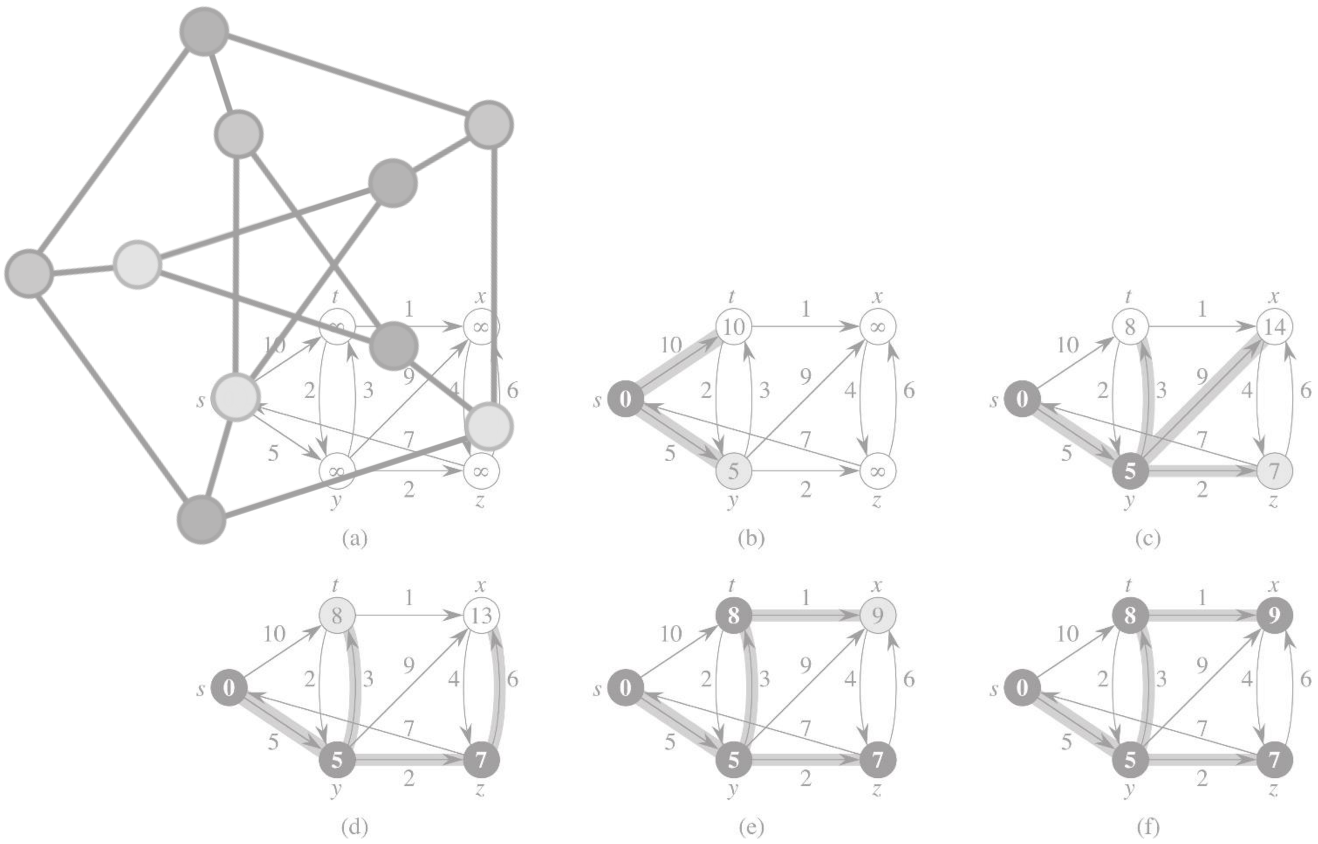
Chapter 3.

Accelerating Graph Algorithm: *Using CUDA*



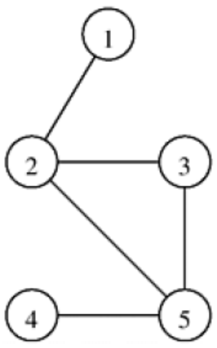
3. Introduction

An **undirected graph** *G* is a pair (*V*, *E*), where *V* is a finite set of points called **vertices** and *E* is a finite set of **edges**. An edge $e \in E$ is an unordered pair (*u*, *v*), where $u, v \in V$. Similarly, a **directed graph** *G*, is a pair (*V*, *E*), where *V* is the set of vertices, but an edge $(u, v) \in E$ is an ordered pair; that is it indicates that there is a connection from *u* to *v*. There two alternative forms of graph representation.

i) **Adjacency matrix representation**: It is preferred when the graph is **dense**, i.e. |*E*| is close to $|v^2|$. The adjacency matrix of this graph is an $n \times n$ array *A*, where n is the number of vertices. Let an element a_{ij} is defined as follows

$a_{ij} = 1$, if $(v_i, v_j) \in E$
=0, otherwise

For weighted graph
$a_{ij} = w(v_i, v_j)$, if $(v_i, v_j) \in$ E
=0, if $i=j$
=∞, otherwise



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Figure 3.1: Adjacency matrix of a graph

ii) **Adjacency list representation**: It is preferred when the graph is sparse, i.e. |*E*| is much less than $|v^2|$. The adjacency list representation of a graph *G* = (*V*, *E*) consists of an array Adj [1..|V|] of lists. For each $v \in V$, Adj [*v*] is a linked list of all vertices u such that *G* contains an edge $(v, u) \in E$. In other words, Adj [*v*] is a list of all vertices adjacent to v.

