Accelerating Computation of Steiner Trees on GPUs

Rajesh Pandian M
www.cse.iitm.ac.in/~mrprajesh

Acknowledgements and disclaimer

- This is a joint work with Rupesh Nasre and N.S.Narayanaswamy
- This work evolved after the PACE Challenge 2018 on Steiner Tree [www.pacechallenge.org]
- Thanks to P100 – GPU server and PACE Lab members.

- This work is in progress / under submission.

Outline

- Steiner Tree Problem
  - Example
  - Properties
  - Hardness
- KMB algorithm

Intro to problem & Algorithm

- Challenges
- Design Choice of KMB
- CPU Optimization

Designing &
CPU Implementation +OPT

- GPU Implamentation
- SSSP Optimization – Sync/Compute/memory
- Pull/Push variants

GPU implementation and Opt

Steiner Tree Problem(STP)

Input : Undirected Graph G(V, E, W, L) W is non-negative edge weights; L $\subseteq$ V terminals
Output : A tree with all terminals
Goal : Minimize the weight of the tree

- **Terminals or** terminal vertices are special vertices which must be present in the tree
- **Non-terminals** or Steiner vertices are optional vertices – generally included in tree to minimize the overall weight of the resulting tree.
- Steiner Tree - tree with all the terminals
- Applications[Hwang et. al. 92]: VLSI design, network/vehicle routing, etc.

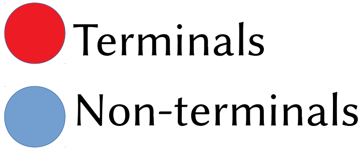
Standard Graph-theoretic notations are used n=|V| , m=|E| and additionally k=|L|

Steiner Tree Problem(STP) - Example

<u>Input</u> : Graph G(V, E, W, L) $W: E \rightarrow Z^+$ and $L \subseteq V$ terminals

<u>Output</u> : Connected subgraph T'(V'$\supseteq$L, E'$\subseteq$E) s.t Min W(E')

// Minimum weighted tree with all terminals



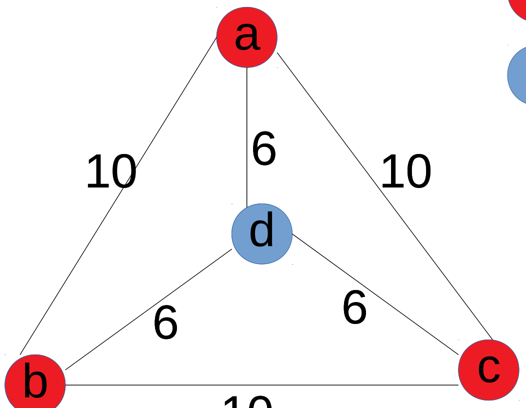


Fig 1 (a)

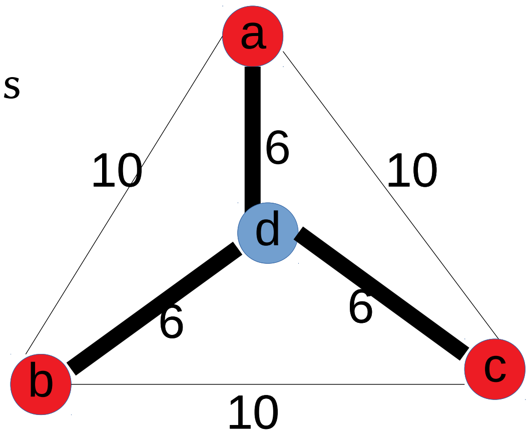


Fig 1 (b)

Steiner Tree Problem(STP) - Example

Input : Graph G(V, E, W, L) W:E→ Z^+ and L⊆V terminals

Output : Connected subgraph T'(V'⊇L, E'⊆E) s.t Min W(E')

// Minimum weighted tree with all terminals

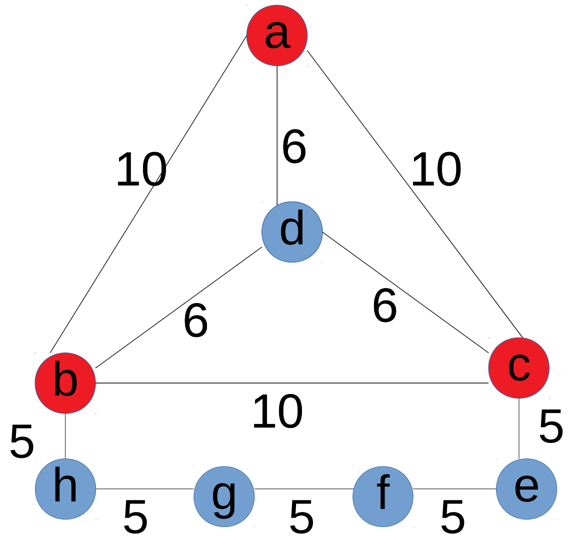


Fig 2 (a)

