Accelerating SSSP for Power-Law Graphs

Yuze Chi
chiyuze@cs.ucla.edu
University of California, Los Angeles

Licheng Guo
lcguo@cs.ucla.edu
University of California, Los Angeles

Jason Cong
cong@cs.ucla.edu
University of California, Los Angeles

ABSTRACT

The single-source shortest path (SSSP) problem is one of the most important and well-studied graph problems widely used in many application domains, such as road navigation, neural image reconstruction, and social network analysis. Although we have known various SSSP algorithms for decades, implementing one for large-scale power-law graphs efficiently is still highly challenging today, because ① a work-efficient SSSP algorithm requires priority-order traversal of graph data, ② the priority queue needs to be scalable both in throughput *and* capacity, and ③ priority-order traversal requires extensive random memory accesses on graph data.

In this paper, we present SPLAG to accelerate SSSP for power-law graphs on FPGAs. SPLAG uses a coarse-grained priority queue (CGPQ) to enable high-throughput priority-order graph traversal with a large frontier. To mitigate the high-volume random accesses, SPLAG employs a customized vertex cache (CVC) to reduce off-chip memory access and improve the throughput to read and update vertex data. Experimental results on various synthetic and real-world datasets show up to a 4.9× speedup over state-of-the-art SSSP accelerators, a 2.6× speedup over 32-thread CPU running at 4.4 GHz, and a 0.9× speedup over an A100 GPU that has 4.1× power budget and 3.4× HBM bandwidth. Such a high performance would place SPLAG in the 14th position of the Graph 500 benchmark for data intensive applications (the highest using a single FPGA) with only a 45 W power budget. SPLAG is written in high-level synthesis C++ and is fully parameterized, which means it can be easily ported to various different FPGAs with different configurations. SPLAG is open-source at `https://github.com/UCLA-VAST/splag`.

CCS CONCEPTS

• **Theory of computation → Shortest paths**; • **Computer systems organization → Reconfigurable computing**; **High-level language architectures**.

KEYWORDS

SSSP, priority queue, cache, power-law, graph, FPGA, HLS

ACM Reference Format:
Yuze Chi, Licheng Guo, and Jason Cong. 2022. Accelerating SSSP for Power-Law Graphs . In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22), February 27-March 1, 2022, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. `https://doi.org/10.1145/3490422.3502358`

FPGA '22, February 27-March 1, 2022, Virtual Event, CA, USA

ACM ISBN 978-1-4503-9149-8/22/02.
`https://doi.org/10.1145/3490422.3502358`

1 INTRODUCTION

The graph is a universal data structure that models relationships, connections, and structures. The single-source shortest path (SSSP) problem, one of the most important and well-studied graph problems, finds its prevalent application in road navigation [25], telecom network routing [47], neural image reconstruction [39, 41], and social network analysis [4]. Although we have known Dijkstra's algorithm [22] and its priority queue–based variants [23, 31] for several decades, these algorithms are not easily parallelizable, because increasing parallelism is often at the cost of increasing the total amount of work as well. As such, efficient parallelization of SSSP algorithms are still an active field of research [18, 35, 42, 51, 54, 57].

Compared with CPUs and GPUs, FPGAs have the unique capability of customizing the control flow and data paths, which has demonstrated tremendous potential in various application domains, including stencil computations [7, 8, 19, 38], neural networks [33, 52, 56], and general graph algorithms [5, 28, 55]. This makes the FPGA a naturally good candidate platform for SSSP acceleration, since the high-throughput on-chip priority queues [2, 36] enable effective control over the trade-off between parallelism and the amount of work [1, 35]. However, such on-chip priority queue–based approach has been applied only to uniform-degree planar graphs, yet many real-world graphs have skewed degree distributions, which are often modeled using the power law [14]. Compared with planar graphs, power-law graphs have a much larger frontier of active vertices, which requires a priority queue with a much larger capacity. Even worse, such a capacity requirement increases rapidly as the size of graph grows, making it infeasible to keep the priority queue on-chip. This is demonstrated in Figure 1.

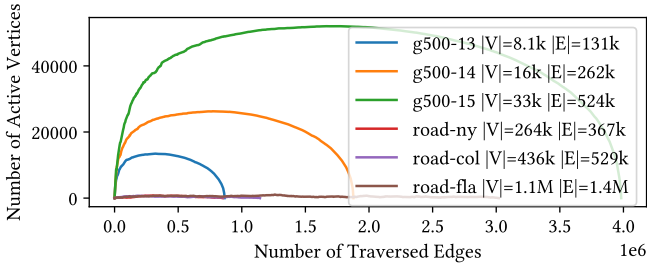

Figure 1: Change of the number of active vertices as edges are traversed. The g500 graphs are power-law graphs and the road graphs are planar graphs.

Another challenge to efficiently implement priority queue–based SSSP algorithms is that priority-order graph traversal prohibits many reordering techniques used in many graph accelerators [5, 16, 17, 28, 49, 58], which are vitally important to reducing external memory traffic and achieving high performance. As such, many graph accelerators [5, 28, 58] implement the Bellman-Ford algorithm [50] that does not require a priority queue at all. However, these accelerators work best for algorithms whose amount of work

is insensitive to the traversal order (e.g., SpMV and PageRank). For SSSP, the total amount of traversed edges (i.e., amount of work) can be very different with different traversal orders. We will show in Section 4.4 that, while the Bellman-Ford algorithm is good for parallelization and raw traversal throughput, its highly redundant edge traversal leads to a poorer overall performance for SSSP.

In this paper, we present a new architecture called SPLAG to accelerate SSSP for power-law graphs. Our contributions include:

- We present a coarse-grained priority queue (CGPQ) that manages the off-chip memory with an on-chip priority queue and on-chip buffers. The CGPQ enables high-throughput priority-order traversal for large-scale power-law graphs.
- We design a customized vertex cache (CVC) to reduce the amount of random off-chip memory traffic required by the priority-order graph traversal. By providing application-specific *push* and *pop* operations instead of general-purpose *read* and *write* operations, the CVC also reduces the amount of on-chip memory traffic.
- We implement a parallel variant of Dijkstra's algorithm on the FPGA with both high graph traversal throughput and high work-efficiency using the CGPQ and the CVC. Written in high-level synthesis (HLS) and open-source at `https://github.com/UCLA-VAST/splag`, one can easily port SPLAG to another FPGA.
- We evaluate SPLAG using both synthetic and real-world datasets and compare it with state-of-the-art SSSP systems. Using all 32 HBM channels on the Alveo U280 FPGA, SPLAG demonstrates up to 763 MTEPS throughput, a 4.9× speedup over state-of-the-art accelerators, a 2.6× speedup over multi-thread CPU, and a 0.9× speedup over an A100 GPU that has 4.1× power budget and 3.4× HBM bandwidth. Evaluated using the Graph 500 [46] benchmark for data-intensive applications, SPLAG could be placed in the 14th position with only a 45 W power consumption.

2 BACKGROUND AND RELATED WORK

Given a directed graph[1] $G = (V, E)$ where each edge $e_{i,j} \in E$ has a non-negative[2] weight $w_{i,j} \geq 0$, a *path* P is a sequence of vertices $P = (v_1, \cdots, v_n) \in V \times \cdots \times V$ such that v_i and v_{i+1} are connected by an edge $e_{i,i+1} \in E$ for $1 \leq i < n$. Such a path is called a path from $u = v_1$ to $v = v_n$. The *shortest path* between u and v is the path that minimizes the *distance* from u to v, i.e., $\sum_{i=1}^{n-1} w_{i,i+1}$. The *single-source shortest path* (SSSP) problem aims to find the shortest path from a given vertex u (called the *root*) to all vertices in the graph. That is to say, for each vertex $v \in V$, we not only need to find the shortest distance from u, but also the sequence of vertices that connects u to v. This can be effectively represented by storing the parent (v_{i-1}) of each v_i in the output.

2.1 Single-Source Shortest Path Algorithms

Dijkstra's algorithm [22] keeps two sets of vertices, the *visited* set and the *active* set. Initially, the visited set is empty, and the active set contains only the root vertex. All vertices are initialized with a *tentative* distance of ∞, except that the root has distance 0. The algorithm iteratively removes vertex u with the minimum distance from the active set, traverses the neighbors of u, and moves u to the visited set. For each neighbor v of u, a new tentative distance can be generated by adding the edge weight to the tentative distance of their parent vertex u. If this new tentative distance is smaller than the previously known distance, v will get a new tentative distance. If v is not in the active set nor the visited set, it will be moved to the active set. This compare-and-update operation is called *relaxation*. The algorithm terminates when the active set is empty.

