Parallel Bidirectional A* Search for GPU-Accelerated Pathfinding

Hadi Al Khansa
American University of Beirut
Beirut, Lebanon
hma153@mail.aub.edu

Juan Gómez-Luna
NVIDIA
Zurich, Switzerland
jgomezluna@nvidia.com

Amer E. Mouawad
American University of Beirut
Beirut, Lebanon
University of Waterloo
Waterloo, Canada
aa368@aub.edu.lb

Izzat El Hajj
American University of Beirut
Beirut, Lebanon
izzat.elhajj@aub.edu.lb

Abstract

A* search is an important point-to-point shortest path finding algorithm, with applications in many domains such as navigation services, robotics, gaming, and network routing. However, it is an inherently sequential algorithm due to its reliance on a priority queue, typically implemented as a heap data structure, to perform a best-first search. A few prior works have attempted to parallelize A* search on GPUs either by using many heap-based priority queues or by using a batched heap-based priority queue. However, the latency of heap-based operations remains a fundamental limitation.

We propose a parallel A* search implementation for GPUs that replaces heap-based priority queues with a bucket-based priority queue for fast parallel extraction and insertion operations. Our implementation also leverages bidirectional search to further extract parallelism by executing the forward and backward searches in parallel. Our evaluation shows that our implementation achieves a geometric mean speedup of 8.56× (up to 15.05×) over a CPU baseline and 11.91× (up to 43.33×) over the state-of-the-art GPU baseline.

ACM Reference Format:
Hadi Al Khansa, Juan Gómez-Luna, Amer E. Mouawad, and Izzat El Hajj. 2026. Parallel Bidirectional A* Search for GPU-Accelerated Pathfinding. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3797905.3805620

1 Introduction

Pathfinding and shortest-path queries lie at the heart of countless applications including navigation services, robotics, gaming, and network routing [1, 11]. At the core of many of these systems is the A* search algorithm [26], an informed search method that combines exact cost-to-come and heuristic estimates to guarantee optimal paths while pruning large portions of the search space. However, as problem sizes grow and real-time demands tighten, CPU implementations of A* can become slow, which motivates the desire to accelerate A* search on GPUs.

GPUs offer massive parallelism and high memory bandwidth that have been leveraged to accelerate many graph and search algorithms [30, 36, 38, 41, 60]. However, the inherently sequential nature of A* search, particularly its reliance on a priority queue data structure and data-dependent expansions to perform a best-first search, poses a fundamental challenge for GPU parallelization. Prior GPU implementations of A* search relax the priority queue access to trade-off work efficiency for increased parallelism. They also replace the monolithic and typically heap-based priority queue with a large number of heap-based priority queues [27, 69] or a batched heap-based priority queue [15]. However, the latency of the operations needed to update the heap data structures by rearranging the tree nodes after each insertion and extraction remains a challenge.

To overcome this challenge, we propose a parallel GPU implementation of the A* search algorithm that abandons heap data structures entirely. Instead, our design leverages a bucket-based priority queue that avoids long-latency tree-based operations, thereby enabling efficient parallel extraction and insertion operations. We also optimize the memory access patterns of these operations and ensure efficient handling of duplicate insertions. While such a bucket-based priority queue has been used on GPUs before in other contexts [58, 68], to the best of our knowledge, our work is the first to apply it in the context of A* search. Furthermore, we extract additional parallelism from the search by employing a bidirectional search [46], executing the forward and backward frontiers of the search in parallel until they meet. To the best of our knowledge, our work is the first to parallelize bidirectional A* search on GPUs.

We evaluate our GPU implementation on a variety of grid graph types and sizes. We show that it outperforms a well optimized CPU implementation by a geometric mean of 8.56× (up to 15.05×) and the state-of-the-art parallel GPU implementation by a geometric mean of 11.91× (up to 43.33×). We also show that our technique can achieve reasonable work efficiency for non-trivial graph types.

We make the following contributions:

- We design and implement a bucket-based priority queue for A* search suitable for GPU architectures, and a corresponding kernel that leverages this priority queue to parallelize the search.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICS '26, Belfast, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2522-7/2026/07
https://doi.org/10.1145/3797905.3805620

- We adapt our design and implementation to support bidirectional A* search, increasing the parallelism extracted and often reducing the total number of node expansions.
- We evaluate our approach on a wide range of grid graphs and show consistent and significant performance improvements over CPU and GPU baselines.

Our implementation has been open sourced to enable reproducibility and further research on parallel A* search, and more generally, best-first search algorithms on GPUs.[1]

2 Background

In this section, we give an overview of A* search (Section 2.1) and bidirectional search (Section 2.2). We then describe prior works on parallelizing A* search on GPUs (Section 2.3). Other related prior works are discussed in Section 6.

2.1 A* Search

The A* search algorithm is a widely used informed search technique that efficiently computes the shortest path between a source and target vertex in a weighted graph. It extends Dijkstra's algorithm by employing a heuristic function to guide its search towards the target vertex more efficiently, significantly reducing computational overhead. It operates by evaluating candidate vertices for the shortest path using a cost function given by:

$$f(v) = g(v) + h(v) \tag{1}$$

where:

- $g(v)$ is the true cost of going from the source vertex to vertex v.
- $h(v)$ is a heuristic function that estimates the cost of going from vertex v to the target vertex.
- $f(v)$ is the estimated total cost of the path going from the source vertex to the target vertex through vertex v.

To ensure optimality and completeness, the heuristic function $h(v)$ must be admissible and consistent. The function is *admissible* if it never overestimates the true minimum cost from vertex v to the target. Formally:

$$h(v) \leq h^*(v), \quad \forall v \tag{2}$$

where $h^*(v)$ is the true minimal cost of going from v to the target vertex. The function is *consistent* (or *monotonic*) if it satisfies the triangle inequality. For every vertex v and each of its successors v', the heuristic satisfies:

$$h(v) \leq d(v, v') + h(v') \tag{3}$$

where $d(v, v')$ is the true cost of going from vertex v to vertex v'.

Algorithm 1 shows the pseudocode for the A* search algorithm. The algorithm maintains two primary data structures:

(1) **Open set** (*open*): A priority queue containing vertices to be evaluated that are prioritized by their estimated cost f.
(2) **Closed set** (*closed*): A set of vertices already evaluated.

It initializes the open set to the source vertex s (line 2) and iterates until the open set is depleted (line 3). In each iteration, it extracts the vertex v with the lowest f-value from the open set (line 4) and adds it to the closed set (line 5). If the vertex v is the target vertex t (line 6), then the shortest path has been found and it returns the constructed path (line 7). Otherwise, it iterates over the neighbors of v (line 8) that are not in the closed set (line 9). For each neighbor u of v, it finds the distance g' to u through v (line 10). If u has not been visited before (line 11), it inserts it to the open set (line 12), otherwise if the new distance g' to u is longer than the previously found distance (line 13), it stops processing u (line 14). If u has not been visited before or if a shorter distance has been found, it updates the g-value, h-value, f−value, and parent of u (lines 15-18).

Algorithm 1 A* Search Algorithm

```
1:  Input: Graph G = (V, E), source s, target t
2:  Initialize:
        open ← {s}
        closed ← ∅
        g(s) ← 0
        h(s) ← heuristic(s, t)
        f(s) = g(s) + h(s)
3:  while open ≠ ∅ do
4:      v ← EXTRACT(open)
5:      closed ← closed ∪ {v}
6:      if v = t then
7:          return reconstructed path from s to t
8:      for u ∈ N(v) do
9:          if u ∉ closed then
10:             g′(u) ← g(v) + w(v, u)
11:             if u ∉ open then
12:                 INSERT(open, u)
13:             else if g′(u) ≥ g(u) then
14:                 continue
15:             g(u) ← g′(u)
16:             h(u) ← heuristic(u, t)
17:             f(u) ← g(u) + h(u)
18:             parent(u) ← v
19: return no path exists
```

The EXTRACT and INSERT operations are priority queue operations that remove the vertex with the lowest f-value and insert a vertex based on its f-value, respectively. Priority queues are typically implemented as heaps, which means each of these operations involves long-latency updates to a tree data structure. This long latency can be a challenge for parallel implementations as we discuss in Section 2.3.

In our work, we specifically focus on two-dimensional grid graphs containing obstacles. Grid graphs are graphs that can be visualized as a grid of points when embedded in a Euclidean space. Admissible and consistent heuristic functions can be defined for grid graphs based on the Manhattan distance for four-way graphs and the Chebyshev or the octile distance for eight-way graphs. In our evaluation, we focus on eight-way graphs and use the octile distance as the heuristic, which treats diagonal moves as $\sqrt{2}$ times more expensive than horizontal or vertical moves. The octile distance is calculated as follows:

$$max(x, y) + (\sqrt{2} - 1) \cdot min(x, y) \tag{4}$$

[1]https://github.com/HadiKhansaa/BBAStar

where x is the horizontal distance between two grid points and y is the vertical distance between them.