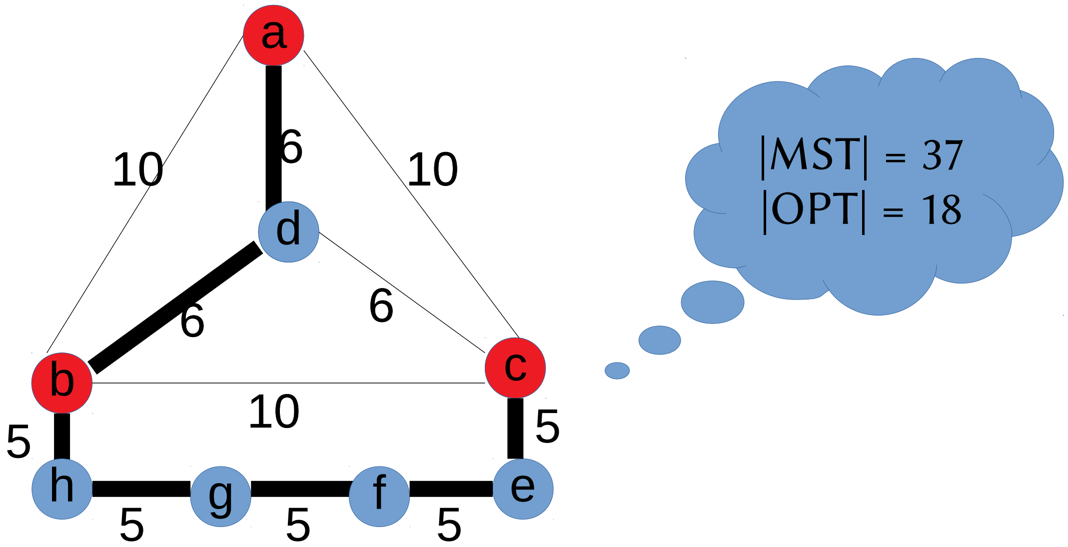


Fig 2 (b)

Steiner Tree Problem(STP) - Hardness

Input : Graph G(V, E, W, L) $W: E \rightarrow Z^+$ and $L \subseteq V$ terminals
Output : Minimum weighted tree with all terminals

2		n

Special cases

- $L = \{u,v\}$ or $k = 2$ STP=ShortestPath_In_G(u,v)
- $L = V$ or $k = n$ STP=MST(G)

In P Time

- Othercases STP is NP-Hard

Approximation algorithms

Standard Graph-theoretic notations are used $n=|V|$, $m=|E|$ and additionally $k=|L|$

How to deal with NP-Hardness

- No Polynomial time algorithm can find optimal solution unless P != NP
- What could be naive solutions? Enumerate all Spanning trees.

- **Approximation algorithms**
- Runs in P time
- Outputs an approximate solution with some guarantee.
  - e.g 2 or some constant, log n, etc.

- There are several algorithms
  - Kou, Markowsky and Berman[KMB81]
  - Robins and Zelikovsky [RZ2000]
  - Melhorn [M88]
  - ..

|ALG| <= 2 |OPT|

Our work

- Optimized CPU implementation for KMB Algo
- GPU optimizations for SSSP in undirected
- GPU Implementation of KMB algo.
  - Speed-up 10x (average 3x) over sequential CPU

KMB Algorithm G(V, E, W, L)

Phase 1

Computes the shortest distance between every pair of terminals

Phase 2

// Construct G'= K_L

Build a graph G' over terminals, having edge-weights corresponding to the shortest distances computed Phase 1

//Every edge in G' corresponds to a path in G

MST (G')

Phase 3

// Construct G''

For every edge in MST(G') substitute the edges with the corresponding shortest path in G

//collect all the edges & vertices of the corresponding path to construct G''

MST(G'')

KMB Algorithm G(V,E,W,L)

