Accelerating large graph algorithms on GPU using CUDA

Archisman Pathak, Kousshik Raj, Koustav Chowdhury, Rounak Patra
Satyam Porwal, Siddhant Agarwal, Sriyash Poddar
-
Department of Computer Science and Engineering, IIT Kharagpur

ABSTRACT

In recent times, the use of large-scale graphs involving millions of vertices are quite common in various practical applications like social networks, search engines, maps, etc., and often take up high computational resources when processing such graphs for traversal or finding shortest path between nodes. Although practical sequential implementations are possible using high-end computers and have been reported, such resources are not easily accessible. Moreover, the performance of such algorithms degrades drastically with increasing graph size. To this end, possiblity of parallelisation of graph algorithms using Graphical Processing Units (GPU) have gained significant importance, considering their high computational power and affordability, though their restrictive programming model is often difficult to get past. In this paper, we present the parallelisation of three fundamental class of graph problem - Breadth First Search, Single Source Shortest Path, and All-Pair Shortest Path, using CUDA based implementation on GPUs. We have also profiled the performance of the algorithms on a diverse set of generated and procured large graphs and have observed significant speedups as compared to the results of a sequential algorithm.

KEYWORDS

Breadth-First search, GPU, CUDA, Graph Algorithms, Parallel Algorithms, Shortest Path Algorithms

1 INTRODUCTION

Graph algorithms are a very common requirement in several problem domains including several scientific and engineering applications, which involves processing large graphs that have millions, if not billions, of vertices. Fundamental graph operations like breadth-first search (BFS), depth-first search (DFS), shortest path, etc., occur frequently in these domains. While fast sequential implementations of the famous algorithms for these problems exists [1, 2], they are of the order of number of vertices and edges. On very large graphs, these algorithms become practically impractical. But when we switch to using parallel algorithms, we can achieve much more reasonable and practical implementations, in terms of both time and hardware cost on these basic graph problems [3].

Commodity graphics hardware has become a cost-effective parallel platform to solve many general problems. Many problems in the field of image processing, computer vision, signal processing [4], etc., have benefited from its speed and parallel processing capabilities. Paulius et. al [5] provided a GPU based implementation of All-Pairs Shortest Paths Problem, but this was severely limited by the memory capacity and architecture of the existing GPUs of those times. These were then optimized for graphics operations and their programming model is highly restrictive. Hence all the algorithms were disguised as graphic rendering passes with programmable shaders to interpret the data.

Nvidia's Compute Unified Device Architecture (CUDA) offers an alternative to such a programming model. In this paper, we will discuss various CUDA-based implementation of the solutions to these few fundamental graph problems and show results obtained by Breadth First Search (BFS), Single-Source Shortest Path (SSSP) and All-Pair Shortest Path (APSP) for large graphs on a Nvidia GPU.

The paper is organised as follows. Section 2 tackles the fine grained details of the implementation of any generic graph algorithm on CUDA. In Section 3, 4, 5, we present some parallelised approaches for the BFS, SSSP, and APSP problems. Then, we discuss the experiments performed on the presented algorithms using large graph datasets in Section 6. And finally in Section 7 we conclude our work.

2 CUDA IMPLEMENTATION

In general, though GPUs offer high computing powers by using the SIMD class of architectures, they often come up with highly restrictive programming model. Because of this, we try to use CUDA, which provides a high-level interface which can be used as an extension of the C/C++ programming language, thereby making it quite straightforward to implement algorithms in it. Any GPU tries to execute a parallel algorithm by creating multiple threads and scheduling them for execution in batches called warps. Each thread executes a common set of instructions (called kernel) along with a unique thread identifier which allows us to differentiate the tasks it performs. For example, in the parallel BFS (Section 3) and SSSP (Section 4) algorithms each thread is responsible for a single vertex, whereas in the naive parallel Floyd-Warshall (Section 5.2) algorithm, each thread takes care of a pair of vertices. Now we will look at some implementation technicalities.

2.1 Graph Representation in CUDA

In order to fully utilise the functionalities that a GPU paired with CUDA can offer, we need a suitable way to represent our input graph for the algorithms to work with. As we know, an input graph can be represented by an adjacency list or an adjacency matrix. But directly working with such a format will not result in optimal performance because of memory access of uncontinguous segment of data. To overcome this, we try to reframe the given input into a form which facilitates contiguous memory access during execution. We will do so both for the adjacency list as well as the adjacency matrix.

Assume, we are given a graph $G(V, E, W)$, where V is the set of vertices, E and W are the set of edges and their corresponding weights. Let's ignore the weights for now and focus on the vertices and edges, as it is easy to incorporate the weights into any graph

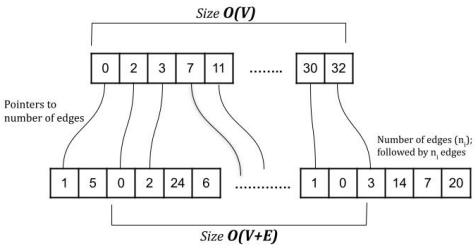

Figure 1: Graph representation with vertex list pointing to a packed edge list

representation. We are going to convert the original adjacency list representation into a compact adjacency list form of a single contiguous array E_a. To do so, we number the vertices from 1 to $|V|$, after which we try to store all the endpoints of edges with one of the endpoint being vertex i, $\forall 1 \leq i \leq |V|$ in order in E_a. To determine the endpoint of each edge list, we start each of them with their size. But, with this array its difficult to find the starting point of a edge list of a particular vertex in $O(1)$. To make this possible, we add another auxiliary array V_a of size $|V|$ where each element in it points to the start position of the edge set of the corresponding vertex in E_a. Fig. 1 exemplifies this clearly. To incorporate weights into this representation, its enough to introduce another array W_a of same size as E_a with each element in it holding the weight of the corresponding edge in E_a.

