, the WTBs may still be performing tasks that spawn additional work that belongs in the head bucket. Should that bucket be deallocated (i.e. the head pointer is incremented), that new work will be placed into the next highest priority bucket. We have seen this situation cause continuous cramming of work into ever fewer buckets, eventually rendering the entire priority scheme ineffective.

Performance can be optimized by allowing the MTB to allocate work from multiple high priority buckets, instead of just the head bucket. This optimization has two benefits: (1) It avoids situations in which small amounts of remaining work in a nearly-empty head bucket limit the concurrency of active worker threads, and (2) it gives the MTB control over the number of buckets from which allocation can be made, which along with delta adjustment, allows the MTB to control the tradeoff between the amount of concurrency and work efficiency. When assigning items, higher priority buckets are considered first and lower priority buckets are considered only if there are idle WTBs after all higher priority work has been assigned.

Our algorithm terminates when it detects for two consecutive sweeps of the work queue that no work has been assigned to any bucket. Two sweeps are needed to ensure that all work in progress has been completed.

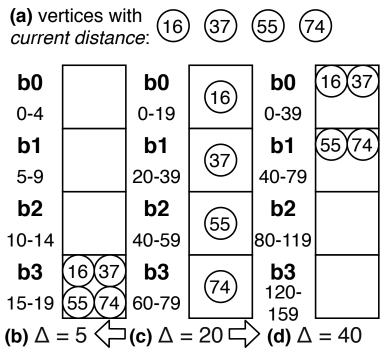


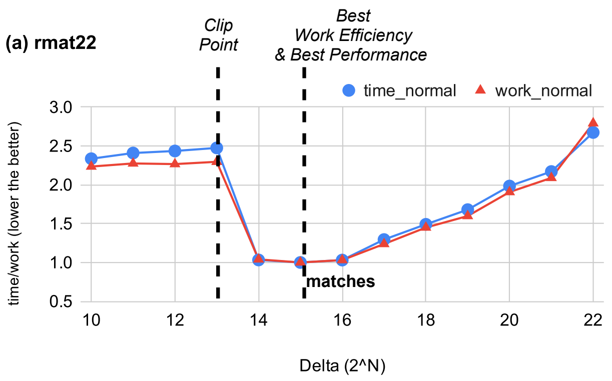
Figure 6. How Δ Affects Work Efficiency and Concurrency. Adding 4 vertices (a) to 4 buckets under 3 scenarios: Δ=20 (c) produces the best work efficiency; Δ=40 (d) improves parallelism; Δ=5 (b) clips vertices to the last bucket.

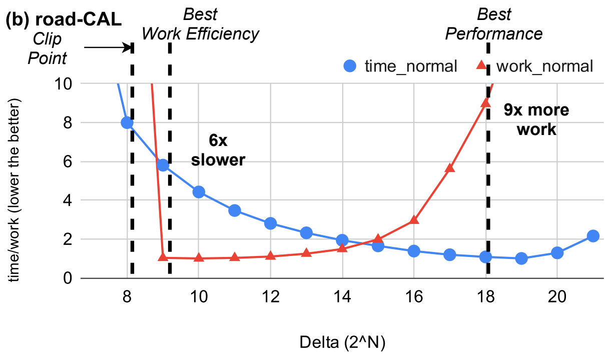
5.5 Dynamically Setting the Δ Value

The value of Δ can significantly impact performance. In this section, we first explains how the value of Δ impacts work efficiency and parallelism. We then introduce a mechanism for dynamically setting the Δ value based on run-time information.

Using a simple example where 4 vertices are added to 4 buckets under 3 different Δ values, Figure 6 shows how the value of Δ affects performance. If Δ=20 (case (c)), we get a precise ordering of vertices and optimal work efficiency. However, this Δ value might not provide sufficient parallelism to fully utilize the hardware, so increasing Δ to 40 (case (d)) increases the number of of work items in each bucket and therefore increases parallelism.

The interesting case occurs when Δ is decreased to 5 (case (b)). We might expect that decreasing the value of Δ always improves ordering, but if the number of buckets is smaller than the number of priorities, then vertices out of range will be **clipped** to the last bucket, and ordering will be lost in the last bucket. Thus, the value of Δ should be large enough to minimize the chance of clipping.





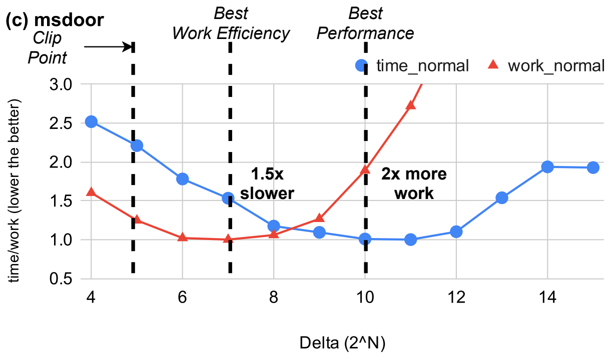


Figure 7. Execution Time and Work Performed vs. Δ. The choices of Δ are predetermined and fixed during execution; both *time* and *work* are normalized to the lowest point (lower is better). 32 buckets are used.

