CS 406 Project Final Report

Efe Şencan, Gökberk Yar, Ahmet Enes Deveci
Alper Giray, Zeynep Özge Ergin

28 April 2021

Abstract

Depth first search (DFS) is a common algorithm that is used to traverse graph data structures. The algorithm starts by selecting an arbitrary root node as a starting point and it keeps traversing nodes starting from the root node as deep as possible until some stopping conditions are met. The goal of this project is to count the number of cycles (k = 3,4,5) in a cyclic undirected graph using the DFS algorithm as efficiently as possible by utilizing parallelization. For this purpose, we implemented several algorithms using OpenMP and CUDA and experimented with various parallelization methods on high performance computing (HPC) clusters. Based on our observations, we improved the sequential version of the implementation significantly both on CPU and GPU. In Amazon dataset, we improved sequential CPU runtime for k=3 case by a factor of 19 and in Dblp dataset we got more than 6 times speed up in k=4.

Keywords— Openmp, Cuda, Multicore, Cycle Count

Contents

1 Introduction

Un-directed graph implies $u, v \in V$ and $(u, v) \in E \implies (v, u) \in E$.A k-cycle in G defined as $p = (v_0, v_1, v_2 ... v_{k-1}, v_k)$ where $v_0 = v_k$ and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, ..., k$ and $v_i \neq v_j$ if $i \neq j$ for $i = 0, 1, ..., k$. Given an un-directed **graph** G and a **number** k where $2 < k < 6$. $G = (V, E)$, where V is **vertices set**, and E is the **edges set**, our goal is to find the number of k-cycles for $\forall v \in$V.

Depth first search (DFS) is a common algorithm that is used to traverse graph data structures. The DFS algorithm contains many sub-tasks, in particular graph traversals which can be initialized with multiple processors. Since, one traversal starting from a particular root node j, does not depend on another traversal starting from j, DFS can be implemented in parallel. Moreover, as the size of the graph G grows, the runtime of the single thread approach can cost a significant amount of computation time (exponential time). For these reasons, HPC is applicable to DFS to reduce the runtime.

2 Data Description and Preprocess

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix. In this project, we have 2 sparse matrices that we can conduct tests on, named Dblp and Amazon. These matrices are taken as inputs where input format is like:

$$u_1 \; v_1$$

$$u_2 \; v_2$$

$$\cdots$$

$$u_m \; v_m$$

$$where \; u_i < v_i \; and \; (u_i, v_i) \in E, \forall i.$$

To be able to do the DFS, we need to have an adjacency matrix(list). So, we take the inputs and transform it into a sparse adjacency matrix. However, in both cases, we have two sparse matrices with Dblp having 425956 columns and rows, and Amazon having 548551 rows and columns. Since storing these kinds of matrices consumes too much memory and is nearly impossible, we used a method called Compressed Sparse Row (CSR) representations. The CSR representation uses two arrays, adj and xadj for storing the column indices of the nonzeros within each row of the matrix adjacently. In CSR, the length of the array adj is equal to the number of nonzeros, and it keeps the column indices for the rows. The array xadj keeps the starting index for each row in adj. Hence, for a row i, the sub-array of column indices of nonzeros at row i starts with the entry adj[xadj[i]] and ends with the entry adj[xadj[i+1]] 1. To simplify the implementations and make the previous statement correct for the last row, the length of xadj is set to n + 1, and the last element, xadj[n], is set to the number of nonzeros. An example CSR representation for a toy sparse matrix is given in Figure 1.

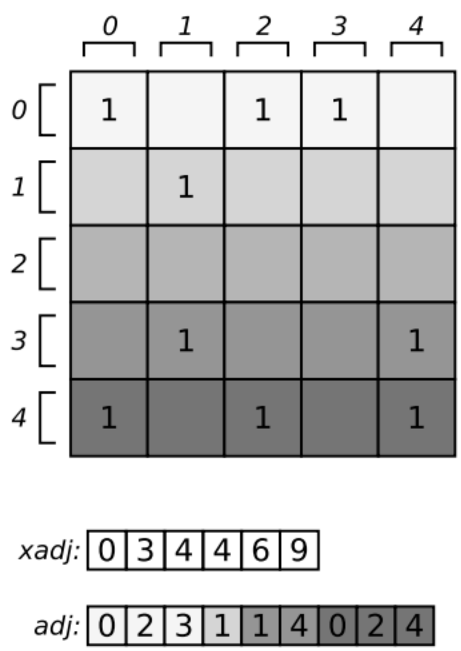

Figure 1: CSR Matrix Visualization

3 General DFS

There are approaches that utilize Breadth First Search (BFS) algorithm for detection of cycles in an undirected graph. However, this approach does not count number of cycles for each vertex. For this purpose(counting cycles of length k from a starting vertex i), Depth First Search(DFS) algorithm is more suitable.

The DFS algorithm starts by selecting an arbitrary root node as a starting point and it keeps traversing nodes starting from the root node as deep as possible until some stopping conditions are met. The essential part is, graphs may contain cycles and a particular node in a graph may be visited more than once. To avoid processing a node more than once, a boolean array is used to keep track of the visited nodes. Different from the regular DFS approach, since we are trying to find the cycles of length k, k-depth search is enough to determine whether the subgraph starting from node i is a cycle of length k. Therefore, we implemented DFS algorithm with k level nested for loops, since k is small.

Algorithm 1 Recursive_DFS(G,u):

Require: G, undirected graph
Require: u, starting vertex
if $visited[u]$ = true **then**
  return
end if
$visited[u] \leftarrow true$
for $v \in G[u].neighbors$ **do**
  Recursive_DFS(G,v)
end for

4 Methods and Results

4.1 Conversion of inputs to CSR format

As it mentioned in the "Data Description & Preprocess" section, the graph that we provided as input is mostly sparse. In other words, most of the vertices do not have any neighbours. Therefore, it is more efficient to store matrices with CSR format to be memory efficient. To begin with our input data preprocessing method, we first read the input file line by line. For each vertex, we store it's neighbours as key-value pairs by using unordered_map in C++. The major reason for using unordered map is to perform find and insert operations in constant time. After filling the unordered map, the keys would be the vertices in our graph and the values are the corresponding neighbours of these vertices.