Its much easier to convert the adjacency matrix of a graph representation into a suitable form for our purpose. An adjacency matrix is a 2D array A of size $|V| \times |V|$, in which the element $A[u, v]$ holds the weight of the edge from vertex u to v (this means it can represent only simple graphs). In this case, we will just flatten the matrix A into a 1D array E of size $|V|^2$, with an element $A[u, v]$ mapping to the corresponding element $E[u * |V| + v]$. Now, we can use the array E in our algorithms.

3 BREADTH FIRST SEARCH

In the BFS problem you are given an undirected, unweighted graph $G(V, E)$ and a source vertex S, and we need to find the minimum number of edges needed to reach every vertex in G from source vertex S. An asymptotically optimal sequential solution for the problem takes $O(V + E)$ time. BFS has been used in state space searching, graph partitioning, automatic theorem proving, etc., and is one of the most used graph orientation in various practical graph algorithms. We will now discuss various CUDA based approaches to tackle the problem of BFS.

3.1 First Approach: Parallel BFS

In this approach, we will use level synchronization to solve BFS. The graph is traversed level by level and once visited it is not visited again. The BFS *distance* stores the level at which the nodes were processed. The *level* variable keeps track of the level being processed. Maintaining a queue for each vertex would incur additional overheads and thereby slow down the speed of execution. Instead in this implementation we give one thread to each vertex. In each iteration, if a vertex is at the same level as the level being processes, it fetches its cost from the cost array and updates its neighbours if more than 'its cost + 1' i.e. the neighbour was univisted. Here, a global flag keeps track of the changes made in each iteration. If no change is made in an iteration it implies that all vertices have been visited and BFS completed. The above approach has been presented in Algorithm 1 and the corresponding kernel in Algorithm 2.

Algorithm 1 parallelBFS_Host

```
1:  Input: V_a, E_a, S                          ▷ The graph G(V, E) and source S
2:  Create distance array Dist_a, and parent array P_a of size |V|
3:  Initialise all elements of Dist_a, P_a to ∞
4:  D_a[S] = 0
5:  level = 0
6:  flag = True
7:  while flag do
8:      flag = False
9:      Invoke parallelBFS_kernel(level, V_a,E_a,Dist_a,flag).
10:     level = level + 1
```

Algorithm 2 parallelBFS_kernel

```
1:  Input: level, V_a,E_a,Dist_a,flag
2:  tid = getThreadID
3:  f = False
4:  if tid < Va_size and Dist_a[tid] = level then
5:      u = tid
6:      for all v = neighbours of u do
7:          if level + 1 < Dist_a[v] then
8:              Dist_a[v] = level + 1
9:              f = True
10:     if f = True then
11:         flag = True
```

3.2 Second Approach: Queue BFS

In this approach as well, we will use level synchronization to solve BFS. Like we have seen earlier, the graph is traversed level by level and once visited it is not visited again. But here, we maintain a queue cQ of the vertices to be visited. A cost array, C_a, stores the minimal number of edges of each vertex from source S. In each iteration, we traverse through the queue, and for each vertex u, if we find a unvisited neighbour v, we update the distance value for v, and add it to the queue of vertices to be expanded next, nQ. We use *atomicMin* and *atomicAdd* operations to avoid race conditions. The above approach has been presented in Algorithm 3 and the corresponding kernel in Algorithm 4.

3.3 Third Approach: Scan BFS

A work-efficient parallel BFS algorithm should perform $O(V + E)$ work. To achieve this, each iteration should examine only the edges and vertices in that iteration's logical edge and vertex-frontiers, respectively. Edge frontiers are vertices that needs to be examined in the next step of the BFS. In this algorithm, frontier is managed out-of-core and is fully produced in off-chip memory for consumption by the next BFS iteration after a global synchronization step.

Algorithm 3 queueBFS_Host

Input: V_a, E_a, S ▷ The graph $G(V, E)$ and source S
Create cost array $Dist_a$ and parent array P_a of size $|V|$ and initialise all values to ∞
Create two array cQ and nQ, and initialise it to S and *null* respectively.
$Dist_a[S] = 0$
$P_a[S] = -1$
$l = 0$ ▷ Start with the source vertex
while $cQ_{size} > 0$ **do**
    Invoke queueBFS(l, V_a, E_a, $Dist_a$, P_a, cQ, nQ)
    $swap(cQ, nQ)$
    Set nQ to *null*
    $l = l + 1$

Algorithm 4 queueBFS_kernel

Input: l, V_a, E_a, $Dist_a$, P_a, cQ, nQ ▷ The graph $G(V, E)$ and source S
tid = getThreadID
if $tid < cQ_{size}$ **then**
    $u = cQ[tid]$
    for all v = neighbours of u **do**
        if $Dist_a[v] = \infty$ and $atomicMin(Dist_a[v], l + 1) = \infty$ **then**
            $P_a[v] = u$
            $pos = atomicAdd(nQ_{size}, 1)$
            $nQ[pos] = v$

The intuition behind the linear optimisation of BFS is quite same as the standard BFS. We maintain a queue of unvisited vertex, level-synchronised, and we terminate when the queue is empty. The only challenge is to populate this queue efficiently. Here, we employ prefix scan to find out the position of vertices in queue for the next iteration. For each vertex in the queue, we assign a new thread to compute the frontiers for the next iteration.

The above approach has been presented in Algorithm 5, and the kernels in the subsequent Algorithms 6, 7, 8 and 9.

