# Delta-Stepping for Parallel SSSP

Daniel Li & Enrico Green

## Introduction

- In general, graph algorithms are difficult to parallelize. Many problems have low arithmetic intensity, require visiting vertices in a specific order making any algorithm inherently sequential, and results vary drastically based on the input graph.

- We implement the Delta-Stepping algorithm, which has decent parallelism and low total work, to solve the Single-Source Shortest Path Problem in multiple parallel computing paradigms. Our goal is to achieve speedup over traditional algorithms as well as measure and optimize speedup against the sequential version of our algorithm.

- We design a version of the Delta-Stepping algorithm which is compatible with CUDA, and we weigh the benefits of an OpenMP implementation versus a CUDA implementation leveraging the GPU.

## Motivation

- Dijkstra's Algorithm - Inherently sequential
  - Priority queue orders the vertices that we visit
  - Can only relax the edges of one vertex at a time
- Bellman-Ford - Easily parallelizable, but lots of redundant work
  - O(|V| |E|) total work, compared to O(|E| log |V|) for Dijkstra's
- Solution: Delta-Stepping
  - Replace priority queue with "buckets" of width $\Delta$
  - Entire bucket explored at the same time $\Rightarrow$ Parallelism!
  - By tuning $\Delta$, we can choose how much we prioritize parallelizability vs. not doing redundant work

# Delta-Stepping Pseudocode

```
Preprocessing:  Split edges into "light" and "heavy"
    Set d(source) = 0 and place the source vertex in bucket 0
        Loop: While there exists a nonempty bucket, do:
            Let B be the bucket with the minimum tentative distance
            Set $R = \varnothing$
            While $B = \varnothing$, do:
                Create requests for all light edges in B
                Set $R = R \cup B$
                Set $B = \varnothing$
                Relax all created requests
            Create requests for all heavy edges in R
            Relax all created requests
```

| | |
|---|---|
| Requests: | (dest. vertex, new tentative distance) |
| Relaxation: | If new tentative $d(v) < d(v)$, update $d(v)$ and move v to corresponding bucket |

## OpenMP Implementation

- Relaxation simple: Iterate over each (vertex, tentative distance) pair in parallel and update their weights
    - Use locks to protect buckets and distances from race conditions
- Finding requests is less simple
    - C++ sets don't support OpenMP parallel-for ⇒ Have to use tasks! (overhead)
    - Have to access shared data structure each time we process a node from the bucket (contention)
- Solution: convert the bucket contents to a vector and parallelize both the relaxation and request finding steps at the same time
    - Each thread adds requests to a local vector
    - After all requests are found and the bucket is cleared, each thread relaxes all requests in its local list
    - This solution may cause load imbalance issues on some inputs, but we found it gave a significant speedup on the graphs we were using for testing

## CUDA Implementation

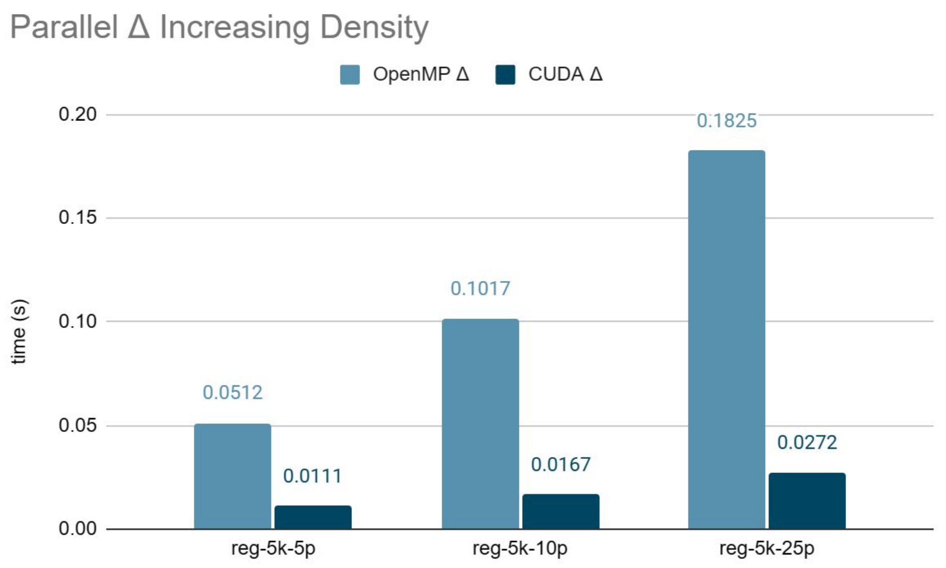
- Finding requests was tedious to parallelize
  - No access to C++ vectors, have to malloc fixed-length arrays and store offsets for different vertices
  - Map threads to edges and have each CUDA thread generate one request
  - Each thread does a binary sort to find the tentative distance of the destination vertex
- It's not all bad – extra performance on GPU
  - Parallel sort to find the request with the lowest tentative distance for each destination vertex
  - Prefix sum to collect the relaxations which minimize the tentative distance
  - Outcome: We find way fewer requests while still preserving correctness
- We were unable to parallelize the relaxation step
  - CUDA cannot represent the bucket set structure – must store buckets on CPU
  - Since relaxation moves vertices between buckets, it is done on the CPU
  - It's not that bad – since we generate way fewer requests with CUDA, even sequential CPU relaxation is pretty fast