After reading the inputs, adj and xadj vectors must be filled correctly. For this purpose, there is a for loop that iterates number of vertices many times. First, the next element of the unordered map is taken. Then, we checked whether this element has connection to any vertex or not. If the element is not connected, its index of the xadj is filled with the previous element's value. If it is connected, we take the number of neighbors it has and fill the xadj with it. After filling the xadj, we started to visit every neighbor of the element and fill the adj array accordingly.

4.2 Multi-core CPU implementation with OpenMP

As it mentioned earlier in the "General DFS Algorithm", our implementation was based on the DFS. For each vertex in the graph, we applied the DFS algorithm but only with depth k where k is the length of the cycle. When the algorithm reaches kth depth, it checks whether the starting vertex is the neighbor of it. If that is the case, the count for this vertex is incremented by one.

To parallelize this algorithm, we distributed the vertices to threads. We were calling the DFS function in a loop so we parallelized this loop. In each iteration, a thread arrives and takes an available vertex. After traversing and detecting the k-cycles, the thread takes another vertex until no available vertex is left. We used different scheduling options and guided options worked best as expected because the number of neighbours of each vertex is different and it leads to unbalanced workload. This unbalanced workload is more problematic in Dblp case, most of the CPU threads becomes idle and wait for the responsible thread which executes vertex with the most neighbours. In Table 1, efficiencies for Amazon is almost 1, our algorithm scales with number of processors. In Dblp, efficiency is less than Amazon since it is denser but speed up still keeps increasing with number of threads used.

OPENMP				
K=3	Thread Count	Duration	Speedup	Efficiency
amazon	1	0.603501	1	1
	2	0.310298	1.944907798	0.9724538992
	4	0.156428	3.858011353	0.9645028384
	8	0.0786051	7.677631604	0.9597039505
	16	0.0402396	14.99768884	0.9373555527
dblp	1	1.8533	1	1
	2	1.12785	1.643214967	0.8216074833
	4	0.659231	2.8113059	0.7028264751
	8	0.353933	5.236301786	0.6545377232
	16	0.178739	10.36874997	0.6480468728
K=4	Thread Count	Duration	Speedup	Efficiency
amazon	1	4.55978	1	1
	2	2.29525	1.986615837	0.9933079185
	4	1.19381	3.819519019	0.9548797547
	8	0.578946	7.876002252	0.9845002815
	16	0.287246	15.8741288	0.9921330497
dblp	1	66.5606	1	1
	2	34.1796	1.947377968	0.9736889841
	4	20.5717	3.235542031	0.8088855078
	8	12.5448	5.305831898	0.6632289873
	16	9.59927	6.9339231	0.4333701938
K=5	Thread Count	Duration	Speedup	Efficiency
amazon	1	43.1156	1	1
	2	21.9676	1.962690508	0.9813452539
	4	10.9534	3.936275494	0.9840688736
	8	5.42065	7.953953862	0.9942442327
	16	2.86459	15.05122897	0.9407018107
dblp	1	-	-	-
	2	-	-	-
	4	-	-	-
	8	-	-	-
	32	977	-	-

Table 1: Multicore OpenMP Implementation

4.3 GPU Parallelization

4.3.1 Single GPU Recursive Implementation

After implementing the sequential and OpenMP algorithms, to run these algorithms in the GPU, we tried to give each vertex to a thread. To do that, we used a single dimensional grid and block. These threads are sent to a single kernel that gives each thread a startVertex. When a thread takes this startVertex, it starts to dive k-depth and checks if the kth vertex is a neighbor of the startVertex. For each cycle, the count is incremented and at the end, results are written a global array according to the global id of the thread.

One the other hand, there is a drawback of using recursion operations in CUDA. First of all, the stack size that is allocated for each recursive call is limited, and the programmer should be careful about not exceeding the memory limit. In addition, we realized that recursive calls become costly in CUDA which brings us inefficiency in our implementation. Hence, we abandoned the recursive implementation and decided to implement a non-recursive algorithm.

4.3.2 Single GPU Non-Recursive Implementation

The most common method for converting a recursive algorithm to a non recursive version is to use stack data structure and simulate the recursion by a first come last out manner. However, since the cycle lengths in our problem definition is limited within the range 3 to 5, instead of utilizing stack, we wrote nested for loops of k many times. We implemented three different kernels for each cycle length scenario. After determining the root node, we computed its neighbour by looking at the xadj array. Then, for every neighbour of the starting point, this time we computed the neighbours of that neighbour. This process continues until we reach the k^{th} neighbour of the root node. Finally, we checked whether the k^{th} neighbour is equal to the starting node. If they are equal, then this means that we find a cycle of length k. In Table 2, one can see that single GPU with non-recursive implementation produces 13 threads equivalent work for Amazon and nearly 4 threads worth work for Dblp case. When k=4, performance nearly dropped by 50 percent. When k = 5, performance dropped by 25 more percent on k=4 case for Amazon. Dblp requires too much time that's why it is not reported. As an observation GPU performs best when sparsity of matrix is large and performance drops dramatically as density of graph increases. Increasing k or using dblp are ways to increase density of graph. The reason why performance drops is most of the GPU threads become idle and wait for the bottleneck in the calculation (vertex with most neighbours).