4 SINGLE-SOURCE SHORTEST PATH

In the Single Source Shortest Path (SSSP) problem you are given a weighted graph $G(V, E, W)$ with non-negative weights (we do not consider negative weights in our algorithm) and a source vertex S, and we need to find a path to every vertex V from source S such that the sum of weights in it is the smallest of all such paths. One of the optimal sequential algorithm is the well known Dijkstra's algorithm [6] which has a time complexity of $O(VlogV + E)$ when implemented with Fibonacci Heaps [7]. In this section, we will try to parallelise the above algorithm to speed up its execution. We will consider the input graph in the above mentioned compact adjacency list format with elements in V_a pointing to the start of their adjacency list, E_a and W_a consisting of all the edges and their corresponding weights, respectively.

Algorithm 5 ScanBFS_Host

Input: V_a, E_a, S ▷ The graph $G(V, E)$ and source S
Create updating cost array Deg_a, $PreDeg_a$ of size $|V|$ and initialise all values to 0
Create cost array $Dist_a$ of size $|V|$ and initialise all values to ∞
Create mask array P_a of size $|V|$ and initialise all values to -1
Create two array cQ and nQ, and initialise it to S and *null* respectively.
$Dist_a[S] = 0$
$P_a[S] = -1$
$l = 0$ ▷ Start with the source vertex
while $cQ_{size} > 0$ **do**
    Invoke nextLayer(l, V_a, E_a, P_a, $Dist_a$, cQ)
    Invoke countDegrees(V_a, E_a, P_a, cQ, Deg_a)
    Invoke scanDegrees(cQ_{size}, Deg_a, $PreDeg_a$)
    Perform Prefix Sum on Deg_a, and store the results in $PreDeg_a$
    $nQ = PreDeg_a[cQ_{size}/NUM\_THREADS]$
    Invoke populateNextQueue(V_a, E_a, P_a, cQ,nQ, Deg_a, $PreDeg_a$)
    $cQ = nQ$
    $l = l + 1$

Algorithm 6 nextLayer

Input: l, V_a, E_a, P_a, $Dist_a$, cQ
tid = getThreadId() ▷ Get the Id of the thread
if $tid < cQ_{size}$ **then**
    $u = cQ[tid]$
    for all v = neighbours of u **do**
        if $Dist_a[v] > l + 1$ **then**
            $Dist_a[v] = l + 1$
            $P_a[v] = u$

Algorithm 7 countDegrees

Input: V_a, E_a, P_a, cQ, Deg_a
tid = getThreadId() ▷ Get the Id of the thread
if $tid < cQ_{size}$ **then**
    $u = cQ[tid]$
    $d = 0$
    for all v = neighbours of u **do**
        if $P_a[v] = E_a.index(v) and v \neq u$ **then**
            $d = d + 1$
    $Deg_a[tid] = d$

4.1 First Approach: Bugged Parallel Djikstra

One of the foremost algorithms that was presented for this purpose is by Harish et. al [8]. The algorithm presented in their work relies on the fact that a sequential algorithm of the Djikstra's algorithm finalises the shortest distance from the source of only a single vertex every iteration, which is the bottleneck for the algorithm. But in large scale graphs, in every iteration the distance of scores of such vertices can possibly be confirmed. This fact is leveraged by their work, and we have the following algorithm.

Algorithm 8 scanDegrees

Input: $cQ_{size}, Deg_a, PreDeg_a$
tid = getThreadId() ▷ Get the Id of the thread
if $tid < cQ_{size}$ **then**
  Create a shared array $preSum$ of size $NUM\_THREADS$
  $m = threadId.x$
  $preSum[m] = Deg_a[tid]$
  $sync\_threads$
  $n = 2$
  while $n \leq NUM\_THREADS$ **do**
    if $bitwiseAnd(m, n-1) = 0$ and $tid + (2 * n) < cQ_{size}$ **then**
      $preSum[m] += preSum[tid + (2 * n)]$
    $sync\_threads$
    $n = 2 * n$
  if $m = 0$ **then**
    $PreDeg_a[tid/NUM\_THREADS + 1] = preSum[m]$
  $n = NUM\_THREADS$
  while $n > 1$ **do**
    if $bitwiseAnd(m, n-1) = 0$ and $tid + (n/2) < cQ_{size}$ **then**
      $temp = preSum[m]$
      $preSum[m] += preSum[tid + (n/2)]$
      $preSum[tid + (n/2)] = temp$
    $sync\_threads$
    $n = n/2$
  $Deg_a[tid] = preSum[m]$

Algorithm 9 populateNextQueue

Input: $V_a, E_a, P_a, cQ, nQ, Deg_a, PreDeg_a$
tid = getThreadId() ▷ Get the Id of the thread
if $tid < cQ_{size}$ **then**
  Initialise a shared variable i
  if $threadId.x = 0$ **then**
    $i = PreDeg_a[NUM\_THREADS]$
  $sync\_threads$
  $s = 0$
  if $threadId.x \neq 0$ **then**
    $s = Deg_a[tid - 1]$
  $u = cQ[tid]$
  $c = 0$
  for all v = neighbours of u **do**
    if $P_a[v] = E_a.index(v)$ and $v \neq u$ **then**
      $nQ[i + s + c] = v$
      $c = c + 1$

In the algorithm, other than the input, an array C_a to store the shortest distance from the source vertex, a boolean mask array M_a where it marks whether a potential shortest path to some vertex passes through this vertex, and the updating cost array U_a which acts as an auxillary for updating the cost array which otherwise would result in data hazards. It also maintains a boolean variable $flag$ to determine the termination of the algorithm.