We focus on grid graphs because they are a standard setting for pathfinding, robotics, navigation, and game maps where A* search is commonly used. Grid graphs have the advantage over general graphs that admissible and consistent heuristic functions can be defined naturally. We discuss further benefits of grid graphs to our approach and possible future extensions to general graphs in Section 7.

2.2 Bidirectional A* Search

Bidirectional A* search [6, 46, 51] is a variant of A* search that explores two search frontiers simultaneously, one from the source vertex and one from the target vertex. The search terminates when the frontiers meet, hopefully in the middle of the shortest path. The main benefit of bidirectional A* search is that if the two frontiers meet in the middle of the shortest path, the total number of vertices visited would be reduced. Intuitively, a frontier usually grows larger the deeper it explores into the graph. Hence one large frontier is likely to visit more vertices than two frontiers each with half the depth. In our work, we support the bidirectional variant of A* search not only for the purpose of reducing the number of vertices visited, but also for the purpose of increasing the amount of parallelism extracted to utilize GPU resources well.

2.3 Parallel A* Search on GPUs

There are few prior works implementing A* search on GPUs [15, 27, 69]. These works rely on extracting multiple vertices from the open set and processing them in parallel. In other words, line 4 of Algorithm 1 is converted into a batched extraction operation that extracts the top k vertices from the priority queue, and these vertices are processed in parallel. These parallel implementations thereby relax the strict priority ordering between vertices in order to extract more parallelism and better utilize the GPU resources. To process the vertices in the input batch in parallel, the rest of the operations in the loop body of Algorithm 1 such as inserting into the open set and updating the values associated with each vertex have to be done atomically. The notion of a closed set is no longer relevant because the relaxation allows a vertex to be re-added to the open set after it has been removed if a better path to that vertex is found. However, if the same neighbor vertex u is visited by multiple threads coming from different vertices v in the input batch, care must be exercised to avoid simultaneously inserting or subsequently extracting u from the open set multiple times and processing it redundantly. Overall, one of the key challenges of parallelizing A* search on GPUs is designing a data structure for the open set that enables batched extraction and concurrent insertion operations efficiently. While sequential implementations typically use a heap-based priority queue, monolithic heaps are fundamentally sequential data structures and are not suitable for highly parallel access.

One of the prior GPU implementations of A* search, GA* [69], supports concurrent access to the open set by replacing the monolithic heap with thousands of small sequential heaps, and having different threads access different heaps. A synchronization mechanism is routinely executed to ensure that vertices are not duplicated across these heaps to avoid processing them redundantly. DA* [27] extends the technique proposed by GA* to support multiple GPUs.

Another prior work, BGPQ [15], presents a heap-based priority queue for GPUs that enables concurrent insert and extract operations via batched tree nodes. Although the focus of this work is not on A* search, it includes an implementation of A* search in its evaluation for one type of graphs. The implementation leverages a single instance of the proposed BGPQ data structure to implement the open set.

Compared to these prior works, our proposed approach abandons the use of heap-based priority queues for the open set, replacing them with bucket-based priority queues that further relax the ordering between vertices for more efficient parallel extraction and insertion options. Furthermore, while these prior works only parallelize unidirectional A* search, we propose to use bidirectional A* search to further extract more parallelism from the computation. We evaluate our approach on multiple types of grid graphs. We use GA* [69] as one of the baselines in our evaluation, and show that we outperform it consistently and significantly. We use GA* [69] as our evaluation baseline, not BGPQ [15], because GA* specializes in A* search and because it performs better than BGPQ for the graphs reported in those works.

Some prior works [7, 10, 18, 49] have explored performing a large number of A* searches on small graphs, each search leveraging a small heap-based priority queue. The focus of our work is on parallelizing a single A* search on a large graph. Bucket-based priority queues have been used on GPUs in the context of other computations such as Dijkstra's algorithm for the single-source shortest path problem [9, 17, 58, 68]. We discuss these prior works in Section 6. To the best of our knowledge, our work is the first to tailor bucket-based priority queues and use them in the context of A* search on GPUs.

3 Design and Implementation

This section describes the design and implementation of our parallel bidirectional A* search approach for GPUs. We first give an overview of how control flows and how work is assigned to threads in Section 3.1. We then describe the design of our bucket-based priority queue in Section 3.2. These two sections are discussed in the context of parallelizing unidirectional A* search. Next, Section 3.3 describes how bidirectional search is supported to extract more parallelism. Finally, Section 3.4 mentions a few notable optimizations.

3.1 Control Flow and Work Assignment

3.1.1 Iterative approach. Our parallel implementation of A* search extracts batches of vertices from the open set priority queue in an iterative manner similar to prior works. In each iteration, it extracts a batch of vertices from the priority queue, processes these vertices in parallel and updates their data, inserts new vertices into the priority queue concurrently, and performs a barrier synchronization to ensure that all insertions complete before a new batch of vertices is extracted. The implementation keeps iterating until the promising vertices in the priority queue have been depleted (see Section 3.2.8 for the termination condition).

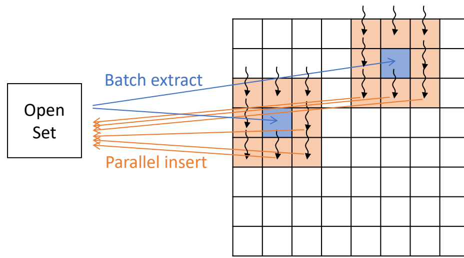


Figure 1: Example of threads assignment

3.1.2 Assignment of threads. We assign a group of threads to each vertex in the extracted batch such that each neighbor of the vertex is processed by one thread. For eight-directional grid graphs, each vertex in the batch is assigned eight threads and each thread processes one neighbor of the vertex. For example, in Figure 1, a batch of two vertices is extracted from the open set, each vertex is assigned eight threads, and each thread processes one neighbor of the vertex which it may or may not insert into the open set. This approach amounts to parallelizing the nested loop on line 8 of Algorithm 1. Capturing this nested parallelism helps increase the amount of parallelism extracted from the computation, and balance the load to reduce control divergence when different vertices in the input batch have a different number of neighbors that need to be expanded.

3.1.3 Synchronization between iterations. We use CUDA cooperative groups to perform grid-wide barrier synchronizations between iterations to ensure that all threads finish inserting newly visited vertices to the open set before a new batch is extracted. To enable performing a grid-wide barrier, the number of threads launched is equal to the maximum number of threads that can be resident on the GPU simultaneously. The entire search is performed by a single grid launched by a single kernel call. We also tried using different kernel calls for each iteration, but it degraded performance significantly due to the launch overhead and the need to reload data into shared memory and registers each iteration.

3.1.4 Work efficiency. Although extracting a batch of vertices from the priority queue concurrently exposes more parallelism, it may degrade work efficiency. In particular, extracting and working on multiple vertices concurrently increases the chances of working on less promising vertices that a sequential implementation would not have worked on. Hence, our parallel implementation of A* search, like prior works [15, 27, 69], trades off parallelism and work efficiency. This trade-off is quite typical in parallel algorithms that leverage priority queues [58]. However, our implementation takes several measures to tame the work inefficiency. We extract just enough work to utilize GPU resources and avoid extracting excess work (see Section 3.2.4). We also reduce reliance on large input batches by extracting parallelism from other sources including nested parallelism (see Section 3.1.2) and bidirectional search (see Section 3.3). We show that our implementation is effective at taming work inefficiency for non-trivial graphs in Section 5.1.4.

3.2 Bucket-based Priority Queue Design

Our implementation leverages a priority queue for the open set and a global array of variables associated with each vertex to facilitate tracking which vertices have been visited and their best paths so far. It does not use a closed list data structure for the reasons explained in Section 2.3. This section describes the design of our bucket-based priority queue data structure.

3.2.1 Motivation. For GPUs, a large batch of vertices needs to be extracted from the priority queue to fully utilize the GPU resources in each iteration. In addition, a large number of vertices needs to be inserted concurrently. Accordingly, a heap-based priority queue design would be expensive and unnecessary. It would be expensive because the latency of the heap update operations to the tree data structure would present a significant bottleneck for contending extract or insert operations. It would be unnecessary because when a large batch of vertices is extracted concurrently, one only needs to ensure that the extracted vertices are the highest priority in the queue. It is unnecessary to enforce a sorted ordering between the vertices within the batch.

