CS 314 Principles of Programming Languages
Project 3: Efficient Parallel Graph Matching

THIS IS NOT A GROUP PROJECT! You may talk about the project in general terms, but must not share your code. In this project, you will be asked to implement a component of a parallel graph matching algorithm. The program takes a graph file (in matrix-market format) as input and returns a graph matching.

You will encounter considerable amount of design and implementation issues while you are working on this project. Identifying these issues is part of the project. You should start early, allowing enough time for debugging, testing, and improving of your code.

1 Background

1.1 Graph Matching Problem

Given a graph $G = (V, E)$, where V is the set of vertices (also called nodes) and $E \subset |V|^2$. A **matching** M in G is a set of pairwise non-adjacent edges such that no two edges share a common vertex. A vertex is **matched** if it is an endpoint of one of the edges in the matching. A vertex is **unmatched** if it does not belong to any edge in the matching. In Fig. 1, we show examples of possible matchings for a given graph.

A **maximum matching** can be defined as a matching where the total weight of the edges in the matching is maximized. In Fig. 1, (c) is a **maximum matching**, where the total weight of the edges in the matching is 7. Fig. (a) and (b) respectively have the total weight of 3 and 2.

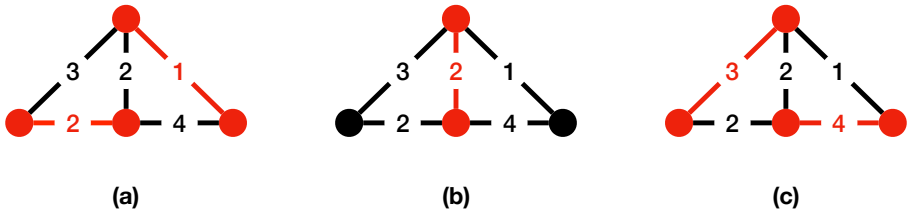


Figure 1: Graph Matching Examples

1.2 Parallel Graph Matching

Most well-known matching algorithms such as *blossom algorithm* are embarrassingly sequential and hard to parallelize. In this project, we will adopt the handshaking-based algorithm which is amenable to parallelization on GPUs.

In the handshaking-based algorithm, a vertex v extends a hand to one of its neighbours and the neighbor must be on the maximum-weight edge incident to v. If two vertices shake hands, the edge between these two vertices will be added to the matching. An example is shown in Fig. 2 (b) where node A extends a hand to D since edge(A,D) has the largest weight among all edges incident to node A; Nodes C and F shake hands because they extend a hand to each other.

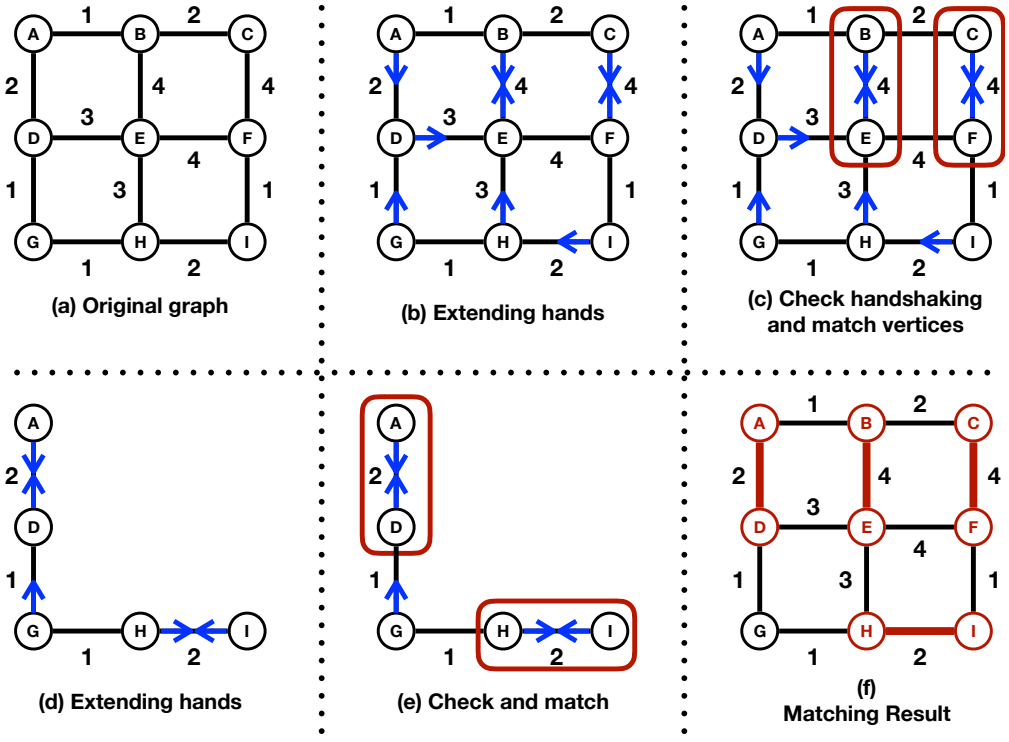


Figure 2: One-Way Handshaking Matching Example

It is possible that multiple incident edges of a node have maximum weight. In this project, we let the algorithm pick the neighbor vertex that has the **smallest vertex index**. For example, in Fig. 2(b), among the maximum-weight neighbors of vertex E, we pick vertex B since it has the smallest index (in alphabetical order) among all E's edges that have maximum-weight 4.