Non Recursive		
K=3	**Duration**	**Speedup**
amazon	0.045381	13.29853904
dblp	0.487654	3.800440476
K=4	**Duration**	**Speedup**
amazon	0.514845	8.856607328
dblp	44.543335	1.494288652
K=5	**Duration**	**Speedup**
amazon	6.447344	6.687342881
dblp	-	-

Table 2: GPU Non-Recursive

4.3.3 Multi-GPU Non-Recursive Implementation

With Multi-GPU Non-Recursive implementation, we designed the algorithm such that it could run on up to 4 GPUs concurrently. In fact, it can work as much as GPUs that the computer allows and our testing environment has 4 GPUs available. The algorithm is essentially very similar to the single GPU Non-Recursive implementation. The appropriate kernel function is found by the main processor and every thread starts working on a vertex. Main difference of this approach is that since there are multiple GPUs, it first divides the workload among the GPUs. In other words, the total number of vertices was divided into the GPUs equally. GPU 1 takes the first forth of the data, GPU 2 takes the second forth data and continues. GPUs in Nebula are heterogenous, there are 2 faster and 2 slower GPUs. This method's performance is directly affected by this fact. For run time, we took the slowest GPU among 4 GPUs. Our observation is generally other GPUs complete their work significantly early and become idle for a duration. This idle time shows there is still room for improvement with custom workloading the GPUs. For example giving more work to better GPUs or applying a preprocessing step for computation order of the vertices can improve performance. In Table 3, we achieve 19 threads worth speed up for the Amazon dataset. This is about 50 percent faster than single GPU implementation. This speed up pattern continues for k=4 and k=5.

Multigpu - Non Recursive		
K=3	**Duration**	**Speedup**
amazon	0.030755	19.62285807
dblp	0.49196	3.767176193

K=4	**Duration**	**Speedup**
amazon	0.372918	12.2272993
dblp	57.406826	1.159454452

K=5	**Duration**	**Speedup**
amazon	5.285642	8.157116959
dblp	-	-

Table 3: Multi-GPU Non-Recursive

4.4 Combining GPU and CPU

4.4.1 Multi-GPU + CPU Non-Recursive Implementation

With the inspiration we got from Multi-GPU Non-Recursive Implementation, we thought that we could run the code with multiple GPUs and CPUs together. In this implementation, we used 4 GPU and 28 CPU together. Similar to the multi-GPU case, it can work as much as GPUs and CPUs that the computer allows and our testing environment has 4 GPUs and 28 CPUs available. For this purpose, we created 32 threads initially and assigned the first 4 of them to the GPU and rest to the CPU. First, we divided the total work to the threads chunk by chunk and distributed these chunks to the processors equally. Based on our experiments, threads in the GPU finished their job, whereas threads in the CPU still have work to do. Since, the GPU has more available threads than the CPU, overall, GPU can handle more vertices for a particular time period. Hence, it has a better throughput compared to CPU. Therefore, we distributed more chunks to the GPU threads. After conducting some runs, we figured out that GPU threads can handle 10 times more vertices than CPU threads. Then, GPU threads found the appropriate kernel according to cycle length and started to work on this kernel like in the previous non-recursive implementations. On the other hand, CPU threads started to perform DFS like Multi-core CPU implementation with OpenMP. In Table 4, except the most sparse case Amazon dataset k=3, adding CPU threads improved the performance. For instance in Dblp k=3, speedup doubled compared to multigpu case. Also for Amazon k=4 and k=5 speedups are better when CPU is added on top.

Multigpu + CPU Non Recursive		
K=3	**Duration**	**Speedup**
amazon	0.037041	16.29278367
dblp	0.239933	7.724239684
K=4	**Duration**	**Speedup**
amazon	0.274205	16.62909137
dblp	22.658237	2.93758954
K=5	**Duration**	**Speedup**
amazon	4.201431	10.2621226
dblp	2200	-

Table 4: Multi-GPU + CPU Non-Recursive

4.4.2 Multi-GPU + CPU Non-Recursive Dynamic Workqueue Implementation

After running some tests on the Multi-GPU + CPU Non-Recursive Implementation, despite the GPU threads having 10 times more workload, there were still CPU and GPU threads waiting whereas a single thread works so much longer. To optimize the usage of the threads, we did not divide all the work into the threads initially. Instead, we had created chunks and calculated the total number of chunks that needed to be completed. After that, until we reached the total number of chunks, the available threads both in GPU and CPU came forward and took the next available chunk. We were aware that picking up the next available chunk caused a race condition. That's why we took those parts into the critical region. But this critical region created an overhead for threads and that caused the algorithm to run slower. A solution to this problem would be the incrementing the chunk size, since for larger chunks threads will spend more time on that, hence the threads would need less to get the next chunk. However, for larger chunk sizes, it would be similar to Multi-GPU + CPU Non-Recursive Implementation, a non-dynamic method, so there would be idle threads and that would also increase the run time. In Table 5, only Dplb k=4 case performed better and it is actually the best case among all GPU versions. However, speedup dropped under 1 for Amazon cases. For this method, parameters like chunck size and GPU multiplier are very important, one should find the best pair that is consistent for in all datasets and cycle lengths.

Multigpu + CPU Non Recursive Dynamic Workload		
K=3	**Duration**	**Speedup**
amazon	2.076866	0.2905825412
dblp	3.700613	0.50080892
K=4	**Duration**	**Speedup**
amazon	5.658017	0.8058971898
dblp	10.617888	6.268723121
K=5	**Duration**	**Speedup**
amazon	8.319901	5.182225125
dblp	-	-

Table 5: Multi-GPU + CPU Non-Recursive Dynamic Workload

5 How to reproduce results

A makefile is shared with the project. One can reproduce the results by running different commands in the makefile. For all testcase, there is run that only measures the run time and another run that produces the desired output.

6 Future Work

After observing GPU performs best when sparsity is large and idling increase when workload is imbalanced, one should preprocess the input and sort vertices accordingly their required work amount. One heuristic that can be used to estimate required work for a vertex is its neighbour count (one vertex could have less neighbours but its neighbours could potentially have many neighbours, so it is only a heuristic). Also, there is no need to re-compute same cycles for each vertex in the path, by using a better algorithm, shared memory and atomic memory accesses redundant calculation could be minimized. For GPU implementation, one could implement different kernels for different neighbour count vertices. For example, neighbour count space could be splitted into 3 : less than 32, 32 to 256 and more than 256. For less than 32 case, one can pack different vertices into single GPU warp and run together. For 32 to 256, one can use block level implementation and vertices more than 256 one can use grid level implementations to reduce idling.