3.2.2 Bucket-based design. Based on these observations, we propose a bucket-based priority queue design that groups elements in the queue into distinct buckets according to their priority values (f-values in the context of A* search). Each bucket corresponds to a continuous range of priority values. Such bucket-based priority queues have been used on GPUs before in the context of Dijkstra's algorithm for the single-source shortest path problem [9, 17, 58, 68]. To the best of our knowledge, our work is the first to tailor such a priority queue and use it in the context of A* search. We discuss the distinctions in design considerations between the two algorithms in Section 6.2.

3.2.3 Insert operations. To insert elements concurrently into the bucket-based priority queue, each thread simply identifies the bucket with the corresponding priority range and atomically increments that bucket's counter to reserve a spot in the bucket for its element. The level of contention of the atomic operations depends on the bucket's priority range. A large range implies more elements will map to the bucket causing higher contention. For this reason, our design favors using buckets with small priority ranges to reduce the contention between insert operations. However, the buckets cannot be made too small for the reasons described in Section 3.2.4.

3.2.4 Extract operations. To extract elements concurrently from the bucket-based priority queue, threads in a grid collaboratively load a bucket's elements, consuming all elements in the bucket and clearing the bucket's counter. Consecutive threads access consecutive bucket elements in a manner that ensures that the memory accesses are coalesced. However, since our design favors using buckets with small priority ranges to reduce insert contention, a single bucket is unlikely to have a sufficient number of vertices to satisfy all the threads that can run simultaneously on the GPU. For this reason, our extract operation consumes elements from multiple buckets simultaneously. In particular, we execute a prefix sum operation on the bucket counters to identify how many of the highest priority buckets are needed to utilize the largest number of GPU threads without exceeding the maximum number launched.

We refer to these buckets as the *active buckets* during an iteration. By extracting just enough vertices to fully utilize GPU resources without exceeding them, our implementation tames work inefficiency by incurring just enough of it to utilize resources that would otherwise be idle. The only infrequently occurring exception is when a single bucket contains a number of vertices that requires more resources than the entire GPU has, in which case the threads in the grid iterate over multiple vertices each to consume the entire bucket.

The prefix sum operation for identifying the active buckets is performed sequentially because we can stop when the desired number of buckets is reached and do not need to sum the entire set of counters for all buckets. Since the number of buckets is typically small, there is no benefit to parallelizing the prefix sum operation. Once we identify the number of buckets, the threads in the grid collaboratively gather and consume all the buckets as one batch. Threads perform a binary search operation on the scanned bucket counters to identify, based on their global thread index, which bucket they must access and which element in that bucket they are responsible for. Although we use small bucket ranges, we ensure they are not too small because having too many active buckets could also degrade performance. In particular, having too many active buckets would increase the latency of the sequential prefix sum operation, the extent of the binary search operation, and the fragmentation of the gather operation. In practice, the typical number of active buckets is between 50 and 100 buckets out of 10,000 buckets in total.

3.2.5 Handling concurrent insertion and extraction. Within each iteration, threads may want to extract and insert vertices into the same bucket. To avoid synchronizing between extraction and insertion operations in the same iteration, we use separate memory buffers to extract from and insert into the same bucket. That is, for each active bucket marked for batch extraction during the iteration, another buffer is used to absorb any insertions into that same bucket. After the iteration is over, the buffered insertions are then written to the original bucket before the next iteration begins.

3.2.6 Example. Figure 2 shows an example of extracting and inserting vertices to the priority queue during one iteration of our implementation. In this example, the grid consists of 12 threads and each vertex in the priority queue is processed by two threads that visit its neighbors in parallel, which means the grid can process up to six vertices from the priority queue in parallel. In practice, a grid consists of tens to hundreds of thousands of threads, and each vertex receives eight threads for eight-directional grid graphs or four threads for four-directional grid graphs. Since the grid can process up to six vertices in parallel, we execute a prefix sum on the bucket counters and identify that the largest number of buckets that does not produce more than six vertices is three buckets (i.e., buckets 0, 1, and 2) out of the six buckets in total. Accordingly, these buckets are considered the active buckets in this iteration. Once the active buckets are identified, each thread finds which vertex it is responsible for and loads the vertex from the bucket. These loads embody the batch extract operation. For example, threads 2 and 3 are both working on vertex b so they both load it from the bucket and each thread processes a different neighbor. After processing, some threads may insert vertices to the priority queue. The threads

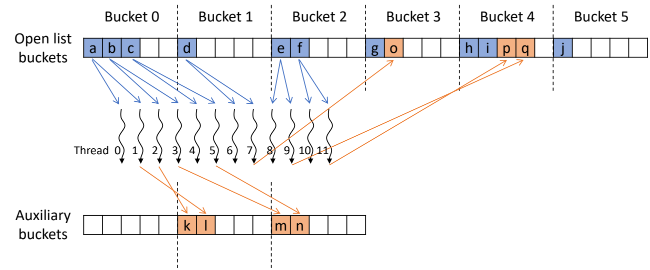


Figure 2: Example of using the bucket-based priority queue for the open set

inserting to the active buckets (e.g., threads 1, 2, 3, and 5) all insert into the auxiliary buckets to avoid race conditions with the batch extraction operation. On the other hand, the threads inserting into the inactive bucket range (e.g., threads 7, 9, and 11) insert into the original priority queue. The insert operations are performed by atomically incrementing the bucket counter, and there is no sort order enforced between vertices that are inserted into the same bucket. Once the iteration is over, the threads collaboratively replace the previously active buckets with the auxiliary buckets and start a new iteration.

3.2.7 Handling duplicate insertions. If multiple vertices in the extracted batch have the same neighbor vertex, that neighbor vertex may be processed by multiple threads with each thread visiting the vertex from a different path and having a different f-value. In this case, each thread may attempt to insert the same vertex into the priority queue or update it with a different priority. Preventing the vertex from being inserted multiple times would require the whole operation of checking if the vertex is in the open set and inserting it or updating its f-value to be atomic. It would also require support for moving the vertex from one bucket to another, which would require support for removing individual vertices from the buckets and compacting them. To avoid such expensive operations, we instead allow the same vertex to be inserted multiple times into the priority queue with a different f-value each time. However, we cannot allow all these duplicates to be processed in the next iteration because that would result in an exponential explosion of redundantly processed vertices. For this reason, we associate a shared variable with each vertex that tracks the minimum f-value for that vertex found so far. Whenever a thread adds a vertex to the open set, it also atomically updates that shared variable. At extraction time, the threads that extract that vertex check if the f-value of the extracted vertex matches the minimum f-value in that vertex's shared variable. Only the threads that extract the matching open set entry will process the vertex. The other threads recognize that they have extracted a duplicate vertex coming from a higher cost path and cease to process that vertex.

3.2.8 Termination condition. In the sequential A* search, the search terminates when the target vertex is removed from the open set. However, in the parallel search, because we have relaxed the extraction from the priority queue to increase parallelism, removing the target vertex from the open set does not indicate termination. It is possible that a better path exists but the vertex that leads to that

path is still being processed by another thread. For this reason, the parallel algorithm continues to iterate until we deplete the buckets up to the highest priority bucket where the target vertex was found. In other words, when the target vertex is extracted from a bucket in the open set, this bucket is remembered. Subsequently, the algorithm only consumes vertices and inserts vertices into that bucket and higher priority buckets until these buckets are depleted. All lower priority buckets (i.e., buckets with higher f-value) are ignored because the vertices in these buckets cannot yield a better path than the one already found.

3.3 Supporting Bidirectional Search

3.3.1 Motivation. We have seen that the main challenge with parallelizing A* search on GPUs is extracting a sufficient amount of parallelism to utilize GPU resources well. To extract parallelism, we use a bucket-based priority queue for fast batch-extraction to process vertices in parallel. We also assign multiple threads per vertex to process their neighbors in parallel. Yet another optimization that we apply to increase the amount of parallelism we extract is to perform a bidirectional search and expand both search directions in parallel with each other. As discussed in Section 2.2, bidirectional search may also be applied in a sequential implementation, but the objective is usually to reduce the number of vertices visited. The sequential implementation alternates between expanding each frontier until the two frontiers meet. For a parallel implementation, a bidirectional search has the added benefit of increasing the amount of work that can be done in parallel, which is much needed for a GPU implementation.

3.3.2 Assignment of threads. To perform the bidirectional search, the threads in the grid are divided in half across the search directions, with each search direction being assigned an equal number of threads. We maintain separate open set priority queues, auxiliary buckets, and range boundaries for each search direction. Since we have half the number of threads in each search direction compared to the unidirectional search, the maximum batch size extracted from each of the two priority queues is half the size. We use a common set of vertex metadata to facilitate detecting when the two search directions meet, such that vertices explored by the forward search are not explored by the backward search, and vice versa.