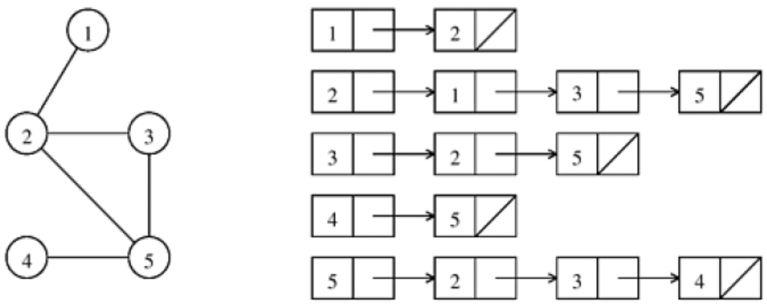


Figure 3.2: Adjacency list of a graph

3.1. Managing Graphs on CUDA

There are two popular data parallel programming approaches [1]:

- The iterative mask based approach, in which a set of vertices take part in execution at each iteration. We process each vertex in the mask in parallel. Synchronization occurs after execution of all vertices at every iteration.

- The divide-and-conquer approach, in which we divide the problem into its simplest constituent and process each constituent in parallel while merging them recursively as we move up the hierarchy. We give one thread to each constituent and process them in parallel.

In all implementations we map the problem to a data parallel scenario. We assume there can exist a thread for each vertex/edge in the graph. A **bulk synchronous** parallel programming model is used in implementing all algorithms.

3.1.1. Graph Representation:

Adjacency matrix representation is not suitable for large sparse graphs because of its *O(V^2)* space requirements, restricting the size of graphs that can be handled by the GPU. Adjacency list is a more practical representation for large sparse graphs requiring *O(V +E)* space. We represent graphs using a *compact adjacency list* representation with each vertex pointing to its starting edge list in a packed adjacency list of edges (Figure 3.3). CUDA model treats memory as general arrays and can support such representation efficiently. We assume the GPU can hold entire data into memory using this representation.

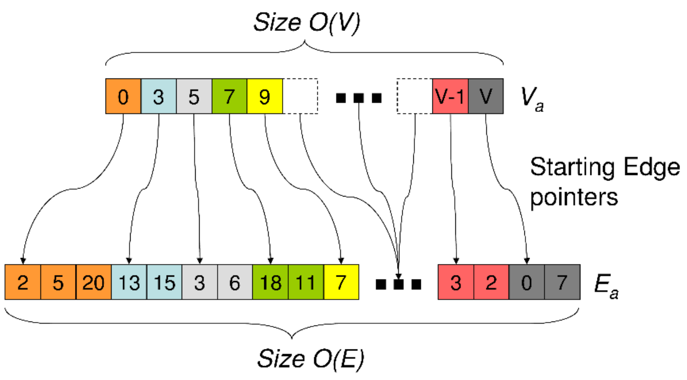


Fig: 3.3 Graph representation is in terms of a vertex list that points to a packed edge list.

Table 3.1
GENERAL VARIABLES USED IN GRAPH REPRESENTATION AND THE CPU_SKELETON CODE

Variable	Purpose
V_a	Holds starting index of edge list in E_a
E_a	Holds vertex id of outgoing vertex
W_a	Holds the weight of every edge
$Terminate$	Global variable written over by all threads to achieve consensus using logical OR

Table 3.1 states the variables used for representing graph in adjacency list format. The vertex list V_a points to its starting index in the edge list E_a. Each entry in the edge list E_a points to a vertex in vertex list V_a. W_a holds the edge weight for each edge. We deal with undirected graphs resulting in each edge having one entry for each of its end vertices.

3.1.2. Algorithm Outline on CUDA

The CUDA hardware can be seen as a multicore/manycore co-processor in a bulk synchronous parallel mode when used in conjunction with the CPU. Synchronization of CUDA threads can be achieved with the CPU deciding the barrier for synchronization. Broadly a bulk synchronous parallel machine follows three steps: (a) **Concurrent computation**: Asynchronous computation takes place on each processing element (PE). (b) **Communication**: PEs exchange data between each other. (c) **Barrier Synchronization**: Each PE waits for all PEs to finish their task. Concurrent computation takes place at the CUDA device in the form of program kernels with communication through the global memory. Synchronization is achieved only at the end of each kernel. Following Algorithm outlines the CPU code in this scenario. The skeleton code runs on the CPU while the kernels run on a CUDA device.