We see this behavior in practice. Figure 7 shows how performance and work efficiency correlate with Δ for 3 dramatically different graphs. For each graph, we highlight three important Δ values: the one that achieves the best work efficiency (*best-work-point*), the one that achieves the best performance by balancing work efficiency and parallelism (*best-perf-point*), and the one that causes clipping (*clip-point*). These choices roughly correspond to cases (c), (d), and (b) in Figure 6.

For the *RMAT* graph (Figure 7(a)), execution time correlates strongly with the amount of work performed, which indicates that there is sufficient work to keep the hardware fully utilized no matter the choice of Δ, so the Δ with the least amount of work (*best-work-point*) also achieves the best performance (*best-perf-point*). By contrast, for the *ROAD* graph (Figure 7(b)), the *best-work-point* severely underutilizes the hardware, so the *best-perf-point* is 6× faster despite doing 9× more work; thus, hardware utilization is an important factor when choosing the Δ value for such graphs. The *MSDOOR* graph (Figure 7(c)) lies midway between *RMAT* and *ROAD*, and the tradeoff is much less extreme. Finally, for all 3 graphs, the *clip-point* always performs worse than the *best-work-point*, since it causes dramatically more work without improving parallelism.

Selecting an Optimal* Δ *Value. Based on the above observations, we develop a run-time method that automatically selects a point near *best-perf-point* for a given graph. Before execution starts, we choose an initial Δ value using a heuristic similar to the one described in Section 4.3. During execution, the MTB periodically gathers run-time information and then either increases or decreases Δ. The process is a continuous feedback loop in which the MTB guides Δ closer to the optimal value at each period.

To avoid clipping, our system sets a lower bound on the Δ value by observing the point at which clipping occurs. We empirically determine that this bound occurs when the tail bucket contains at least 65% of the total number of assigned work items.

Above this lower bound, we can safely assume that smaller Δ values increase work efficiency while decreasing parallelism, and vice versa. Our goal is to keep Δ near a point where the hardware is *nearly* fully utilized, which represents the optimal tradeoff between work efficiency and parallelism; this point corresponds to the *best-perf-point* in Figure 7 (b) and (c). Beyond this point, a larger Δ value increases parallelism, but because the hardware is already fully utilized, it is likely to produce pointless extra work by relaxing ordering, so increasing Δ does not improve performance. On the hand, below this point, a smaller Δ decreases hardware utilization by decreasing concurrency and thus hurts performance.

To keep Δ near the optimal point, we define upper and lower limits on the allowed utilization based on the total number of hardware threads on the GPU; in addition, we take into account the effect of memory access divergence on bandwidth usage by correlating the number of threads with the average degree of the input graph. During execution, the hardware utilization is monitored by the MTB based on the number of work items that it currently has assigned at any time. The MTB then adjusts Δ to keep the utilization between the two limits, ensuring that the hardware is near full utilization.

For several reasons, it is desirable to avoid adjusting Δ too frequently. First, frequent adjustments can negatively impact work efficiency by mixing work items of different priorities in the same buckets. Second, some utilization fluctuations will dampen without changing Δ. For example, when a new bucket that has accumulated many work items is first being processed, utilization will temporally jump and then gradually fall with a fixed Δ, so adjusting it is likely to be counterproductive. Third, after a change in Δ value, utilization changes gradually instead of instantaneously.

To avoid overshooting the optimum setting, our scheme waits some time for utilization to settle before again changing the Δ value. This settling time varies depending on the graph input and the current Δ value. Generally speaking, it takes longer for the utilization to settle when Δ is larger, so the wait time is scaled with Δ by waiting for a fixed number of *head bucket switches*. Since the number of work items in each bucket is proportional to the Δ value, the settling time scales naturally with the value of Δ.

Fluctuations in utilization can also be avoided by assigning work from multiple high priority buckets instead of from just the head bucket, so in addition to low-frequency Δ adjustments, the MTB can make higher-frequency fine-grained adjustments by dynamically varying the number of these high priority buckets in response to measured utilization changes. This more sensitive mechanism can dampen many utilization changes that would otherwise occur, thereby enhancing the effectiveness of low-frequency Δ adjustments.

6 Evaluation

We now present our comprehensive evaluation of ADDS.

6.1 Methodology

To evaluate ADDS and prior solutions, we use an NVIDIA RTX 2080 ti GPU (Turing, TU102) [16] with driver 440.44 and an NVIDIA RTX 3090 (Ampere) [18] with driver 455.38, both using CUDA toolkit version 10.0 [17] (see Table 1 for more details). We use the RTX 2080 ti for the bulk of our results, but we include results for the more modern RTX 3090 to measure the robustness of ADDS to hardware changes. We run shared memory and serial CPU implementations on an Intel Core i9-7900X CPU, which has 10 cores and 20 hardware threads running at 3.3 GHz.

6.1.1 Graph Inputs. Our experiments use a set of 226 graph inputs from the Lonestar benchmark suite [11] and the SuiteSparse Matrix Collection [8]. We select all graphs that fit the following criteria: (1) They have at least 100K vertices and 1M edges, and (2) they are suitable for SSSP traversal, where at least 75% of the vertices can be reached

	RTX 2080 ti	RTX 3090
SM Count	68	82
Threads Per SM	1024	1536
Max Clock Rate	1.75 GHz	1.8GHz
GDDR6 Bandwidth	616 GB/s	936 GB/s
DRAM Size	11 GB	24 GB
L2 Size	5.5 MB	6 MB
Scratchpad Per SM	48 KB	48 KB
Compute Capability	7.5	8.6

Table 1. Hardware specifications for the RTX 2080 ti GPU and RTX 3090

from one of the vertices. We exclude a few large graphs that the baselines, which use more memory than ADDS, cannot run. Finally, we convert any negative edge weights to positive weights. We see from Table 2 that the selected graphs are highly diverse in terms of average degree and diameter.