Phase 1 & 2

Observe: Two For loop

```
For u in L {
  For v in L {
    Puv = ShortestPath(u,v)
    W'(u,v) = |Puv|
   }
}
T' = MST( G'(L, - ,W') )
```

Phase 3

```
For (u,v) in edges of T'
   G'' = G'' ∪ P uv
  //Add vertices & edges of P uv
End

T' = MST ( G"( - , - , W) )
```

KMB Algorithm G(V,E,W,L)

Input: Graph G(V , E, W, L)
Output: 2-approx Steiner Tree T (V_T , E_T) $V_T \supseteq L$

for u ∈ L do
    SSSP(G, W , L, u)
    Compute W' incrementally
end
T' = MST(G' ,W')

Compute G'' and its vertices, adjList using T'
T'' = MST(G''' ,W)

return T'' ;

Single for loop but runs SSSP fully

KMB Algorithm G(V,E,W,L)

Input: Graph G(V , E, W, L)

Output: 2-approx Steiner Tree T (V_T , E_T) $V_T \supseteq L$

for u ∈ L do

  parallel SSSP(G, W , L, u);

  Compute W ′ incrementally;

 end

T ′ = **parallel** MST(G ′, W ′);

Compute G ″ and its vertices, adjList ;

T ″ = **parallel** MST(G ″, W);

return T ″ ;

Single for loop but runs SSSP fully

Our next work: In fact we want to run multiple SSSP from different source in parallel

[Detour] SSSP : Dijkstra vs BellmanFordMoore

- Runs in time $O(n^2)$ / $O(m+n \log n)$ / $O((m+n) \log n)$
- Uses Binary/Fib Min-Heap
- At each iteration,
  - Pick up node from Q
  - RELAX'es all its neighbours

In parallel setting must use Queue
In some form //it is difficult.

- Runs in time O(nm)
- No heap
- All edges are RELAX'ed at most (n-1) times

```
For i from 1 to n-1:
  For each edge (u, v) in E
       RELAX(u,v, W(u,v))
```

RELAX all edges
Launched using n threads or m

Dijkstra and its RELAX operations

```
INPUT: G(V,E,W), src
OUTPUT: d[], p[]

INITIALIZE-SINGLE -SOURCE (G, src)
Q = G.V
while(! Q.empty() ) {
   u = ExtractMin(Q);
   For v in Adj[u]
      RELAX(u,v, W)
}
```

```
RELAX(u, v, W){
   If u.d + W(u,v) < v.d {
      v.d =u.d + W(u,v)
      v.p = u
   }
}
```

```
INITIALIZE-SINGLE -SOURCE(G , src)
   For each v in G.V
      v.d = ∞
      v.p = NIL
}
src.d = 0
```

Source : CLRS book

Dijkstra and its RELAX operations

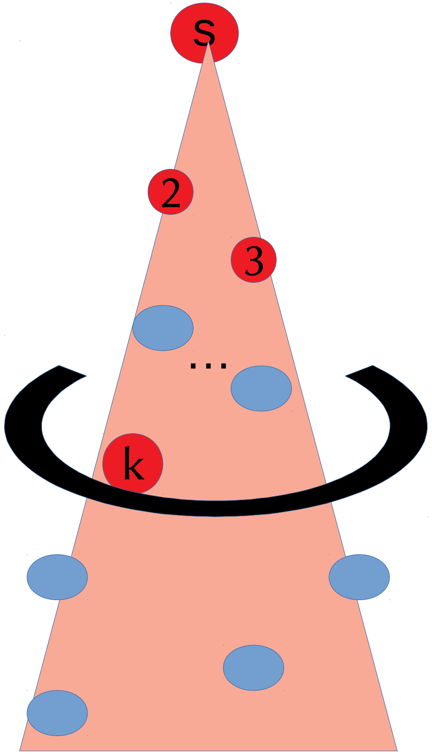
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        distance[v] ← INFINITY
        parent[v] ← NIL
        add v to Q
     distance[source] ← 0


    while Q is not empty:
        u ← vertex in Q with min distance[u]


        remove u from Q
        // only v that are still in Q
        for each neighbor v of u:
            newD ← distance[u] + Weight(u, v)
            if newD < distance[v]:
                distance[v] ← newD          //RELAX(u,v, W)
                parent[v] ← u

