Warp-Centric CUDA Programming for SSSP Algorithms

15-418 Project Final Report

Repo: https://github.com/victorsongyw/15418-project

Yiwen(Victor) Song (yiwenson), Xingran Du (xingrand)

May 13, 2021

1 Summary

We implemented and studied a warp-centric CUDA optimization on three SSSP algorithms, Dijkstra's, Bellman-Ford, and Delta-Stepping. We achieve a speedup of 6x on Bellman-Ford and 1.15x on Delta-Stepping over a variety of graph inputs.

2 Background

Graph algorithms typically involve iterating through graph nodes or edges. Nodes or edges in the graph can often be processed in parallel, so graph algorithm can benefit from a parallel implementation. In this project, we explore solving a particular graph problem - the SSSP (Singel Source Shortest Path) problem, using a particular computer architecture - CUDA programming on NVIDIA GPUs.

Naive CUDA implementations to parallelize graph algorithms would be to assign each node to a task running on a CUDA thread, assuming that the graph algorithm performs the same operation on each node in one iteration. Since a GPU can schedule hundreds or thousands of threads to run in parallel, this allows us to potentially achieve a lot of speedup. However, GPUs execute instructions in a SIMD (single instruction, multiple data) fashion: threads are grouped into warps, which executes the same instruction on each clock cycle. This means that GPUs' performance suffer greatly from branch divergence and imbalanced workloads, which, unfortunately, are very common in graph problems.

In this project, we explore the benefits of a warp-centric CUDA programming paradigm targeting workload imbalance and irregular accesses, proposed in the paper "Accelerating CUDA Graph Algorithms at Maximum Warp" by Sungpack Hong et al. [4]. This paper presents an idea for CUDA programming that reduces workload imbalance and utilizes locality in memory accesses,

and it includes code snippets for a baseline and an optimized CUDA implementation for the BFS (breadth-first search) algorithm. We adopt this programming paradigm to implement and optimize CUDA implementations of three algorithms that solve the SSSP problem: Dijkstra's algorithm, Bellman-Ford algorithm, and Delta-stepping algorithm. Note that we are not concerned with the scope of problems that each algorithm can solve; we are only interested in optimizing and comparing their performance on directed graphs with positive edge weights.
Below we provide a brief walk-through of each algorithm. You can find the sequential implementations under `src/` with filenames `xxx_seq.cpp`.

The inputs are in compressed sparse row (CSR) format loaded into `input_graph.h` (more details on this in section 4.2). The output is an array mapping each node to its distance to the source node (node 0). See the `README.md` for more details.

2.1 Dijkstra's Algorithm

At a very high level, Dijkstra's algorithm maintains a mapping of nodes to their distances from the origin and a set of visited nodes; at each iteration, the algorithm visits one additional node that has the shorted distance with the current set of visited nodes, and update the distances of all its neighbors. This algorithm maintains a priority queue of unfinished nodes, with their temporary distance as their priority.

2.2 Bellman-Ford Algorithm

At a very high level, the Bellman-Ford algorithm maintains a mapping of nodes to their distances from the origin; at each iteration, the algorithm updates the mapping by "relaxing" all edges, where "relaxing" an edge $e = (u, v)$ is simply updating $d(v) = min(d(v), d(u) + w(e))$. The algorithm terminates after n iterations, given that there are no negative weight cycles.

2.3 Delta-stepping Algorithm

This is a rather new algorithm proposed in 2003 by U. Meyer and P. Sanders [6]. It is essentially a hybrid of Dijkstra's and Bellman-Ford. The idea is that we maintain an array B of buckets such that $B[i]$ stores $\{v \in V : v \text{ is not finalized and } d(v) \in [i \times \Delta, (i+1) \times \Delta)\}$, where Δ is a predetermined constant. We start with the smallest bucket (smallest i) and in each *phase*, we finalize a bucket, until there are no buckets left. Within a *phase*, we remove all nodes from the bucket and relax all light edges, where a light edge has $w(e) \leq \Delta$. This might lead to nodes being moved from a larger bucket to a smaller bucket, potentially to the current bucket. We loop until the current bucket is empty, by which time we are done with this *phase*.