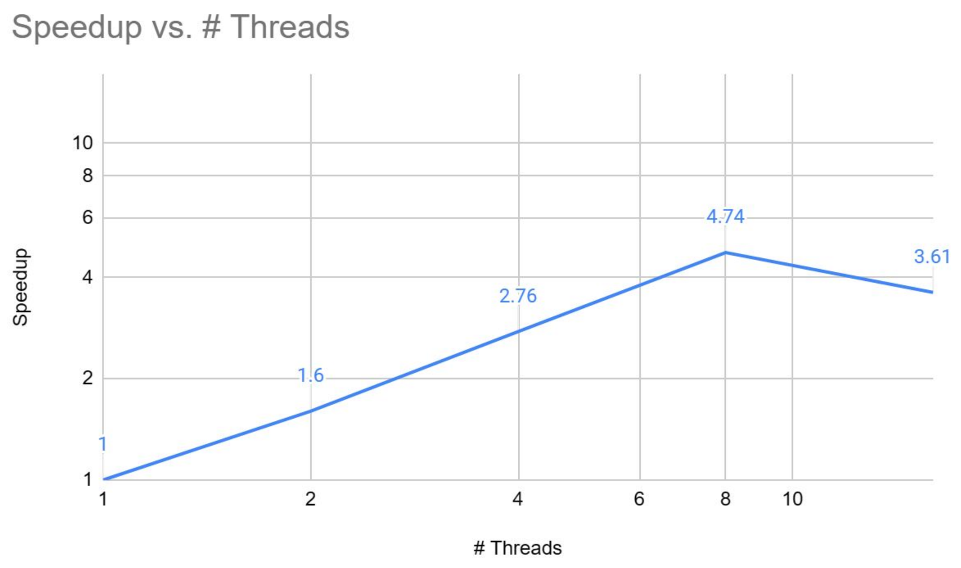
## Performance

| | Dijkstra | Bellman-Ford | OpenMP Δ | CUDA Δ |
|---|---|---|---|---|
| complete-1k | 0.1819 | 1.5167 | 0.0175 | 0.0056 |
| random-20k | 0.0023 | 3.1203 | 0.0002 | 0.0003 |
| local-100k | 0.0169 | (timeout) | 0.0006 | 0.0038 |
| tree | 0.2354 | (timeout) | 0.1222 | 0.0204 |
| cycle | 0.1564 | (timeout) | 0.6688 | 7.0821 |

- Parallel Delta-Stepping algorithms perform significantly better than classical alternatives on graphs a high edge density
- Both Delta-Stepping algorithms (but especially the CUDA implementation) performed poorly on the cycle input (100,000 vertices)