3.3.3 Termination condition. The key challenge with parallel bidirectional search is identifying when to terminate the search. In a sequential bidirectional search, the search terminates simply when the two directions meet. However, in the parallel bidirectional search, because we have relaxed the extraction from the priority queue to increase parallelism, having the search directions meet does not mean that the best path has been found. It is possible that a better path exists but the vertices that lead to that path are still being explored. For example, in Figure 3, the bidirectional search meets for the first time in iteration 5 but the path is not the shortest path. The shortest path is found later in iteration 6. To overcome this challenge, our parallel implementation maintains a shared variable that tracks the best path found so far. Whenever a thread in either search direction reaches a vertex that has been visited in the other search direction (i.e., the searches meet), we merge the forward path and the backward path and check if the merged path is better than the best path found so far. If yes, the thread atomically updates the best path. Once a candidate best path is found, we stop expanding vertices that cannot yield a better path (i.e., they have a larger f-value). Once the buckets containing vertices that can yield a better path are depleted, the search terminates and the best path is found.

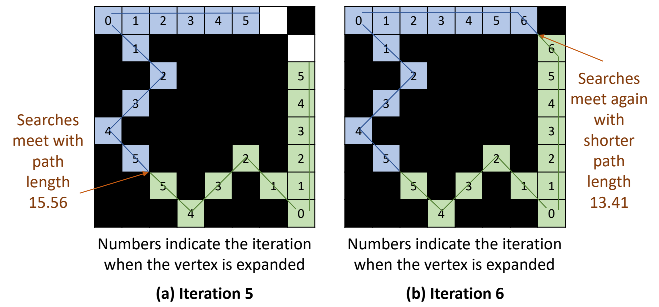


Figure 3: Example of termination with parallel bidirectional search

3.4 Other Optimizations

3.4.1 Privatization of open set buckets in shared memory. To reduce contention between threads when inserting into the same open set buckets, we apply privatization [29] to the active open set buckets in shared memory. In other words, each thread block maintains a shared memory array that is partitioned into buckets equal to the number of active open set buckets. Threads in that thread block insert their vertices into the shared memory buckets while processing. When all threads in the thread block are done processing, the vertices in the shared memory buckets are then written to the open set buckets in global memory in bulk, thereby reducing the contention of the atomic operations and improving memory coalescing when writing the inserted vertices to global memory.

3.4.2 Integer arithmetic. The g-values, h-values, and f-values are real numbers because of the presence of $\sqrt{2}$ in Equation 4. However, we scale the distance values by 1,000 and use integers instead of floating point values for cheaper arithmetic. Since the overall kernel computation is latency-bound, we have observed that replacing floating point arithmetic with lower-latency integer arithmetic improves performance by a small amount.

4 Experimental Methodology

4.1 Hardware and Software

Our GPU implementations were executed on a an NVIDIA V100 GPU with 32 GB of device memory. Our CPU implementations were executed on an Intel Xeon E5-2665 CPU with 16 cores and 64 GB of main memory.

All tests were executed within a Linux (CentOS 7) environment. We used the `nvcc 11.7` compiler for CUDA code and `g++ 10.1.0` for C++ code compilation. Both nvcc and g++ were compiled with the -O3 flag.

4.2 Implementations and Baselines

Our evaluation includes the following implementations and baselines:

1. **CPU**: A sequential unidirectional A* search implementation for CPUs written in C++. This implementation follows the classical A* search algorithm with careful tuning of data structures and memory access patterns to optimize single-threaded performance. The objective of comparing to this baseline is to demonstrate the utility of parallelizing A* search on GPUs.
2. **CPU-B**: A sequential bidirectional A* search implementation for CPUs written in C++. The objective of comparing to this baseline is to show how the benefit of bidirectional search on GPUs differs from that on CPUs.
3. **LockA***: A parallel GPU baseline that uses a monolithic heap-based priority queue and locks the entire queue on extraction and insertion. The objective of comparing to this baseline is to demonstrate the importance of enabling concurrent extract and insert operations in the priority queue design.
4. **GA*** [69]: The state-of-the-art parallel GPU baseline which uses a large number of small and local priority queues distributed across thread blocks. We execute the code of this prior work on our GPU for fair comparison. The objective of comparing to this baseline is to demonstrate how our work outperforms prior work due to the benefit of using bucket-based priority queues.
5. **UBA* (Uni-directional Bucket-based A*)**: Our parallel GPU implementation that performs unidirectional A* search using a bucket-based priority queue. The objective of including this implementation is to isolate the benefit of using bucket-based priority queues from the benefit of incorporating bidirectional search.
6. **BBA* (Bidirectional Bucket-based A*)**: Our parallel GPU implementation that performs bidirectional A* search using a bucket-based priority queue. This implementation is the ultimate one proposed in our work. We compare it against all preceding implementations and baselines.

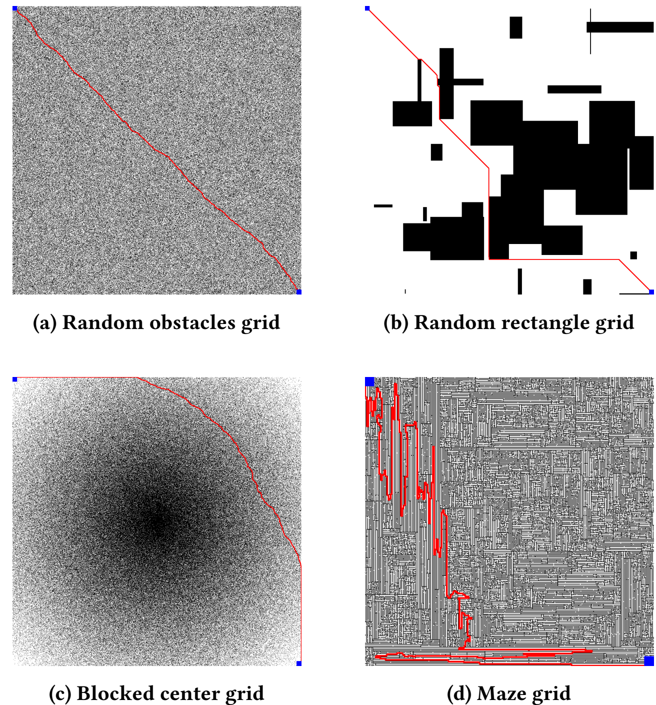
We also compare to Orionet [20], a state-of-the-art parallel CPU implementation, in Section 5.1.5.

For the UBA* and BBA* implementations, we use a circular buffer of 200 buckets where each bucket has a size of 20,000 entries, which is sufficient to avoid overflow on all tested grids. The bucket f-value range is 3,000, matching the integer-scaled octile distance used in our eight-directional grids, where horizontal and vertical moves have a cost of 1,000 and diagonal moves have cost 1,414.

4.3 Datasets

We evaluate all implementations and baselines on multiple types of grid graphs. These types are listed below, ordered by increasing informativeness of the heuristic function

- **Empty**: Grid graphs with no obstacles. For these graphs, the heuristic function guides the search to the exact shortest path.
- **Random**: Grid graphs with randomly distributed single-vertex obstacles with a 20% density. For these graphs, the heuristic function does a reasonable job at guiding the search, but the algorithm needs to make slight deviations to avoid obstacles.
- **Random rectangle**: Grid graphs with randomly distributed rectangular obstacles spanning multiple vertices. For these graphs, the algorithm needs to make larger deviations from the heuristic to avoid the larger obstacles.
- **Blocked center**: Grid graphs with randomly distributed single-vertex obstacles congested at the center causing a blockage. For these graphs, the heuristic function initially performs poorly by guiding the search towards the center where there is no path, but does a reasonable job in the second half of the search once the congestion in the center has been bypassed.
- **Maze**: Grid graphs with obstacles forming a maze. for these graphs, the heuristic function is least effective at guiding the search.



(a) Random obstacles grid (b) Random rectangle grid (c) Blocked center grid (d) Maze grid

Figure 4: Examples of different types of grid graphs and their shortest paths marked in red

For each type of grid graph, we consider randomly generated square grids with different dimensions ranging from $10,000 \times 10,000$ to $30,000 \times 30,000$. An example of each type of grid is also shown in Figure 4, with black pixels indicating obstacles and the red line indicating the shortest path. We also compare to real grid graphs from the MovingAI benchmark suite [50] in Section 5.1.5.