7 Appendix

7.1 File Reading (Spare Matrix Creation)

```
void  read_mtxbin(string fname, int k){
  ifstream infile(fname);
  int a, b;
  int nnv = 0;
  unordered_map<int, vector<int> > hashmap;

  int  maxElement  = -1;

  while (infile >> a >> b)
  {
      nnv+=2;
      hashmap[a].push_back(b);
      hashmap[b].push_back(a);

      if(b > maxElement){
        maxElement = b;
      }
  }

  int nov = maxElement +1;

  int * adj  = new int[nnv];
  int * xadj = new int[nov+1];
  xadj[0]=0;

  int j = 0;
  int maxSize = -1;

  for(int i=0; i < nov ; i++ ){
    auto current = hashmap.find(i);
    if (current == hashmap.end()){
        xadj[i+1] = xadj[i];
    }
    else{
        int size = current->second.size();
        maxSize = max(size,maxSize);

        xadj[i+1] = xadj[i] + size;
        for(auto val : current->second) {
            adj[j] = val;
            j++;
        }
    }
  }

  wrapper(xadj,adj,k,nov,nnv);
}
```

7.2 OpenMP Implementation

```
void DFS_sparse(int xadj[], int adj[], bool marked[], int n,
         int vert, int start, int &count)
             //vert: bulundugu konum //start: baslangıc noktası
{
    marked[vert] = true;
    int start_index = xadj[vert];
    int path_length = xadj[vert+1];
    if (n == 0){
        marked[vert] = false;
        for(int i = start_index; i < path_length; i++){
            if(adj[i] == start){
                count++;
                break;
```

```
            }
        }
        return;
    }
    for(int i=start_index; i < path_length; i++){
        if(!marked[adj[i]]){
            DFS_sparse(xadj, adj, marked, n-1, adj[i], start, count);
        }
    }
    marked[vert] = false;
}

void countCycles_sparse(int *xadj, int *adj, int n,  int nov)
{
    double start, end;
    start = omp_get_wtime();
    int *arr = new int[nov];

    #pragma omp parallel num_threads(16)
    {
        bool *marked = new bool[nov];
        memset(marked, false, nov * sizeof(bool)); // bu belki silinebilir

        #pragma omp for schedule(guided)
        for(int i = 0; i < nov; i++){
            int localcount = 0;
            DFS_sparse(xadj, adj, marked, n - 1, i, i, localcount);
            arr[i] = localcount;
        }
    }
    end = omp_get_wtime();
}
```

7.3 Recursive Cuda (Single GPU) Kernel

```
__device__ bool check(int marked[], int round, int val){
  for(int i = 0; i < round; i++){
    if(marked[i] == val){return false;}
  }
  return true;
}

__device__ void DFS_sparse(int xadj[], int adj[], int marked[], int n,
         int vert, int start, int &count, int round)
             //vert: bulundugu konum //start: baslangıc noktası
{
    marked[round] = vert;

    int start_index = xadj[vert];
    int path_length = xadj[vert+1];

    if (n == 0){
      marked[round] = -1;
        for(int i = start_index; i < path_length; i++){
            if(adj[i] == start){
                count++;
                break;
            }
        }
        return;
    }
    for(int i=start_index; i < path_length; i++){
        if(check(marked, round,adj[i])){
            DFS_sparse(xadj, adj, marked, n-1, adj[i], start, count, round +
                1);
        }
    }
    marked[round] = -1;
}

__global__ void kernel(int* adj, int* xadj, int* output, int n, int nov){
```

```
  int index = threadIdx.x + (blockIdx.x * blockDim.x);
  __shared__ int marked[THREADS_PER_BLOCK][10];
  if(index < nov){
      //int *marked = new int[n];
      //memset(marked, -1, n * sizeof(int)); // bu belki silinebilir
      int localcount = 0;
      int round = 0;
      DFS_sparse(xadj, adj, marked[threadIdx.x], n - 1, index, index,
          localcount, round);
      output[index] = localcount;
    }
}

void wrapper(int *xadj, int *adj, int n,  int nov, int nnz){
  cudaSetDevice(0);
  int *adj_d;
  int *xadj_d;
  int *output_d;
  int *output_h = new int[nov];
  int numBlock = (nov + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
  cudaEvent_t start, stop;
  float elapsedTime;

  gpuErrchk(cudaMalloc((void**)&adj_d, (nnz) * sizeof(int)));
  gpuErrchk(cudaMalloc((void**)&xadj_d, (nov + 1) * sizeof(int)));

  gpuErrchk(cudaMalloc((void**)&output_d, (nov) * sizeof(int)));

  //gpuErrchk(cudaMallocHost((void **)&output_h, (nov) * sizeof(int)));

  gpuErrchk(cudaMemcpy(adj_d, adj, (nnz) * sizeof(int),
      cudaMemcpyHostToDevice));
  gpuErrchk(cudaMemcpy(xadj_d, xadj, (nov + 1) * sizeof(int),
      cudaMemcpyHostToDevice));

  cudaEventCreate(&start);
  cudaEventRecord(start, 0);

  kernel<<<numBlock, THREADS_PER_BLOCK>>>(adj_d, xadj_d, output_d, n, nov);

  gpuErrchk(cudaDeviceSynchronize());

  gpuErrchk(cudaMemcpy(output_h, output_d, (nov) * sizeof(int),
      cudaMemcpyDeviceToHost));
  //printArray(output_h,nov);
  cudaEventCreate(&stop);
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);

  cudaEventElapsedTime(&elapsedTime, start, stop);
  //printf("GPU scale took: %f s\n", elapsedTime/1000);

  cudaFree(adj_d);
  cudaFree(xadj_d);
}
```