The original Dijkstra's algorithm uses a list to store the active vertices, which necessitates $\Theta(|V|)$ time to find the minimum-distance vertex. This can be improved by using a priority queue to store the active vertices, which reduces the time complexity of this step to $\Theta(\log |V|)$ [23, 31]. If all weights are small integers bound by a constant C, Dial's algorithm [21] can further decrease this time complexity to $\Theta(C)$ with a bucket queue. Dijkstra's algorithm and its priority queue–based variants guarantee each edge is visited *at most once*, however, at the cost of being hard to parallelize, since edges from only one vertex can be relaxed at a time.

The *Bellman-Ford algorithm* [50] employs a different and parallelizable approach to solve the SSSP problem. Instead of selecting the edges from the minimum active vertex for relaxation, this algorithm traverses and relaxes all edges iteratively. Allowing parallel relaxation on all vertices enables massive parallelism, although doing so will relax each edge many times and thus is work-inefficient. Unlike Dijkstra's algorithm and its variants, the Bellman-Ford algorithm can handle negative weights and detect negative cycles. However, its worse-case time complexity of $\Theta(|V||E|)$ makes it highly inefficient when all the edge weights are non-negative, which is quite common in real-world applications.

The trade-off between parallelism and work-efficiency has motivated many researchers. The "eager" Dijkstra's algorithm [20] exposes more parallelism by relaxing edges in parallel from more than one vertex with minimal distances. Crauser et al. [15] further define heuristics to decide edges from how many vertices should be relaxed in parallel. *Δ-stepping* [42] and its variants [18, 54] generalize Dial's algorithm [21] by dividing the active vertices into buckets based on their distances and only select active vertices from the first non-empty bucket with the smallest distances.

2.2 Other Shortest Path Problems

The single-source shortest path problem is not the only possible type of shortest path problems. In fact, we can define four shortest path problems on a given graph G:

- The *single-pair shortest path* problem finds the shortest path from a given source vertex u to a given destination vertex v.
- The *single-source shortest path* problem finds the shortest path from a given vertex u to all vertices in the graph.
- The *single-destination shortest path* problem finds the shortest path from all vertices to a given vertex v in the graph.
- The *all-pairs shortest path* problem finds the shortest path between all pairs of vertices.

The single-pair shortest path problem can be solved using Dijkstra's algorithm with an early termination condition. For multiple single-pair shortest path queries on the same graph, one can solve them more efficiently by pre-computing the SSSP of some

[1] An undirected graph is modeled as a directed graph with bidirectional edges.
[2] Dijkstra's algorithm and its variants, including SPLAG, cannot be used on negative-weighted graphs. In practice, the edge weights represent distances, and are often non-negative by definition, e.g., network latency, strength of connection, etc.

selected landmarks [25]. The single-destination shortest path problem can be reduced to SSSP by reverting the direction of edges. The all-pairs shortest path problem utilizes a different algorithm than SSSP [3]. However, due to its $\Theta(|V|^3)$ time complexity, a complete solution to the all-pairs shortest path problem is often computationally intractable for large-scale graphs, in which cases SSSP of selected/sampled vertices can be used [4]. Altogether, solving the SSSP problem efficiently can be helpful for all four types of shortest path problems. We will focus on SSSP in the rest of this paper.

2.3 Single-Source Shortest Path Accelerators

2.3.1 Dijkstra's Algorithm Accelerators. Takei et al. [51] accelerates the original Dijkstra's algorithm and parallelizes both relaxation and the linear search for the minimum active vertex with a SIMD architecture. Lei et al. [35] implements Dijkstra's algorithm with an on-chip systolic priority queue [36]. Since the systolic priority queue operates on every data element on each clock cycle, it cannot leverage dense on-chip storage elements (e.g., BRAMs) and its capacity does not scale well. A two-level linear-search structure is used when the number of active vertices grows beyond the capacity of the queue. Chronos [1] exploits massive speculative parallelism and can implement the eager version of Dijkstra's algorithm efficiently. Chronos uses an on-chip pipelined heap [2] to store the active vertices, which scales better than the systolic priority queue but still is limited by the size of on-chip storage. Only planar graphs are evaluated for the above accelerators.

2.3.2 Bellman-Ford Algorithm Accelerators. HitGraph [58] and its earlier version [57] implement an edge-centric graph accelerator. Leveraging the larger sequential bandwidth, HitGraph writes the intermediate relaxation results to DRAM when generated and reads them back when needed. ThunderGP [5] is an HLS-based graph processing template that implements highly-parallel graph accelerators under the vertex-centric gather-apply-scatter model. Unlike HitGraph, ThunderGP updates on-chip vertices directly without generating off-chip intermediate results. GraphLily [28] is an HLS-based graph linear algebra overlay implemented on an FPGA equipped with high-bandwidth memory (HBM). GraphLily can implement the Bellman-Ford algorithm along with other graph linear algebra algorithms without reprogramming the FPGA.

In summary, all Dijkstra's algorithm accelerators are evaluated using uniform-degree planar graphs only and cannot handle power-law graphs well due to the demanding requirement of the priority queue capacity (Figure 1). The Bellman-Ford algorithm accelerators are evaluated using large-scale power-law graphs, but their work-efficiency is overlooked. Only the raw edge traversal throughput is reported, which demonstrates the performance of the graph processing system, but not the algorithm itself. Even worse, to reduce the bandwidth requirement, none of the accelerators records the parent vertex together with the shortest distance. Without the parent vertex as part of the output, we will not be able to construct the shortest path tree out of the result, which reduces the usefulness of the result. Table 1 summarizes the related work.

3 THE SPLAG ACCELERATOR

SPLAG aims to enable high-throughput and work-efficient SSSP queries for large-scale power-law graphs. This is achieved using the architecture shown in Figure 2. The whole SPLAG accelerator is composed of three major components:

Table 1: Summary of related work. MTEPS measures the algorithm throughput, which is defined as the number of undirected edges in the connected component divided by the execution time. Since some systems did not report the execution time, an upper-bound estimation (Section 4.4) is listed here.

System	Lang.	Work-eff.?	Power-law?	Priority queue?	Vertex cache?	MTEPS
Chronos [1]	RTL	Yes	No	P-heap [2]	App.-agnostic	360
GraphLily [28]	HLS	No	Yes	No	Scratchpad	<232
HitGraph [58]	RTL	No	Yes	No	Scratchpad	46.9
Lei et al. [35]	RTL	Yes	No	ExSAPQ [35]	No	9.2
Takei et al. [51]	RTL	Yes	No	No	On-chip only	0.4
ThunderGP [5]	HLS	No	Yes	No	Scratchpad	<122
SPLAG	HLS	Yes	Yes	CGPQ (Sec. 3.2)	CVC (Sec. 3.3)	763

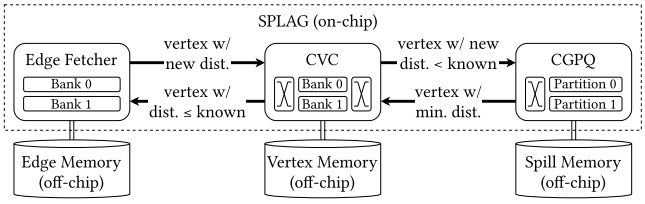

Figure 2: Architecture overview of the SPLAG accelerator.

- The *coarse-grained priority queue (CGPQ)* implements a high-throughput bucket-based priority queue that is scalable to a large capacity by buffering active vertices on-chip and storing excessive vertices in the off-chip *spill memory* as fixed-size chunks. Section 3.2 will provide more details about the CGPQ.
- The *customized vertex cache (CVC)* provides high-throughput access to the vertex data, which are initially stored in the off-chip *vertex memory*. Unlike a standard cache with read and write interfaces, the CVC provides application-specific interfaces for *updating* vertices and *filtering* redundant updates. Section 3.3 will review the internals of the CVC.
- The *edge fetcher (EF)* traverses neighbors of an active vertex and calculates the new tentative distance. The off-chip *edge memory* stores the edge list in the compressed sparse row (CSR) format. Section 3.4 will discuss the edge fetcher.