The algorithm is executed in multiple iterations, and each iteration consists of two phases. In the first phase, all the vertices marked in M_a are treated as potential vertices through which a shortest path from source to some vertex can pass through, and the distance to the neighbouring vertices of such vertices are updated if they are found to be shorter by passing through them. In this phase, all the updates occur in U_a and the shortest distances are read from C_a. This is done to prevent Read After Write (RAW) data hazards that crop up due to the parallelised execution. The pseudo code for this is shown in Algorithm 11.

In the end of this phase we have a possibly shorter path at U_a, which has to be transferred to the array C_a for the next iteration. This is taken care of in the second phase of the algorithm. Here, the cost of each vertex in C_a is compared to the one maintained in U_a and if a better cost is encountered, the corresponding distance in C_a is updated. Furthermore, this vertex is marked *true* in M_a, as now there might be a shortest path through this vertex. If there is any such vertex, then the $flag$ variable is marked true as well. Otherwise, $flag$ will not be updated, resulting in the termination of the algorithm. The parallel execution takes place in these two phases, where each vertex is managed by a thread. But care has to be taken to synchronise all the threads before calling the next phase. This phase and the host code is exemplified in Algorithm 10 and Algorithm 12, respectively. The final shortest distance is stored in the array C_a.

Algorithm 10 SSSP_Host

Input: V_a, E_a, W_a, S ▷ The graph $G(V, E, W)$ and source S
Create updating cost array U_a of size $|V|$ and initalise all values to ∞
Create cost array C_a of size $|V|$ and initalise all values to ∞
Create mask array M_a of size $|V|$ and initialise all values to $false$
$U_a[S] = 0$
$C_a[S] = 0$
$M_a[S] = flag = true$ ▷ Start with the source vertex
while $flag$ **do**
  $flag = false$
  for all $v \in V$ in parallel **do**
    Invoke SSSP_Phase1(V_a, E_a, W_a, C_a, U_a, M_a)
    Invoke SSSP_Phase2(C_a, U_a, M_a, $flag$)

Algorithm 11 SSSP_Phase1

Input: $V_a, E_a, W_a, C_a, U_a, M_a$
tid = getThreadId() ▷ Get the Id of the thread
if $M_a[tid] = true$ **then**
  $M_a[tid] = false$
  for all neighbours nid of tid **do** ▷ Line 6, 7 must be atomic
    if $U_a[nid] > C_a[tid] + W_a[nid]$ **then**
      $U_a[nid] = C_a[tid] + W_a[nid]$

Algorithm 12 SSSP_Phase2

```
1: Input: C_a, U_a, M_a, flag
2: tid = getThreadId()                    ▷ Get the Id of the thread
3: if C_a[tid] > U_a[tid] then
4:     M_a[tid] = flag = true
5:     C_a[tid] = U_a[tid]
```

4.2 Second Approach: Corrected Version

The above algorithm might look perfectly fine at a first glance, but hidden within it is an unsuspecting mistake with a devastating consequence. Let us iterate through the algorithm once more. The host code in Algorithm 10 will work fine as long as the individual phases does what they are supposed to do. Now let's move on to phase two in Algorithm 12, assuming the phase one works fine. There seems to be a race condition in line 4 when assigning $true$ to the variable $flag$. But, in CUDA, multiple writes to the same location by different threads will have an undetermined effect except for the fact that at least one of the writes will execute, which is enough for our scenario. So there is nothing wrong in phase two as well.

But, in phase 1, a similar race condition occurs which is not as benign. The U_a array is used so that directly updating C_a might result in RAW data hazard. But this still fails to resolve the WAW data hazard that occurs in U_a in line 6 and 7 of Algorithm 11. For example, suppose two threads are executing on behalf of the vertex v_1 and v_2 respectively and both of which happen to have a common neighbour *nid*, for which they execute line 6 simultaneously. Moreover, assume that the condition evaluates to $true$ in both cases. Then, they both will execute line 7 for *nid* and update $U_a[nid]$, but only one of the two values will remain in the end.

For our algorithm to be correct, we need the lesser of the two values to remain, but this cannot be ensured during runtime. Such errors might start cascading, thereby resulting in a completely erroneous result. To rectify this, we must ensure that at the end of simultaneous multiple executions of line 7 for a particular node, the smallest value remains. This might be difficult to manage directly, but if we try to execute both **line 6 and 7 atomically** [9], we have an indirect solution at hand. Though this update might result in a slight slow down of the program, we can ensure the validity of our output.

4.3 Thread Approach: Thread Coarsening

Thread Coarsening is an optimisation in which instructions executed by a number of different threads are merged into a single thread [10]. Thus in effect we are executing a smaller number of larger i.e. more coarse-grained threads in comparison to before [11].

Thread Coarsening evidently leads to a reduction in parallelism. This can have both beneficial and detrimental effects , and the observed result closely depends on the *thread coarsening factor*, the number of threads that we are merging into one. On one hand , Thread coarsening would lead to lesser number of thread launches, and thus potentially reducing the number of thread synchronisation barriers and exploiting hardware instruction-level parallelism [12], whereas on the other hand increased computation by a single thread could lead to increased resource consumption of the kernel and also increased cache pressure [11].

Selecting the best coarsening factor thus is a trade-off between exploiting thread-level parallelism and avoiding execution of redundant instructions [13] and forms the crux of any coarsening based optimisation.

Thread Coarsening can be implemented either manually or as a semi-automatic, auto-tuning based or as a machine-learning based optimisation [11]. In our work, we have used the manual approach, whereby we manually merged the computation of *coarsening factor* number of threads into a single thread in the SSSP kernels. The results we obtained are summarised in Section 6.3.2 below.