A few types of graphs are worth describing explicitly, since we will analyze them in some detail. First, the road network graphs from the Lonestar suite are relatively uniform graphs with low bounded degree that are approximately planar, so they have high diameters. Second, graphs such as rmat22, again from the Lonestar suite, are power law graphs, which display a power-law distribution in their vertex degree. Thus, a small number of vertices have extremely high degree, while the vast majority of vertices have low degree. Such graphs are characteristic of social networks and many communication graphs, including large distributed systems such as the Internet. Third, the characteristics of random graphs depend on the distribution used, but these typically use a binomial distribution of node degrees.

Average Degree					
<4	4 - 8	8 - 16	16 - 32	32 - 64	>=64
17 (8%)	59 (26%)	34 (15%)	23 (10%)	71 (31%)	22 (10%)
Diameter					
<40	40 - 80	80 - 169	160 - 320	320 - 640	>=640
54 (24%)	33 (15%)	49 (22%)	29 (13%)	32 (14%)	29 (13%)

Table 2. The Distribution of Graph Characteristics—# of benchmarks (% of 226 graphs)

6.1.2 Baseline Implementations. We compare ADDS against six baselines: four for GPUs, one for shared memory CPU systems, and one for sequential processors. The four GPU implementations include the highly-optimized implementation of Near-Far [7] from the LonestarGPU 4.0 benchmark suite [4], which we refer to as **NF**; an implementation of Bellman-Ford from Gunrock 1.0 [23], which we refer to as **Gun-BF**; a Near-Far implementation from Gunrock 0.2, which we dub **Gun-NF**; and NVIDIA's proprietary SSSP implementation from CUDA 10.0, which we refer to as **NV**.

To provide a fair comparison, we make a few small modifications to the baselines. First, instead of using a default value or accepting user input to define the value of Δ, all of our parallel baselines use the equation described in the Near-Far paper [7]. Second, since current GPUs do not provide hardware support for the atomicMin operation on floating point values, we modify the code to use a software atomicMin routine from Gunrock1.0 [23]. Our implementation of ADDS handles floating point weights in the same way. Third, for NF, we modify the code pertaining to warp-level cooperative operations so that it works properly on the more recent Volta/Turing 2080 ti GPUs. The original code uses deprecated warp-level primitives, e.g. ballot() instead of ballot_sync().

Our shared memory baseline, **CPU-DS**, is an implementation of delta-stepping [15] for shared memory CPUs from Galois 4.0 [12, 20]. This implementation uses multiple fine-grained buckets to implement its priority queue.

Our sequential baseline is **Dijkstra**, a highly tuned **serial** implementation of Dijkstra's algorithm [9] from Galois 4.0, which implements the priority queue using a binary heap.

6.2 Performance Results

Comparison Against GPU Baselines. In comparing ADDS against the four baseline GPU implementations, we find that NF performs significantly better than the others, while ADDS performs significantly better than NF. For the 226 input graphs, ADDS achieves average speedups of 2.9×, 5.8×, 9.6×, and 13.4× over NF, Gun-NF, Gun-BF, and NV, respectively.

Table 3 summarizes the distribution of this performance advantage by showing the number of graphs for which ADDS obtains various ranges of speedups. For example, we see that ADDS performs worse than NF for only 4% of the graphs. For 6% of the graphs, the two perform about the same, and for 19% of the graphs, ADDS sees between 1.1× and 1.5× speedup. However, for the vast majority (78.8%) of the graphs, ADDS sees speedup of at least 1.5×, including 35% of the graphs for which the speedup is at least 3×.

To see the actual magnitudes of these speedups, these same results for NF are shown as a scatter plot in Figure 8, where the log-scale y-axis represents ADDS' speedup over NF, while the x-axis represents the average degree of the input graph. Figure 9 shows individual speedups segregated by the diameter of the input graph.

Together, these scatter plots show that ADDS' speedup over NF is largely independent of the graph's degree or diameter, which is true because ADDS optimizes both parallelism and work efficiency, and because ADDS' dynamic mechanism for selecting Δ values is able to automatically pick values that balance parallelism and work efficiency.

Comparison Against CPU-DS and Dijkstra. To show the benefit of using GPUs, we also compare ADDS running

	<0.9×	0.9× – 1.1×	1.1× – 1.5×	1.5× – 2×	2× – 3×	3× – 5×	>=5x
NF	8 (4%)	13 (6%)	27 (12%)	44 (19%)	54 (24%)	59 (26%)	21 (9%)
Gun-NF	20 (9%)	2 (1%)	10 (4%)	14 (6%)	27 (12%)	40 (18%)	113 (50%)
Gun-BF	16 (7%)	3 (1%)	8 (4%)	8 (4%)	20 (9%)	49 (22%)	122 (54%)
NV	18 (8%)	4 (2%)	7 (3%)	12 (5%)	13 (6%)	24 (11%)	148 (65%)
CPU-DS	7 (3%)	2 (1%)	1 (0%)	8 (4%)	28 (12%)	38 (17%)	142 (63%)
Dijkstra	2 (1%)	0 (0%)	0 (0%)	0 (0%)	3 (1%)	14 (6%)	207 (92%)

Table 3. Distribution of Speedup of ADDS over NF, Gun-NF, Gun-BF, NV, CPU-DS (on CPUs), and Dijkstra (serial). For example, for 8 of the 226 graphs, the speedup of ADDS over NF is < 0.9×.