It is guaranteed that when we are done with a phase i, all nodes with distance $d(v) < (i+1) \times \Delta$ have been found and finalized. Detailed proofs can be found in the paper and are omitted here. For integer weights and $\Delta = 1$, this algorithm is the same as Dijkstra's algorithm. For $\Delta = \infty$, this becomes the Bellman-Ford algorithm.

Note that a modified version of this algorithm called Radius-stepping is proposed quite recently in 2016 by Guy E. Blelloch et al. at CMU [2], but since the workload is very similar, we only focus on the original Delta-stepping algorithm in this project.

3 Approach

We work on one of the GHC machines containing a Intel Core i7 processor (complete specification can be found here) and a NVIDIA GeForce RTX 2080 B GPU, which supports CUDA compute capability 7.5. We use C++11 to program on the CPU and CUDA 10.2 to program on the GPU. CUDA implementations can be found under `src/` with filenames `xxx.cu`.

For the sequential Dijkstra's and Bellman-Ford, we referred to existing C++ programs available online. For the sequential Delta-stepping algorithm, we referred to the pseudo-code in the paper [6]. We implemented all CUDA versions ourselves.

3.1 Parallelizing Dijkstra's

Dijkstra's is essentially a sequential algorithm. To make it run on GPUs, we had to sacrifice work. Instead of maintaining a global queue, we only maintain an array mapping each node to its temporary distance and a boolean array keeping track of which nodes are finalizes. We spawn CUDA threads to find the minimum-distance non-finalized node. After obtaining a global min, we spawn CUDA threads to update the distances of all its neighbors.

3.1.1 Baseline CUDA implementation of Dijkstra's

In the baseline implementation, to find the minimum-distance non-finalized node, we spawn CUDA threads where each thread is responsible for a node. We use `atomicMin` to update the global min. To update its neighbors, we spawn CUDA threads where each thread is responsible for updating one neighbor node.

3.1.2 Warp-Centric CUDA implementation of Dijkstra's

In the warp-centric implementation, we used shared memory to coalesce memory access. We first copy the relevant chunk of the arrays to shared memory, and then have the CUDA threads read from shared memory (instead of global memory). However, in this case, the access pattern was

already consecutive, so there is no real benefit, which is verified by our results shown in the next section.

3.2 Parallelizing Bellman-Ford

Bellman-Ford is a highly parallel algorithm. In each iteration, all edges can be relaxed in any order in parallel, using the `atomicMin` primitive to update the distances. This naturally maps well to GPUs. We just need to synchronize after each iteration, and we are done when we reach N iterations.

3.2.1 Baseline CUDA implementation of Bellman-Ford

In the baseline implementation, we spawn CUDA threads where each thread is responsible for a node. For each node, we loop through all its neighbor and relax these edges. This results in huge workload imbalance since node degrees can vary greatly. CUDA threads whose assigned node has a very small degree will have to wait idly for other threads in the warp to finish.