To enable concurrent processing, we partition all the three major components internally. We use multi-stage switch networks [34] to improve the clock frequency without sacrificing the throughput [11] when all-to-all concurrent communication is required. Besides the three major components, the SPLAG accelerator also contains a dispatcher responsible for injecting the first active vertex, controlling program termination, and collecting statistics. The host program initializes the vertex and edge memory.

3.1 The SPLAG Algorithm

The SPLAG architecture implements a variant of Dijkstra's algorithm, shown in Algorithm 1. The algorithm is designed to expose as much parallelism as possible while minimizing the amount of work. It exploits two levels of parallelism by ① relaxing edges from multiple active vertices at the same time (Line 4 in Algorithm 1), and ② relaxing multiple edges of the same active vertex at the same time (Line 7 in Algorithm 1). Moreover, SPLAG executes the

Algorithm 1 SPLAG's variant of Dijkstra's algorithm.

Require: A graph $G = (V, E)$ and $root \in V$
Ensure: $vertices$ represent the shortest-path tree from $root$

```
1:  vertices = [{dist = ∞, parent = null}, ···]
2:  vertices[root] = [{dist = 0, parent = root}]
3:  queue = [{id = root, dist = 0, parent = root}]
4:  while not queue.empty() in parallel do
5:    u = queue.pop()                                   ▷ CGPQ
6:    if u.dist ≤ vertices[u.id].dist then              ▷ CVC
7:      for all e = u.id → vid ∈ E in parallel do       ▷ Edge Fetcher
8:        if vid ≠ u.parent then                        ▷ Edge Fetcher
9:          d = u.dist + e.weight                       ▷ Edge Fetcher
10:         if d < vertices[vid].dist then[3]           ▷ CVC
11:           vertices[vid] = {dist = d, parent = u.id} ▷ CVC
12:           queue.push({id = vid, dist = d, parent = u.id}) ▷ CGPQ
```

algorithm asynchronously, which means the next iteration of the outer loop (Line 4) can start before the previous one finishes, avoiding load imbalance caused by skewed degree distribution. This, however, makes it possible to terminate the program prematurely because the *queue* may be temporarily empty before active vertices are pushed to the *queue* in Line 12. To solve this problem, we delay the program termination by a short, fixed amount of clock cycles to make sure any in-progress operation has been completed.

Highly parallel execution of Dijkstra's algorithm may lead to highly redundant amount of work. That is, the number of edge traversal may be greater than the number of edges in the connected component. SPLAG reduces the amount of work using the conditional statement shown in Line 6 of Algorithm 1. It can filter out vertices that are updated many times. For example, for an SSSP query from root vertex A on the graph shown in Figure 3, vertex A generates a path to D with a tentative distance of 8 and vertex B generates a path to D with a tentative distance of 3+2=5. Without Line 6 in Algorithm 1, neighbors of vertex D will be traversed twice because D will be popped twice in Line 5 with two different distances. With Line 6 and the priority queue, vertex D with the smaller tentative distance 5 will be popped first, and the second pop will be filtered out because a smaller distance is already known.

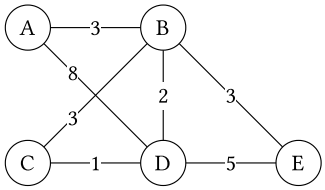

Figure 3: A graph with 5 vertices and 7 edges.

To further reduce redundant edge traversal, SPLAG applies another optimization named *never-look-back*. Noticing that a vertex always has a smaller distance than its children in the SSSP tree, SPLAG skips the parent of a vertex v when it traverses the neighbors of v. For example, for an SSSP query from root vertex A in Figure 3, A generates a path to B with tentative distance 3 and parent A. When SPLAG traverses neighbors of B, it will skip A and only traverse C, D, and E, since A is the parent of B, and we know A must already have a smaller distance than B.

[3]Line 10 and Line 11 must be atomic. The CVC takes care of this in SPLAG.

3.2 The Coarse-Grained Priority Queue

A high-throughput and work-efficient SSSP accelerator for power-law graphs requires high-throughput priority-order graph traversal. Therefore, there are two design objectives for the priority queue: ① the priority queue must have a **large capacity** and utilize off-chip memory efficiently, and ② the priority queue must support **high throughput** push and pop operations. In this section, we present our solution named the coarse-grained priority queue (CGPQ). Noticing that a strict priority queue exposes too little parallelism and is not necessary for correctness, we take a coarse-grained *bucket*-based approach to achieve the two design objectives. Using a pre-selected Δ, we can divide the active vertices into many buckets based on the distance from the root, and, e.g., store a vertex with tentative distance d in bucket $\left\lfloor \frac{d}{\Delta} \right\rfloor$. Vertices in the same bucket are considered to have the same priority and can be accessed in simple first-in-first-out (FIFO) order.

While it seems trivial to implement such a simple bucket-based CGPQ, the dynamic nature of the SSSP problem actually imposes significant challenges: the maximum size of each bucket is unknown before the program execution. While we can pessimistically reserve consecutive memory space for each bucket, doing so will likely result in a significant waste of memory since the overall utilization of memory would be low, which limits the scalability in terms of capacity. Figure 4 shows an example of how the sizes of 32 buckets change as edges are traversed. We can see that different buckets are utilized differently. If we reserve memory based on the maximum size of all the buckets, 63% of the reserved memory will be unnecessary. In fact, we must reserve even more because we must account for the worse case among all SSSP queries on all datasets. To avoid such memory waste, one can employ a linked list to allocate memory space dynamically, but such a data structure not only has the storage overhead for the node pointers, but also is slow due to random accesses. A commonly used data structure that achieves a compromise between a fixed-size array and a linked list is often called a double-ended queue (*deque*), which is a linked list of fixed-size arrays. The use of linked lists, however, are still inefficient for implementation on FPGA.

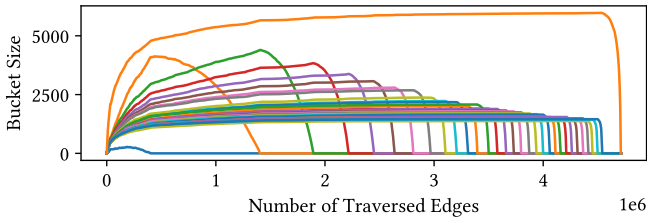

Figure 4: Sizes of 32 buckets as edges are traversed in the g500-15 dataset. Each line represents a bucket.

The CGPQ is inspired by the deque data structure. Similar to a deque, the CGPQ manages off-chip active vertices in a unit of fixed-size *chunks*. Unlike a deque, the CGPQ manages the position and priority of the chunks with an on-chip priority queue, instead of linked lists. Figure 5 demonstrates how the on-chip *chunk priority queue* (CPQ) orchestrates the off-chip memory accesses to enforce the priority ordering of vertices. While the example shows 4-vertex chunks, in practice, the chunks are typically hundreds or thousands of vertices large to make sure the on-chip priority queue does not

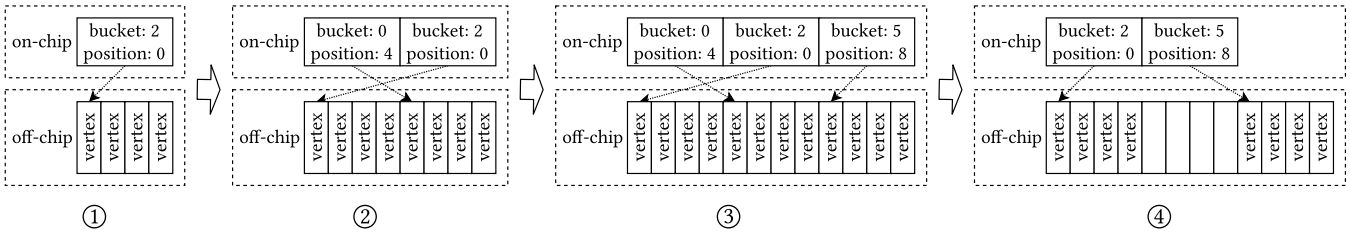

Figure 5: An example of the CGPQ orchestrating chunks of vertices. ① Initially the CGPQ contains one chunk of four vertices stored off-chip and its reference stored on-chip. The on-chip reference stores the bucket number and a pointer to the off-chip memory position. ② Another chunk of four vertices is added. The new chunk belongs to Bucket 0 and thus has higher priority. Therefore, it is stored on-chip at a position with a higher priority. The on-chip reference of the old chunk is moved to a position with a lower priority. Meanwhile, the off-chip memory only needs to append the newly added vertices without moving existing ones. ③ Another chunk for Bucket 5 is added. ④ The chunk with the highest priority is removed. The chunk reference is popped from the on-chip priority queue and the pointer is used to read the vertices from the off-chip memory. On-chip chunk references are reorganized to maintain the priority queue structure while off-chip data are not moved.

overflow. Therefore, the off-chip memory is always accessed in large chunk of vertices, which guarantees high memory bandwidth utilization. As such, the CGPQ can scale to a large capacity without a significant waste of the memory space or bandwidth.