CPU SKELETON:

```
1. Create and initialize working arrays on CUDA device.
2. While NOT Terminate do
3.          Terminate ← true
4.          For each vertex/edge/color in parallel:
5.          Invoke Kernel1
6.          Synchronize
7.          For each vertex/edge/color in parallel:
8.          Invoke Kernel2
9.          Synchronize
10.         etc. ...
11.         For each vertex/edge/color in parallel:
12.         Invoke Kernel n and modify Terminate
13.         Synchronize
14.         Copy Terminate from GPU to CPU
15. End while
```

3.2. Breadth First Search (BFS)

The BFS problem is to find the minimum number of edges needed to reach every vertex in graph *G* from a source vertex *s*. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex reachable from *s*, the path in the breadth-first tree from *s* to *v* corresponds to a "shortest path" from *s* to *v* in *G*, that is the path containing the smallest number of edges.

BFS is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance *k* from *s* before discovering any vertices at distance *k+1*.

BFS was invented in the late 1950s by E. F. Moore, who used it to find the shortest path out of a maze, and discovered independently by C. Y. Lee as a wire routing algorithm (published 1961) [2].

Breadth-first search can be used to solve many problems in graph theory, such as, Copying garbage collection, Cheney's algorithm, Finding the shortest path between two nodes *u* and *v* (with path length measured by number of edges), Testing a graph for bipartiteness, Ford–Fulkerson method for computing the maximum flow in a flow network.

3.2.1. Sequential Procedure

The algorithm takes a graph *G*(*V*, *E*) and a vertex *s* as input. It then labels all verices reachable from *s*. The outline of the algorithm is given below. It uses an queue Q, for internal operations.

Algorithm 3.1: BFS_sequential

```
1.  Q. enqueue(s)
2.  label s as discovered and set distance[s] = 0
3.  while Q is not empty
4.          s ← Q.dequeue()
5.          process s
6.          for all adjacent vertices w of s do
7.                  if w is not labeled as discovered
8.                  Q.enqueue(w)
9.                  label w as discovered
10.                 Set distance[w] = distance[s] +1
11.                 end if
12.         end for
13. end while
```

3.2.2. An Example

The working of BFS procedure is illustrated in figure 3.4. Here white colour represents undiscovered state. Gray represents discovered not fully explored state. Black represents fully explored state.