    return distance[], parent[]
```

Source : Wikipedia

[coming back] CPU Implementation- Optimization

- SSSP halt optimization

Steps of SSSP execution



Dijkstra Property: when a node u is picked from Q for processing then the distance[u] saturated using all the visited node

Halt the SSSP when all terminals are visited

GPU Implementation

- Style of implementation

Topology-driven [TD]

- All nodes are assumed to be active
- Operator is applied on all the nodes
- Some nodes might have no work

- Easy to implement.

- Works well on GPU

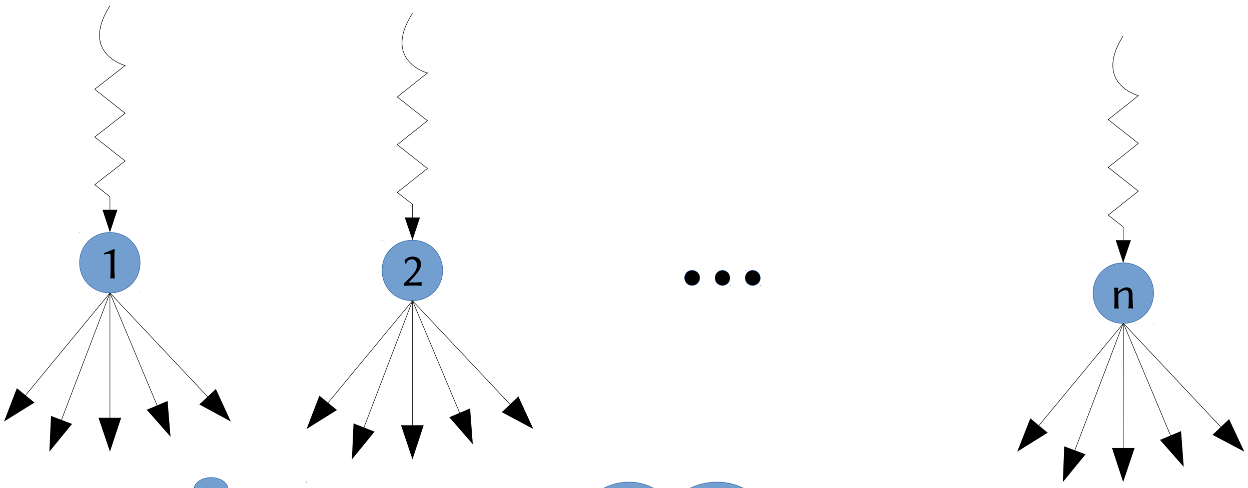
Data-driven [DD]

- Only some nodes are active
- Operates on some nodes that might have work to do

- Requires addition effort to maintain worklist(WL)

- Works well for CPU parallelism

GPU Implementation - SSSP



- n-threads
- One thread for each node
- Perform RELAX in parallel
- RELAX's its Nbrs
- Till there is no change

In some sense,
Every node/thread
pushses the data
to Nbrs

KMB Algorithm G(V,E,W,L)

MAIN-GPU

```
For s in L {
  ThdsPerBlk = 512; // or 1024
  Blks = ⌈n/ThdsPer Blk⌉;
  Do {
     INIT-KER<Blks,ThdsPerBlk>(s, d_s , p_s , n);
     SSSP-KER<Blks,ThdsPerBlk>(..,s, d_s , p_s , changed, n);
     CopyTo(DArray, d_s );            // From Device to Host.
     CopyTo(PArray, p_s );            // From Device to Host.
     CopyTo(hChanged, changed); // From Device to Host.
  }while (hChanged);
}
```

- Note we reuse d[] p[] across iterations
- We need the P[] for know the intermediate vertices tin the shortest path

KMB Algorithm G(V,E,W,L)

SSSP KERNEL(..,s, d_s , p_s , done, n)

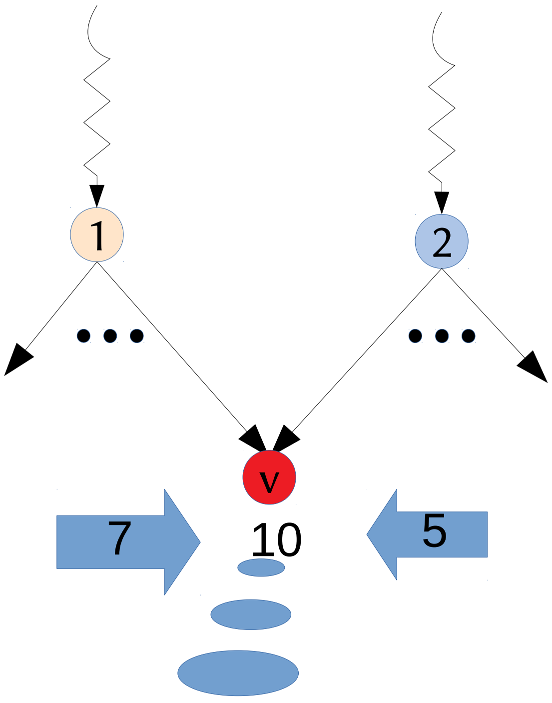
```
u=tid  // compute tid;
if tid < n  {
   for v ∈ adjacent[u] { // Using CSR arrays
      // Relax Operation (u, v, d_u )
      newCost = d_s[u] + W (u, v) ;
      old = d_s[v];
      if newCost < old then
          Atomic-MIN(d_s[v], newCost);
      // Updates Parent array
      if Atomic-MIN is success {
          p_s[v] = u;
          changed = true;
      }
}}
```

Note

- Parent should updated only if atomic min Success

Is it enough?

Parent update - Challenge



Suppose: 2 thread wants to update distance of its common Nbr v