5 ALL PAIR SHORTEST PATH

In the All-Pair Shortest Path problem (APSP) you are given a weighted graph $G(V, E, W)$ with non-negative weights and we are required to find the least weighted path from every vertex to every other vertex in the graph. One obvious approach to this problem is to run Djikstra's algorithm repeatedly from every source. This algorithm runs in $O(V^2 logV + EV)$. Though this requires complex data structures such as Fibonacci Heaps, it runs significantly faster than other known algortihms for graphs that are not particularly dense. Another one of the most popular algorithms is Floyd Warshall's Algorithm [14]. It takes $O(V^3)$ time and $O(V^2)$ space, irrespective of the number of edges. While the asymptotic complexity is much worse for the generic graph, this runs faster for dense graphs due to the much smaller overhead. We will now explore possible parallel solutions for this problem.

5.1 First Approach: Using SSSP

As mentioned above, a parallelised solution for APSP can be implemented by running the parallelised SSSP from all the vertices, as source, sequentially. Though we reduce the scope for parallelism through such a method, it does have a significant advantage of saving space while running on the GPU. We might need an overall $O(V^2)$ space to store the final results, but we need only $O(V)$ space in the GPU, as each source vertex runs independent of the other. This significantly reduces the cost of processing a large graph, albeit by trading off time. This is demonstrated in Algorithm 13.

5.2 Second Approach: Naive Floyd-Warshall

The problem with our previous approach is that running SSSP for each vertex sequentially, drastically reduces our parallelism. So, by using a significantly higher space in the GPU, at the order of $O(V^2)$, we look for an algorithm that has much higher scope of running faster when parallelised, and this has led us to a parallelised version of Floyd-Warshall's algorithm [5]. Since this requires $O(V^2)$ space, it is not feasible to go beyond a few thousand vertices on commonly available GPUs due to memory limitations, but this possibly gives us a much higher speedup when implemented smartly. We will first start with a naive parallelisation of Floyd-Warshall.

We use an adjacency matrix to represent our weighted graph instead of the compact adjacency list used for the previous algorithms, as this is much more practical for this algorithm. In the sequential version, we will find this algorithm implemented in $|V|$ iterations, and in each iteration, every vertex pair is updated their

Algorithm 13 APSP_Using_SSSP

1: **Input:** V_a, E_a, W_a ▷ The graph $G(V, E, W)$
2: Create updating cost array U_a of size $|V|$
3: Create cost array C_a of size $|V|$
4: Create mask array M_a of size $|V|$ and initialise all values to $false$
5: Create a 2d output array O_a of size $|V|$ x $|V|$
6: **for all** $S \in V$ **do**
7: Assign ∞ to all values of U_a and C_a
8: $U_a[S] = C_a[S] = 0$
9: $M_a[S] = flag = true$ ▷ Start with the source vertex
10: **while** $flag$ **do**
11: $flag = false$
12: **for all** $v \in V$ in parallel **do**
13: Invoke SSSP_Phase1(V_a, E_a, W_a, C_a, U_a, M_a)
14: Invoke SSSP_Phase2(C_a, U_a, M_a, $flag$)
15: Copy the distances in C_a to $O_a[S]$

distance by assuming the shortest path passes through a particular vertex k, which remains constant throughout a iteration and is different for every iteration. Parallelising this, we arrive at Algorithm 14. Though line 5 might look like it suffers from RAW or WAR inconsistencies, the algorithm works properly irrespective which version of the value used, either the updated value or the old value. This is because the value updated could not be any worse than the currently existing one, and using the updated value might result in a faster saturation of its cost but never an erroneous output.

Algorithm 14 APSP_Naive_FW

1: **Input:** E ▷ $G(V, E, W)$ as adjacency matrix of size $|V|$x$|V|$
2: Initalise the non-edges in E as ∞
3: **for all** $k \in V$ **do**
4: **for all** $(u, v) \in V \times V$ in parallel **do**
5: $E[u, v] = min(E[u, v], E[u, k] + E[k, v])$

5.3 Third Approach: Blocked Floyd-Warshall

There are quite a few similarities between matrix multiplication and Floyd-Warshall's algorithm. Every element $E[i, j]$ accesses elements present in the row i or column j only, but here the access happens one at a time over several iterations. It can be easily seen that there will be repetitive access to the array elements. A little bit of math will show us that each element is accessed $O(V)$ times. In algorithm 14, a lot of time is wasted in global memory transfer but the computation done by each thread is fairly simple. Following the common optimisation of matrix multiplication, we may adopt ways to bring chunks of data to the shared memory so that the repetitive access can happen from a much faster shared memory rather than a DRAM.

Katz et. al. [15] discuss an implementation of Floyd Warshall's algorithm that uses shared memory and brings about a significant speedup. Like the previous algorithm, it also uses the adjacency matrix representation of the graph. We will be representing the array elements as $E[i, j]$, where i and j are the row and column indices. FW performs the DP updates $n = |V|$ times on every element. We represent the number of updates or the iterations performed as per the original Floyd-Warshall as k. So k varies from 0 to $n - 1$.

The 2D array is partitioned into blocks of equal size. At every iteration, a block along the main diagonal (starting from the topmost) is processed. These blocks are called primary blocks. This means, for a graph with n vertices, the algorithm performs $\frac{n}{BLOCK_SIZE}$ iterations. This means that in every iteration, k increases by $BLOCK\_SIZE$. Each iteration is further broken down into three phases.

5.3.1 ***First Phase: Independent Phase.*** In the first phase, only the values in the primary block are computed. This phase is called the "independent" phase because the computations depend only on the primary block. Only one block is launched and only one SM is utilised making it very lightweight. Here, the entire primary block is transferred to the shared memory of the SM. At every iteration, the corner points of the primary block are (p_{start}, p_{start}) and (p_{end}, p_{end}), where $p_{start} = primary\_block\_id \times BLOCK\_SIZE$ and $p_{end} = p_{start} + BLOCK\_SIZE - 1$. This means that Floyd Warshall is performed on array elements with i and j in the range $[p_{start}, p_{end}]$. Also k runs from p_{start} to p_{end}.