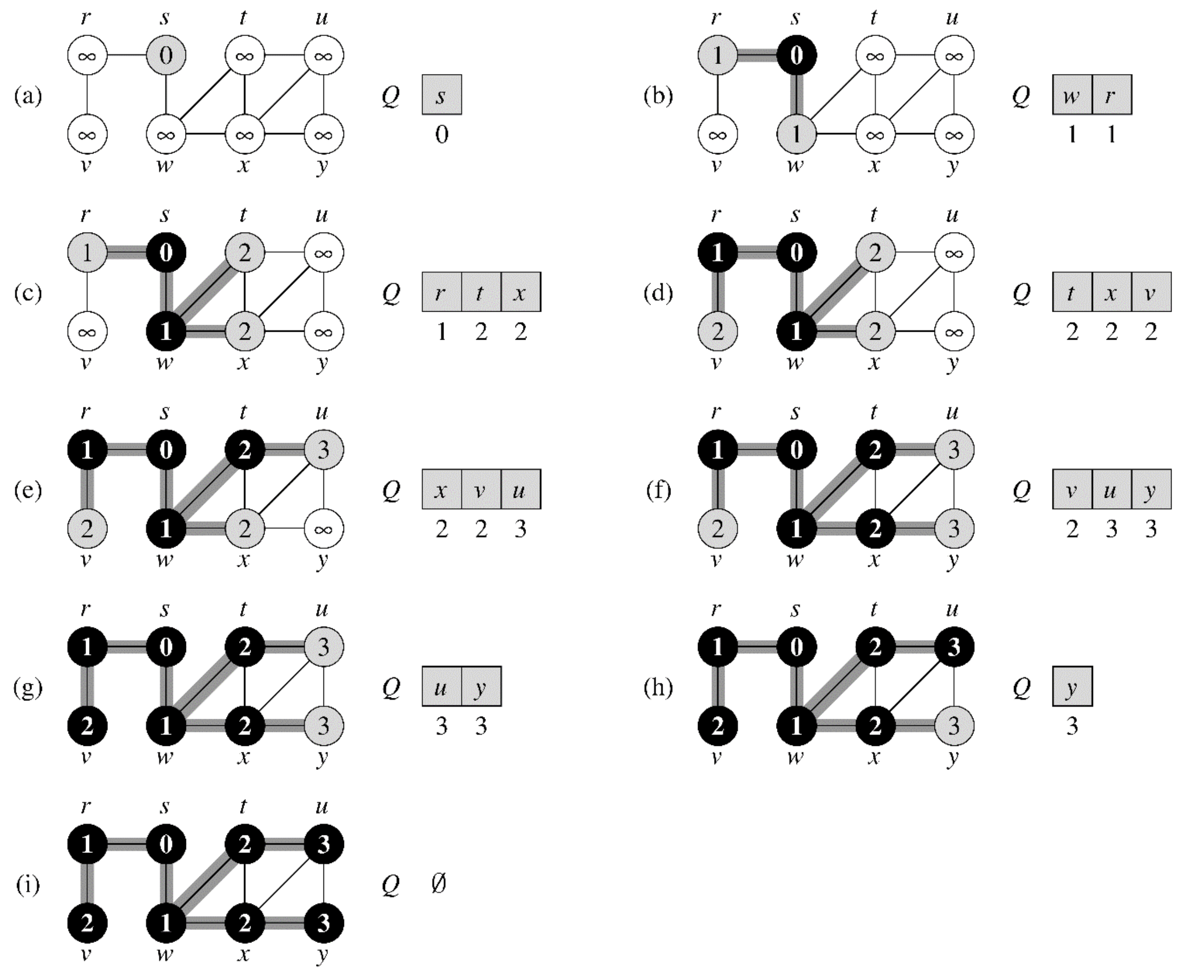


Figure 3.4: An example of breadth first search

3.2.3. Parallel Formulation

Fo this GPU is treated as a bulk synchronous device and is used as level synchronization to implement BFS. BFS traverses the graph in levels, once a level is visited it is not visited again during execution. This is used as barrier and threads are synchronized at each level. A BFS frontier corresponds to all vertices at the current level, see Figure 3.5. Concurrent computation takes place at the BFS frontier where each vertex updates the cost of its neighboring vertices by assigning cost values to their respective indices in the global memory.

Here one thread is assigned to every vertex. So queue is not required. Here we keep two Boolean arrays, frontier (F_a) and visited (X_a), of size $|V|$, which store the BFS frontier and the visited vertices. Another integer array, cost, C_a, stores the minimal number of edges of each vertex from the source vertex s. Initially, X_a is set to false and F_a contains only the source vertex. In the first algorithm (Algorithm 3.2), each thread looks at its entry in the frontier array F_a, if present, it updates the cost of its unvisited neighbors by writing its own cost plus one to its neighbor's index in the global cost array C_a. Another Boolean array F_{ua} is used to resolve write after read inconsistencies.

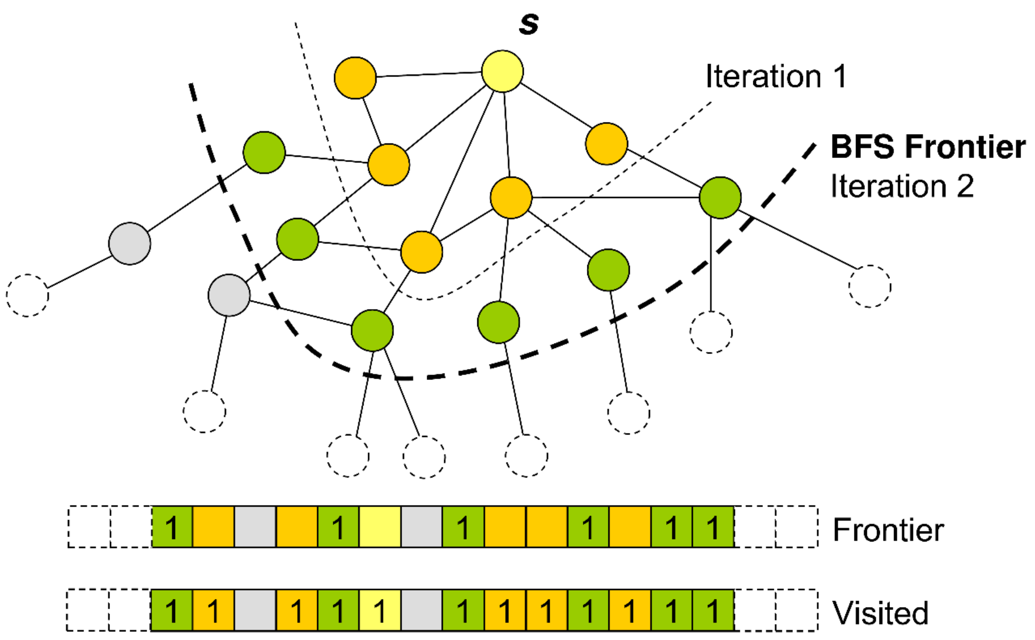


Figure 3.5: Paralllel BFS, vertices in the frontier list execute in parallel in each iterarion. Executuion stops when the frontier is empty.

Algorithm 3.2: BFS_KERNEL1

```
1.  $tid \leftarrow$ getThreadID
2.   if $F_a[tid]$ then
3.          $F_a[tid] \leftarrow$ false
4.          for all neighbors $nid$ of $tid$ do
5.                 if NOT $X_a[nid]$ then
6.                        $C_a[nid] \leftarrow C_a[tid]+1$
7.                        $F_{ua}[nid] \leftarrow$ true
8.                 end if
9.          end for
10. end if
```

Algorithm 3.3: BFS_KERNEL2

```
1.  $tid \leftarrow$ getThreadID
2.   if $F_{ua}[tid]$ then
3.          $F_a[tid] \leftarrow$ true
4.          $X_a[tid] \leftarrow$ true
5.          $F_{ua}[tid] \leftarrow$ false
6.          $Terminate \leftarrow$ false
7.   end if
```

3.3. ST-CONNECTIVITY (STCON)

St-connectivity or STCON is a decision problem asking, for vertices s and t in a directed graph, if t is reachable from s. The st-Connectivity problem resembles the BFS problem closely. Given an unweighted graph G(V,E) and two vertices, s and t, find a path from s to t assuming one exists.