3.2.2 Warp-Centric CUDA implementation of Bellman-Ford

In the warp-centric implementation, we used the idea proposed in [4]. Specifically, instead of assigning one task to each thread, we "allocate a chunk of tasks to each warp and execute distinct tasks as serial" [4]. This prevents branch divergence and workload imbalance, since all threads in a warp is executing in a SIMD fashion. To speed up memory access, we copy the relevant portion of input into shared memory, which is shared among all threads in a warp. See the kernel code below:

```
__global__
void warp_BF_kernel(uint *nodes, uint *edges, uint *weights, uint *dists, uint num_nodes)
{
    uint warp_offset = threadIdx.x % WARP_SIZE;
    uint warp_id = threadIdx.x / WARP_SIZE;

    // this is the range of indexes of nodes for which this warp is responsible
    uint chunkStart = blockIdx.x * NODES_PER_BLOCK + warp_id * CHUNK_SIZE;
    if (chunkStart >= num_nodes) return;
    uint chunkEnd = chunkStart + CHUNK_SIZE;
    if (chunkEnd > num_nodes) chunkEnd = num_nodes;

    // shared memory across threads in a block
    __shared__ uint block_nodes[NODES_PER_BLOCK + WARPS_PER_BLOCK];
    __shared__ uint block_dists[NODES_PER_BLOCK];

    // pointers to the start of the region corresponding to this warp
    uint *warp_nodes = block_nodes + warp_id * (CHUNK_SIZE+1);
    uint *warp_dists = block_dists + warp_id * CHUNK_SIZE;

    warp_memcpy(chunkStart, warp_offset, chunkEnd+1, warp_nodes, nodes);
    warp_memcpy(chunkStart, warp_offset, chunkEnd, warp_dists, dists);

    // iterate over my work
    for (uint v = 0; v < chunkEnd - chunkStart; v++)
    {
        uint nbr_start = warp_nodes[v];
        uint nbr_end = warp_nodes[v+1];
        warp_update_neighbors(nbr_start + warp_offset, nbr_end, edges, dists, warp_dists, weights, v);
    }

}
```

All threads in a virtual warp process one node at a time, iterating through all its neighbors. Here are the helper functions used:

```
__inline__ __device__
void warp_memcpy(uint start, uint offset, uint end, uint *warp_array, uint *array)
{
    for (uint i = start+offset; i < end; i += WARP_SIZE)
        warp_array[i-start] = array[i];
}

__inline__ __device__
void warp_update_neighbors(uint start, uint end, uint *edges, uint *dists, uint *warp_dists, uint *weights, uint v)
{
    for (uint i = start; i < end; i += WARP_SIZE)
    {
        uint u = edges[i];
        // updating an edge from v to u
        uint new_dist = warp_dists[v] + weights[i];
        atomicMin(&(dists[u]), new_dist);
    }
}
```

Notice here that when each thread executes the loop, it starts with an offset and in each iteration jumps ahead by `WARP_SIZE`. This ensures that as a SIMD program, the access to the arrays are coalesced and can be served by a single memory fetch.

3.3 Parallelizing Delta-stepping

Delta-stepping, as introduced previously, is a hybrid of the other two algorithms. Nodes in the same bucket can be processed in arbitrary order in parallel, but there is a sequential dependency across buckets. To implement this in CUDA, we keep track of an array mapping nodes to their buckets. This array is updated during edge relaxing using the `atomicMin` primitive. To find the next bucket after we finish a phase, we simply spawn N CUDA threads and use `atomicMin` to find the smallest bucket that is larger than the last bucket. We also use a boolean to keep track of whether new nodes have been added to the current bucket.

3.3.1 Baseline CUDA implementation of Delta-stepping

In the baseline implementation, we spawn CUDA threads where each thread is responsible for a node in the current bucket. For each node, we loop through all its edges, decide whether it is a light edge or a heavy edge, and relax the edge if needed.

3.3.2 Warp-Centric CUDA implementation of Delta-stepping

In the warp-centric implementation, we used the same idea as in that of Bellman-Ford. A warp of threads is responsible for a chunk of nodes in the bucket, and we execute them serially in a SIMD fashion. The code is omitted here.

4 Results

4.1 Experiment Setup

We use the wall-clock time (in milliseconds) it takes for each SSSP implementation to finish one run of one test input as the measurement of performance.

We measure speedup as the ratio of CUDA baseline implementation runtime and warp-based implementation runtime, for each of the three algorithms. We believe this is a fair comparison to evaluate the effectiveness of the warp-centric approach.

For each algorithm, we benchmark the sequential CPU version, the CUDA baseline version, and the CUDA warp-centric version. For the CUDA versions, we further separate the time it takes to setup the CUDA SSSP kernels (mostly memory copy, which is an overhead the sequential CPU version does not have), and the time for the actual SSSP computation (a.k.a. "Kernel Time").