5.3.2 ***Second Phase: Partially dependent phase.*** In this phase, all the blocks that are dependent on the primary block and itself are computed. These are the blocks that share a row or column with the primary block. The current block and the primary block is loaded in the shared memory. Note that for every partially dependent block, the primary block will be loaded in the memory, i.e. we have not been able to do away with all the repetitions in memory access. This phase is called "partially dependent" as each block depends on one other block i.e. the primary block.

There are two types of blocks, ones which share the rows with the primary block and the ones which share the columns. In this phase, Floyd Warshall's algorithm is performed for the elements with i and j in the range $[p_{start} \rightarrow p_{end}, 0 \rightarrow n - 1]$ for the row aligned blocks and $[0 \rightarrow n - 1, p_{start} \rightarrow p_{end}]$ for the column aligned blocks. Note that these ranges also include the primary block but the primary block is computed in the previous phase, and not in this phase. As already discussed, FW is performed for $BLOCK\_SIZE$ iterations with k ranging from p_{start} to p_{end} in this phase.

5.3.3 ***Third phase: Double dependent phase.*** In the last phase, all the other blocks are computed. These blocks are dependent on the blocks computed in the second phase. This phase is called "double dependent" phase. Here, three blocks need to be loaded in the shared memory, the current block and the two dependent blocks that were computed in the previous phase. Here all the elements with their i and j indices not in $[p_{start}, p_{end}]$ are computed. Like the previous phases, k runs from p_{start} to p_{end}.

So in all the three phases, the basic algorithm remains the same. Each phase performs $BLOCK\_SIZE$ iterations of FW. All that varies is the blocks that are computed. Figure 2 shows the blocks computed in the three stages.

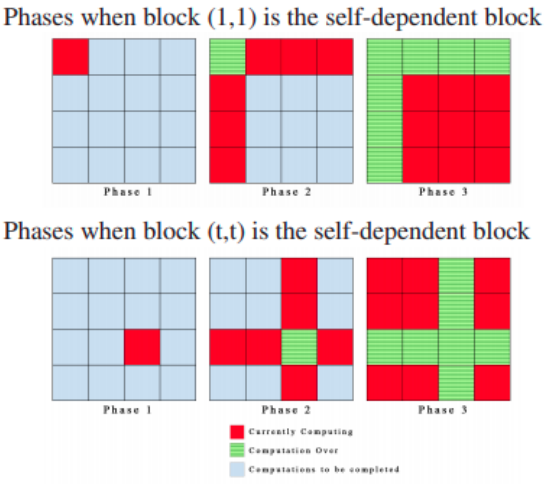

Figure 2: Blocks processed in the three phases, for the first and t^{th} iteration.

6 EXPERIMENTAL RESULTS

Now, to test the significant speedups brought about by the above algorithms over the sequential ones, we run various experiments over diverse graphs and analyse the results obtained.

6.1 Datasets

A proper test suite is required to capture the performance of every algorithm presented above accurately. To that extent, a wide range of graphs of varying sizes and densities from diverse fields such as Genetics, Social Network Circles, Geometric Graphs, Cryptocurrencies, etc., along with a mix of randomly generated graphs were taken and used in the testing of the algorithms. Suitable graphs were first identified, processed to remove redundant data, and were then converted into the required format. The graphs used are further segregated into two classes. For the experiments carried, we used two classes of graphs.

6.1.1 ***APSP Graphs.*** As we know that running APSP algorithms for graphs of size over a few thousand vertices is not feasible, owing to memory and time constraints, we restrict this class of graphs to a size of less than ten thousand vertices to accommodate such algorithms. The following five graphs are used in the testing of our APSP parallel algorithms.

- **Graph 1**:- This is a randomly generated graph of 2, 700 vertices and 1, 808, 853 edges using the Mersenne Twister pseudorandom generator. The graph has an average degree of 1, 340 and is **extremely dense**.
- **Graph 2**:- This is a graph modelling the Facebook Friends Circle taken from Stanford Network Analysis Project (SNAP) [16, 17]. It has 4, 039 vertices and 88, 234 edges, with an average degree of 44, making it a **moderately sparse** graph.
- **Graph 3**:- This graph represents the Biological Gene Network taken from Network Repository [18, 19]. It has 4, 412 vertices and 108, 818 edges, making it a **moderately sparse** graph with an average degree 49.
- **Graph 4**:- Modelled after the Bitcoin OTC Network[20] taken from SNAP, this graph has 5, 881 vertices and 21, 492 edges, with the average degree amounting to 7. This is a **very sparse** network.
- **Graph 5**:- This graph is a product of random generation using the Mersenne Twister pseudorandom number generator. It has 7, 500 vertices and 837, 083 edges and with an average degree of 233, it is a graph with an **average density**.

6.1.2 ***BFS and SSSP Graphs.*** Unlike APSP algorithms, BFS and SSSP do not face any particular constraint. Even then, in this class of graphs, we still do not go above than a few million vertices in size owing to the fact that storage and handling of the data becomes increasingly difficult after a certain point. We have five graphs in this class as well.