3.3.1. Sequential Procedure

The algorithm takes a graph $G(V, E)$, source vertex s and terminating vertex t as input. It determines wheather t is reachable from s or not. It also gives the distance between s and t, if connected. The outline of the algorithm is given below. It uses two queues Q_R and Q_G, for internal operations.

Algorithm 3.4: ST-CON_Sequential

```
1.  Enqueue $s$ into $Q_R$
2.  Set $Rvisited[s]$ as true and $distance[s]$ as $0$
3.  Enqueue $t$ into $Q_G$
4.  Set $Gvisited[t]$ as true and $distance[t]$ as $0$
5.  Repeat
6.          Dequeue an item from $Q_R$ and put it into $v$
7.          For all adjacent vertices $u$ of $v$ do
8.                  If $Rvisited[u]$ is false
9.                          If $Gvisited[u]$ is true
10.                                 $dist \leftarrow distance[u] + distance[v] + 1$
11.                                 Terminate
12.                         End if
13.                         Enqueue $u$ into $Q_R$
14.                         Set $Rvisited[u]$ as true and $distance[u] \leftarrow distance[v] + 1$
15.                 End if
16.         End for
17.         Dequeue an item from $Q_G$ and put it into $v$
18.         For all adjacent vertices $u$ of $v$ do
19.                 If $Gvisited[u]$ is false
20.                         If $Rvisited[u]$ is true
21.                                 $dist \leftarrow distance[u] + distance[v] + 1$
22.                                 Terminate
23.                         End if
24.                         Enqueue $u$ into $Q_G$
25.                         Set $Gvisited[u]$ as true and $distance[u] \leftarrow distance[v] + 1$
26.                 End if
27.         End for
28. Until not terminated
```

3.3.2. Parallel Formulation

Our approach starts BFS concurrently from s and t with Red and Green colors assigned respectively to them. In each iteration, colors are propagated to neighbors along with the BFS cost. Termination occurs when both colors meet. Evidently, both BFS frontiers hold the smallest distance to the current processing vertex from their respective source vertices. The smallest path from s to t is reached when frontiers come in contact with each other. Figure 3.6 depicts two termination conditions due to merging of frontiers, either at a vertex or an edge. We set the Terminate variable to false in this implementation and each thread writes a true in this variable if termination condition is reached.

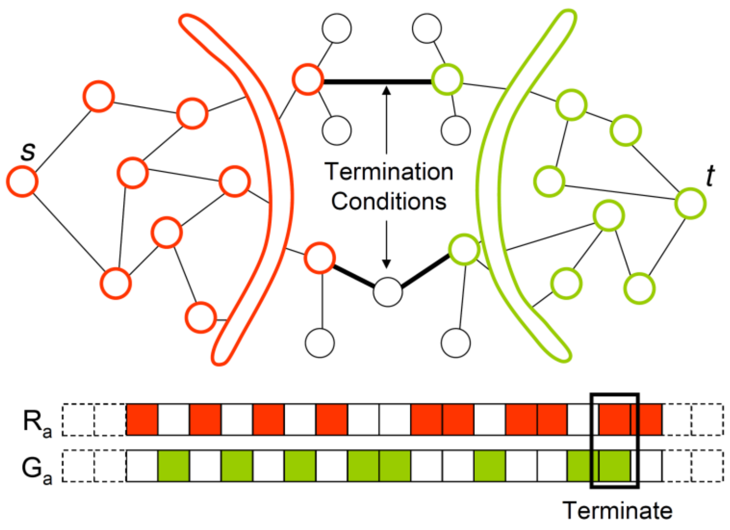


Fig: 3.6 Parallel ST-CON with colours expanding from s and t vertices.

Here we use Va, Ea, Fa, and Ca. beside this, we keep two boolean arrays Ra and Ga, for red and green colors, of size |V| as the vertices visited by s and t frontiers respectively, see Table 3.2.

Table 3.2

Variable	Purpose
Fa	Holds active vertices in each iteration
Ca	Holds the total length per vertex
Ra	Vertices visited by Red frontier
Ga	Vertices visited by Green frontier
Rf	Holds the current Red frontier value
Gf	Holds the current Green frontier value
Fua, Gua, Rua	Resolves read after write inconsistencies

R_a and G_a are set to false and F_a contains the source and target vertices. To keep the state of variables intact and avoid read after write inconsistencies, alternate updating arrays R_{ua}, G_{ua} and F_{ua} of size $|V|$ are also used in each iteration. Variables R_f and G_f keep track of the Red and Green frontier lengths at current execution. Each vertex, if present in F_a, reads its color in both R_a and G_a and sets its own color to one of the two. This is exclusive as a vertex can only exist in one of the two arrays, an overlap is a termination condition for the algorithm. Each vertex updates the cost of its unvisited neighbors by adding 1 to its own cost and writing it to the neighbor's index in C_a. Based on its color, the vertex also adds its neighbors to its own color's visited vertices by adding them to either R_{ua} or G_{ua}. The algorithm terminates if any unvisited neighbor of the vertex is of the opposite color. We need not update both frontiers for termination at an edge, only the Red frontier is updated in this case as shown in following Algorithm. The vertex removes itself from the frontier array F_a and adds its neighbors to the updating frontier array F_{ua}.