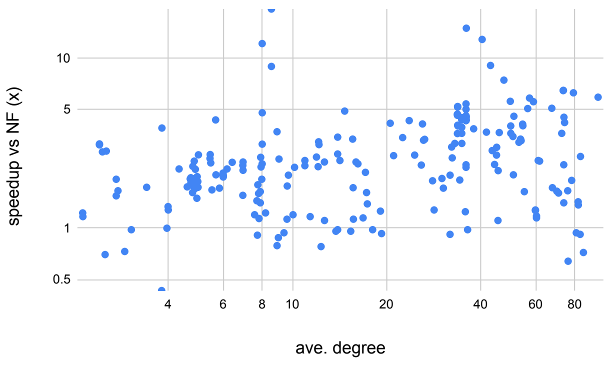


Figure 8. Speedup of ADDS over NF vs. graph degree.

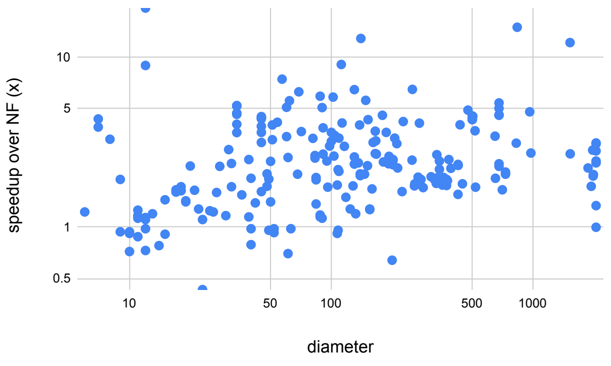


Figure 9. Speedup of ADDS over NF vs. graph diameter.

on a GPU against CPU-DS running on an Intel Core i9 and Dijkstra's running serially on the same Intel Core i9.

Table 3 summarizes the distribution of ADDS' performance advantage over CPU-DS, showing the clear superiority of running SSSP on GPUs. Not evident from the Table is that ADDS' average speedup over DS is 14.2×. ADDS' average speedup over the serial Dijkstra's is 34.4×.

6.3 Work Efficiency

Table 4 summarizes the distribution of ADDS' work efficiency relative to NF. Compared to NF, ADDS achieves nontrivial work savings (<0.75×) for 20% of the graphs. These savings are achieved by using 32 buckets, which provide a more precise priority queue than NF's 2 buckets. For 36% of the graphs, ADDS does noticeably more work (>1.5×), which occurs because ADDS' dynamic mechanism will pick a larger Δ value if the GPU is underutilized, which improves parallelism but leads to more work. In addition, ADDS does not have the duplicate vertex ID removal filter used by NF, since that requires a BSP model. On the other hand, ADDS' use of multiple buckets can mitigate some of the work inefficiency; for 44% of the graphs, ADDS does a similar amount of work (0.75× to 1.5×).

We designed ADDS to improve performance rather than to minimize work. If the improved hardware utilization outweighs the loss in work efficiency, overall performance still improves. Therefore, on average, ADDS achieves 2.9× speedup over NF despite processing 1.55× more vertices.

	<0.25×	0.25× - 0.5×	0.5× - 0.75×	0.75× - 1×	1× - 1.5×	1.5-3×	>3×
NF	10 (4%)	22 (10%)	13 (6%)	24 (11%)	75 (33%)	70 (31%)	12 (5%)
Gun-NF	50 (22%)	25 (11%)	38 (17%)	35 (15%)	34 (15%)	36 (16%)	8 (4%)
Gun-BF	61 (27%)	21 (9%)	20 (9%)	28 (12%)	64 (28%)	25 (11%)	7 (3%)
CPU-DS	18 (8%)	21 (9%)	29 (13%)	18 (8%)	39 (17%)	37 (16%)	64 (28%)
Dijkstra	0 (%)	0 (%)	0 (%)	0 (%)	30 (13%)	49 (22%)	147 (65%)

Table 4. Distribution of normalized vertex processing count of ADDS over prior implementations (lower is better for ADDS). For example, for 10 of the 226 graphs, ADDS processes < 0.25× as many vertices as NF. NV results are not included here because without the source code, we cannot obtain this metric.

6.4 Performance Analysis

This section analyzes ADDS' performance gains in more detail.