5 Evaluation

Table 1 compares the execution time of our implementations and baselines, and Figure 5 shows their speedup relative to the CPU baseline. To assist with understanding the performance trends, Table 2 shows the number of vertices expanded by some of the

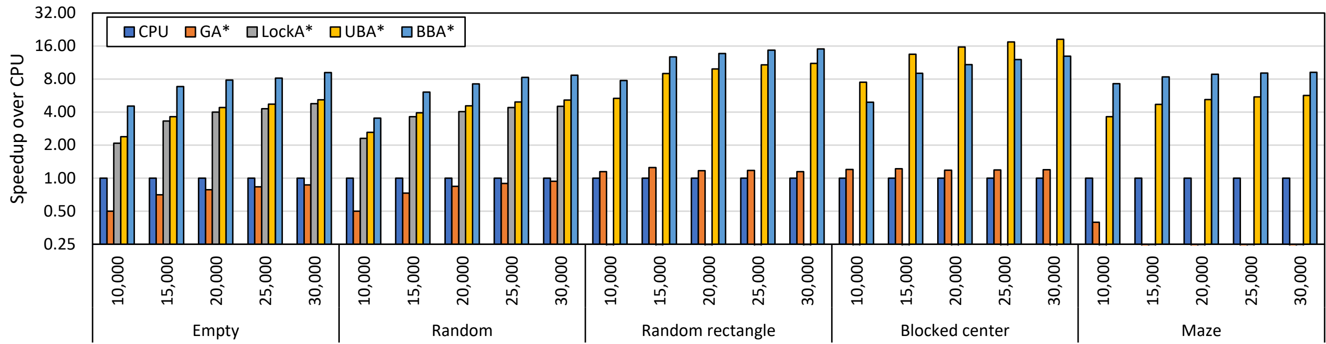


Figure 5: Speedup over baselines (missing columns indicate speedup lower than 0.25×)

Table 1: Comparing the execution time of different implementations and baselines

Grid type	Grid dimension	Path length	Time (s)					
			CPU	CPU-B	GA*	LockA*	UBA*	BBA*
Empty	10,000	10,000	0.50	1.19	1.00	0.24	0.21	**0.11**
	15,000	15,000	1.16	2.70	1.65	0.35	0.32	**0.17**
	20,000	20,000	1.80	4.30	2.30	0.45	0.41	**0.23**
	25,000	25,000	2.45	6.37	2.95	0.57	0.52	**0.30**
	30,000	30,000	3.20	13.78	3.70	0.67	0.62	**0.35**
Random	10,000	11,365	0.60	1.77	1.20	0.26	0.23	**0.17**
	15,000	17,052	1.34	2.44	1.84	0.37	0.34	**0.22**
	20,000	22,739	2.10	4.56	2.50	0.52	0.46	**0.29**
	25,000	28,426	2.82	6.91	3.15	0.64	0.57	**0.34**
	30,000	34,113	3.56	14.03	3.80	0.79	0.69	**0.41**
Random rectangle	10,000	13,125	2.4	3.5	2.1	72	0.45	**0.31**
	15,000	19,112	6.1	10.3	4.9	> 200	0.68	**0.48**
	20,000	26,132	9.0	14.6	7.7	> 200	0.91	**0.66**
	25,000	33,552	12.3	19.2	10.5	> 200	1.14	**0.84**
	30,000	38,158	15.2	27.1	13.3	> 200	1.37	**1.01**
Blocked center	10,000	16,072	3.0	4.7	2.5	81	**0.40**	0.61
	15,000	24,170	8.20	10.2	6.8	> 200	**0.61**	0.91
	20,000	32,267	13.0	17.9	11.0	> 200	**0.83**	1.20
	25,000	40,364	18.1	25.3	15.3	> 200	**1.04**	1.50
	30,000	48,462	23.2	37.4	19.5	> 200	**1.26**	1.79
Maze	10,000	363,910	8.7	12.2	22	140	2.40	**1.2**
	15,000	474,509	17.6	25.3	76	> 200	3.75	**2.1**
	20,000	585,108	26.5	40.2	120	> 200	5.10	**3.0**
	25,000	695,707	35.4	66.3	164	> 200	6.45	**3.9**
	30,000	806,306	44.3	72.2	208	> 200	7.80	**4.8**

implementations and baselines. Throughout this section, we discuss the performance comparison between our implementations and the CPU baselines (Section 5.1), our implementations and the GPU baselines (Section 5.2), and our unidirectional and bidirectional implementations together (Section 5.3).

5.1 Comparison with CPU baselines

5.1.1 Comparison between CPU and CPU-B. We observe from Table 1 that the bidirectional CPU-B implementation is consistently slower than the unidirectional CPU implementation for the graphs that we evaluate on. If we look at the number of vertices expanded by the CPU-B implementation in Table 2, they are usually lower than that of the unidirectional CPU implementation but only by a small amount. Hence, the number of vertices saved does not outweigh the additional overhead of the implementation. For this reason, we use the unidirectional CPU implementation as our main CPU baseline in the rest of the evaluation section.

5.1.2 Time and speedup comparison. We observe from Table 1 and Figure 5 that both our implementations consistently and significantly outperform the unidirectional CPU baseline. For the largest grid graph of size $30,000 \times 30,000$, the speedup of our unidirectional implementation (UBA*) ranges between 5.16× and 18.41× while the speedup of our bidirectional implementation (BBA*) ranges between 8.68× and 15.05× across different types of grid graphs.

5.1.3 Number of vertices expanded. To better understand why the speedup of our implementation varies across graph types, we inspect the number of vertices expanded by each implementation. Table 2 shows the number of vertices expanded by the CPU implementation and its ratio to the shortest path length (i.e., CPU / path length). It is clear that the speedup of our GPU implementations is larger when the ratio of the number of vertices expanded by the CPU implementation compared to the shortest path length is larger. For example, the graphs with the lowest speedups, which are the empty and random graphs, have a ratio close to one. On the other hand, the graphs with the highest speedups, which are the random rectangle and blocked center graphs, have ratios in the hundreds. The execution time of the CPU implementation is most affected by the number of vertices it expands since these expansions happen sequentially and fall on the critical path of execution. On the other hand, the execution time of the GPU implementation is more affected by the length of the shortest path. The GPU implementation needs to execute at least as many iterations as the shortest path length so the vertices on the shortest path form a critical path to the execution. However, in each iteration, the unpromising vertices are processed in parallel with the promising vertices so the spuriously expanded vertices are absorbed by the parallelism and do not usually fall on the critical path. For this reason, when the difference between the number of vertices expanded and the number of vertices on the critical path is larger, the difference in execution time between the CPU implementation and the GPU implementation is naturally larger.

5.1.4 Work efficiency. Table 2 also shows the number of vertices expanded by our GPU implementations and their ratios to that of the CPU implementation (i.e., UBA / CPU and BBA* / CPU). We observe that the number of vertices expanded by the GPU implementations is significantly higher than that of the CPU implementations for

Table 2: Comparing the number of vertices visited between different implementations and baselines

Grid type	Grid dimension	Path length	Expanded vertices				Ratio of expanded vertices				
			CPU	CPU-B	UBA*	BBA*	CPU / path length	UBA* / CPU	BBA* / CPU	CPU-B / CPU	BBA* / UBA*
Empty	10,000	10,000	10,000	10,000	11,929,520	8,180,242	1.00×	1,192.95×	818.02×	1.00×	0.69×
	15,000	15,000	15,000	15,000	18,020,263	13,129,049	1.00×	1,201.35×	875.27×	1.00×	0.73×
	20,000	20,000	20,000	20,000	24,111,006	17,049,926	1.00×	1,205.55×	852.50×	1.00×	0.71×
	25,000	25,000	25,000	25,000	30,201,749	22,435,585	1.00×	1,208.07×	897.42×	1.00×	0.74×
	30,000	30,000	30,000	30,000	36,292,492	25,135,512	1.00×	1,209.75×	837.85×	1.00×	0.69×
Random	10,000	11,365	11,985	10,921	25,929,520	19,608,814	1.05×	2,163.50×	1,636.11×	0.91×	0.76×
	15,000	17,052	18,023	17,055	36,510,763	22,586,413	1.06×	2,025.79×	1,253.20×	0.95×	0.62×
	20,000	22,739	24,061	23,332	47,092,006	35,357,919	1.06×	1,957.19×	1,469.51×	0.97×	0.75×
	25,000	28,426	30,099	29,120	61,673,249	36,700,128	1.06×	2,049.01×	1,219.31×	0.97×	0.60×
	30,000	34,113	36,137	35,011	110,254,492	81,762,041	1.06×	3,051.01×	2,262.56×	0.97×	0.74×
Random rectangle	10,000	13,125	4,341,022	4,211,053	41,234,914	25,016,815	330.74×	9.50×	5.76×	0.97×	0.61×
	15,000	19,112	6,812,154	6,723,132	59,128,956	38,296,664	356.43×	8.68×	5.62×	0.99×	0.65×
	20,000	26,132	8,122,716	8,011,311	74,202,912	49,299,956	310.83×	9.14×	6.07×	0.99×	0.66×
	25,000	33,552	11,254,312	10,152,116	92,287,016	63,170,342	335.43×	8.20×	5.61×	0.90×	0.68×
	30,000	38,158	16,554,871	15,214,167	150,271,030	86,437,785	433.85×	9.08×	5.22×	0.92×	0.58×
Blocked center	10,000	16,072	7,349,062	7,491,165	31,086,812	45,334,934	457.26×	4.23×	6.17×	1.02×	1.46×
	15,000	24,170	17,695,959	18,102,215	82,196,669	112,818,958	732.15×	4.64×	6.38×	1.02×	1.37×
	20,000	32,267	28,042,856	29,122,361	127,499,966	180,302,982	869.09×	4.55×	6.43×	1.04×	1.41×
	25,000	40,364	38,389,753	39,129,552	184,070,347	247,787,006	951.09×	4.79×	6.45×	1.02×	1.35×
	30,000	48,462	48,736,650	49,135,252	218,437,785	315,271,030	1,005.67×	4.48×	6.47×	1.01×	1.44×
Maze	10,000	363,910	56,077,676	54,057,172	45,385,784	31,121,680	154.10×	0.81×	0.55×	0.96×	0.69×
	15,000	474,509	83,432,656	81,132,224	102,248,895	74,488,338	175.83×	1.23×	0.89×	0.97×	0.73×
	20,000	585,108	110,787,636	108,223,110	159,092,106	112,500,776	189.35×	1.44×	1.02×	0.98×	0.71×
	25,000	695,707	138,142,616	135,020,116	215,945,117	160,116,373	198.56×	1.56×	1.16×	0.98×	0.74×
	30,000	806,306	165,497,596	162,222,156	272,798,228	189,020,201	205.25×	1.65×	1.14×	0.98×	0.69×