The second Kernel copies the updating arrays F_{ua}, R_{ua}, G_{ua} to actual arrays F_a, R_a and G_a for all newly visited vertices. It also checks the termination condition due to merging of frontiers and terminates the algorithm if frontiers meet at any vertex. Variables R_f and G_f are updated to reflect the current frontier lengths. The length of the path between s and t can be obtained by adding R_f and G_f. The algorithm needs a maximum of the radius of the Graph G iterations to terminate.

Algorithm 3.5: STCON_KERNEL1

```
1.  $tid \leftarrow$ getThreadID
2.  if $F_a[tid]$ then
3.      $F_a[tid] \leftarrow$ false
4.      for all neighbors $nid$ of $tid$ do
5.          if ($G_a[nid]$ | $R_a[nid]$) then
6.              if ($R_a[tid]$&$G_a[nid]$) then
7.                  $R_f \leftarrow C_a[tid]+1$
8.                  $Terminate \leftarrow$ true
9.              end if
10.             if ($G_a[tid]$&$R_a[nid]$) then
11.                 $Terminate \leftarrow$ true
12.             end if
13.         Else
14.             if $G_a[tid]$ then $G_{ua}[nid] \leftarrow$ true
15.             if $R_a[tid]$ then $R_{ua}[nid] \leftarrow$ true
16.             $F_{ua}[nid] \leftarrow$ true
17.             $C_a[nid] \leftarrow C_a[tid]+1$
18.         end if
19.     end for
20. end if
```

Algorithm 3.6: STCON_KERNEL2

```
1.  $tid \leftarrow$ getThreadID
2.  if $F_{ua}[tid]$ & not terminated then
3.      $F_a[tid] \leftarrow$ true
4.      if $R_{ua}[tid]$ then
5.          $R_a[tid] \leftarrow$ true
6.          $R_f \leftarrow C_a[tid]$
7.      end if
8.      if $G_{ua}[tid]$ then
9.          $G_a[tid] \leftarrow$ true
10.         $G_f \leftarrow Ca[tid]$
11.     end if
12.     $F_{ua}[tid] \leftarrow$ false
13.     $R_{ua}[tid] \leftarrow$ false
14.     $G_{ua}[tid] \leftarrow$ false
15.     if $G_a[tid]$ & $R_a[tid]$ then $Terminate \leftarrow$ true
16. end if
```

3.4. Single Source Shortest Path

For a given graph $G = (V, E)$, the single-source shortest paths problem is to find the shortest paths from a specified vertex $s \in V$ to all other vertices in V. The BFS algorithm can be used to find the shortest path from a given vertex. But it works only on an unweighted graph.

For a weighted graph, shortest path from *s* to *v* is a minimum-weight path. Depending on the application, edge weights may represent time, cost, penalty, loss, or any other quantity that accumulates additively along a path and is to be minimized. In the following section, we present Dijkstra's algorithm, which solves the single source shortest-paths problem on both directed and undirected graphs with non-negative weights.

Dijkstra's algorithm maintains a set V_T of vertices whose final shortest path weight from the source *s* have already been determined. That is, for each vertex $u \in V - V_T$, Dijkstra's algorithm stores $l[u]$, the minimum cost to reach vertex *u* from vertex *s* by means of vertices in V_T.

3.4.1. Sequential Procedure

The algorithm takes a weighted graph $G(V, E)$, source vertex *s* as input. It gives the minimum-weight path between *s* and all others vertices. The outline of the algorithm is given below.

Algorithm 3.7: SSSP_Sequential

```
1.  $V_T \leftarrow \{s\}$
2.  for all $v \in (V - V_T)$ do
3.      If $(s, v)$ exists set $l[v] \leftarrow w(s, v)$
4.      else set $l[v] \leftarrow \infty$
5.   while $V_T \neq V$ do
6.   begin
7.       find a vertex u such that $l[u] \leftarrow \min\{l[v] \mid v \in (V - V_T)\}$
8.       $V_T \leftarrow V_T \cup \{u\}$
9.       for all $v \in (V - V_T)$ do
10.          $l[v] \leftarrow \min\{ l[v], l[u] + w(u, v) \}$
11. end while
```

Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - V_T$ to add to set V_T so, we say that it uses a greedy strategy. Greedy strategies do not always yield optimal results in general, but Dijkstra's algorithm does indeed compute shortest paths [15].

3.4.2. Example

The execution of Dijkstra's algorithm is illustrated in figure 3.7. The source *s* is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate the edge used for cost calculation. Black vertices are in the set V_T, and white vertices are in the set $V - V_T$. (a) The situation just before the first iteration. The shaded vertex has the minimum value in *l* and is chosen as vertex u. (b)–(f) the situation after each successive pass of the while loop. The shaded vertex in each part is chosen as vertex u of the next iteration. (f) shows the final values.