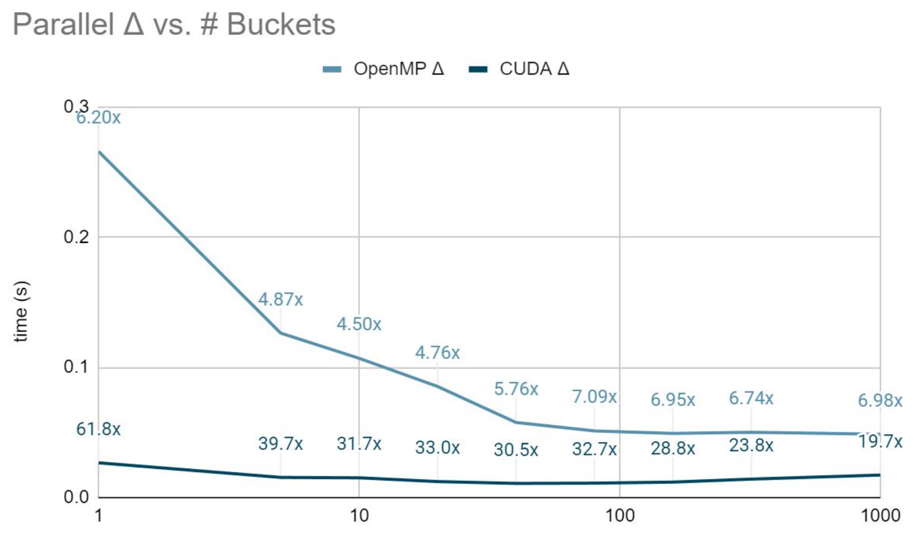
# Varying Input Size and Edge Density

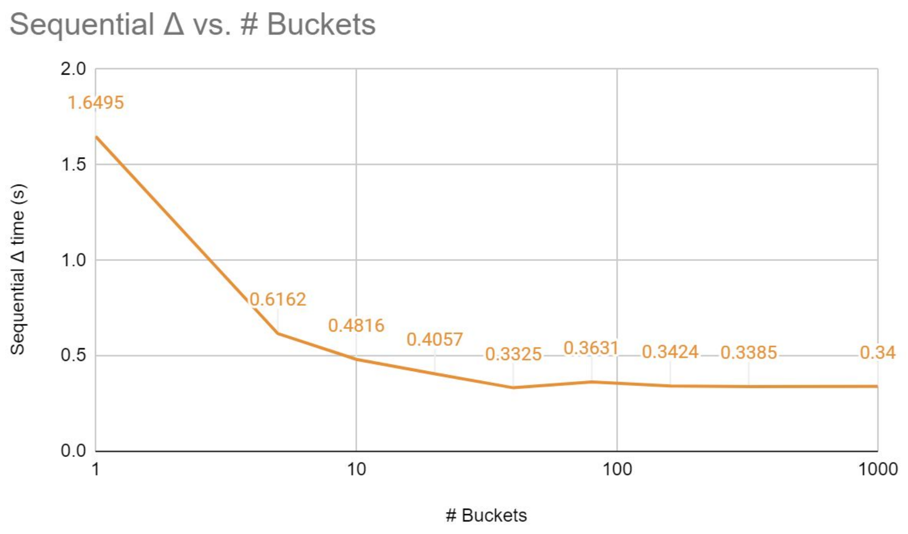




- Runtime of both CUDA and OpenMP Delta-Stepping increases slightly less than linearly with respect to the number of edges
- OpenMP delta stepping performance does not scale linearly with number of threads
- Tests were done on an 8-core CPU, and hence running with 16 threads results in a performance loss