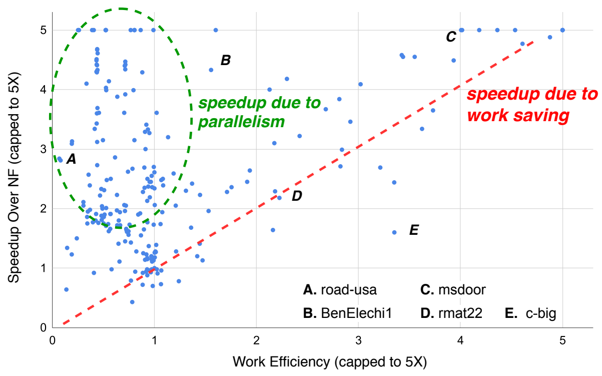


Figure 10. Correlation between speedup and work efficiency (inverse of vertex count); for both higher is better.

.

Figure 10 shows the correlation between speedup (ADDS over NF) and work efficiency for all 226 graphs. The diagonal line represents perfect correlation between work efficiency and speedup; so for graphs on or around this line (e.g.

D.rmat22 and C.msdoor), the speedup is mainly due to work efficiency.

For graphs in the upper left region (e.g. A.road-USA), ADDS does more work yet achieves better performance, with many graphs clustered in this region. For these graphs, NF underutilizes the hardware, and ADDS achieves speedups by increasing parallelism.

In the lower right region, ADDS reduces work but, by doing so, decreases parallelism, so the speedup is less than the work savings. There is just 1 graph (E.c-bag) in this region that lies far off the diagonal line.

For graphs between the two regions (e.g. B.BenElechi1), the speedup is due to both increased parallelism and improved work efficiency.

Figures 11 to 15 examine the two regions in detail by showing how the amount of parallelism changes over time, where parallelism is defined in terms of edge count (vertex-count × average-degree).

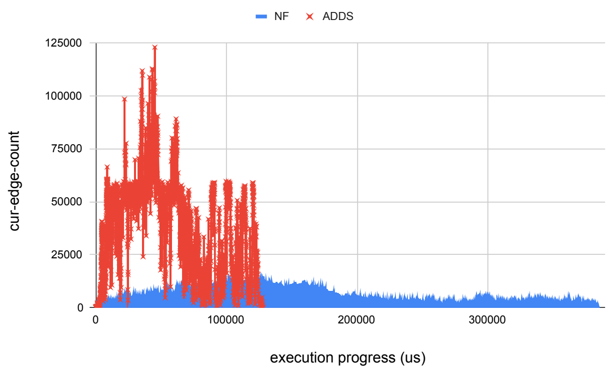


Figure 11. A.road-USA: s:3.09×, w:0.19× (s:speedup, w:work efficiency), the figure plots the amount of parallelism (edge count) during the progress of execution (us).

The Road-USA graph (Figure 11) represents an extreme case, where ADDS achieves 3× speedup yet does 5× more work. Although NF achieves good work efficiency for Road-USA, we can see from the figure that it severely underutilizes the GPU. By contrast, ADDS achieves much higher parallelism, allowing ADDS (red curve) to complete much sooner. Here, ADDS' asynchronous work scheduler allows newly active vertices to be processed immediately, whereas NF's double-buffering does not. Moreover, ADDS is able to dynamically increase the value of Δ when hardware utilization is low.

Although for Road-USA ADDS trades off work efficiency for parallelism compared to NF, ordering is still extremely important for road network graphs. For example, Gunrock's Bellman-Ford implementation does 78× more work than ADDS while being 318× slower. So we see that ADDS' dynamic mechanism works well, since it is able to select a Δ value that provides sufficient parallelism, while not letting the behavior degenerate into a Bellman-Ford solution.

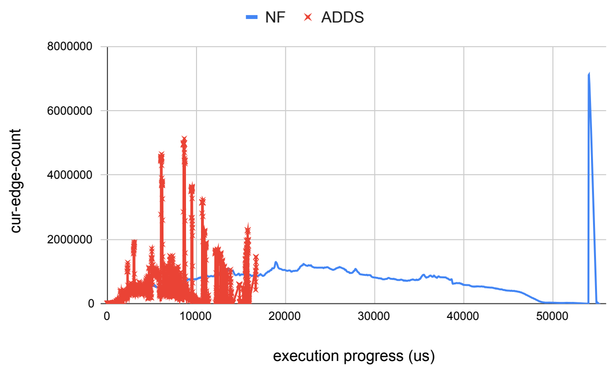


Figure 12. B.BenElechi1: s:4×, w:2.12×.

The BenElechi1 graph (Figure 12) represents cases where NF still underutilizes the hardware—though not as poorly as with the road-USA input—but ADDS is still superior in parallelism, and it sees a 2× improvement in work efficiency, so the combined effect is a speedup of 4×.

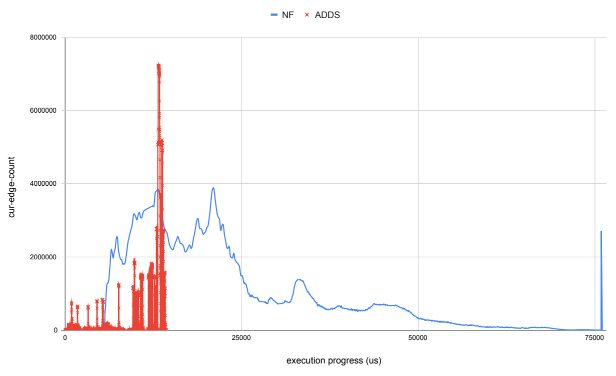


Figure 13. C.msdoor: s:5.57×, w:4×.