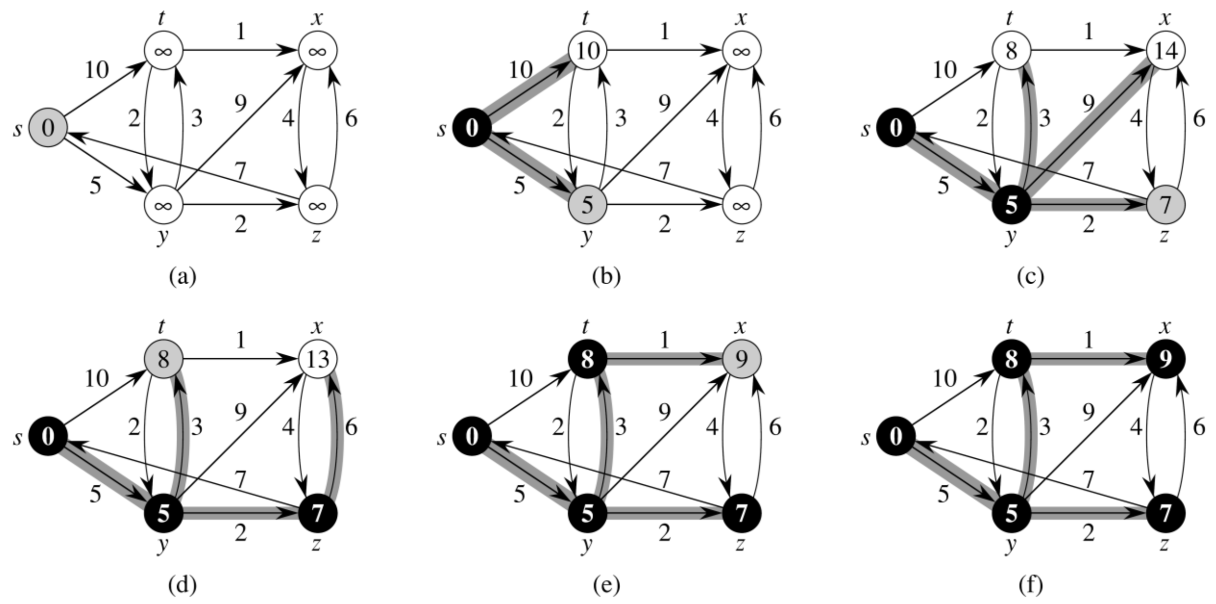
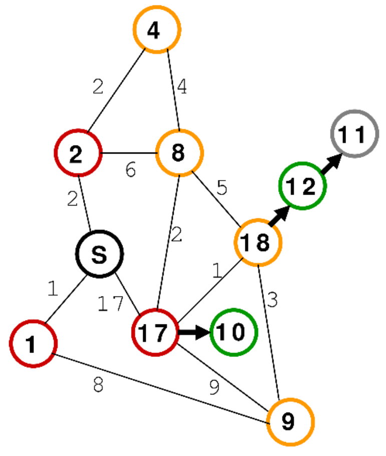


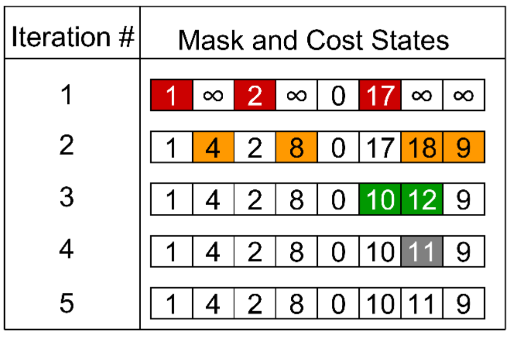
Figure 3.7: The execution of Dijkstra’s algorithm

3.4.3. Parallel Formulation

Single source shortest path does not traverse a graph in levels, as cost of a visited vertex may change due to a low cost path being discovered later in the execution. In our implementation simultaneous updates are triggered by vertices undergoing a change in cost values. These vertices constitute an execution mask. Termination condition is reached with equilibrium when there is no change in cost for any vertex.

We assign one thread to every vertex. Threads in the execution mask execute in parallel. Each vertex updates the cost of its neighbors and removes itself from the execution mask. Any vertex whose cost is updated is put into the execution mask for next iteration of execution. This process is repeated until there is no change in cost for any vertex. Figure 3.8 shows the execution mask (shown as colors) and cost states for a simple case, costs are updated in each iteration, with vertices undergoing re-execution if their cost changes.





Iteration #	Mask and Cost States							
1	1	∞	2	∞	0	17	∞	∞
2	1	4	2	8	0	17	18	9
3	1	4	2	8	0	10	12	9
4	1	4	2	8	0	10	11	9
5	1	4	2	8	0	10	11	9

Change in cost with each iteration Execution terminates when mask is empty

Fig: 3.8: SSSP execution

For our implementation we keep a Boolean mask M_a and cost array C_a of size $|V|$. W_a holds the weights of edges and an updating cost array C_{ua} is used for intermediate cost values.

Initially the mask M_a contains the source vertex. Each vertex looks at its entry in the mask M_a. If true, it updates the cost of its neighbours if greater than its own cost plus the edge weight to the corresponding neighbour in an alternate updating cost array C_{ua}. The alternate cost array C_{ua} is used to resolve read after write inconsistencies in the global memory. Updates in C_{ua} need to lock the memory location before modifying the cost value, as many threads may write different values at the same location concurrently. We use the `atomicMin` function supported on CUDA (lines 5 – 9, following Algorithm) to resolve this.

