CUDA Programming
Fall 2020

Instructions

- You are asked to fill in code in the file **tnt_counting.cu**, and keep the other files unchanged. Please submit a single source file - your modified **tnt_counting.cu**
- **No late submissions will be accepted.**
- Your submission will be run with the commands:

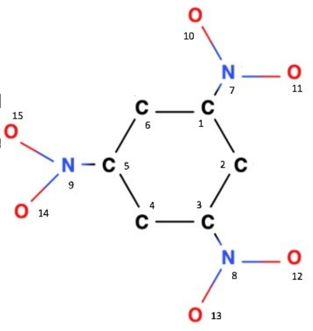
  ```
  ./run_cuda.sh
  ```

  Please make sure your submission is executable. If it is not, you will get 0 mark.

NOTE: We will perform code similarity checks. In case a submission is confirmed to have code similarity issues, we will deduct partial marks or full marks on a case-by-case basis.

1. Problem Overview

In this assignment, you will implement a CUDA C function tnt_counting that employs the GPU to find all occurrences of **2,4,6-Trinitrotoluene (TNT)** structures in a large chemica compound graph. In our simplified problem, a TNT quer graph is identified as a six-C ring with three non-neighboring C atoms on the ring each connected with an NO2 structure. The structure NO2 consists of one N atom bound with two O atoms. The figure on the right illustrates an example TNT structure in the graph.



2. Input

We store the input graph in a plain text file, for example, data.txt in the zip package provided to you. We consider only four kinds of edges (C-C, C-N, C-H and N-O) in the input graph. For example, in data.txt, the first line is "c c <#C-C edges>" where <#C-C edges> is the number of C-C edges in the graph. The following lines are this number of C-C edges with each edge in a line. Each edge is represented as a pair of vertex IDs. The other types of edges are stored in the input file the same way: the leading line is the type of edge and number of this type of edges, and the following lines are edges of that type, one edge in a line. Each edge in the graph appears exactly once in the file.

In main.cu, the read_file function loads all edges from the input file and stores them into their corresponding edge arrays. These edge arrays and numbers of edges are shown in Table 1.

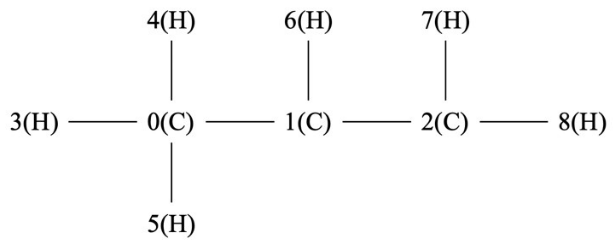
Table 1: Edge arrays and their sizes

Variables	Description
c_c	Pointer to the array of C-C edges
c_n	Pointer to the array of C-N edges
c_h	Pointer to the array of C-H edges
n_o	Pointer to the array of N-O edges.
c_c_size	The number of edges in c_c edge array.
c_n_size	The number of edges in c_n edge array.
c_h_size	The number of edges in c_h edge array.
n_o_size	The number of edges in n_o edge array.

Each edge array stores two rows in sequence, with the first row containing the first endpoint of each edge and the second row containing the second endpoint of each edge. This way, the elements at the same column of the two rows represent an edge as the vertex IDs of the two end points. We store edges this way so that accesses to the first endpoints of the edges can be coalesced on the GPU, and so are the accesses to the second endpoints of the edges.

The c_c edge array is special since the two endpoints are both C. We store each C-C edge twice by reversing the order of the two endpoints so that we can find all C-C edges by only checking in one direction. As a result, c_c_size is twice the number of unique c_c edges in the graph. The other three kinds of edges are stored only once in their corresponding edge arrays.

For example, suppose our input file describes the following chemical compound graph, where each vertex is shown with "id(atom)" in the following figure.



There are two C-C edges (0,1) and (1,2), and we also store them in reverse vertex order (1,0) and (2,1). There are six C-H edges (0,3), (0,4), (0,5), (1,6), (2,7) and (2,8), of which we only store one copy. The C-C and C-H edges can be shown as the following 2D format:

0	1	1	2
1	2	0	1

0	0	0	1	2	2
3	4	5	6	7	8

After executing the read_file function, the edge arrays, c_c and c_h are constructed as follows.

0	1	1	2	1	2	0	1

0	0	0	1	2	2	3	4	5	6	7	8

The values of c_c_size and c_h_size are 4 and 6 correspondingly.

In your program you need to **copy** the edge arrays from the host CPU to the device before using them in your GPU kernel programs.

3. Output

The final_results array stores 15 rows in a sequence, with each row containing final_result_size columns. Each row corresponds to a vertex in the 15-vertex TNT structure in the input graph, so the same column of all 15 rows form a mapping from a TNT structure in the input graph to our given TNT query graph. Again, this design of the array structure facilitates the coalesced access of vertices on the GPU.

In your tnt_counting function you need to **copy** the final results from the GPU device to the final_results array on the CPU and set the final_result_size.

The final_results array should contain all possible mappings and have no duplicate mappings. Specifically, each TNT structure in the input graph can map to the given TNT query graph in different ways – the three Ns in the input graph can map to the three Ns in the query in six ways, and each O of an NO2 in the input graph can map to one of the two Os in the query. So a single TNT structure in the input graph will produce 6*2*2*2 = 48 mappings in the result.

4. Helper Functions

We provide a device function, idx, in helpers.h, which can convert a 2d array index to a 1d array index. You can add any other helper function and data structure in tnt_counting.cu.

5. Compilation and Testing

You can compile and run your code by running the following command:

```
./run_cuda.sh
```

We provide a sequential tnt_counting binary file and a cuda tnt_counting binary file for testing the results. You can run them using the following commands:

```
./tnt_counting_sequential data.txt
```

or

```
./tnt_counting_cuda data.txt 8 512
```

You can also use a smaller file simple.txt. The order in the final results is unimportant.

6. Grading

1. Full marks will be deducted, if your submission cannot be compiled.
2. Partial marks will be deducted, if your code has runtime error, returns wrong results or runs slower than the sequential version binary on the given data file.
3. The top 3 fastest submissions will get bonus points (champion 3 points, runner-up 2 points, and second runner-up 1 point).