For the msdoor and rmat22 graphs (Figures 13 and 14), NF achieves good hardware utilization, so ADDS's advantage here come primarily from being more work efficient. For msdoor, NF's parallelism is low during the last quarter of execution, so ADDS' speedup is still higher than the work reduction. For the rmat22 input, both NF and ADDS are able to fully saturate the hardware, so the speedup correlates perfectly with improved work efficiency.[1]

Finally, for the c-big input (Figure 15), ADDS achieves 3.35× work saving but a smaller speedup of 1.6×. Here, the total execution is short (3 ms), so ADDS' dynamic Δ is unable

[1]In the figure, the edge count for NF is the amount of available work at the beginning of each BSP super-step, which is much larger than the GPU's thread count for the rmat graph; for ADDS, parallelism is the currently assigned work; so the figure does not indicate that NF processes more work concurrently than ADDS.

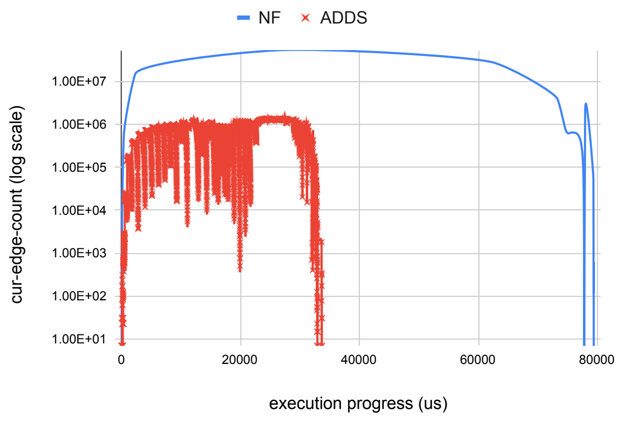


Figure 14. D.rmat22: s:2.29×, w:2.18×.

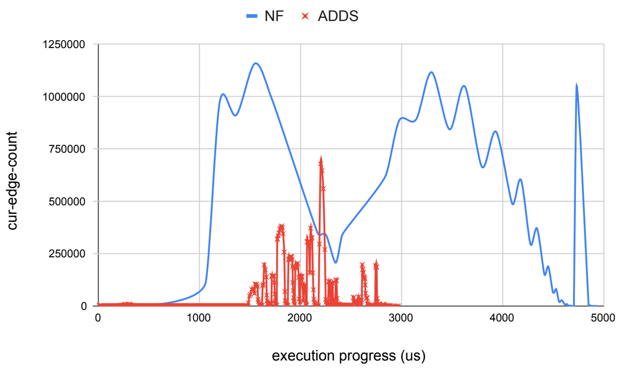


Figure 15. E.c-big: s:1.6×, w:3.35×.

to increase the value of Δ quickly enough, and the parallelism remains low in the first half of execution.

In summary, ADDS' concurrent read-write multiple-bucket design has the ability to improve both parallelism and work efficiency. Moreover, ADDS' dynamic mechanism is able to effectively utilize the multiple buckets by choosing an appropriate Δ according to the graph's run-time characteristics. As a result, ADDS is able to achieve good performance for a variety of graphs.

6.5 Evaluation on an RTX 3090 GPU

To evaluate ADDS' performance on different types of GPUs, we evaluate our solution on a newly released RTX 3090 GPU. On an RTX 3090, ADDS achieves an average speedup of 3.5× over NF, compared with a speedup of 2.9× on an RTX 2080 ti. The first two rows of Table 5 show the distributions of speedup; many graphs in 1-2× range on RTX 2080 ti are now shifted to above the 2× range on the RTX 3090.

As discussed in Section 6.4, a key reason for ADDS' performance advantage over NF is its superior utilization of hardware resources, and this utilization is even more important for the RTX 3090, which has 52% greater peak DRAM bandwidth than the RTX 2080 ti [18]. Thus, ADDS achieves higher speedup on the newer GPU.

This result also demonstrates the robustness of ADDS' mechanism for dynamically selecting Δ values, which performs well on the newer hardware with no tuning of the source code.

	<0.9×	0.9× - 1.1×	1.1× - 1.5×	1.5× - 2×	2× - 3×	3× - 5×	>=5×
RTX2080ti	8 (4%)	13 (6%)	27 (12%)	44 (19%)	54 (24%)	59 (26%)	21 (9%)
RTX3090	8 (4%)	11 (5%)	16 (7%)	23 (10%)	66 (30%)	73 (33%)	25 (11%)
Static-Δ	32 (14%)	18 (8%)	28 (13%)	29 (13%)	57 (26%)	44 (20%)	14 (6%)
2-Buckets	22 (10%)	22 (10%)	52 (23%)	40 (18%)	35 (16%)	37 (17%)	14 (6%)

Table 5. The first two rows show speedup of ADDS over NF on an **RTX2080ti** (same results as from Table 3) and an **RTX3090**. The next two rows show speedups over NF for ablated versions of ADDS on an RTX3090: Static-Δ uses a static Δ value and **2-Buckets** uses both a static Δvalue and just two buckets.

Ablation Studies. To get a better understanding of the sources of ADDS' performance benefit, we perform an ablation study on an RTX 3090 GPU. The last two rows of Table 5 show the results.