Algorithm 3.5: SSSP_KERNEL1

```
1.  $tid \leftarrow$ getThreadID
2.  if $M_a[tid]$ then
3.      $M_a[tid] \leftarrow$ false
4.      for all neighbors $nid$ of $tid$ do
5.          Begin Atomic
6.          if $C_{ua}[nid] > C_a[tid] + W_a[nid]$ then
7.              $C_{ua}[nid] \leftarrow C_a[tid] + W_a[nid]$
8.          end if
9.          End Atomic
10.     end for
11. end if
```

Algorithm 3.6: SSSP_KERNEL2

```
1.  $tid \leftarrow$ getThreadID
2.  if $C_a[tid] > C_{ua}[tid]$ then
3.      $C_a[tid] \leftarrow C_{ua}[tid]$
4.      $M_a[tid] \leftarrow$ true
5.      $Terminate \leftarrow$ false
6.  end if
7.  $C_{ua}[tid] \leftarrow C_a[tid]$
```

Atomic functions resolve concurrent writes by assigning exclusive rights to one thread at a time. The clashes are thus serialized in an unspecified order. The function compares the existing $C_{ua}(v)$ cost with $C_a(u) + W_a(u, v)$ and updates the value if necessary. A second kernel (kernel2) is used to reflect updating cost C_{ua} to the cost array C_a. If C_a is greater than C_{ua} for any vertex, it is set for execution in the mask M_a and the termination flag is toggled to continue execution. This process is repeated until the mask is empty. The algorithms takes the order of diameter of the graph to converge to equilibrium.

3.5. Analysis

Here we present performance analysis of our algorithms and also compare them against their sequential formulations. For this purpose we also developed sequential programs using standard C++ with STL support using GNU C/C++ compiler.

3.5.1. Graphs used

We choose some real world graphs [4] ranging from a few to 1M vertices for all algorithms. The larger graphs are based on New York City and Northwest USA.

3.5.2. Experimental Setup

We conducted our experiments on single Nvidia GT 520 graphics adapter with 2048MB memory, controlled by a quad core processor (AMD Athlon II x4) with 8GB RAM running Ubuntu 14.04 LTS. We implemented our programs with CUDA 7.0-28 SDK with display driver 346.46. We will refer to this configuration as Machine1 now on.

We also took another hardware setup. Which is identical to the previous one except, this one has Nvidia GT 730 graphics adapter instead of GT 520. We will refer to this configuration as Machine2 now on.

3.5.3. Results

Analysis of BFS

Our BFS algorithm takes order of diameter of the graph to terminate. This makes it very efficient. The same can be shown with the running time of our algorithm as given in Table 3.3. As the graph size increases our parallel algorithm gives better result. The table also express that using newer GPU the algorithms becomes more powerful.

Table 3.3

Vertices	Edges	Sequential (Athlon II x4 630)	Machine1 (GT 520)	Machine2 (GT 730)
6	20	5 msec	62.982 msec	118 msec
9	16	7 msec	71.047 msec	121 msec
264,346	733,846	3.52 sec	1.38 sec	341 msec
1,207,945	2,840,208	10.6 sec	1.93 sec	1.06 sec

Analysis of STCON

Our STCON algorithm takes order of diameter of the graph to terminate. The running time of our algorithm as given in Table 3.4, which signifies the efficiency of our algorithm. As the graph size increases, our parallel algorithm gives better result. The table also express that using newer GPU the algorithms becomes more fruitful. However, they can also need more setup time as seen in the table for small graphs.

Table 3.4

Vertices	Edges	Sequential (Athlon II x4 630)	Machine1 (GT 520)	Machine2 (GT 730)
6	20	6 msec	60.455 msec	116 msec
9	16	7 msec	61.41 msec	119 msec
264,346	733,846	1.81 sec	833.37 msec	145 msec
1,207,945	2,840,208	4.64 sec	1.667 sec	580 msec

Analysis of SSSP

Our SSSP algorithm takes order of diameter of the graph to converge. Table 3.5 shows the running time of our algorithms; which proves our parallel algorithm to be efficient. As the graph size increases, our parallel algorithm gives better result. The table also express that using newer GPU the algorithms becomes more powerful. However, they can also need more setup time as seen for small graphs.

Table 3.5

Vertices	Edges	Sequential (Athlon II x4 630)	Machine1 (GT 520)	Machine2 (GT 730)
6	20	10 msec	59.724 msec	121 msec
9	16	12 msec	64.897 msec	124 msec
264,346	733,846	7min 7sec	4.712 sec	825 msec
1,207,945	2,840,208	2hr 31min	42.025 sec	1.06 sec

References

[1] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA” in HiPC 2007
[2] https://en.wikipedia.org/wiki/Breadth-first_search
[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, “Introduction to Algorithms”
[4] “The Ninth DIMACS implementation challange on shortest paths” available at http://www.dis.uniroma1.it/challenge9/