7.4 Non-recursive Cuda

```
__global__ void kernel3(int* adj, int* xadj, int* output, int nov){
  int index = threadIdx.x + (blockIdx.x * blockDim.x);
  if(index < nov){
      //int *marked = new int[n];
      //memset(marked, -1, n * sizeof(int)); // bu belki silinebilir
      int localcount = 0;
      // int round = 0;

      // 0-->
      int s0 = xadj[index];
      int e0 = xadj[index+1];

      for(int i=s0; i < e0; i++){
```

```
        // 0 --> 1

        int neighbour_1 = adj[i];
        int s1  = xadj[neighbour_1];
        int e1  = xadj[neighbour_1+1];

        for(int j=s1;j < e1; j++){
          // 0 --> 1 --> 2

          int neighbour_2 =  adj[j];
          if (neighbour_2 == index) continue;
          int s2  = xadj[neighbour_2];
          int e2  = xadj[neighbour_2+1];

          for(int k=s2; k < e2; k++){

            // 0 --> 1 --> 2 --> 3
            int neighbour_3 =  adj[k];
            if (neighbour_3 == index){
              localcount+=1;
              break;
            }
          }
        }
      }
      output[index] = localcount;
    }
}



void wrapper(int *xadj, int *adj, int n,  int nov, int nnz){

  // int X = nov;
  // int Y = maxSize;
  // int Z = maxSize;
  //
  // dim3 threadsPerBlock(8, 8, 8);
  // dim3 numBlocks(X/threadsPerBlock.x,  /* for instance 512/8 = 64*/
  //                Y/threadsPerBlock.y,
  //                Z/threadsPerBlock.z);
  //





  cudaSetDevice(0);
  int *adj_d;
  int *xadj_d;
  int *output_d;
  int *output_h = new int[nov];
  int numBlock = (nov + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
  cudaEvent_t start, stop;
  float elapsedTime;

  gpuErrchk(cudaMalloc((void**)&adj_d, (nnz) * sizeof(int)));
  gpuErrchk(cudaMalloc((void**)&xadj_d, (nov + 1) * sizeof(int)));

  gpuErrchk(cudaMalloc((void**)&output_d, (nov) * sizeof(int)));

  //gpuErrchk(cudaMallocHost((void **)&output_h, (nov) * sizeof(int)));

  gpuErrchk(cudaMemcpy(adj_d, adj, (nnz) * sizeof(int),
      cudaMemcpyHostToDevice));
  gpuErrchk(cudaMemcpy(xadj_d, xadj, (nov + 1) * sizeof(int),
      cudaMemcpyHostToDevice));

  cudaEventCreate(&start);
  cudaEventRecord(start, 0);

  kernel5<<<numBlock, THREADS_PER_BLOCK>>>(adj_d, xadj_d, output_d, nov);

      //combination<<<numBlocks, threadsPerBlock>>>(adj_d, xadj_d, output_d, n, nov);

  gpuErrchk(cudaDeviceSynchronize());

  gpuErrchk(cudaMemcpy(output_h, output_d, (nov) * sizeof(int),
      cudaMemcpyDeviceToHost));

  cudaEventCreate(&stop);
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);

  cudaEventElapsedTime(&elapsedTime, start, stop);
  // printf("GPU scale took: %f s\n", elapsedTime/1000);
```

```
  printArray(output_h,nov);
  cudaFree(adj_d);
  cudaFree(xadj_d);
}
```