Figure 6 shows the architectural overview of a CGPQ with two push ports and two pop ports, which allows two vertices to be pushed and two vertices to be popped in parallel. Each input vertex will be assigned a bucket by the *bucket assigner* using a pre-selected Δ. The CGPQ then buffers on-chip at least one chunk of active vertices for each bucket, enabling high-throughput push and pop operations. Section 3.2.1 discusses more details about the *chunk buffer* and its two-level partitioning mechanism, which further enables concurrent operations and makes it possible to achieve our high-throughput design objective. When the buffer is (almost) full, vertices will be offloaded to the off-chip *spill memory* as a whole chunk. The *chunk priority queue* orchestrates the chunks between the on-chip chunk buffer and the off-chip spill memory, as demonstrated in Figure 5. Section 3.2.3 further discusses how we dynamically and collaboratively schedule the off-chip operations together with the on-chip push and pop operations. Such dynamic management allows the memory space to be used in a compact way, making it possible to achieve our large-capacity design objective.

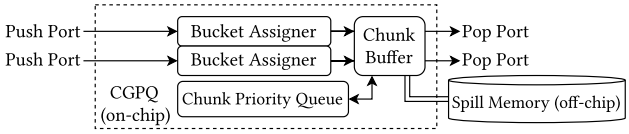

Figure 6: A CGPQ with two push ports and two pop ports. The number of ports can be different for push and pop and is larger than two in the actual design (Table 3 on page 8).

3.2.1 The Chunk Buffer. The chunk buffer is the key to achieving highly concurrent push and pop throughput while supporting large queue capacity. As a priority queue, the two basic operations are *push* and *pop*. To achieve the large-capacity design objective, the chunk buffer must additionally support the *spill* and *refill* operations to store excessive vertices in the off-chip *spill memory*. The spill operation moves a chunk of vertices from the on-chip chunk buffer to the off-chip spill memory. The refill operation moves a chunk of vertices from the off-chip spill memory back to refill the on-chip chunk buffer. To achieve the high-throughput design objective, all four operations must be able to parallelize.

To support concurrent push operations, the chunk buffer is internally partitioned by buckets so that different buckets can be accessed in parallel. This is called *inter-bucket parallelism*. Figure 7 shows the architecture of the chunk buffer. Since each vertex may belong to any *bucket partition*, this requires an all-to-all communication pattern. After each incoming active vertex is assigned a bucket and is sent to the chunk buffer, it will first be routed through the switch network based on its bucket partition ID.

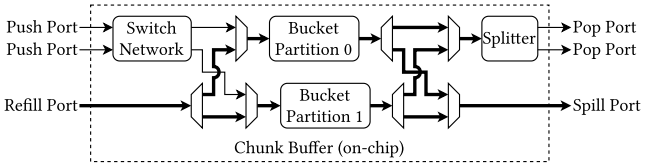

Figure 7: The chunk buffer in Figure 6. Data paths in bold transfer multiple data elements in lockstep.

Each bucket has its own on-chip storage in the chunk buffer called the *bucket buffer (BB)*. Each BB is accessed in FIFO order as a circular buffer. The chunk buffer maintains the write and read pointers of each BB. Figure 8 shows the data layout of the chunk buffer in Figure 7. The numbers in brackets are the array indices of each vertex in each BB. For example, the first three active vertices assigned to Bucket 0 will be written to memory positions `[0]`, `[1]`, and `[2]` in BB 0 in three clock cycles, which can happen in parallel when Bucket 1 or Bucket 3 (but not Bucket 2) is being written.

In Figure 7, only one vertex may be routed to each bucket partition. While each bucket partition can in fact consume multiple vertices in each clock cycle, we do not exploit the parallelism to push vertices with each bucket partition. The rationale is as follows. On the one hand, we observe that the incoming vertices are roughly evenly distributed among all buckets in the beginning of execution where the push operations are the most intensive. Figure 4 shows such an example: almost all bucket sizes increase rapidly before the bucket sizes hit ~500. The switch network can further absorb temporary unbalances. Therefore, pushing only one vertex to each bucket partition does not impose a significant throughput decrease. On the other hand, we observe that unlike pop operations (which

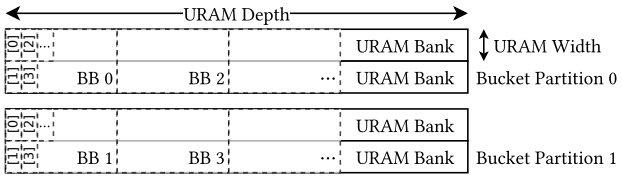

Figure 8: Data layout of the chunk buffer in Figure 7. Each push port requires a bucket partition and each pop port requires a URAM bank, so there are two bucket partitions and each bucket partition has two banks. Each bucket buffer (BB) is used as a circular buffer. The numbers in brackets are the array indices of the vertices in each BB.

we will discuss later), push operations cannot be coalesced and aligned, since we can never know when/if the next vertex to the same bucket will arrive. Therefore, each incoming vertex may fall in any bank in the bucket partition. Unlike the routing problem among different bucket partitions whose destination is determined solely by its distance and thus can use a multi-stage switch network, the bank ID to which a vertex shall be written is determined by the runtime conditions in the buffer. Restricting the push rate to each bucket partition not only reduces resource utilization, but also removes a potential critical path in the whole accelerator. As such, we only exploit inter-bucket parallelism for push operations.

For pop operations, since we only pop from a single non-empty bucket with the smallest distance, the inter-bucket parallelism among different bucket partitions cannot be exploited. Therefore, we further partition each bucket partition cyclically into *URAM banks* so that multiple vertices can be accessed at the same time. This is called the *intra-bucket parallelism*. With the intra-bucket parallelism, we can then pop multiple vertices from the same bucket in a single clock cycle. These vertices then go through coalesced data paths in lockstep and are eventually split into individual data paths. In Figure 7 and Figure 8, since there are two pop ports, each bucket partition is divided into two banks and the coalesced data paths are two-element-wide to match the data rate of pop operations.

Note that a bucket may have fewer valid vertices than the intra-bucket parallel factor. In such cases, *null vertices* padding will be filled in the coalesced data paths. The *splitter* in Figure 7 detects and removes the null vertices when sending the coalesced vertices into individual pop ports. To simplify the logic to determine the validity of popped vertices, we require that the pop operations are always aligned; null vertices will be written to the BB in case a pop operation performs a partial read. For example, let there be three valid vertices in Bucket 0 in Figure 8 stored in `[0]`, `[1]`, and `[2]`. The first pop operation will read two vertices from `[0]` and `[1]` respectively, which are aligned to the intra-bucket parallel factor 2. The second pop operation will only read one valid vertex from `[2]`. This is not an aligned operation. If we allow such unaligned operations and later a new vertex is written to `[3]` in Bucket 0, we will have to be able to read one single valid vertex from `[3]` while marking vertices from other banks null. This complicates the design. Instead, in case of unaligned operations, we will force alignment and adjust the write pointer in addition to the read pointer to fill in the unaligned locations. Using the same example above, when the second pop operation reads the vertex from `[2]`, it will move both the read pointer and the write pointer to `[4]` so that the next incoming vertex will be stored in `[4]` instead of `[3]`. Note that this alignment enforcement does not sacrifice the maximum capacity of each circular buffer. As such, unaligned pop operations will only insert null vertices in higher locations, which simplifies the pop operation logic without affecting other operations.