First, we disable dynamic Δ selection and instead use the same static Δ value as NF. With this configuration, the speedup of ADDS over NF is decreased from 3.5× to 2.4×, which indicates that dynamic Δ selection has a major impact on performance.

Second, we combine this static Δ value with the use of just 2 buckets. The speedup of ADDS over NF is decreased further to 2.2×. With this configuration, the remaining advantage of ADDS over NF is its asynchronous work scheduling and its delegation-based MWMR worklist, which alone achieves better resource utilization and thus higher performance. This result might seem to suggest that there is little benefit to using more than two buckets, but the use of multiple buckets is difficult to separate from the dynamic selection of Δ values, because a static Δ value reduces the effectiveness of using multiple buckets.

7 Conclusions

GPUs present a tradeoff. They provide tremendous compute power and power-efficiency, but because they are designed for highly regular data parallel computations, they often force programmers to transform efficient algorithms to less efficient algorithms for the sake of regularity. This tradeoff has been particularly striking for the SSSP problem, where the state-of-the-art algorithm for CPUs is the delta-stepping algorithm, but the previous state-of-the-art for GPUs dramatically simplifies the delta-stepping algorithm to fit GPUs.

In this paper, we have shown how this tradeoff can be broken for the SSSP problem, as we have presented ADDS, an adaptation of the delta-stepping algorithm that retains the algorithmic benefits of the original algorithm by implementing a sophisticated coarse-grained priority queue that can be efficiently executed on modern GPUs.

The key ideas are (1) to decouple the irregular operations on the priority queue from the highly regular data parallel operations and (2) to find ways to express these irregular operations as data parallel operations in their own right. For example, ADDS includes a coarse-grained priority queue that uses a Manager thread block to perform read operations on behalf of multiple Worker thread blocks. Thus, instead of having multiple threads that each read to and write from different buckets of the priority queue in an irregular fashion, the single Manager amortizes the overhead of metadata access across a large number of read operations. And since this design does not need to resolve conflicts among multiple readers, the overall design is greatly simplified.

As future work, we believe that this general approach can be applied to other irregular computations for GPUs, though the details will likely require significant engineering.

Acknowledgments

We thank Chirag Sakhuja and the anonymous referees for their helpful feedback on earlier versions of this paper. This work was funded in part by DARPA contract # HR0011-17-S-0054, NSF Grant CCF-1823546, and a gift from Intel Corporation through the NSF/Intel Partnership on Foundational Microarchitecture Research.

References

[1] Saman Ashkiani, Andrew Davidson, Ulrich Meyer, and John D. Owens. 2016. GPU multisplit. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 12, 13 pages. https://doi.org/10.1145/2851141.2851169

[2] Richard Bellman. 1958. On a routing problem. *Quart. Appl. Math.* 16, 1 (1958), 87–90. http://www.jstor.org/stable/43634538

[3] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: an asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/3018743.3018756

[4] M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[5] F. Busato and N. Bombieri. 2016. An efficient implementation of the Bellman-Ford Algorithm for Kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 27, 08 (August 2016), 2222–2233. https://doi.org/10.1109/TPDS.2015.2485994

[6] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. 1998. A parallelization of Dijkstra's Shortest Path Algorithm. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS '98)*. Springer-Verlag, Berlin, Heidelberg, 722–731.

[7] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-efficient parallel GPU methods for Single-Source Shortest Paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, USA, 349–359. https://doi.org/10.1109/IPDPS.2014.45

[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[9] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[10] Pawan Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC'07)*. Springer-Verlag, Berlin, Heidelberg, 197–208.

[11] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. 2009. Lonestar: a suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*. http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf

[12] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/1250734.1250759

[13] Shailendra Kumar, Alok Misra, and Raghvendra Tomar. 2011. A modified parallel approach to Single Source Shortest Path Problem for massively dense graphs using CUDA. *2011 2nd International Conference on Computer and Communication Technology, ICCCT-2011*, 635–639. https://doi.org/10.1109/ICCCT.2011.6075214

[14] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes. 2009. CUDA solutions for the SSSP Problem. In *Proceedings of the 9th International Conference on Computational Science: Part I (ICCS '09)*. Springer-Verlag, Berlin, Heidelberg, 904–913. https://doi.org/10.1007/978-3-642-01970-8_91

[15] U. Meyer and P. Sanders. 2003. Delta-stepping: a parallelizable Shortest Path Algorithm. *J. Algorithms* 49, 1 (Oct. 2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[16] NVIDIA. 2018. NVIDIA Turing GPU Architecture. (2018). https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[17] NVIDIA. 2019. Tuning CUDA applications for Turing. (2019). https://docs.nvidia.com/cuda/turing-tuning-guide/index.html

[18] NVIDIA. 2020. NVIDIA AMPERE GA102 GPU ARCHITECTURE. (2020). https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

[19] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano. 2013. A new GPU-based approach to the Shortest Path Problem. In *2013 International Conference on High Performance Computing Simulation (HPCS)*. 505–511. https://doi.org/10.1109/HPCSim.2013.6641461

[20] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, and et al. 2011. The Tao of parallelism in algorithms. *SIGPLAN Not.* 46, 6 (June 2011), 12–25. https://doi.org/10.1145/1993316.1993501

[21] G. G. Surve and M. A. Shah. 2017. Parallel implementation of Bellman-ford Algorithm using CUDA architecture. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, Vol. 2. 16–22. https://doi.org/10.1109/ICECA.2017.8212794

[22] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-Graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY,

USA, 38–52. https://doi.org/10.1145/3293883.3295733

[23] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

A Artifact Description

The artifact is available on Zenodo (https://zenodo.org/record/4365954#.X-2s8nVKiZQ). It contains the source code package and graph input files used for evaluation.