7.5 Multi GPU (Non-recursive) Kernel

```
void DFS_sparse(int xadj[], int adj[], bool marked[], int n,
         int vert, int start, int &count)
             //vert: bulundugu konum //start: baslangıc noktası
{
    marked[vert] = true;
    int start_index = xadj[vert];
    int path_length = xadj[vert+1];

    if (n == 0){
        marked[vert] = false;
        for(int i = start_index; i < path_length; i++){
            if(adj[i] == start){
                count++;
                break;
            }
        }
        return;
    }

// if(path_length-start_index <=1) return;

    for(int i=start_index; i < path_length; i++){
        if(!marked[adj[i]]){
            DFS_sparse(xadj, adj, marked, n-1, adj[i], start, count);
        }
    }
    marked[vert] = false;
}

__global__ void kernel3(int* adj, int* xadj, int* output, int nov, int
    novStart){

  int index = novStart + threadIdx.x + (blockIdx.x * blockDim.x);
  if(index < nov){
      // if(index ==0)printf("called gpu \n");
      //int *marked = new int[n];
      //memset(marked, -1, n * sizeof(int)); // bu belki silinebilir
      int localcount = 0;
      // int round = 0;

      // 0-->
      int s0 = xadj[index];
      int e0 = xadj[index+1];

      for(int i=s0; i < e0; i++){
        // 0 --> 1

        int neighbour_1 = adj[i];
        int s1  = xadj[neighbour_1];
        int e1  = xadj[neighbour_1+1];

        for(int j=s1;j < e1; j++){
          // 0 --> 1 --> 2

          int neighbour_2 =  adj[j];
          if (neighbour_2 == index) continue;
          int s2  = xadj[neighbour_2];
          int e2  = xadj[neighbour_2+1];

          for(int k=s2; k < e2; k++){

            // 0 --> 1 --> 2 --> 3
            int neighbour_3 =  adj[k];
            if (neighbour_3 == index){
              localcount+=1;
```

```
                break;
              }
            }
          }
        }
        output[index-novStart] = localcount;
      }
}

void wrapper(int *xadj, int *adj, int n,  int nov, int nnz){

  int *output_h = new int[nov];

  double start_cpu, end_cpu;
  start_cpu = omp_get_wtime();


  #pragma omp parallel num_threads(PARALEL_THREAD_COUNT)
  {

    int threadId=omp_get_thread_num ();
    // cout<< threadId<<endl;

    int virtual_thread_count  = GPU_MULTIPLIER *4 + PARALEL_CPU;
    int novForThread = (nov+virtual_thread_count-1)/virtual_thread_count;



    if(threadId <=3)
    {
      int novStart = GPU_MULTIPLIER * novForThread * threadId;
      int novEnd   = GPU_MULTIPLIER * novForThread * (threadId+1);
      if (novEnd > nov) novEnd = nov;
      int numBlock = (novEnd-novStart + THREADS_PER_BLOCK-1) /
          THREADS_PER_BLOCK;
      // printf("nov s %d e %d \n", novStart,novEnd);


      cudaSetDevice(threadId);

      int *adj_d;
      int *xadj_d;
      int *output_d;
      cudaEvent_t start, stop;
      float elapsedTime;

      gpuErrchk(cudaMalloc((void**)&adj_d, (nnz) * sizeof(int)));
      gpuErrchk(cudaMalloc((void**)&xadj_d, (nov + 1) * sizeof(int)));

      gpuErrchk(cudaMalloc((void**)&output_d, (novEnd-novStart) *
          sizeof(int)));

      //gpuErrchk(cudaMallocHost((void **)&output_h, (nov) * sizeof(int)));

      gpuErrchk(cudaMemcpy(adj_d, adj, (nnz) * sizeof(int),
          cudaMemcpyHostToDevice));
      gpuErrchk(cudaMemcpy(xadj_d, xadj, (nov + 1) * sizeof(int),
          cudaMemcpyHostToDevice));

      cudaEventCreate(&start);
      cudaEventRecord(start, 0);
      double start_gpu = omp_get_wtime();
      cudaStream_t stream1;
      cudaStreamCreate ( &stream1) ;

        // printf("threadId entry to kernel %d GPU \n", threadId );
      if      (n==3)kernel3<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d,
          xadj_d, output_d, novEnd,novStart);
      else if (n==4)kernel4<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d,
          xadj_d, output_d, novEnd,novStart);
       else if (n==5)kernel5<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d,
           xadj_d, output_d, novEnd,novStart);

          //combination<<<numBlocks, threadsPerBlock>>>(adj_d, xadj_d, output_d, n, nov);
      // printf("threadId exit to kernel %d GPU \n", threadId );
      double end_gpu = omp_get_wtime();


      gpuErrchk(cudaDeviceSynchronize());
      cudaEventCreate(&stop);
      cudaEventRecord(stop, 0);
      cudaEventSynchronize(stop);
      cudaEventElapsedTime(&elapsedTime, start, stop);
```

```
      printf("GPU scale took: %f s on gpu  %d \n", elapsedTime/1000, threadId);

      gpuErrchk(cudaMemcpy(output_h+novStart, output_d, (novEnd-novStart) *
          sizeof(int), cudaMemcpyDeviceToHost));
      cudaFree(adj_d);
      cudaFree(xadj_d);

    }
    else{
      // printf("Entered \n" );

      // int novForThread = (nov+PARALEL_THREAD_COUNT-1)/PARALEL_THREAD_COUNT;
      // int novStart = novForThread * threadId;
      // int novEnd   = novForThread * (threadId+1);
      // if (novEnd> nov) novEnd = nov;

          // int numBlock = (novEnd-novStart + THREADS_PER_BLOCK-1) / THREADS_PER_BLOCK;
      int novStart = 4 * GPU_MULTIPLIER*novForThread +  1 * novForThread *
          (threadId-4);
      int novEnd   =  novStart + 1*  novForThread ;
      if (novEnd> nov) novEnd = nov;


      bool *marked = new bool[nov];
      memset(marked, false, nov * sizeof(bool)); // bu belki silinebilir

      double start_thread = omp_get_wtime();
      for(int i = novStart; i < novEnd; i++){
          int localcount = 0;
          DFS_sparse(xadj, adj, marked, n - 1, i, i, localcount);
          output_h[i ] = localcount;

      }
      double end_thread = omp_get_wtime();
      printf("Took %f secs \n", end_thread -start_thread );

    }

  }
  end_cpu = omp_get_wtime();
}
```