4.2 Test Inputs

We evaluate the algorithms with generated graphs, and real-world graphs [5], following the evaluation in the warp-centric optimization paper [4]. For both generated graphs and real-world graphs, we represent both "regular" and "irregular" graphs, which describes the variance in node degrees, and helps us evaluate the algorithms' ability to deal with workload imbalance and irregular memory access.

We use Python to write the adjacency matrix of the graphs into arrays of the CSR format, in a C header file, and compile with the algorithm source files so that the arrays are accessible as global variables.

We are limited by the storage space available on AFS, and the generate and compile time of the graphs, so we are not able to run the exact same graph inputs as used in the original paper. After some preliminary experiments, we find that the number of nodes $n = 1 \times 10^5$ is the largest reasonable size we can use, and the following are based on this assumption.

4.2.1 Generated Graphs

We use the NetworkX Python package [3] to generate random graphs that are:

- Irregular: The Barabasi-Albert model [1] uses preferential attachment to create scale-free graphs that are similar to many real-world graph instances. The parameter m for this function is an approximate of average node degree.

- Regular: The $G_{n,m}$ model chooses a graph uniformly at random from the set of all graphs with n nodes and m edges.

The generated graphs used in the paper[4] has $m = 12n$, where m is the number of edges and n is the number of nodes, and we do the same.

4.2.2 Real-World Graphs

We choose graphs in the same category as the Hong et al. paper [4] but has n around 1×10^5. The parameters could be found in Table 4.2.2.

- For irregular graphs, we use the `ego-Facebook` and `ego-Twitter` social network datasets.
- For regular graphs, we use the `cit-HepPh` citation network dataset.

Graph	n	m
ego-Facebook	4,039	88,234
ego-Twitter	81,306	1,768,149
cit-HepPh	34,546	421,578

Table 1: Real-world Graph Parameters

4.3 Execution Time Summary

Figure 1 shows the execution time of all benchmarks on all algorithms.

We observe that we achieve an overall average (geometric mean) speedup of up to 6.3x for warp-centric vs. baseline, as shown in Figure 2. Dijkstra's algorithm does not explicitly traverse neighbors and thus exposes little warp-centric optimization opportunity, and thus we observe no speedup. Delta-Stepping, on the other hand, has neighbor updates which is the typical optimization opportunity exploited by the warp-centric method. Bellman-Ford algorithm, less work efficient than Delta-Stepping, has a lot more work and memory accesses; better work balancing and memory coalescing in the warp-centric version greatly helps with speedup.

Further more, we observe a larger speedup in irregular graphs than regular graphs, because irregular graphs tend to have worse workload balance without the warp-centric optimization.

We notice that the Bellman-Ford algorithm performs a lot worse than Dijkstra's and Delta-Stepping across the board, and for the latter two the runtime seems too small to draw solid conclusions. Therefore, we perform additional benchmarking with even larger graphs to further explore the respective performance of Dijkstra's and Delta-Stepping. We have only applied minimal optimization

DELTA=5	(unit: ms)		regular
			irregular

Real-world Graphs

Facebook			
	sequential	CUDA baseline	CUDA warp
Dijkstra's	0.292	65.631	65.314
Bellman-Ford	445.077	484.827	77.17
Delta-Stepping	12.399	1.139	1.421

Twitter			
	sequential	CUDA baseline	CUDA warp
Dijkstra's	0.399	3018.247	3004.876
Bellman-Ford	282025.613	72932.377	10318.645
Delta-Stepping	264.433	7.99	6.369

cit-HepPh			
	sequential	CUDA baseline	CUDA warp
Dijkstra's	0.33	842.627	843.445
Bellman-Ford	26110.25	7458.839	1566.129
Delta-Stepping	71.789	2.466	2.047

Generated Graphs

Scale-free	n=1e5		
	sequential	CUDA baseline	CUDA warp
Dijkstra's	0.445	4228.424	4232.601
Bellman-Ford		119339.643	16982.434
Delta-Stepping	229.031	8.827	5.873