- **Graph 1**:- Same as the **Graph 1** in Section 6.1.1.
- **Graph 2**:- This is a graph modelling the Human Genetic Network taken from Network Repository [21]. It has 21, 853 vertices and 12, 323, 648 edges. With an average degree of 1, 128, this is a **reasonably dense** graph.
- **Graph 3**:- This graph represents the Slashdot Social Circle taken from SNAP [22]. It has 82, 168 vertices and 504, 230 edges, making it a **very sparse** graph of average degree 12.
- **Graph 4**:- Modelled after a network of Autonomous Systems in the internet [23] and taken from SNAP, this graph has 1, 694, 616 vertices and 11, 094, 209 edges, with the average degree amounting to 13. This is an **extremely sparse** network.
- **Graph 5**:- This is a Geometric Graph randomly generated between points of a unit square whose distance is greater than a certain value. It has 16, 777, 214 vertices and 132, 557, 200 with an average degree of 16, making it an **extremely sparse** graph as well.

6.2 Experiments

All the experiments were conducted on Colab Notebook, due to lack of proper resources and to provide a uniform ground for testing. The device specifications of the resource: 25GB RAM, Intel® Xeon® CPU @ 2.00GHz Quadcore processor running on Python 3 Google Compute Engine backend (GPU) with one Tesla-P100. The applications were tested on CUDA version 11.2 with Nvidia Driver Version: 460.32. The CPU applications were implemented in C++ using standard template library. The source vertices for the BFS and SSSP algorithms were chosen at random in such a way that they lie in the largest connected component in the graph.

6.3 Results

The results for various algorithms run across the diverse test suite are presented in Fig. 3, Fig. 4, Fig. 5, and Fig. 6 all of whose time of execution has been capped at 4 minutes.

6.3.1 BFS. We implemented three variants of parellel BFS, each having its pros and cons. As we can in the results, the simple quadratic optimization of BFS, *parallelBFS* gives the best results in all possible cases. The two other variants of linear implementation

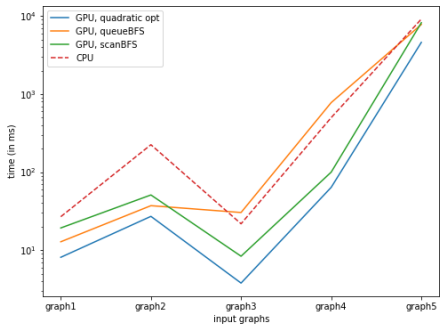

Figure 3: BFS times for various approaches

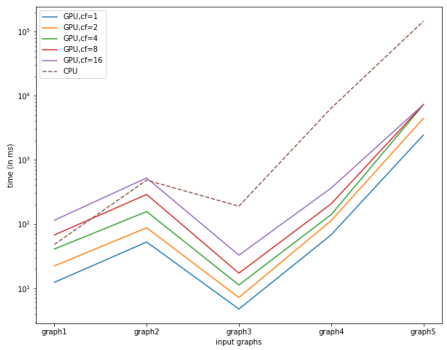

Figure 4: SSSP times for various approaches

of BFS - *scanBFS* and *queueBFS* are suited for different types of graphs. *scanBFS* performs well on sparse graphs and *queueBFS* performs well on dense graphs. It is because of the fact *queueBFS* launches threads for every node in queue, and in dense graphs it helps in parallelism of the vertex frontiers update, whereas in sparse graphs, this parallelism is lost due to the overhead of launching of the kernel. It can be even worse than CPU implementation of BFS in those cases. *scanBFS* involves the additional overhead of *4* global synchronisation steps which becomes a bottleneck for dense graphs and hence its performance stoops in comparison to *queueBFS*. We believe that, as the size of the graph increases, this overhead of global synchronisation ceases to be a bottleneck in the performance of *scanBFS* and it can even outperform *parallelBFS*. We could not benchmark those cases due to memory constraints.

6.3.2 SSSP. This algorithm is heavily dependent on the number of edges present in the graph, and as the number of edges increases, the running time increases as well in both CPU and GPU. This can be attributed to the fact that when a particular node is examined, every edge with that node as a endpoint is assumed to be a part of a shortest path to some node and is thereby analysed to verify the hypothesis. This trend can be observed in the Fig. 4 as well, where *Graph 3* has a lesser running time than *Graph 2* in spite of having higher number of vertices.

Another interesting fact is the speedup offered by the parallel algorithm as compared to the sequential one. Let's define the density of the graph as the ratio between the average degree of the vertices and the number of vertices in the graph. From the Fig. 4, it is obvious that lower the density, higher the speedup offered. For example, with a little bit of math, we will find that the densities of *Graph 1* and *Graph 5* are approximately 0.5 and 7×10^{-6}, respectively, whereas the speedup of the parallel algorithms compared to the sequential one is $3.9x$ for the former and $94.4x$ for the later. This is because, with a higher density, the number of sequential instructions executed by the threads on average significantly increases, resulting in the decreased speedup.

Now we will see why we get a decreased speedup with the increase in coarsening factor for a particular graph (Fig. 5). Coarsening of the thread was carried out with the aim of increasing the per thread activity assuming the thread is not fully utilising the resources in the GPUs. But with each thread iterating through the edges of its corresponding vertex, the per thread activity is already

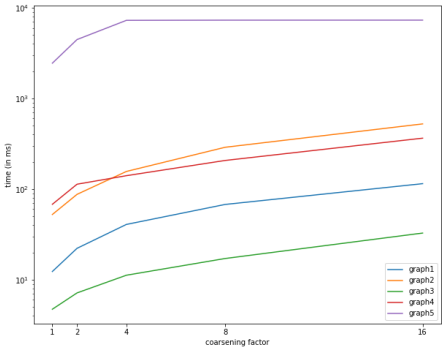

Figure 5: SSSP times for different coarsening factors

significant. Increasing it further will result in a drastic loss of parallelism, which is even more prominent in dense graphs like *Graph 1* and *Graph 2* with a CF of 16, where the parallel version is significantly slower even when compared to the sequential algorithm (Fig. 4).