7.6 Multi GPU (Non-recursive)+ CPU Kernel

```
void DFS_sparse(int xadj[], int adj[], bool marked[], int n,
         int vert, int start, int &count)
             //vert: bulundugu konum //start: baslangıc noktası
{
    marked[vert] = true;
    int start_index = xadj[vert];
    int path_length = xadj[vert+1];

    if (n == 0){
        marked[vert] = false;
        for(int i = start_index; i < path_length; i++){
            if(adj[i] == start){
                count++;
                break;
            }
        }
        return;
    }

// if(path_length-start_index <=1) return;

    for(int i=start_index; i < path_length; i++){
        if(!marked[adj[i]]){
            DFS_sparse(xadj, adj, marked, n-1, adj[i], start, count);
        }
```

```
    }
    marked[vert] = false;
}



__global__ void kernel3(int* adj, int* xadj, int* output, int nov, int
    novStart){

  int index = novStart + threadIdx.x + (blockIdx.x * blockDim.x);
  if(index < nov){
      // if(index ==0)printf("called gpu \n");
      //int *marked = new int[n];
      //memset(marked, -1, n * sizeof(int)); // bu belki silinebilir
      int localcount = 0;
      // int round = 0;

      // 0-->
      int s0 = xadj[index];
      int e0 = xadj[index+1];

      for(int i=s0; i < e0; i++){
        // 0 --> 1

        int neighbour_1 = adj[i];
        int s1  = xadj[neighbour_1];
        int e1  = xadj[neighbour_1+1];

        for(int j=s1;j < e1; j++){
          // 0 --> 1 --> 2

          int neighbour_2 =  adj[j];
          if (neighbour_2 == index) continue;
          int s2  = xadj[neighbour_2];
          int e2  = xadj[neighbour_2+1];

          for(int k=s2; k < e2; k++){

            // 0 --> 1 --> 2 --> 3
            int neighbour_3 =  adj[k];
            if (neighbour_3 == index){
              localcount+=1;
              break;
            }
          }
        }
      }
      output[index-novStart] = localcount;
    }
}


void wrapper(int *xadj, int *adj, int n,  int nov, int nnz){

  int *output_h = new int[nov];

  double start_cpu, end_cpu;
  start_cpu = omp_get_wtime();


  #pragma omp parallel num_threads(PARALEL_THREAD_COUNT)
  {

    int threadId=omp_get_thread_num ();
    // cout<< threadId<<endl;

    int virtual_thread_count  = GPU_MULTIPLIER *4 + PARALEL_CPU;
    int novForThread = (nov+virtual_thread_count-1)/virtual_thread_count;



    if(threadId <=3)
    {
      int novStart = GPU_MULTIPLIER * novForThread * threadId;
      int novEnd   = GPU_MULTIPLIER * novForThread * (threadId+1);
      if (novEnd > nov) novEnd = nov;
      int numBlock = (novEnd-novStart + THREADS_PER_BLOCK-1) /
          THREADS_PER_BLOCK;
      // printf("nov s %d e %d \n", novStart,novEnd);


      cudaSetDevice(threadId);

      int *adj_d;
      int *xadj_d;
      int *output_d;
      cudaEvent_t start, stop;
```

```
        float elapsedTime;

        gpuErrchk(cudaMalloc((void**)&adj_d, (nnz) * sizeof(int)));
        gpuErrchk(cudaMalloc((void**)&xadj_d, (nov + 1) * sizeof(int)));

        gpuErrchk(cudaMalloc((void**)&output_d, (novEnd-novStart) * sizeof(int)));

        //gpuErrchk(cudaMallocHost((void **)&output_h, (nov) * sizeof(int)));

        gpuErrchk(cudaMemcpy(adj_d, adj, (nnz) * sizeof(int), cudaMemcpyHostToDevice));
        gpuErrchk(cudaMemcpy(xadj_d, xadj, (nov + 1) * sizeof(int), cudaMemcpyHostToDevice));

        cudaEventCreate(&start);
        cudaEventRecord(start, 0);
        double start_gpu = omp_get_wtime();
        cudaStream_t stream1;
        cudaStreamCreate ( &stream1) ;

          // printf("threadId entry to kernel %d GPU \n", threadId );
        if      (n==3)kernel3<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d, xadj_d, output_d, novEnd,novStart);
        else if (n==4)kernel4<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d, xadj_d, output_d, novEnd,novStart);
         else if (n==5)kernel5<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d, xadj_d, output_d, novEnd,novStart);

            //combination<<<numBlocks, threadsPerBlock>>>(adj_d, xadj_d, output_d, n, nov);
        // printf("threadId exit to kernel %d GPU \n", threadId );
        double end_gpu = omp_get_wtime();


        gpuErrchk(cudaDeviceSynchronize());
        cudaEventCreate(&stop);
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&elapsedTime, start, stop);

        printf("GPU scale took: %f s on gpu  %d \n", elapsedTime/1000, threadId);

        gpuErrchk(cudaMemcpy(output_h+novStart, output_d, (novEnd-novStart) * sizeof(int), cudaMemcpyDeviceToHost));
        cudaFree(adj_d);
        cudaFree(xadj_d);


      }
      else{
        // printf("Entered \n" );

        // int novForThread = (nov+PARALEL_THREAD_COUNT-1)/PARALEL_THREAD_COUNT;
        // int novStart = novForThread * threadId;
        // int novEnd   = novForThread * (threadId+1);
        // if (novEnd> nov) novEnd = nov;

            // int numBlock = (novEnd-novStart + THREADS_PER_BLOCK-1) / THREADS_PER_BLOCK;
        int novStart = 4 * GPU_MULTIPLIER*novForThread +  1 * novForThread * (threadId-4);
        int novEnd    =  novStart + 1*  novForThread ;
        if (novEnd> nov) novEnd = nov;

        // printf("nov s %d e %d  -cpu \n", novStart,novEnd);

        bool *marked = new bool[nov];
        memset(marked, false, nov * sizeof(bool)); // bu belki silinebilir

        double start_thread = omp_get_wtime();
        for(int i = novStart; i < novEnd; i++){
            int localcount = 0;
            DFS_sparse(xadj, adj, marked, n - 1, i, i, localcount);
            output_h[i ] = localcount;

        }
        double end_thread = omp_get_wtime();
        printf("Took %f secs \n", end_thread -start_thread );

      }
```

```
    }
    end_cpu = omp_get_wtime();

     // printf("Took %f secs \n", end_cpu - start_cpu);
    // printArray(output_h,nov);
}
```