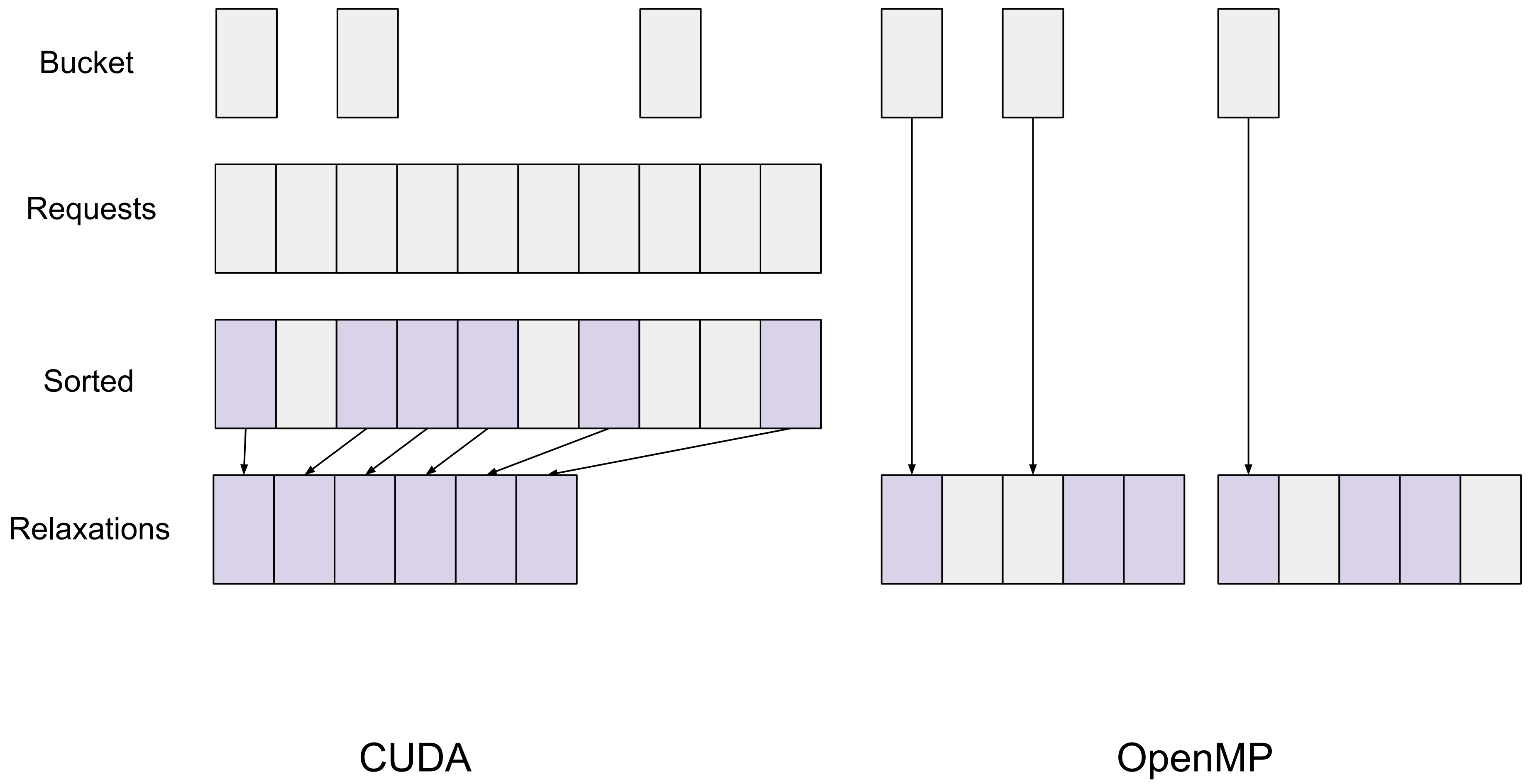
## Varying Δ Value





- Choice of Δ is pretty important – going from 10 buckets to 80 is a 2.47x speedup for OpenMP and a 1.4x speedup for CUDA
- Since different input graphs have different scales, we instead count # buckets = (max edge weight) / Δ
- Buckets are reused after being emptied, and this value is the actual number of buckets in memory

# Random diagram

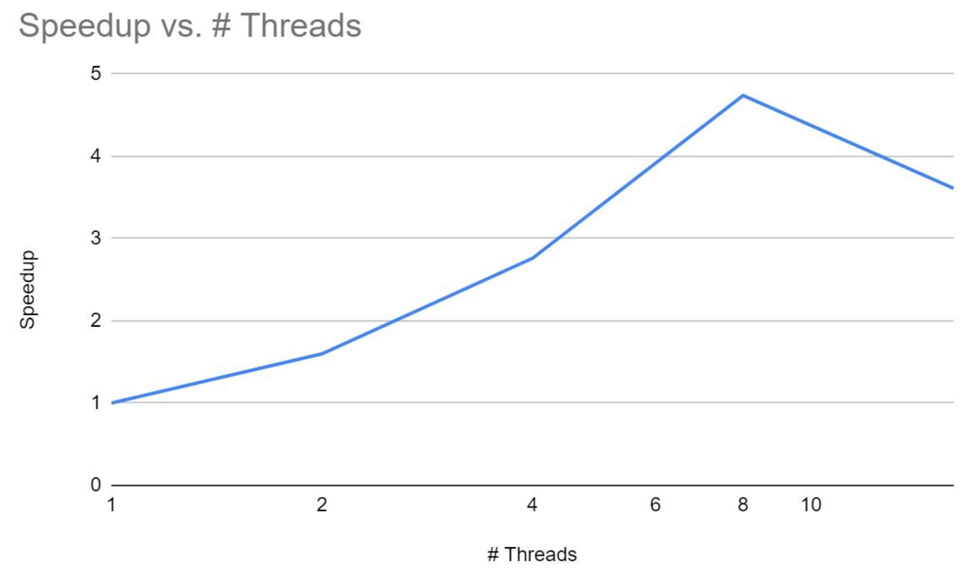


## Extra Notes

Two potential places for parallelization: Relaxation and finding requests

- Bellman-Ford was consistently the slowest algorithm
  - Expected, given O(|V||E|) runtime
- With integer weights, $\Delta = 1$ is the same as Dijkstra's algorithm

- With extremely large $\Delta$, very similar to Bellman-Ford


Speedup vs. # Threads
Speedup
# Threads