Intra-bucket parallelism enables not only concurrent pop operations, but also faster spill/refill operations by reading from/writing to all URAM banks in each bucket partition. Figure 7 shows the data paths for the spill and refill operations. Section 3.2.3 will discuss how the four operations are scheduled. Note that we cannot guarantee each spill/refill operation is aligned. For example, in Figure 8, the read pointer may point to position `[1]` when a spill operation is scheduled, which means the spill operation should read `[1]` and `[2]` in the first clock cycle. This is why we have to partition each bucket partition into individual memory banks instead of reshaping the data structure to use a single-bank memory with a wider width.

3.2.2 The Chunk Priority Queue. The chunk buffer is on-chip and limited in size. To accomplish the large capacity design objective, when a bucket buffer (BB) is almost full, it will *spill* to the off-chip memory. When spilling happens, a *chunk reference* will be created with the off-chip memory pointer and the bucket associated with that chunk. This chunk reference will be pushed into an on-chip *chunk priority queue* (CPQ) so that when the BB has enough space, the off-chip chunks can be *refilled* in priority order. Figure 5 shows an example of spilling and refilling. Since each chunk contains many vertices, the capacity of the CPQ can be much smaller than the whole CGPQ, and the CPQ can be on-chip only. Since the off-chip access has a long latency (> 100 ns [13]), a regular binary heap suffices for the CPQ. Other on-chip priority queue data structures such as the systolic priority queue or pipelined heap require more memory banks and are thus less efficient, so we do not use those.

3.2.3 Scheduling the Operations. The potential bank conflicts and multi-cycle operation of spilling and refilling bring challenges to schedule the four operations correctly without deadlock. Moreover, the spill memory is shared by all bucket partitions to maximize the utilization of the external memory. To avoid deadlock, the general scheduling rules are: ① always make sure multi-cycle operations, i.e., spilling and refilling, can finish without indefinite stalling. This not only simplifies the inter-operation dependency, but also helps to improve off-chip memory utilization since memory requests will not be stalled by the chunk buffer; ② prioritize push operations over pop operations since pop operations may generate more push operations; ③ each bucket partition only has one read port and one write port. Details for scheduling each operation are as follows.

The push operation is scheduled on a bucket partition when an incoming vertex is available on the push port, unless: ① The write port is occupied by an active refill operation. ② There is insufficient buffer space for the target bucket. This includes the case when the BB is full and the case when future refill operations exhaust the available space. For example, let each chunk contain 4 vertices and the BB can hold up to 8 vertices. A refill operation is scheduled when the BB only has 2 vertices. Push operations can be scheduled before refilling data are fetched from the off-chip memory (which takes many clock cycles), until there are 4 vertices in the BB. We will not schedule another push operation when the BB contains 4 vertices since if we do, the refill operation would stall indefinitely

when no pop operation is scheduled and the BB does not have sufficient space for the last vertex.

The pop operation is scheduled for the non-empty BB with the highest priority, unless: ① The output pop port is full. ② The read port is occupied by an active spill operation. ③ There are insufficient vertices. ④ The pop operation is unaligned and the write port is used by a push or refill operation.

A BB is selected for *spilling* if its size exceeds a pre-defined threshold (e.g., 3/4 BB capacity) and there is no spilling or refilling already scheduled (which occupies the off-chip memory). If multiple BBs are almost full, we start spilling the one with the lowest priority. Once a BB is scheduled for spilling, the whole chunk must be moved to the off-chip memory, which takes multiple clock cycles. That BB will continue the spilling operation in the following clock cycles unless the memory channel is busy.

The top bucket in the CPQ is selected for *refilling* if its size is below a pre-defined threshold (e.g., 1/4 BB capacity), and there is no spilling already scheduled. Since there is a long latency between the off-chip memory read request and the data response, we allow at most one refilling operation to be scheduled while another one is in process, which can help to hide this long latency. Once scheduled, the refilling operation will continue in the follow clock cycles unless the memory channel is not ready with the appropriate data.

The above scheduling mechanism guarantees that spilling and refilling operations will not block indefinitely. Moreover, given a finite number of push operations and properly chosen thresholds, spilling and refilling operations will be scheduled for finite times, all of which will eventually complete. Therefore, assuming the spill memory is sufficiently large, spill operations will eventually unblock push operations blocked by insufficient buffer space, so push operations will not block indefinitely either. As a result, the rest of the accelerator is always able to make progress, which will eventually unblock pop operations blocked by full output. This means none of the four operations will block indefinitely and the scheduling is deadlock-free.

3.3 The Customized Vertex Cache

With a high-throughput and large-capacity CGPQ, we still need a carefully designed accelerator that can keep up with the throughput. The priority-order graph traversal generates extensive random memory accesses that are infeasible to reorder for better off-chip bandwidth utilization, making it even more challenging to create a fast SSSP accelerator. We could employ a classic memory cache to mitigate the random accesses on the vertex data, but it would produce lower quality of results due to its application-agnostic nature. Noticing that the tentative distance of each vertex monotonically decreases, we can take advantage of this property and simplify the accelerator design. In SPLAG, we create the customized vertex cache (CVC) to help ① reduce the off-chip memory traffic by caching vertex data on-chip, and ② reduce on-chip memory requests by taking advantages of the fact that the tentative distance of each vertex is monotonically decreasing. The CVC provides two basic operations: ① The *updating* operation consumes as input a vertex with a new distance and its corresponding parent vertex ID. The CVC updates the tentative distance and the tentative parent of a vertex if and only if the input distance is smaller than the existing value. If the update happens, the updated vertex is pushed to the CGPQ. ② The *filtering* operation takes as input a vertex popped from the CGPQ. The CVC compares the tentative distance of the input and the existing value and checks if the input is "stale", i.e., its tentative distance has been updated to a smaller value. Only if the input vertex is not stale, will the CVC forward the input from the CGPQ to the edge fetcher to traverse its neighbors.

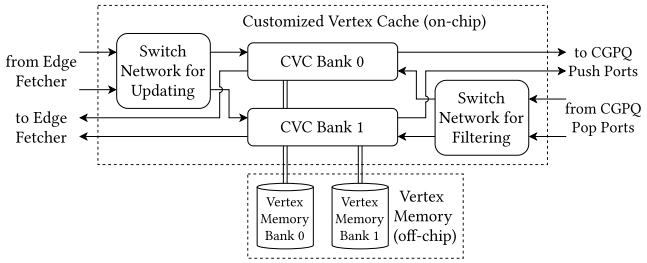

Figure 9: A customized vertex cache with two banks. Both the on-chip and off-chip memory are partitioned into banks. Vertices are cyclically assigned to each bank. The two switch networks route requests based on the bank ID.

Figure 9 shows the architecture of a CVC. For high-throughput memory accesses, the vertices are cyclically partitioned into multiple banks. Each CVC bank has two pairs of ports, one pair for updating and another for filtering. The updating ports are connected to the edge fetcher, which is responsible for generating vertices with new tentative distances. Since each vertex may have neighbors in any CVC bank, a switch network is used to route the updating inputs to the correct bank. Similarly, the filtering ports are connected to the CGPQ and a switch network is used to route the filtering inputs to the correct bank.

The CVC is fully pipelined. Each CVC bank can serve one request per cycle on hit. The initiation interval for miss requests is two, because it takes one cycle to send and another cycle to receive the off-chip memory request. The memory requests are pipelined, and their long latency can be hidden by overlapping them with each other. Each CVC bank implements a direct-mapped write-back cache. We do not employ a set-associative cache since the hit rate improvement does not make up the frequency degradation caused by its complexity. Each cache entry keeps a `dirty` bit to indicate whether its content should be written back on cache miss or program termination. Each entry also keeps a `writing` bit to indicate that its content is being written back, and another dirty cache miss must stall until the write finishes.