7.7 Multi GPU + CPU + Dynamic Workload

```
__host__ void DFS_sparse(int xadj[], int adj[], bool marked[], int n,
         int vert, int start, int &count)
             //vert: bulundugu konum //start: baslangıc noktası
{
    marked[vert] = true;
    int start_index = xadj[vert];
    int path_length = xadj[vert+1];

    if (n == 0){
        marked[vert] = false;
        for(int i = start_index; i < path_length; i++){
            if(adj[i] == start){
                count++;
                break;
            }
        }
        return;
    }

// if(path_length-start_index <=1) return;

    for(int i=start_index; i < path_length; i++){
        if(!marked[adj[i]]){
            DFS_sparse(xadj, adj, marked, n-1, adj[i], start, count);
        }
    }
    marked[vert] = false;
}


__global__ void kernel3(int* adj, int* xadj, int* output, int nov, int
    novStart){

  int index = novStart + threadIdx.x + (blockIdx.x * blockDim.x);
  if(index < nov){
      // if(index ==0)printf("called gpu \n");
      //int *marked = new int[n];
      //memset(marked, -1, n * sizeof(int)); // bu belki silinebilir
      int localcount = 0;
      // int round = 0;

      // 0-->
      int s0 = xadj[index];
      int e0 = xadj[index+1];

      for(int i=s0; i < e0; i++){
        // 0 --> 1

        int neighbour_1 = adj[i];
        int s1  = xadj[neighbour_1];
        int e1  = xadj[neighbour_1+1];

        for(int j=s1;j < e1; j++){
          // 0 --> 1 --> 2

          int neighbour_2 =  adj[j];
          if (neighbour_2 == index) continue;
          int s2  = xadj[neighbour_2];
          int e2  = xadj[neighbour_2+1];

          for(int k=s2; k < e2; k++){

            // 0 --> 1 --> 2 --> 3
            int neighbour_3 =  adj[k];
```

```
            if (neighbour_3 == index){
              localcount+=1;
              break;
            }
          }
        }
      }
      output[index-novStart] = localcount;
    }
}

void wrapper(int *xadj, int *adj, int n,  int nov, int nnz){

  int *output_h = new int[nov];

  double start_cpu, end_cpu;
  start_cpu = omp_get_wtime();

  int totalChunck = (nov + CHUNK_SIZE_OUR -1) /CHUNK_SIZE_OUR;
  int currentChunk = 0; //mask accesed atomicly



  #pragma omp parallel num_threads(PARALEL_THREAD_COUNT)
  {


    int threadId=omp_get_thread_num ();
    // cout<< threadId<<endl;
    //
    // int virtual_thread_count  = GPU_MULTIPLIER *4 + PARALEL_CPU;
    // int novForThread = (nov+virtual_thread_count-1)/virtual_thread_count;



    if(threadId <=(OUR_GPU_COUNT-1))
    {
      // int novStart = GPU_MULTIPLIER * novForThread * threadId;
      // int novEnd   = GPU_MULTIPLIER * novForThread * (threadId+1);
      // if (novEnd > nov) novEnd = nov;

          // int numBlock = (novEnd-novStart + THREADS_PER_BLOCK-1) / THREADS_PER_BLOCK;
      // // printf("nov s %d e %d \n", novStart,novEnd);


      cudaSetDevice(threadId);

      int *adj_d;
      int *xadj_d;
      int *output_d;
      cudaEvent_t start, stop;
      float elapsedTime;

      gpuErrchk(cudaMalloc((void**)&adj_d, (nnz) * sizeof(int)));
      gpuErrchk(cudaMalloc((void**)&xadj_d, (nov + 1) * sizeof(int)));

      gpuErrchk(cudaMalloc((void**)&output_d, (GPU_MULTIPLIER *CHUNK_SIZE_OUR)
          * sizeof(int)));

      //gpuErrchk(cudaMallocHost((void **)&output_h, (nov) * sizeof(int)));

      gpuErrchk(cudaMemcpy(adj_d, adj, (nnz) * sizeof(int),
          cudaMemcpyHostToDevice));
      gpuErrchk(cudaMemcpy(xadj_d, xadj, (nov + 1) * sizeof(int),
          cudaMemcpyHostToDevice));

      cudaEventCreate(&start);
      cudaEventRecord(start, 0);
      double start_gpu = omp_get_wtime();
      cudaStream_t stream1;
      cudaStreamCreate ( &stream1) ;


      while(true){
        int thisChunk;
        #pragma omp critical
        {
          thisChunk=currentChunk; //thisChunk is different on everyone.
          currentChunk+= GPU_MULTIPLIER;

        }
        if (thisChunk  >= totalChunck) break;

        int novStart   = thisChunk * CHUNK_SIZE_OUR;

        int novEnd =  novStart + GPU_MULTIPLIER * CHUNK_SIZE_OUR;

```

```
        // printf("nov s %d e %d \n", novStart,novEnd);
        if(novEnd>nov) novEnd=nov;

        int numBlock = (novEnd-novStart + THREADS_PER_BLOCK-1) /
            THREADS_PER_BLOCK;


        if      (n==3)kernel3<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d,
            xadj_d, output_d, novEnd,novStart);
        else if (n==4)kernel4<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d,
            xadj_d, output_d, novEnd,novStart);
        else if (n==5)kernel5<<<numBlock, THREADS_PER_BLOCK,0,stream1>>>(adj_d,
            xadj_d, output_d, novEnd,novStart);

      gpuErrchk(cudaDeviceSynchronize());
      gpuErrchk(cudaMemcpy(output_h+novStart, output_d, (novEnd-novStart) *
          sizeof(int), cudaMemcpyDeviceToHost));

      }

      double end_gpu = omp_get_wtime();


      cudaEventCreate(&stop);
      cudaEventRecord(stop, 0);
      cudaEventSynchronize(stop);
      cudaEventElapsedTime(&elapsedTime, start, stop);

      printf("GPU scale took: %f s on gpu  %d \n", elapsedTime/1000, threadId);

      cudaFree(adj_d);
      cudaFree(xadj_d);

    }
    else{

    double start_thread = omp_get_wtime();

    while(true){
      int thisChunk;

      #pragma omp critical
      {
        thisChunk=currentChunk; //thisChunk is different on everyone.
        currentChunk++;

      }
      if(thisChunk >= totalChunck) break;

      int novStart   = thisChunk * CHUNK_SIZE_OUR;
      int novEnd =  (thisChunk+1)* CHUNK_SIZE_OUR;
      if(novEnd>nov) novEnd=nov;


        bool *marked = new bool[nov];
        memset(marked, false, nov * sizeof(bool)); // bu belki silinebilir


        for(int i = novStart; i < novEnd; i++){
            int localcount = 0;
            DFS_sparse(xadj, adj, marked, n - 1, i, i, localcount);
            output_h[i ] = localcount;


        }
      }
      double end_thread = omp_get_wtime();
      printf("Took %f secs \n", end_thread -start_thread );

    }

  }
  end_cpu = omp_get_wtime();

   printf("Took %f secs \n", end_cpu - start_cpu);
}
```