```
<snip>
newCost = d[u] + W (tid, v) ;
old = d[v];
if newCost < old
   oldA=Atomic-MIN(d[v], newCost);

// AtomicMin is Success
if oldA != old  {
   // Updates Parent array
    p[v] = u;
    changed = true;
}
```

Parent update - Challenge

Time

Thread 1	Thread 2
Old=10 newCost=7	Old=10 newCost=5
d[v]=7 //oldA=10	
	d[v]=5 //oldA=7
	p[v]=2
p[v] =1	

```
<snip>
newCost = d[u] + W (tid, v) ;
old = d[v];
if newCost < old
   oldA=Atomic-MIN(d[v], newCost);

// AtomicMin is Success
if oldA != old  {
   // Updates Parent array
    p[v] = u;
    changed = true;
}
```

Wrong!

It is a challenge to find-out which "thread" updated distance to minimum

How update both distance and parent at the same time? Locks?

GPU Optimizations

- Synchronization
  - Push
  - Pull
- Computation
  - Data-driven
  - Edge-based
  - Controlled Computation unrolling
    - Δ
    - 2Δ
    - $t\Delta$
- Memory
  - Shared memory

Δ – max degree of the graph

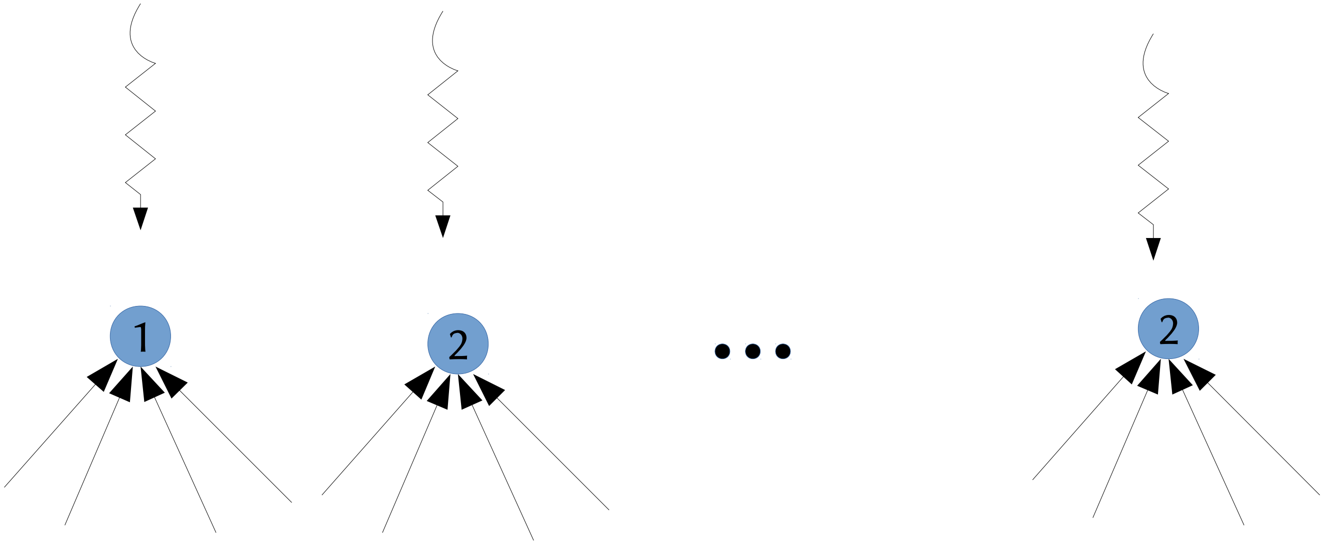
GPU Optimizations

- Synchronization
  - Push
  - **Pull**
- Computation
  - Data-driven
  - Edge-based
  - **Controlled Computation unrolling**
    - Δ
    - 2Δ
    - **$t\Delta$**
- Memory
  - **Shared memory**

These worked best for us!

Δ – max degree of the graph

Synchronization optimization • Pull