Random	n=1e5		
	sequential	CUDA baseline	CUDA warp
Dijkstra's	0.411	4259.685	4256.429
Bellman-Ford		60928.657	8888.568
Delta-Stepping	219.718	6.45	5.403

Figure 1: Runtime for All Benchmarks

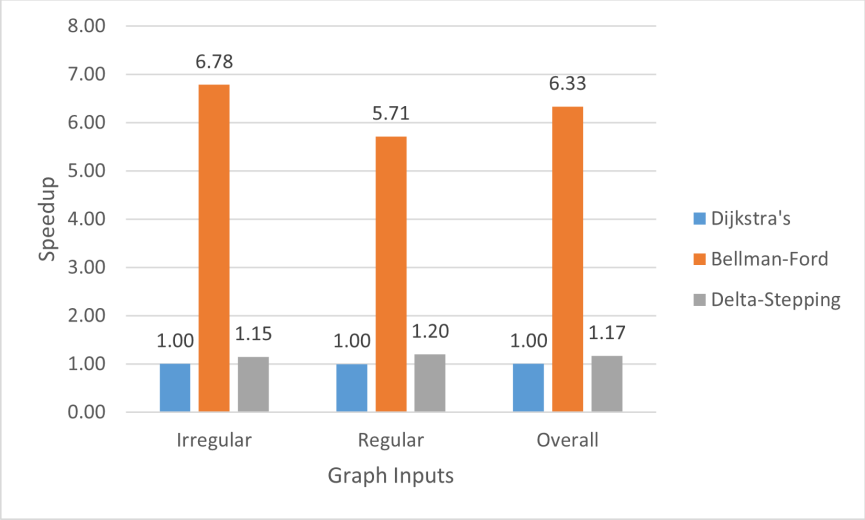

Figure 2: Speedup for All Benchmarks

in the Dijkstra's warp version, and previous results confirm that there is little difference between the Dijkstra's baseline and warp runtimes. Therefore we only include one number for the Dijkstra's CUDA runtimes.

Figure 3 shows the performance of Dijkstra's versus Delta-Stepping on two large generated graphs. Comparing with previous results, we can see that the Delta-Stepping version scales reasonably well

with inputs, while Dijkstra's version does not.

This benchmark also confirms previous results that comparing to regular graphs, larger speedup is obtained in irregular graphs. What is more important, the fine-grained timing on large benchmarks reveals that we have very high overhead (mostly memory copy to and from GPU): the total time is almost twice the kernel time.

scale-free	n=1e6		
		Dijkstra's	Delta-Stepping
sequential		0.523	2684.504
CUDA baseline	total		96.208
	kernel		52.111
CUDA warp	total	307535.808	78.87
	kernel	307489.964	33.924

random	n=1e6		
		Dijkstra's	Delta-Stepping
sequential		0.419	2721.123
CUDA baseline	total		89.613
	kernel		45.123
CUDA warp	total	307780.82	86.794
	kernel	307734.51	42.223

Figure 3: Runtime for Larger Benchmarks

We also explore the effect of graph density (i.e. average node degree) on the performance of Dijkstra's versus Delta-Stepping algorithms. All our previous results have a $\frac{m}{n}$ ratio of 12 (following the original paper [4]), which is a sparse graph. But $n = 1 \times 10^4, m = 1000n$ is close to a dense graph ($m = O(n^2)$). Figure 4 shows the results we have for two different graph densities.

scale-free	n=1e5	m=50n	
		Dijkstra's	Delta-Stepping
sequential		0.445	954.143
CUDA baseline	total		32.062
	kernel		14.601
CUDA warp	total	6622.23	19.62
	kernel	4247.563	2.137

scale-free	n=1e4	m=1000n	(dense)
		Dijkstra's	Delta-Stepping
sequential		0.319	2005.701
CUDA baseline	total		46.515
	kernel		13.066
CUDA warp	total	217.906	35.976
	kernel	184.456	2.599

Figure 4: Runtime for Denser Benchmarks

The results though brief show that the patterns we observed above still apply to denser graphs.