empty and random graphs with ratios in the hundreds and thousands, indicating poor work efficiency for these graphs. The reason is that the shortest path for these graphs aligns well with the heuristic function, allowing the CPU to make quick progress towards the solution without exploring many additional vertices that are off the shortest path. On the other hand, for the remaining types of graphs where the heuristic does not perfectly align with the shortest path, the GPU implementations only expand a modest number of vertices more than the CPU implementation with ratios less than ten, indicating reasonable work efficiency for these scenarios. Accordingly, our approach of extracting just enough work from the priority queue to utilize the GPU hardware without exceeding its capacity is an effective approach at taming the work inefficiency caused by relaxing the priority queue access.

5.1.5 Comparison with a state-of-the-art parallel CPU implementation. Table 3 compares BBA* to Orionet's [20] parallel CPU implementations of unidirectional and bidirectional A* search for our synthetic graphs, while Table 4 makes the same comparison for real grid graphs from the MovingAI benchmark suite [50]. It is clear that BBA* is consistently faster than Orionet for all datasets for the CPU and GPU used. Naturally, the specific performance difference would depend on the specific CPU-GPU pairing.

5.2 Comparison with GPU baselines

5.2.1 Comparison with LockA.* We observe from Table 1 and Figure 5 that both our implementations consistently and significantly outperform LockA*, and so does GA* for the graphs where the heuristic does not align well with the shortest path. In fact, for these non-trivial graphs, LockA* fails to finish within reasonable time. This result demonstrates the scalability challenge faced by using a monolithic heap-based priority queues on GPUs and motivates the need for supporting concurrent extraction and insertion operations the way GA* and our implementations do.

Table 3: Comparing execution time (in seconds) to Orionet [20] parallel CPU implementations

Grid type	Grid dimension	Orionet [20]		BBA*
		Unidirectional A*	Bidirectional A*	
Empty	3,000	0.452	0.407	**0.010**
	3,500	0.608	0.707	**0.012**
	4,000	0.783	0.680	**0.017**
	5,000	1.188	1.080	**0.018**
Random	3,000	0.392	0.320	**0.058**
	3,500	0.528	0.511	**0.064**
	4,000	0.647	0.561	**0.095**
	5,000	0.973	0.918	**0.130**
Random rectangle	3,000	0.253	0.300	**0.065**
	3,500	0.341	0.340	**0.080**
	4,000	0.214	0.334	**0.097**
	5,000	0.543	0.660	**0.141**
Blocked center	3,000	0.337	0.275	**0.114**
	3,500	0.435	0.432	**0.127**
	4,000	0.528	0.459	**0.165**
	5,000	0.760	0.670	**0.237**
Maze	3,000	0.599	0.587	**0.382**
	3,500	0.776	0.971	**0.396**
	4,000	1.548	0.476	**0.462**
	5,000	1.797	1.757	**0.483**

Table 4: Comparing execution time (in milliseconds) to Orionet [20] parallel CPU implementations for real grid graphs from the MovingAI benchmark suite [50]

Map name	Grid dimensions	Orionet [20]		BBA*
		Unidirectional A*	Bidirectional A*	
lak513d	637×389	1.19	2.04	**0.25**
hrt000d	876×408	6.52	2.25	**0.25**
ost000a	969×487	12.17	3.63	**0.24**
ost000t	971×487	7.61	3.89	**0.26**
ost100d	1025×1024	3.75	7.06	**0.28**

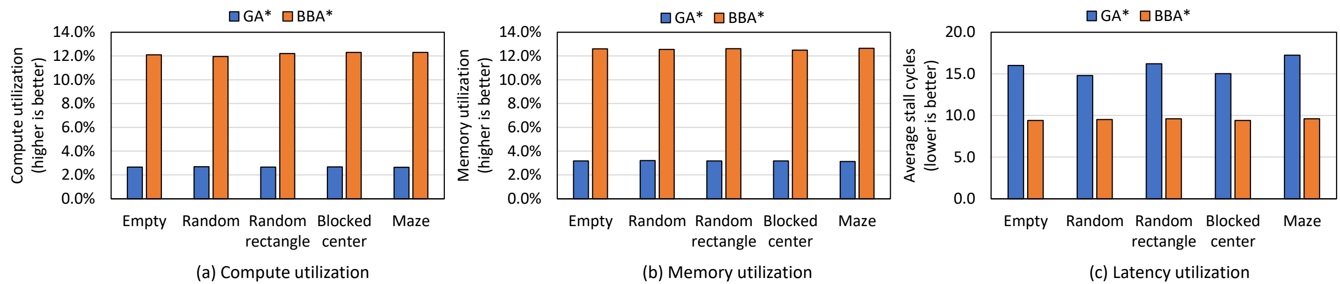


Figure 6: Profiling analysis and comparison of our implementation to the state-of-the-art GPU baseline

5.2.2 Comparison with GA.* We observe from Table 1 and Figure 5 that both our implementations consistently and significantly outperform the state-of-the-art GPU baseline GA* [69]. For the largest grid graph of size $30,000 \times 30,000$, the speedup of our unidirectional implementation (UBA*) ranges between 5.51× and 26.67× while the speedup of our bidirectional implementation (BBA*) ranges between 9.27× and 43.33× across different types of grid graphs. It is noteworthy that for all types of graphs, the larger the graph, the higher the speedup of our implementations over GA*. This result indicates that our bucket-based priority queue provides more scalability than GA*'s distributed heap-based priority queues.

5.2.3 Profiling analysis. To better understand the advantage of our BBA* implementation over the GA* baseline, Figure 6 compares the compute utilization, memory utilization, and average latency between consecutive instructions of the two implementations. As expected, the compute utilization in Figure 6(a) and memory utilization in Figure 6(c) are low for both implementations because A* search is a fundamentally latency bound computation. However, they are much lower for GA* than for BBA*. When we look at the latency analysis in Figure 6(c), we observe that GA* incurs much higher average latency between instructions than BBA*. This result demonstrates the effectiveness of our implementation at reducing the latency of priority queue and vertex processing operations.

5.3 Comparison between UBA* and BBA*

We observe from Table 1 and Figure 5 that our bidirectional search implementation (BBA*) outperforms our unidirectional search implementation (UBA*) for four out of the five types of graphs. For the largest grid graph of size $30,000 \times 30,000$, the speedup of BBA* over UBA* is 1.77×, 1.68×, 1.36×, and 1.63× for empty, random, random rectangle, and maze graphs respectively. This result shows the effectiveness of supporting bidirectional search at extracting more useful parallelism from the computation. This observation is clear from Table 2 where BBA* consistently expands fewer vertices than UBA* for these four types of grid graphs.

It is notable from Table 2 that the reduction in vertices explored by the bidirectional search for the parallel GPU implementation (i.e., BBA* / UBA*) is more significant than that for the sequential CPU implementation (i.e., CPU-B / CPU). For the sequential implementation, the benefit of bidirectional search is replacing the second half of the forward search with the first half of the backward search, which tends to expand fewer vertices. However, for the parallel GPU implementations, the bidirectional search has an additional benefit of replacing the parallel expansion of less promising vertices in the forward search with more promising vertices in the backward search, resulting in fewer spurious vertices processed overall. This observation explains why the parallel GPU implementation benefits more than the sequential CPU implementation from leveraging bidirectional search.