```
<snip>
newCost = d[v] + W (tid, v) ;
old = d[u];
if newCost < old {
   d[u] = newCost
   p[u] = tid;
   changed = true;
}
```

No Atomics
Parent update is easy

Because, one thread is wrting on one index

Compute optimization

- Data-driven
  - Needs Worklist
  - Active/Change nodes are inserted into WL
  - Only size of WL many threads launched
  - Need synchronization while inserting nodes in WL
- Edge based optimization
  - m-threads are launched
  - RELAXes one or group of edges
  - Representation needs to be modified.
- Computation Unrolling
  - Instead of one thread doing Δ work, perform more work per thread
  - Update also neighbours of neighbours (Δ^2)
  - **Repeat the work**; Say 2 times or t times (2Δ or $t\Delta$); e.g we do pull 3 times in the kernel • 3-pull
  - When t=3 is performance peaked for our testcase!

Memory optimization

- Programmable shared memory can be useful
- When there are multiple reads to DRAM
- We can move data to shared memory

- For e.g. In 3-pull, we moved CSR Adj list to shared
- As the neighbours Adjlist is accessed 3 times
- Of the total 48K per block
- when using 512 threadPerBlock we have 24 words to store per thread

- Hence, if degree(node) < 25 we use shared, we move CSR AdjList[node] to Shared
- With shared memory we achieve 24% of improvement in 3-pull

Key take-aways

- Solving Steiner Tree Problem is hard
- KMB Algorithm, an 2-approximation algorithm
- CPU Implementation has SSSP halt OPT
- SSSP with parent array parent update was challenging!
- Pull based SSSP is great for GPUKMB even without SSSP halt !

Experimental setup

t#	Graph	n	m	k	AvgDeg	MaxDeg	AvgWt	MaxWt
t1	t3-instance137.gr	97,928	1,28,632	902	2.63	14	2,486.70	2,71,369
t2	t3-instance163.gr	1,17,756	1,65,153	1,879	2.81	16	1,937.63	2,71,369
t3	t3-instance181.gr	1,35,543	2,01,803	3,033	2.98	16	1,795.73	2,58,185