Coordinating memory reads from DRAM is more complicated than writes, because both update requests and filter requests can generate DRAM reads. On cache miss, we store the tentative distance and parent in the incoming vertex as the tentatively known data, and do not generate the output until the off-chip data are available. The CVC treats off-chip read caused by updating and filtering differently when they arrive, therefore each cache entry in the CVC has two different reading states. Figure 10 shows the finite-state machine of a CVC entry for memory reads. When an off-chip read for updating finishes, the CVC compares the distances and generates an update if the incoming vertex has a smaller distance. If another updating request arrives for the same vertex before the read data arrive, the cache entry is updated on-chip to keep only

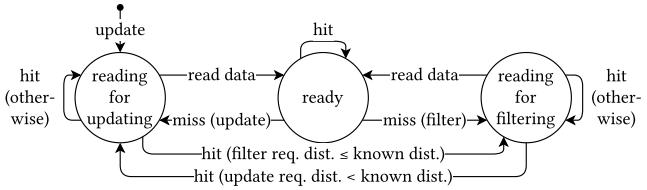

Figure 10: The finite-state machine for memory reads in the CVC. The initial invalid state can only transfer to reading for updating because each vertex cannot be filtered before being updated first. Dirty and writing states can be managed independent of the states for reading and are not included in the figure. Miss on reading will stall the request until the request until the entry is no longer reading, so there is no state transition from reading states on miss.

the smallest tentative distance. This application-specific optimization reduces on-chip memory traffic while hiding off-chip memory latency. When an off-chip read for filtering finishes, the CVC compares the distances and discards the request if the incoming vertex has a greater distance (which means the popped vertex is "stale"). If another filtering request arrives for the same vertex before the read data arrive, the cache entry is updated on-chip to keep only the smallest tentative distance. If an updating request arrives when an entry is reading for a filtering request, the CVC compares the distances and marks the purpose of the reading updating if the updating request has a smaller distance. This is because if the updating request has a smaller distance, the filtering request would become stale and should be discarded. Otherwise, the updating request is not generating an update and the filtering request should continue. Similarly, if a filtering request arrives when an entry is reading for an updating request, the CVC compares the distances and marks the purpose of the reading filtering if the filtering request's tentative distance is smaller than or equal to the updating request.

3.4 The Edge Fetcher

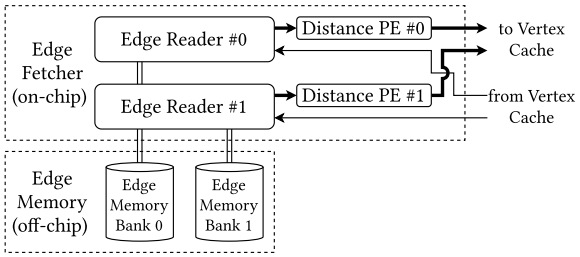

Figure 11: Edge fetcher with two banks. Each bank stores edges whose vertices in the corresponding vertex partition. Bold lines are coalesced data paths that transfer multiple vertices in lockstep.

The edge fetcher traverses neighbors of active vertices filtered by the CVC and calculates the new tentative distances of the neighbors. Figure 11 shows the architecture of the edge fetcher. The edge fetcher exploits two levels of parallelism: ① the edges are partitioned into multiple banks based on the source vertex ID so that neighbors of different vertices are traversed at the same time, and ② the edges are coalesced into wide vectors so that multiple neighbors are traversed at the same time. The edge fetcher is fully pipelined without turnaround time for different input vertices, meaning if there are no bubbles in the input vertices, the edge fetcher will not insert any bubbles to the output (unless the off-chip memory does not keep up with the data rate).

Table 2: Graph datasets evaluated on SPLAG. $d_{\max}$ and d_{avg} denote the maximum and average degree, respectively.

Dataset	$\|V\|$	$\|E\|$	$d_{\max}$	d_{avg}	Source
amzn	2.1M	5.8M	12k	2.7	Amazon product ratings [45]
dblp	540k	15M	3.3k	28	DBLP Paper Coauthors [24]
digg	872k	4.0M	31k	4.5	Users from digg.com [48]
flickr	2.3M	33M	34k	14	Flicker users [43]
g500-N	2^N	2^{N+4}	$2^{0.6N+5}$	16	Graph 500 datasets [46]
hlwd-09	1.1M	58M	12k	50	Actor collaboration [48]
orkut	3.0M	106M	28k	36	Orkut social network [48]
rmat-21	2.1M	91M	214k	44	A Kronecker graph [37]
wiki	274k	2.9M	3.4k	11	Wiki article–word graph [48]
youtube	3.2M	12.2M	130k	3.8	YouTube users [44]

4 EVALUATION

We evaluate SPLAG with both synthetic and real-world graph datasets. Table 2 shows the details of the datasets. All graphs are undirected. For each dataset, we sample 64 vertices that are connected to at least one other vertex and report the harmonic mean since the metrics are ratios. All experimental results are collected from on-board execution. Performance counters are inserted to the accelerator to collect the relevant metrics.

We implement SPLAG using an open-source extension to HLS C++, TAPA [10], to leverage the convenient peeking interfaces, fast software simulation [6, 12], asynchronous memory interfaces, simplified host-kernel interfaces, and coarse-grained floorplanning [26, 27]. Our implementation targets the Alveo U280 board with 32 high-bandwidth memory (HBM) channels. Table 3 summarizes the design parameters. We determine the design parameters as follows: to maximize the utilization of the switch networks, we only select powers of 2 for *#bank* and *#HBM*. We allocate as many *#HBM* as possible to CVC for its intensive random accesses, and evenly distribute the rest between EF and CGPQ. For CVC, *capacity/bank* maximizes the URAM utilization. For EF, *coalescing factor* matches *#bank* of CVC and EF. For CGPQ, *#port* matches *#bank* of CVC. *CPQ capacity* and *#bucket* are maximized without exceeding the timing critical path in CVC. *Chunk size* matches the capacity of HBM and CPQ. *BB capacity* doubles the chunk size. *Spill (refill) threshold* is empirically chosen as ¾ (¼) of *BB capacity*.

Table 3: Design parameters of the SPLAG accelerator.

CGPQ	#Push Port	16	BB Cap.	2048	#HBM	8
	#Pop Port	16	Spill Thre.	1536	#Bucket	128
	Chunk Size	1024	Refill Thre.	512	CPQ Cap.	256k
CVC	Capacity/Bank	64k	#Bank	16	#HBM	16
EF	Coalescing Fac.	2	#Bank	8	#HBM	8

We use Vitis 2021.1 for hardware implementation. The post-implementation reports suggest that the whole accelerator, including the Vitis shell platform, utilizes 75% CLBs, 6.3% DSPs, 14% BRAMs, and 83% URAMs with a 45W power budget (including

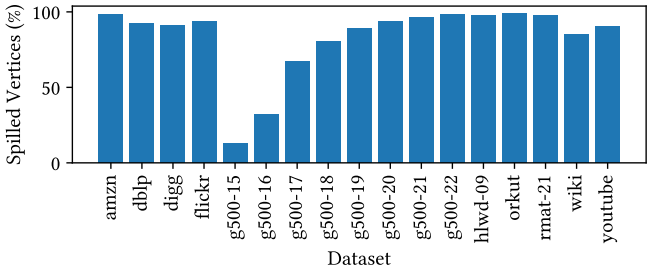

Figure 12: Percentage of spilled vertices among all vertices pushed to the CGPQ.

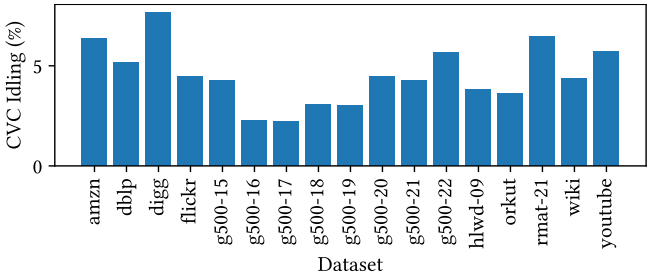

Figure 13: Percentage of CVC idling. Low idling suggests that the CGPQ can pop vertices with a high throughput.

the HBM). There are 162840 CLBs, 9024 DSPs, 2016 BRAMs, and 960 URAMs available. The accelerator is clocked at 130MHz with critical paths caused by the extensive usage of URAMs in the CVC.

4.1 Evaluation of the CGPQ

Figure 12 shows the percentage of spilled vertices among all vertices pushed to the CGPQ. We can see that for large datasets, almost all active vertices are spilled to the off-chip memory. Moreover, the scaling from g500-15 to g500-22 matches the trend of active vertices shown in Figure 1 on page 1. This suggests that our CGPQ design has accomplished the large-capacity design objective.

Figure 13 shows the percentage of idling cycles of the CVC. Note that CVC idling can be caused by either empty CGPQ or insufficient pop throughput; the performance counters cannot tell the reason for idling. Moreover, the CVC never stalls because the CGPQ push port is full in any of the evaluations. <8% CVC idling and 0% CVC stalling caused by the CGPQ suggest that our CGPQ design has accomplished the high-throughput design objective.