6.3.3 APSP. Ignoring the Blocked Floyd-Warshall (BFW), the Floyd-Warshall algorithm performs significantly worse than the APSP using repeated SSSP for graphs, unless the graph is dense (Fig. 6, *Graph 1* and *Graph 2*). This is true even for their corresponding sequential versions. This is because, asymptotically, using repeated SSSP for APSP has a much lesser complexity than Floyd-Warshall whenever the graph is sparse, as the algorithm depends cubically on the number of vertices and ignores the number of edges.

But, in the BFW parallel algorithm, we can see a drastic increase in the performance as compared to all other versions (Fig. 6). It performs at least $500x$ times better than the best sequential run time and reaches as much as $4000x$ in certain cases. Even when compared to the best parallel algorithms run, it offers at least $20x$ speedups. This is because, BFW makes optimal use of memory accesses unlike the naive FW algorithm.

7 CONCLUSION

In this paper we have presented fast parallel implementations for few fundamental graph algorithms: breadth-first traversal, single source shortest path, all pair shortest path, using GPUs and which can provide incredible speedup compared to sequential programs.

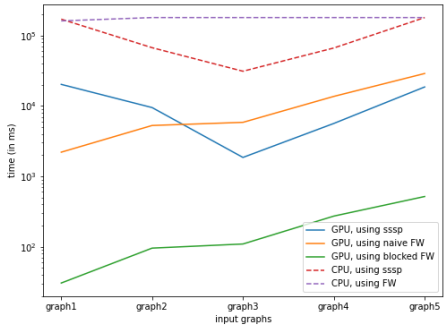

Figure 6: APSP times for various algorithms

All the algorithms when parallelized on the GPU brought about some speedup. Performing more optimizations lead to even more speedups. For example, while implementing the naive Floyd Warshall, the kernel showed only about a speedup of 6-7 times. Whereas when using various memory optimizations, we got a huge speedup in blocked Floyd Warshall, 500-4000 times, asymptotically better than the sequential algorithm.

Some of the algorithms performed really well for small graphs but were bottle necked for dense graphs. Moreover the resource constraints proved to be a driving factor of the speedup or lack thereof, for example, in case of coarsening in SSSP, we observed a decrease in speedup with increasing coarsening factor. Using CUDA, we can easily perform various optimizations which would help us fully utilise the power of modern GPUs.

REFERENCES

[1] Jun-Dong Cho, Salil Raje, and Majid Sarrafzadeh, "Fast approximation algorithms on maxcut, k-coloring, and k-color ordering for vlsi applications," *IEEE Trans. Comput.*, vol. 47, no. 11, Nov. 1998.

[2] Thomas Lengauer and Robert Endre Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, Jan. 1979.

[3] P. Narayanan, "Single source shortest path problem on processor arrays," *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 553–556, 1992.

[4] John Owens, Shubhabrata Sengupta, and Daniel Horn, "Assessment of graphic processing units (gpus) for department of defense (dod) digital signal processing (dsp) applications," 10 2005.

[5] Paulius Micikevicius, "General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem.," 01 2004, vol. 3, pp. 1359–1365.

[6] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[7] Michael L. Fredman and Robert Endre Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, July 1987.

[8] Pawan Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *High Performance Computing – HiPC 2007*, Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, Eds., Berlin, Heidelberg, 2007, pp. 197–208, Springer Berlin Heidelberg.

[9] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes, "Cuda solutions for the sssp problem," in *Proceedings of the 9th International Conference on Computational Science: Part I*, Berlin, Heidelberg, 2009, ICCS '09, p. 904–913, Springer-Verlag.

[10] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2013, SC '13, Association for Computing Machinery.

[11] Nicolai Stawinoga and Tony Field, "Predictable thread coarsening," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 2, June 2018.

[12] Alberto Magni, Christophe Dubach, and Michael O'Boyle, "Exploiting gpu hardware saturation for fast compiler optimization," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, New York, NY, USA, 2014, GPGPU-7, p. 99–106, Association for Computing Machinery.

[13] Alberto Magni, Christophe Dubach, and Michael O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, New York, NY, USA, 2014, PACT '14, p. 455–466, Association for Computing Machinery.

[14] P. Z. Ingerman, "Algorithm 141: Path matrix," *Commun. ACM*, vol. 5, no. 11, pp. 556, Nov. 1962.

[15] & Kider J.T Katz, G.J., "All-pairs shortest-paths for large graphs on the gpu," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH '08), 47-55.* 2008, Penn Libraries.

[16] Jure Leskovec and Andrej Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[17] Julian J. McAuley and Jure Leskovec, "Discovering social circles in ego networks," *CoRR*, vol. abs/1210.8182, 2012.

[18] Ryan A. Rossi and Nesreen K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[19] Ara Cho, Junha Shin, Sohyun Hwang, Chanyoung Kim, Hongseok Shim, Hyojin Kim, Hanhae Kim, and Insuk Lee, "Wormnet v3: a network-assisted hypothesis-generating server for caenorhabditis elegans," *Nucleic acids research*, vol. 42, no. W1, pp. W76–W82, 2014.

[20] Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and VS Subrahmanian, "Rev2: Fraudulent user prediction in rating platforms," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 2018, pp. 333–341.

[21] Mukesh Bansal, Vincenzo Belcastro, Alberto Ambesi-Impiombato, and Diego Di Bernardo, "How to infer gene networks from expression profiles," *Molecular systems biology*, vol. 3, no. 1, 2007.

[22] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *CoRR*, vol. abs/0810.1355, 2008.

[23] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, New York, NY, USA, 2005, KDD '05, p. 177–187, Association for Computing Machinery.