t4	t3-instance183.gr	1,20,866	1,87,312	3,224	3.10	9	2,349.20	2,71,369
t5	t3-instance185.gr	66,048	1,10,491	3,343	3.35	16	1,03,010.50	2,15,10,249
t6	t3-instance187.gr	63,158	1,07,345	3,458	3.40	9	1,38,645.37	5,38,90,551
t7	t3-instance189.gr	1,72,687	2,55,825	3,902	2.96	10	2,826.30	2,71,369
t8	t3-instance191.gr	85,085	1,38,888	3,954	3.26	9	91,278.60	3,36,66,258
t9	t3-instance193.gr	17,127	27,352	4,461	3.19	4	19.71	126
t10	t3-instance195.gr	89,596	1,48,583	4,991	3.32	10	1,27,022.40	2,58,65,203
t11	t3-instance197.gr	2,35,686	3,66,093	6,313	3.11	14	2,375.55	2,71,369
t12	lin37.gr	38,418	71,657	172	3.73	4	56.17	198
t13	alue7080.gr	34,479	55,494	2,344	3.22	4	8.80	13
t14	Deezer-HR.gr*	54,573	4,98,202	3,000	18.26	420	1.00	1

Table 1. Benchmark graphs and their characteristics. Note *: t14 terminals chosen randomly with unit edge-weights

Experiments – Solutions and execution times

t#	OPT Value O	CPU Value C	GPU Value G	%deviation G vs C	%deviation G vs O	CPU Time T_C (in ms)	GPU Time T_G (in ms)
t1	11,300,427	13,038,882	13,046,669	+0.0597	+15.384	27,251.6	20,536.49
t2	13,391,485	15,075,540	15,082,504	+0.0462	+12.576	85,365.8	58,824.49
t3	20,086,478	23,054,415	23,057,792	+0.0146	+14.776	190,095	112,630.45
t4	24,998,365	28,441,249	28,446,853	+0.0197	+13.772	198,557	104,092.88
t5	793,246,106	804,364,415	804,363,424	-0.0001	+1.402	121,017	61,420.78
t6	863,275,877	869,507,927	869,511,663	+0.0004	+0.722	130,419	62,972.79
t7	40,927,523	47,966,818	47,986,233	+0.0405	+17.199	357,768	197,407.5
t8	977,020,806	989,323,251	989,322,713	-0.0001	+1.259	207,400	94,300.58
t9	184,908	197,752	198,199	+0.2260	+6.946	79,476.1	14,710.29
t10	1,406,041,806	1,424,191,280	1,424,195,265	+0.0003	+1.291	284,033	124,497.23
t11	51,655,792	58,240,533	58,251,361	+0.0186	+12.747	1,088,530	461,788.91
t12	99,560	106,937	107,297	+0.3366	+7.410	1,909.8	714.50