4.2 Evaluation of the CVC

Figure 14 shows the CVC read and write hit rate. We found that the hit rate highly depends on the number of vertices of the dataset: the g500-*N* series show a clear dropping trend when vertex count increases, and larger datasets (e.g., amzn, flickr, orkut, youtube) tend to have lower hit rate in general. Nevertheless, even for the largest datasets, the read and write hit rate is still higher than 80% and 50%, indicating effective caching.

Figure 15 shows the percentage of traversed edges that generated an active vertex with a new distance. We can see that CVC filtering is very effective in reducing redundant edge traversal.

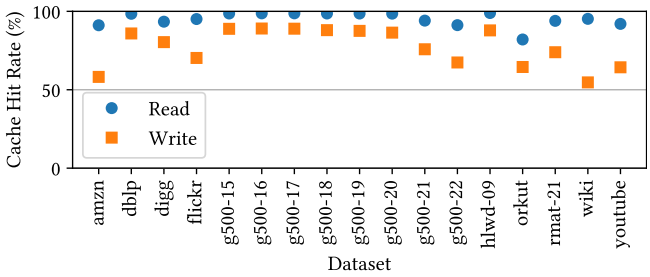

Figure 14: Read and write hit rate of the CVC.

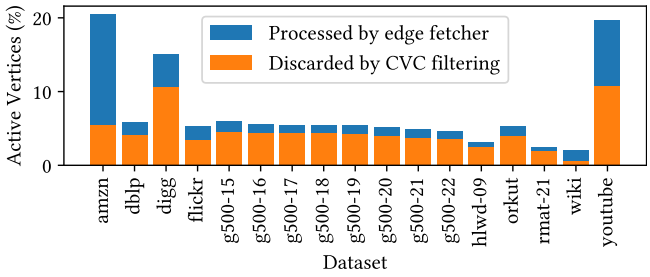

Figure 15: Percentage of active vertices among all traversed edges. Active vertices are either ① discarded by CVC filtering, or ② processed by the edge fetcher. The rest of traversed edges did not generate active vertices during CVC updating.

4.3 Overall Evaluation of SPLAG

Figure 16 shows the throughput achieved by SPLAG. The *traversal throughput* is defined as the number of traversed edges divided by the kernel execution time, which reflects the processing capability of the hardware but not the performance of the algorithm itself. The traversal throughput counts directed edges because only one direction can be traversed at a time. The *algorithm throughput* is defined as the number of *undirected* edges in the connected component of the root vertex divided by the kernel execution time, which measures the overall performance of the SSSP algorithm, including both the graph traversal throughput and the work efficiency. The algorithm throughput metric is used by the Graph 500 benchmark for data intensive applications [46]. We measured 504 MTEPS throughput under this metric using the g500-21 dataset, which could be ranked at the 14th position of the Graph 500 June 2021 SSSP list [40]. To the best of our knowledge, SPLAG is the first FPGA accelerator that can achieve such a ranking. The immediate preceding system on that list (at the 13th position) used an 8-node/128-core cluster to achieve 656 MTEPS throughput, while SPLAG works on a single FPGA board with only 45 W power budget. Beyond the Graph 500 datasets, the dblp dataset achieves the highest 763 MTEPS algorithm throughput.

Figure 17 further shows the work efficiency achieved by SPLAG. The work efficiency metric, *amount of work*, is normalized to the number of *directed* edges in the traversed connected component. Therefore, Dijkstra's algorithm generally achieves the amount of work of 1. Thanks to the never-look-back optimization (Section 3.1), SPLAG can even achieve < 1 amount of work for some datasets.

4.4 Comparison with Other SSSP Systems

Table 4 compares SPLAG against a multi-thread CPU baseline, a GPU baseline, and three state-of-the-art graph accelerators. The

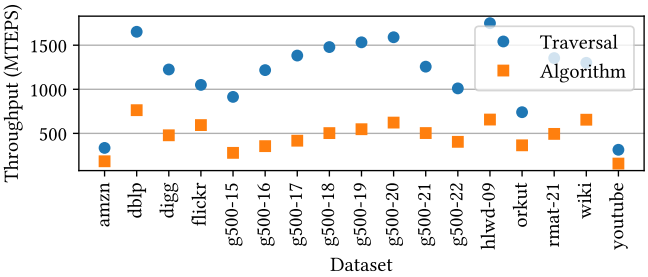

Figure 16: Throughput achieved by SPLAG. "Traversal" throughput measures the number of traversed edges over the kernel execution time. "Algorithm" throughput measures the number of *undirected* edges in the connected component over the kernel execution time.

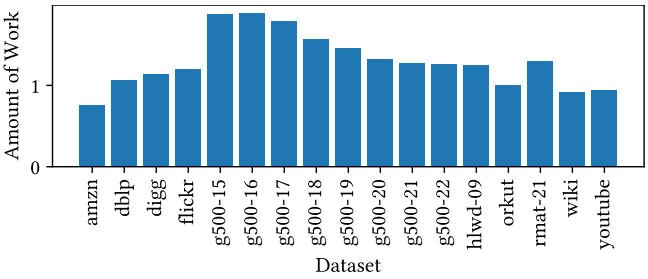

Figure 17: Normalized amount of work achieved by SPLAG. This is defined as the number of traversed edges divided by the number of *directed* edges in the connected component. Lower is better. Some benchmarks have < 1 amount of work because of the *never-look-back* optimization (Section 3.1).

CPU baseline is based on the latest Galois [29, 32], which runs the Δ-stepping [42] algorithm. Galois is a well-established concurrent graph library designed for graph analytics algorithms [29]. We modified the implementation from Galois to use floating-point distances and record the parent IDs [30]. The server has two Xeon Gold 6244 CPUs, which have 32 threads in total running at 4.4 GHz. The CPU baseline achieves better work efficiency than the Bellman-Ford accelerators, but is still less efficient than SPLAG due to its application-agnostic memory system.

The GPU baseline is based on the floating-point version of ADDS [54]. We modified the implementation from [54] to run on the Nvidia A100 GPU and record the parent IDs [53]. The GPU baseline has the highest throughput among all systems (including SPLAG), although that is achieved using a 1555 GB/s HBM and up to 186 W power consumption (reported by `nvidia-smi`), while the U280 FPGA only has a 460 GB/s HBM and a 45 W power budget (post-implementation report; `xbutil` reports lower power).

Previous Dijkstra's algorithm accelerators were not evaluated using power-law graphs. All three FPGA accelerators implement the Bellman-Ford algorithm. We use the performance numbers reported in each paper. Considering the fact that the previous works do not record the parent vertex ID in the vertex data while SPLAG does, we halve the traversal throughput of the previous works. Since ThunderGP [5] and HitGraph [58] use DDR-based FPGAs, we port SPLAG to a DDR-based FPGA (U250) for fair comparisons. Because HitGraph is simulated using a smaller chip, we further restrict the resource usage on U250 for fair comparison. Since ThunderGP and GraphLily [28] do not report the absolute execution time, we calculate the upper-bound of their algorithm throughput based on a CPU implementation of the Bellman-Ford algorithm [9]. This CPU implementation applies push-based graph traversal and edges are traversed only if the vertex is updated in the previous iteration. Due to lower parallelism, push-based traversal usually is only adopted when the graph traversal frontier is small [28], and pull-based traversal generates more redundant traversal. Therefore, our calculation gives a lower-bound of the number of traversed edges. Still, SPLAG is at least 2.3× faster. Note that the three FPGA baselines are powerful *general-purpose* graph processing systems. Many graph algorithms (e.g., PageRank) are very well-accelerated by these systems, yet SPLAG is not capable of the same. However, while they can support some application-specific optimizations like pruning and early-termination for SSSP, further customization by SPLAG, especially with efficient support of order-sensitive edge traversal, leads to better performance at the expense of some lose of generality.

Table 4: SPLAG compared against other SSSP systems.