For the remaining type of grid graphs, blocked center graphs, we observe that our unidirectional GPU implementation outperforms our bidirectional GPU implementation. For the largest grid graph of size $30,000 \times 30,000$, the speedup of UBA* over BBA* is 1.42×. It is not surprising for unidirectional search to outperform bidirectional search for this particular type of graph. In the blocked center graph, the heuristic poorly guides the search towards the blocked center in the first half of the search. However, once the blocked center is bypassed, the heuristic guides the search very well in the second half of the search. As shown in the example in Figure 4c, the second half of the shortest path closely follows an octile-distance path towards the target. Accordingly, the bidirectional implementation replaces the efficient second half of the forward search with an inefficient first half of the backward search, resulting in an overall decline in efficiency. Indeed, it is clear from Table 2 that UBA* expands fewer vertices than BBA* for this particular type of graph. This observation is also true for the sequential CPU implementation where the CPU-B implementation expands more vertices than the CPU implementation only for this type of graph.

6 Related Work

6.1 Priority Queues and Parallel Shortest Path Solutions on CPUs

Priority queues in the context of parallel shortest path problems have been widely explored on CPUs [19, 28, 35, 52]. Recent notable works include GraphIt [67], MBQ [66], and Orionet [20]. GraphIt [67] is based on Δ-stepping for weighted graphs. MBQ [66] uses a multi-level bucket queue with internal locking and is evaluated on graph workloads rather than on grid-based A* search. Orionet [20] is a CPU point-to-point shortest-path (PPSP) system with bidirectional search and additional PPSP-specific optimizations, which outperforms GraphIt and MBQ. We compare the performance of our parallel GPU implementations to Orionet in Section 5.1.5.

6.2 Priority Queues and SSSP on GPUs

Multiple prior works [9, 17, 58, 68] have implemented priority queues for the GPU, particularly for accelerating Dijkstra's algorithm for the single-source shortest path problem (SSSP). Dijkstra's algorithm aims to find the shortest path from a source vertex to all other vertices whereas A* search aims to find the shortest path to a particular target vertex. Moreover, Dijkstra's algorithm uses the priority queue to prioritize expanding from the vertices that are closest to the source, whereas A* search uses the priority queue to prioritize expanding vertices that are most likely to be on the shortest path based on the heuristic function that guides the search towards the target.

The parallel implementations of Dijkstra's algorithm are based on Δ-stepping [39] which involves grouping vertices into buckets with priority range Δ and processing them one bucket at a time, i.e., taking a step of size Δ each time. Near-Far [17] uses just two buckets one for vertices that are relatively near the source and one for vertices that are relatively far from the source. ADDS [58] argues that two buckets are too coarse-grain and uses multiple buckets and a dynamically adapted priority range for the buckets. Zhang et al. [68] also use multiple buckets and make further improvements to data locality and load balancing. Berney et al.[9] improve the cache efficiency of the bucket-based priority queue.

Our proposed bucket-based priority queue can be seen as analogous to Δ-stepping but in the context of A* search. However, applying bucket-based priority queues in the context of A* search is not a straightforward appropriation of Δ-stepping from the context of SSSP. Δ-stepping is built around light/heavy edge relaxation for general weighted graphs and typically processes one bucket at a time. Prior works are usually concerned with adapting Δ to explore arbitrary graphs with power-law distributions. Our work is designed for A* search on grid graphs with structured cost progression and many simultaneously active buckets. The key idea is a GPU-oriented extraction and packing strategy that collects just enough work from multiple buckets to keep the device saturated, combined with iteration-level synchronization and correction. Furthermore, duplicate entries may remain temporarily in the open structure, but inferior duplicates are prevented from being expanded, which preserves the correctness of the final shortest path. Furthermore, while Dijkstra's algorithm is fundamentally unidirectional because it finds the shortest paths to all vertices, our implementation of A* search is bidirectional which allows us to extract more parallelism while taming work inefficiency.

6.3 Graph Search Algorithms on GPUs

There have been many works implementing graph search algorithms on GPUs. One of the most commonly explored search algorithms on GPUs is breadth first search (BFS) [8, 16, 24, 25, 34, 36–38, 40, 44, 48, 55]. Common objectives of these prior works are optimizing the irregular memory accesses as well as exploiting nested parallelism while achieving load balance. A number of works propose automated techniques for optimizing GPU kernels with nested parallelism [12, 21, 42, 53, 56, 57, 61, 64] that especially target BFS and other similar graph algorithms. There are also a number of libraries [22, 30, 33, 41, 59, 60] that optimize a variety of graph algorithms on GPUs including BFS and other traversal algorithms.

While BFS is commonly targeted due to its high amenability for parallelization, depth-first search (DFS) has received much less attention by prior works on graph traversal due to its sequential nature. However, DFS has been accelerated on GPUs [2, 4, 5, 13, 14, 23, 62, 63, 65] in the context of optimization algorithms on graphs that traverse search trees where a BFS traversal would suffer exponential explosion in memory demands. The typical approach adopted by these works is to divide the search space into many small subtrees and execute many concurrent DFS traversals. Common optimization objectives include mitigating the memory demands of sustaining many concurrent traversal stacks as well as balancing the load across thread blocks using oversubscription or a shared worklist data structure [31, 32, 47].

While each of BFS and DFS have their applications, our work targets A* search which is a form of best-first search (BeFS) and where the key challenge is overcoming the sequential nature of the priority queue, a data structure not used in either BFS or DFS.

Bidirectional search strategies have also received attention on GPUs in different contexts. Pajot et al. [43] parallelize bidirectional search in the context of path tracing. Tangjittaweechai et al. [54] implement a parallel bidirectional variant of Dijkstra's algorithm. Pavlik et al. [45] perform bidirectional search on the CPU while offloading parts of the computation to the GPU. ECL-MM [3] uses bidirectional search on GPUs to compute maximum matchings in bipartite graphs. Our work parallelizes bidirectional search on GPUs in the context of A* search, and performs the entire computation on the GPU with a single kernel.

7 Conclusion and Future Work

We propose a parallel GPU implementation of A* search. In contrast with prior works that use adaptations of heap-based priority queues, our work proposes a bucket-based priority queue to improve the efficiency of batch extraction and concurrent insertion operations. To tame work inefficiency, our implementation extracts just enough work from the priority queue to utilize GPU resources without oversubscribing them, and leverages alternative sources of parallelism including nested parallelism and bidirectional search. Our evaluation on multiple types of grid graphs shows that our implementation outperforms the baseline CPU implementation while keeping work efficiency under control for non-trivial graphs. It also outperforms the state-of-the-art GPU implementation consistently and by a significant amount.

Our work focuses on grid graphs because admissible heuristics are natural, edge costs are regular, and the progression of f-values is structured enough to make lightweight bucketing effective with predictable active-bucket windows. The same design ideas may extend to more general weighted graphs, but doing so would require additional mechanisms for parameter selection and for handling broader cost distributions. Extending our approach to support general graphs is the subject of future work.

Acknowledgments

This work was supported by the University Research Board of the American University of Beirut (AUB-URB-28005-104631).

References

[1] 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*. 287–290.

[2] Faisal N Abu-Khzam, DoKyung Kim, Matthew Perry, Kai Wang, and Peter Shaw. 2018. Accelerating vertex cover optimization on a GPU architecture. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 616–625.

[3] Anju Mongandampulath Akathoott and Martin Burtscher. 2025. A Bidirectional GPU Algorithm for Computing Maximum Matchings in Bipartite Graphs. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 297–308.

[4] Mohammad Almasri, Yen-Hsiang Chang, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2023. Parallelizing maximal clique enumeration on gpus. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 162–175.

[5] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. Parallel k-clique counting on gpus. In *Proceedings of the 36th ACM international conference on supercomputing*. 1–14.

[6] Momodou Bah, Ioanna Giorgi, and Giovanni Luca Masala. 2025. A lightweight and rapid bidirectional search algorithm. *Robot Learning* 2, 2 (2025).

[7] Nikolai Baudis, Florian Jacob, and Philipp Andelfinger. 2017. Performance evaluation of priority queues for fine-grained parallel tasks on GPUs. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–11.

[8] Massimo Bernaschi, Giancarlo Carbone, Enrico Mastrostefano, Mauro Bisson, and Massimiliano Fatica. 2015. Enhanced GPU-based distributed breadth first search. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*. 1–8.

[9] Kyle Berney, John Iacono, Ben Karsin, and Nodari Sitchinava. 2019. A parallel priority queue with fast updates for GPU architectures. *arXiv preprint arXiv:1908.09378* (2019).