A.1 Quick Start Guide

1. Download and unzip files from zenodo
   a. **ppopp-code.zip** contains the source code package
   b. **sssp-int.zip** contains the int graph inputs.
   c. **sssp-float.zip** contains the float graph inputs.
2. Put the sssp-int and sssp-float directories in the inputs directory, e.g. the input file structure should be inputs/sssp-int/graph.gr
3. Inside ppopp-code directory, run ./build_all.sh.
4. There are int and float implementations of our solution and of prior solutions (taking int or float graphs)
   a. ads_* is our solution
   b. nf_* is an optimized implementation of Near-Far, which is the prior state-of-the-art GPU solution
   c. nv_* is an nvGRAPH library implementation
   d. cpu_sss_* are implementations of Dijkstra's algorithm and a CPU delta-stepping algorithm.
   e. Note: nf_* and cpu_sssp_* are from their git repository. The build_all.sh script performs the *git clone* command and applies patches.
5. Inside the ppopp-code directory, run ./run_all.sh, and each implementation will produces 2 outputs.
   Using *ads_int* as an example:
   a. *ads_int_result* contains timing and work count results. Each line has 3 fields separated by a space: **Graph_name run_time (in seconds) work_count**
   b. *ads_int_final_dist* is a directory that has the final distance (i.e. SSSP result) for all graphs (used in the next step)
6. To validate performance results, the correctness of SSSP results can be checked by comparing whether two implementations produce the same final node distances.
   a. run ./verify_against_*
      This will check the SSSP result (*_final_dist) between our solution and the target implementation.
   b. The script verify.py will compare files and report a "mismatch" for any lines that differ.
   c. Note: nv_graph uses float data types internally, so we sometimes get conversion problems for int graphs, with the distances differing by 1 between NV and other implementations (ours and prior solutions). We commented out the int version's verification for NV.

A.2 Explanations About the Source Code Package

1. NF is from LonestarGPU 6.0

https://iss.oden.utexas.edu/?p=projects/galois/lonestargpu
The download button links to the Galois git repository:
https://github.com/IntelligentSoftwareSystems/Galois

2. In ./build_all.sh from our code package, we git clone from the Galois repository for nf_* and cpu_*, and then we apply patches to modify the original source code for our experiments.
   We will now explain the contents of the patch.
3. **nf_int.patch:**
   a. *Line 16:* we modify the setup_push_warp_one() function in worklist.h to make the code work on newer Volta/Turing GPUs. The original version does not work due to the new control flow re-convergence model and the way that warp-level primitives are handled on newer GPUs. For example, this problem can be triggered on an RTX2080ti for the sssp-int/rmat22.gr graph.
   b. *Line 81:* we add "unsigned* work_count," to measure the vertex processing count. This routine has very low overhead. (Our solution has the same routine, see line 30, kernel.cu, ads_int.)
   c. *Line 174:* we add a profile_kernel to sample the average weight of the graph, which is used for setting delta based on the equation from the Near-Far paper [7] (in page 7).
      i. Note: the original code expects users to input the delta value manually for each graph; otherwise, a default delta of 10000 is used for all graphs. Therefore, we add this routine for automatically setting the delta. The profile kernel takes much less than 1% of run time. In ADDS, e.g. ads_int, we have the same routine for setting the initial delta (line153 kernel.cu), and then ADDS adjusts the delta dynamically (see, wl.h line 852).
   d. Finally, we run each graph 8 times, and we use average timing. The remaining changes are for output results, etc.
4. **nf_float.patch:**
   a. In addition to the changes in nf_int, this patch adds the ability to take float graphs as input, which is lacking in the original code.
   b. **Line 8:** we change the data type
   c. **Line 112:** we add atomicMin_float() from gunrock 1.0, which is a relatively efficient software routine for float atomicMin (current NVIDIA GPUs don't have hardware atomicMin for float). Other lines change the atomicMin to atomicMin_float.
5. **cpu_sssp_*.patch:**
   We add routines to measure total work and to output results. For the float version, we change type names and change delta_shift (for the int version) to just plain delta values (for the float version)
6. **nv_*:**
   We wrote a wrapper for the nvGRAPH library. See kernel.cu. Line 76 calls nvgraphSssp(), which is black box function. Other lines are for setup and for printing results.

A.3 Explanations About the Graph Input Files

The graph files are in binary *GR* format (http://users.diag.uniroma1.it/challenge9/format.shtml). This format is used by *Galois* [12] as well as ADDS.

Most of our input graphs are from *SuiteSparse Matrix Collection* (https://sparse.tamu.edu). The website provides MTX format graphs in text files. We converted them to GR binaries. The remain graphs are inputs that come with Galois (e.g. rmat*.gr and road-*.gr), which are already in an appropriate format.