t13	62,449	65,529	66,290	+1.1613	+4.932	34,179.3	9,922.34
t14	-	4,629	4,595	-0.7345	-	82,829.8	8,311.58

Table 2. Comparison of solution quality and execution times (optimal value for t14 is not known; GPU time is KMBGPU's time, a pull-based implementation without any GPU optimizations.)

Experiments - Speed-up

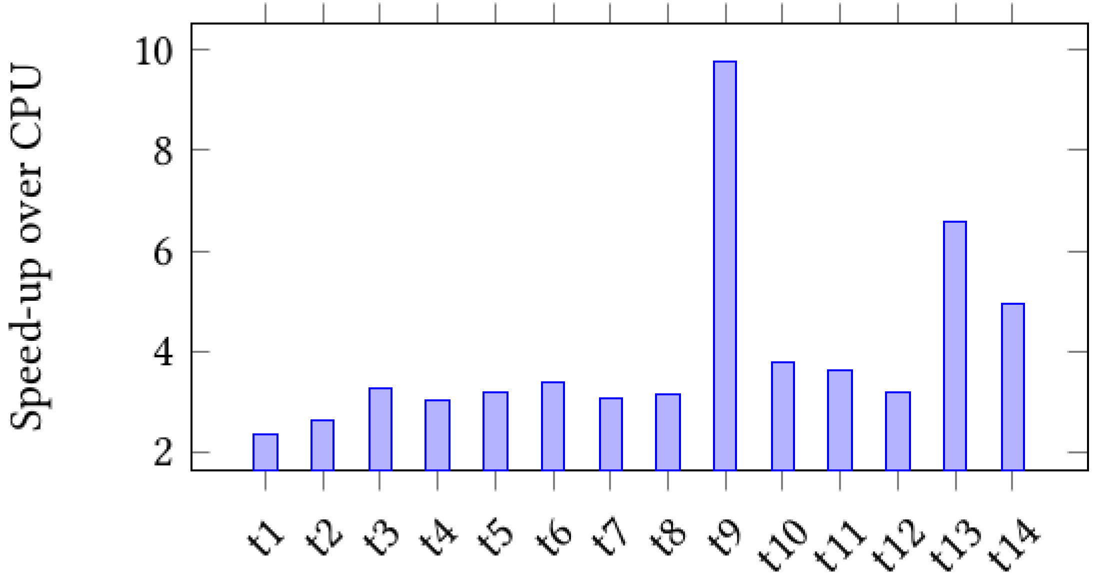


Figure 3. KMBGPU-OPT vs sequential CPU implementation

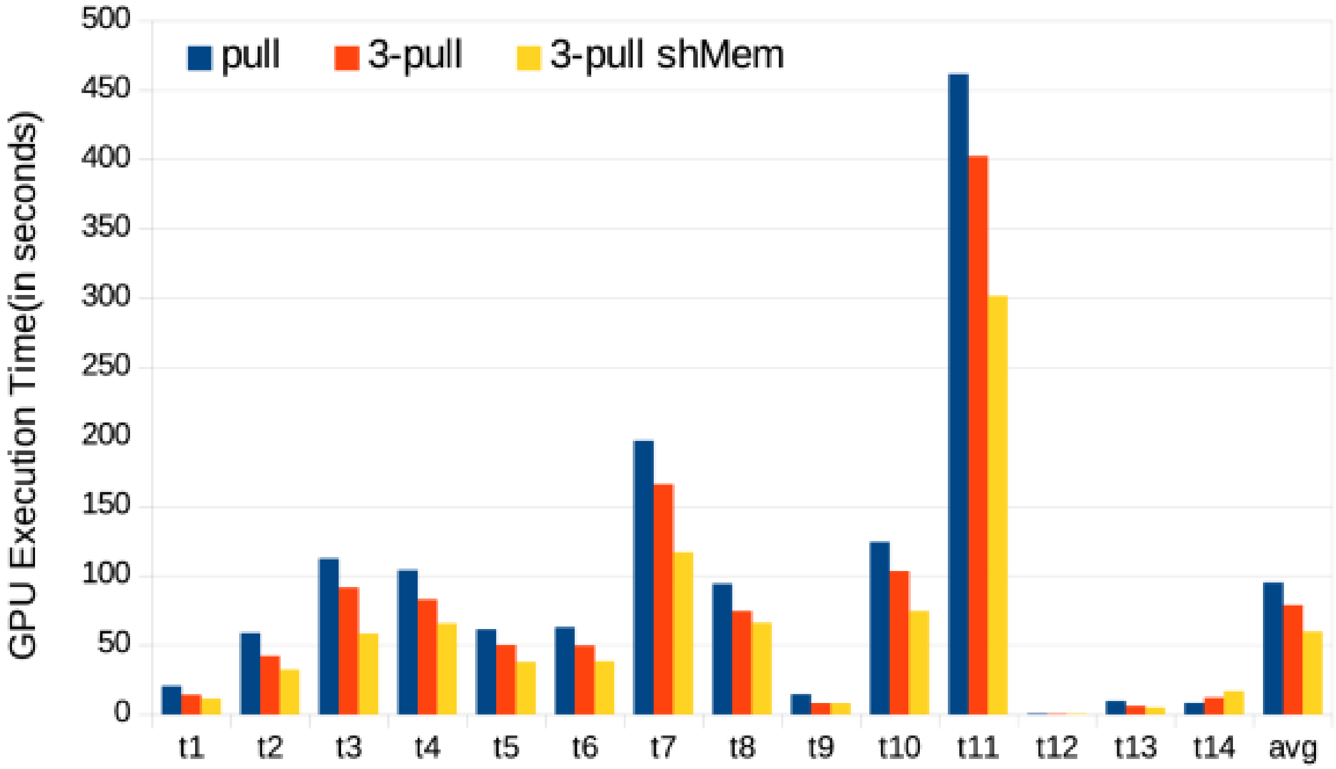
Experiments – SSSP comparision

- Single SSSP
  - Using TD CSR Based
  - Data-driven Based
  - Edge-based

t#	CSR Based C	DD Based D	Edge Based E
t1	27.522	798.555	1,518.323
t2	48.634	1,472.512	2,258.170
t3	63.399	1,994.612	3,271.143
t4	31.039	966.854	2,863.490
t5	23.418	607.420	925.442
t6	15.329	262.385	744.587
t7	57.979	1,827.458	5,767.550
t8	24.292	555.236	1,605.450
t9	4.157	25.633	91.128
t10	29.921	704.242	1,461.143
t11	106.415	4,293.516	11,044.733
t12	7.518	24.018	681.824
t13	5.822	25.646	179.849
t14	4.307	17.395	5,577.330

Table 3. Comparison of CSR, Data-driven and Edge based to compute SSSP distances with source a first vertex. Note: times are in milliseconds

Comparison of GPU time with Shared memory



Questions?

Thank you,
- for your time
- for your patience

References

- Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In High Performance Computing - HiPC 2007, 14*th International Conference, Goa, India, December 18-21, 2007, Proceedings.* 197–208.
  https://doi.org/10.1007/978-3-540-77220-0_21

- Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. *In 27th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013. 463–474.
  https://doi.org/10.1109/IPDPS.2013.28

- L. Kou, G. Markowsky, and L. Berman. 1981. A fast algorithm for Steiner trees. *Acta Informatica* 15, 2 (01 Jun 1981), 141–145.
  https://doi.org/10.1007/BF00288961
- And many…!