[10] Avi Bleiweiss. 2008. GPU accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. 65–74.

[11] Antonios Chatzisavvas, Michael Dossis, and Minas Dasygenis. 2024. Optimizing mobile robot navigation based on A-star algorithm for obstacle avoidance in smart agriculture. *Electronics* 13, 11 (2024), 2057.

[12] Guoyang Chen and Xipeng Shen. 2015. Free launch: optimizing GPU dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*. 407–419.

[13] Xuhao Chen et al. 2022. Efficient and scalable graph pattern mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 857–877.

[14] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1190–1205.

[15] Yanhao Chen, Fei Hua, Yuwei Jin, and Eddy Z Zhang. 2021. BGPQ: A Heap-Based Priority Queue Design for GPUs. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–10.

[16] Mayank Daga, Mark Nutter, and Mitesh Meswani. 2014. Efficient breadth-first search on a heterogeneous processor. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 373–382.

[17] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 349–359.

[18] Aljosha Demeulemeester, Charles-Frederik Hollemeersch, Pieter Mees, Bart Pieters, Peter Lambert, and Rik Van de Walle. 2011. Hybrid path planning for massive crowd simulation on the gpu. In *International Conference on Motion in Games*. Springer, 304–315.

[19] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 293–304.

[20] Xiaojun Dong, Andy Li, Yan Gu, and Yihan Sun. 2025. Parallel point-to-point shortest paths and batch queries. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures*. 458–472.

[21] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu. 2016. KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[22] Alex Fender, Brad Rees, and Joe Eaton. 2022. Rapids cugraph. In *Massive Graph Analytics*. Chapman and Hall/CRC, 483–493.

[23] Samuel Ferraz, Vinicius Dias, Carlos HC Teixeira, George Teodoro, and Wagner Meira. 2022. Efficient strategies for graph pattern mining algorithms on gpus. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 110–119.

[24] Zhisong Fu, Harish Kumar Dasari, Bradley Bebee, Martin Berzins, and Bryan Thompson. 2014. Parallel breadth first search on GPU clusters. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 110–118.

[25] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: eXploring runtime optimizations for breadth-first search on GPUs. In *Proceedings of the 28th International symposium on high-performance parallel and distributed computing*. 121–131.

[26] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[27] Xin He, Yapeng Yao, Zhiwen Chen, Jianhua Sun, and Hao Chen. 2021. Efficient parallel A* search on multi-GPU system. *Future Generation Computer Systems* 123 (2021), 35–47.

[28] Galen C Hunt, Maged M Michael, Srinivasan Parthasarathy, and Michael L Scott. 1996. An efficient algorithm for concurrent priority queue heaps. *Inform. Process. Lett.* 60, 3 (1996), 151–157.

[29] Wen-mei W Hwu, David B Kirk, and Izzat El Hajj. 2022. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.

[30] Seunghwa Kang, Chuck Hastings, Joe Eaton, and Brad Rees. 2023. cuGraph C++ primitives: vertex/edge-centric building blocks for parallel graph computing. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 226–229.

[31] Bernhard Kerbl, Michael Kenzel, Joerg H Mueller, Dieter Schmalstieg, and Markus Steinberger. 2018. The broker queue: A fast, linearizable fifo queue for fine-granular work distribution on the gpu. In *Proceedings of the 2018 International Conference on Supercomputing*. 76–85.

[32] Bernhard Kerbl, Jörg Müller, Michael Kenzel, Dieter Schmalstieg, and Markus Steinberger. 2018. A scalable queue for work distribution on gpus. *ACM SIGPLAN Notices* 53, 1 (2018), 401–402.

[33] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

[34] Hang Liu, H Howie Huang, and Yang Hu. 2016. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*. 403–416.

[35] Itay Lotan and Nir Shavit. 2000. Skiplist-based concurrent priority queues. In *International Parallel and Distributed Processing Symposium*. 263–268.

[36] Lijuan Luo, Martin Wong, and Wen-mei Hwu. 2010. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th design automation conference*. 52–55.

[37] Enrico Mastrostefano and Massimo Bernaschi. 2013. Efficient breadth first search on multi-GPU systems. *J. Parallel and Distrib. Comput.* 73, 9 (2013), 1292–1305.

[38] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.

[39] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[40] Takuji Mitsuishi, Jun Suzuki, Yuki Hayashi, Masaki Kan, and Hideharu Amano. 2016. Breadth first search on cost-efficient multi-GPU systems. *ACM SIGARCH Computer Architecture News* 43, 4 (2016), 58–63.

[41] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[42] Mhd Ghaith Olabi, Juan Gómez Luna, Onur Mutlu, Wen-mei Hwu, and Izzat El Hajj. 2022. A compiler framework for optimizing dynamic parallelism on GPUs. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–13.

[43] Anthony Pajot, Loïc Barthe, Mathias Paulin, and Pierre Poulin. 2011. Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 315–324.

[44] Yuechao Pan, Roger Pearce, and John D Owens. 2018. Scalable breadth-first search on a GPU cluster. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1090–1101.

[45] John A Pavlik, Edward C Sewell, and Sheldon H Jacobson. 2021. Two new bidirectional search algorithms. *Computational Optimization and Applications* 80, 2 (2021), 377–409.

[46] Wim Pijls and Henk Post. 2008. *A new bidirectional algorithm for shortest paths*. Technical Report.

[47] Md Sabbir Hossain Polak, David Troendle, and Byunghyun Jang. 2024. Agile Queue: A Fast Scalable Concurrent FIFO Queue on GPU. In *Workshop Proceedings of the 53rd International Conference on Parallel Processing*. 108–109.

[48] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[49] Andre Silva, Fernando Rocha, Artur Santos, Geber Ramalho, and Veronica Teichrieb. 2011. GPU pathfinding optimization. In *2011 Brazilian Symposium on Games and Digital Entertainment*. IEEE, 158–163.

[50] N. Sturtevant. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games* 4, 2 (2012), 144 – 148. http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

[51] Nathan Sturtevant and Ariel Felner. 2018. A brief history and recent achievements in bidirectional search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[52] Håkan Sundell and Philippas Tsigas. 2005. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel and Distrib. Comput.* 65, 5 (2005), 609–627.

[53] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T Kandemir, and Chita R Das. 2017. Controlled kernel launch for dynamic parallelism in GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 649–660.

[54] Lalinthip Tangjittaweechai, Mongkol Ekpanyapong, Thaisiri Watewai, Krit Athikulwongse, Sung Kyu Lim, and Adriano Tavares. 2016. Fast bidirectional shortest path on GPU. *IEICE Electronics Express* 13, 6 (2016), 20160036–20160036.

[55] Koji Ueno and Toyotaro Suzumura. 2013. Parallel distributed breadth first search on GPU. In *20th Annual International Conference on High Performance Computing*. IEEE, 314–323.

[56] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2016. Laperm: Locality aware scheduler for dynamic parallelism on gpus. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 583–595.

[57] Jin Wang and Sudhakar Yalamanchili. 2014. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 51–60.

[58] Kai Wang, Don Fussell, and Calvin Lin. 2021. A fast work-efficient sssp algorithm for gpus. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 133–146.

[59] 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*. 1–12.

[60] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), 1–49.

[61] Hancheng Wu, Da Li, and Michela Becchi. 2016. Compiler-assisted workload consolidation for efficient dynamic parallelism on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 534–543.

[62] Zicang Xu and Lei Zou. 2025. BUG: Balanced DFS-Based Subgraph Matching with a ReUse Strategy on GPUs. In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[63] Peter Yamout, Karim Barada, Adnan Jaljuli, Amer E Mouawad, and Izzat El Hajj. 2022. Parallel vertex cover algorithms on gpus. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 201–211.

[64] Yi Yang and Huiyang Zhou. 2014. CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications. *ACM SIGPLAN Notices* 49, 8 (2014), 93–106.

[65] Lyuheng Yuan, Da Yan, Jiao Han, Akhlaque Ahmad, Yang Zhou, and Zhe Jiang. 2024. Faster depth-first subgraph matching on gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3151–3163.

[66] Guozheng Zhang, Gilead Posluns, and Mark C Jeffrey. 2024. Multi bucket queues: Efficient concurrent priority scheduling. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. 113–124.

[67] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing ordered graph algorithms with GraphIt. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 158–170.

[68] Yuan Zhang, Huawei Cao, Jie Zhang, Yiming Sun, Ming Dun, Junying Huang, Xuejun An, and Xiaochun Ye. 2023. A bucket-aware asynchronous single-source shortest path algorithm on gpu. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*. 50–60.

[69] Yichao Zhou and Jianyang Zeng. 2015. Massively parallel A* search on a GPU. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 29.