Dataset	System	Algorithm	Hardware	MTEPS Trav.	MTEPS Algo.	SPLAG's Speedup
hlwd-09	Galois [32]	Δ-stepping [42]	Xeon 6244 CPU	1229	211	**2.6×**
	ADDS [54]	ADDS [54]	A100 40G GPU	31242	1455	0.4×
	GraphLily [28]	Bellman-Ford [50]	U280 FPGA	4670	< 232	> **2.3×**
	SPLAG	SPLAG	U280 FPGA	**1744**	**543**	1×
	ThunderGP [5]	Bellman-Ford [50]	U250 FPGA	2454	< 122	> **2.6×**
	SPLAG	SPLAG	U250 FPGA	756	315	1×
rmat-21	Galois [32]	Δ-stepping [42]	Xeon 6244 CPU	930	254	**1.9×**
	ADDS [54]	ADDS [54]	A100 40G GPU	15878	530	0.9×
	GraphLily [28]	Bellman-Ford [50]	U280 FPGA	2823	< 195	> **2.5×**
	SPLAG	SPLAG	U280 FPGA	**1354**	**494**	1×
	HitGraph [58]	Bellman-Ford [50]	VU5P FPGA	2152	46.9	**4.9×**
	SPLAG	SPLAG	VU5P FPGA	**533**	**228**	1×

5 CONCLUSION

We present SPLAG to accelerate the SSSP algorithm for power-law graphs on FPGAs. Two components in SPLAG are key to achieving the acceleration: The coarse-grained priority queue (CGPQ) uses an on-chip priority queue to orchestrate the off-chip memory accesses and enables high-throughput priority-order graph traversal with a large queue capacity. The customized vertex cache (CVC) implements two application-specific operations to reduce the amount of off-chip memory access and improve the throughput of random memory accesses imposed by priority-order graph traversal. Experimental results on various synthetic and real-world datasets on an HBM-equipped FPGA demonstrate up to 763 MTEPS overall throughput and a 4.9× speedup over state-of-the-art accelerators.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments, Jason Lau and Linghao Song for their help with the GPU baseline, and Jason Lau for his effort maintaining our infrastructures. This work is partially supported by the NSF RTML program (CCF1937599), NIH Brain Initiative (U01MH117079), the Xilinx Adaptive Compute Clusters (XACC) program, CRISP, one of six JUMP centers, and support of the CDSC industrial partners[4].

[4] `https://cdsc.ucla.edu/partners`

REFERENCES

[1] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *ASPLOS*.

[2] Ranjita Bhagwan and Bill Lin. 2000. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *INFOCOM*.

[3] Uday Bondhugula, Ananth Devulapalli, James Dinan, Joseph Fernando, Pete Wyckoff, Eric Stahlberg, and P. Sadayappan. 2006. Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths. In *FCCM*.

[4] Ulrik Brandes and Christian Pich. 2007. Centrality Estimation in Large Networks. *International Journal of Bifurcation and Chaos* 17, 07 (2007).

[5] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *FPGA*.

[6] Yuze Chi, Young-kyu Choi, Jason Cong, and Jie Wang. 2019. Rapid Cycle-Accurate Simulator for High-Level Synthesis. In *FPGA*.

[7] Yuze Chi and Jason Cong. 2020. Exploiting Computation Reuse for Stencil Accelerators. In *DAC*.

[8] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with Optimized Dataflow Architecture. In *ICCAD*.

[9] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: An Efficient Graph Processing System on a Single Machine. In *ICDE*.

[10] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *FCCM*.

[11] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. 2021. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. In *FPGA*.

[12] Young-kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. 2020. FLASH: Fast, ParalleL, and Accurate Simulator for HLS. *TCAD* (2020).

[13] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. 2020. *When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization.* arXiv 2010.06075.

[14] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. 2009. Power-Law Distributions in Empirical Data. *SIAM Rev.* 51, 4 (2009).

[15] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. 1998. A Parallelization of Dijkstra's Shortest Path Algorithm. In *MFCS*.

[16] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*.

[17] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *FPGA*.

[18] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *IPDPS*.

[19] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. 2021. StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems. In *CGO*.

[20] Camil Demetrescu, Andrew Goldberg, and David Johnson. 2009. *The Shortest Path Problem.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 74.

[21] RoBERT B Dial. 1969. Algorithm 360: Shortest-Path Forest with Topological Ordering. *CACM* 12, 11 (1969).

[22] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (1959).

[23] M.L. Fredman and R.E. Tarjan. 1984. Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. In *FOCS*.

[24] Robert Geisberger, Peter Sanders, and Dominik Schultes. 2008. Better Approximation of Betweenness Centrality. In *ALENEX*.

[25] Andrew V. Goldberg and Chris Harrelson. 2005. Computing the Shortest Path: A* Search Meets Graph Theory. In *SODA*.

[26] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *FPGA*.

[27] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2022. RapidStream: Parallel Physical Implementation of FPGA HLS Designs. In *FPGA*.

[28] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *ICCAD*.

[29] IntelligentSoftwareSystems. 2021. Galois: C++ Library for Multi-Core and Multi-Node Parallelization. `https://github.com/IntelligentSoftwareSystems/Galois`.

[30] IntelligentSoftwareSystems and UCLA-VAST. 2021. Galois: C++ Library for Multi-Core and Multi-Node Parallelization. `https://github.com/UCLA-VAST/Galois`.

[31] Donald B. Johnson. 1977. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM* 24, 1 (1977).

[32] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. In *PLDI*.

[33] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *FPGA*.

[34] Duncan H. Lawrie. 1975. Access and Alignment of Data in an Array Processor. *Transactions on Computers* C-24, 12 (1975).

[35] Guoqing Lei, Yong Dou, Rongchun Li, and Fei Xia. 2016. An FPGA Implementation for Solving the Large Single-Source-Shortest-Path Problem. *Transactions on Circuits and Systems II* 63, 5 (2016).

[36] Charles E. Leiserson. 1979. *Systolic Priority Queues.* Technical Report.

[37] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *Journal of Machine Learning Research* 11 (2010).

[38] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *FPGA*.

[39] Rui Li, Muye Zhu, Junning Li, Michael S. Bienkowski, Nicholas N. Foster, Hanpeng Xu, Tyler Ard, Ian Bowman, Changle Zhou, Matthew B. Veldman, X. William Yang, Houri Hintiryan, Junsong Zhang, and Hong Wei Dong. 2019. Precise Segmentation of Densely Interweaving Neuron Clusters Using G-Cut. *Nature Communications* 10, 1 (2019).

[40] The Graph 500 List. 2021. June 2021 SSSP. `https://graph500.org/?page_id=944`.

[41] Karl Marrett, Muye Zhu, Yuze Chi, Chris Choi, Zhe Chen, Hong-Wei Dong, Chang Sin Park, X. William Yang, and Jason Cong. 2021. *Recut: A Concurrent Framework for Sparse Reconstruction of Neuronal Morphology.* bioRxiv 2021.12.07.471686.

[42] U. Meyer and P. Sanders. 2003. Δ-Stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms* 49, 1 (2003).

[43] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2008. Growth of the Flickr Social Network. In *WOSN*.

[44] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *IMC*.

[45] Arjun Mukherjee, Bing Liu, and Natalie Glance. 2012. Spotting Fake Reviewer Groups in Consumer Reviews. In *WWW*.

[46] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the Graph 500. In *CUG*.

[47] Larry L. Peterson and Bruce S. Davie. 2010. *Computer Networks: A Systems Approach.*

[48] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*.

[49] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. 2019. Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching. In *FPGA*.

[50] Alfonso Shimbel. 1953. Structural Parameters of Communication Networks. *The Bulletin of Mathematical Biophysics* 15, 4 (1953).

[51] Yasuhiro Takei, Masanori Hariyama, and Michitaka Kameyama. 2015. Evaluation of an FPGA-Based Shortest-Path-Search Accelerator. In *PDPTA*.

[52] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *FPGA*.

[53] Kai Wang and Yuze Chi. 2021. A Fast Work-Efficient GPU Algorithm for SSSP. `https://github.com/UCLA-VAST/adds`.

[54] Kai Wang, Don Fussell, and Calvin Lin. 2021. A Fast Work-Efficient SSSP Algorithm for GPUs. In *PPoPP*.

[55] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. 2021. GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing. In *FPGA*.

[56] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. 2021. FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations. In *FPGA*.

[57] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Accelerating Large-Scale Single-Source Shortest Path on FPGA. In *IPDPSW*.

[58] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput Graph Processing Framework on FPGA. *TPDS* (2019).