4.4 Sensitivity Studies

For all sensitivity studies (and for the above benchmarking results), the default parameters are `DELTA` = 5, `CHUNKSIZE` = 8, and `WARPSIZE` = 32. We perform the sensitivity study on a 1×10^4 nodes graph because it could be compiled more easily, allowing us to explore more parameter settings. We use the Bellman-Ford algorithm because it displayed the most significant speedup in previous results, showing that the computation of Bellman-Ford is the most suitable for warp-centric optimizations.

4.4.1 Effect of `CHUNKSIZE`

The `CHUNKSIZE` variable describes how many nodes a virtual warp processes together at a time. Figure 5 shows that as `CHUNKSIZE` increases, our speedup decreases.

While a larger `CHUNKSIZE` allows SIMD memory copy of graph data to shared memory, with better memory operation coalescing, it increases the task granularity, and if there are additional available warps in the GPU, a finer task granularity will lead to more warps running in parallel and thus gives better speedup. Our interpretation of the data is that SIMD memory copy is not as significant as higher warp parallelism in contributing to speedup, in the Bellman-Ford algorithm specifically.

scale-free	n=1e4	
		CUDA warp
Bellman-Ford		
chunksize=1	total	240.301
	kernel	239.587
chunksize=2	total	305.235
	kernel	304.533
chunksize=4	total	407.852
	kernel	407.122
chunksize=8	total	453.822
	kernel	452.57
chunksize=16	total	599.761
	kernel	599.049
chunksize=32	total	844.851
	kernel	844.124

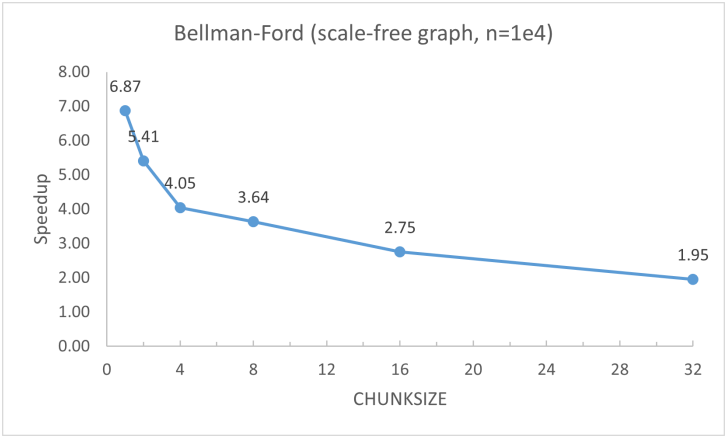

Figure 5: Runtime with Different `CHUNKSIZE`

4.4.2 Effect of `WARPSIZE`

The `WARPSIZE` variable describes how many threads are in a virtual warp. The physical warp contains 32 threads, and so a `WARPSIZE` of 8 would mean there are 4 virtual warps in a physical warp. Figure 6 shows that when packing more virtual warps into a physical warp, the speedup decreases. The intention of having virtual warps is that under-utilized physical warps could benefit from having more than one job [4], intuitively like SMT. However, our results show that packing

more virtual warps into a physical warp lead to longer execution time, potentially due to severe SIMD divergence, because instructions and data access patterns are more different across virtual warps than within the same virtual warp. We have not observed the effect described in the paper where having virtual warps increases the warp utilization and decreases runtime.

scale-free	n=1e4	
		CUDA warp
Bellman-Ford		
warpsize=32	total	453.822
	kernel	452.57
warpsize=16	total	612.036
	kernel	611.306
warpsize=8	total	1053.246
	kernel	1052.536
warpsize=4	total	1613.716
	kernel	1613.716

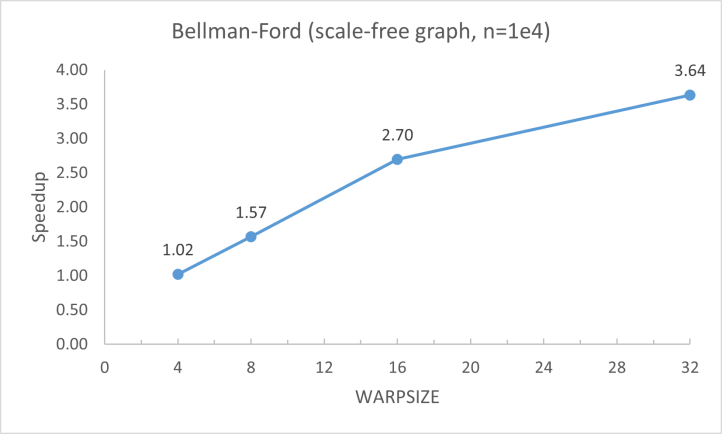

Figure 6: Runtime with Different `WARPSIZE`

4.5 Results Analysis Summary

Based on the above results and discussion, we believe that our speedup is limited by:

- Bandwidth copying graph data and results between CPU and GPU. Although both the baseline and warp-based versions have this memory copy overhead, it is not parallelizable and by Amdahl's Law it is significantly limiting our speedup.
- Low inherent parallelism in the algorithm. Bellman-Ford has a much larger speedup than Delta-Stepping, highlighting that without massively parallel neighbor updates the warp-based approach cannot show its benefits.
- SIMD divergence and irregular memory access could still be the performance bottleneck for the key computations. Our sensitivity study on `WARPSIZE` shows that decreasing the number of virtual warps in a physical warp helps with speedup, most likely by reducing SIMD divergence. Therefore, we speculate that the performance could be further improved if divergence within each single warp could be reduced.

4.5.1 Machine Target Discussion

Our machine target for optimizing the SSSP is GPU because we want to explore optimizations on CUDA warps. Intuitively, GPU's high SIMD parallelism is suitable for parallelizing across a large number of graph nodes or edges. From our results, the best GPU implementation, Delta-Stepping

with warp-centric optimizations, achieves at least 10x speedup over the sequential Delta-Stepping version, so GPU parallelism is well utilized. However, it is still much slower than the best sequential version, Dijkstra's algorithm, which inherently has less work in absence of the need to parallelize.

For the input graph sizes we currently evaluate on, it seems like the sequential Dijkstra's algorithm is quite suffcient. However, because the runtimes of both the sequential Dijkstra's and the warp-based Delta-Stepping versions are too low to put significant stress on the machine, it is yet to see whether for larger graph sizes whether GPU parallelism will outperform Dijkstra's sequential version.

Therefore, we think that it is worth further exploration whether the SSSP problem is best parallelized on a GPU or simply run with the already efficient Dijkstra's algorithm on a CPU.

5 Division of work

We mostly plan our project implementation and evaluation, write code, and discuss the benchmark results together. Each of us do have a focus though: Yiwen does a lot of heavy-lifting in writing the implementations, tidying the algorithms, while Xingran focuses more on setting up the workflow for and performing the evaluation.

References

[1] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. ISSN: 0036-8075. DOI: 10.1126/science.286.5439.509. eprint: https://science.sciencemag.org/content/286/5439/509.full.pdf. URL: https://science.sciencemag.org/content/286/5439/509.

[2] Guy E Blelloch et al. "Parallel shortest paths using radius stepping". In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. 2016, pp. 443–454.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11 –15.

[4] Sungpack Hong et al. "Accelerating CUDA Graph Algorithms at Maximum Warp". In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: Association for Computing Machinery, 2011, 267–276. ISBN: 9781450301190. DOI: 10.1145/1941553.1941590. URL: https://doi.org/10.1145/1941553.1941590.

[5] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection.* http://snap.stanford.edu/data. June 2014.

[6] Ulrich Meyer and Peter Sanders. "Δ-stepping: a parallelizable shortest path algorithm". In: *Journal of Algorithms* 49.1 (2003), pp. 114–152.