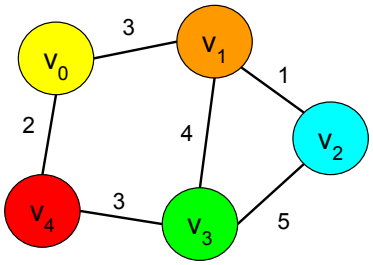
The handshaking algorithm needs to run one or multiple passes. A one-pass handshaking checks all nodes once and only once. At the end of every pass, we remove all matched nodes and check if there is any remaining un-matched nodes. If the remaining un-matched nodes are connected, another pass of handshaking must be performed. We repeat this until no more

edges can be added to the matching. In Fig. 2, we show two passes of handshaking.

The handshaking algorithm is highly data parallel, since each vertex is processed independently and the find-maximum-weight-neighbor step involves only reads to shared data. It is a greedy algorithm that attempts to maximize the total weight in the matching.

2 Implementing Parallel Graph Matching

2.1 Data Structure



src	0	0	1	1	1	2	2	3	3	3	4	4
dst	1	4	0	2	3	1	3	1	2	4	0	3
weight	3	2	3	1	4	1	5	4	5	3	2	3

Figure 3: A graph, and its encoding as an edge list

The graph is encoded as an edge list consisting of three arrays: src, dst, and weight, such that src[n] is the source node for the n-th edge, dst[n] is the destination node for the n-th edge, and weight[n] is the weight of the n-th edge. The graph is undirected, so if src[n]=x and dst[n]=y then there exists an edge m such that src[m]=y and dst[m]=x.

An example of this representation is shown in Fig. 3. The example graph, with five vertices, has a total of six un-directed edges. **The edge list is arranged such that edges that have the same source node are placed together.** For edges that have the same source node, they are sorted by the destination indices. The relative order of the edges is important, please do not modify it.

We have provided graph I/O functions for you. The code given to you will read and parse the graphs stored in matrix format. After the graph is parsed, three arrays src[ ], dst[ ], weight[ ] contain graph information. Pointers to the three arrays: src[ ], dst[ ], weight[ ] are stored in the GraphData struct. This is what GraphData looks like:

```
struct GraphData {
    int numNodes;
    int numEdges;
    int * src;
    int * dst;
```

```
    int * weight;
}
```

The program contains four GPU kernel functions, three of which you are required to implement (while the fourth is already implemented). Their arguments (input and output) are all described in the file *src/gpuHeaders.cuh*. Your implementations should go inside the appropriate files in the *src/gpu_required/* directory.

In Fig. 4 we show an example of the data produced during the first iteration of the handshaking algorithm, based on the same graph as in our edge list example. Note that the notations used in Fig. 4 will be used consistently in the following subsection. Those notations include the keep_edges, new_indices, strongNeighbor, and matches arrays. Please refer to Fig. 4 for a specific example of the results from each of the functions described in the subsections below.

We use the strongNeighbor[ ] arrays (strongNeighbor_cpu[ ] and/or strongNeighbor_gpu[ ], which reside in CPU memory and GPU memory, respectively) to store the results of maximum-weight neighbors. The i-th element of this array stores the maximum-weight neighbor of node i.

Note that by default, the program uses provided CPU code to serially identify the strongest neighbors. For information about the extra credit part of this project, see Section 6.

2.2 Update Matches

With the strongNeighbor results calculated, we can now perform handshaking to update the matching results. This kernel function looks for node-pairs which are both each other's maximum-weight neighbor, and matches them.

```
__global__ void check_handshaking_gpu(int * strongNeighbor, int * matches, int numNodes);
```

The strongNeighbor array is the input, such that strongNeighbor[x]=y if the x-th node's strongest neighbor is the y-th node. The matches array is the output, such that matches[x]=y if the x-th has been matched with the y-th node. Note that some nodes may already be matched from prior iterations, and they should not be changed. An unmatched node is indicated by a matches value of -1.

2.3 Edges Filtering

To filter out the edges we no longer need, we must first distinguish them from the edges to keep. A vertex can be matched with only one vertex – therefore if an edge has a source or destination which has already been matched then we can discard it; otherwise we must keep it.

```
__global__ void markFilterEdges_gpu(int * src, int * dst, int * matches, int * keepEdges,
int numEdges);
```

Figure 4: An Iteration of the Handshaking Algorithm

The input arrays for this function are src (the source array from the edge list), dst (the destination array from the edge list), matches (the current node matches). The output is keepEdges. The src, dst, and keepEdges arrays each have *numEdges* elements.

After this function is complete, keepEdges[x] should be 1 if we want to keep the xth edge, or 0 otherwise.

2.4 Get New Edge Indices

Once we know which edges to keep versus which edges to filter, we can determine each edge's new index in the edge list. This is performed with an exclusive prefix sum.

__global__ void **exclusive_prefix_sum_gpu**(int * oldSum, int * newSum, int distance, int numElements);

The input array is oldSum, and the output array is newSum. Each of these arrays has *numElements* elements. The distance variable tells us which stride to use at each iteration. This function will be called multiple times. A sketch of the parallel algorithm is in lecture 21 slides, page 32.

Since this is an **exclusive** prefix sum, we do not want the sum to include the current element. In other words, final_sum[x] = (input[0] + input[1] + ... + input[x-1]).

The first time **exclusive_prefix_sum_gpu** is launched, it will be given a distance value of 0. In this case, instead of performing addition, it should merely shift everything one element to the right – in other words, each element in the output should be set to the **previous** element in the input when distance is 0 in this kernel function. That is how we implement the exclusivity, after which we can operate like an inclusive prefix scan.

2.5 Filter Edges

This function is already implemented for you. When the program reaches this point, we know where the edges will go in the filtered edge list, but we need to repack them for use in the next iteration of matching.

```
__global__ void packGraph_gpu(int * newSrc, int * oldSrc, int * newDst, int * oldDst, int
* newWeight, int * oldWeight, int * edgeMap, int numEdges)
```

Here oldSrc, oldDst, and oldWeight are the edge list before filtering, each with numEdges elements. The edgeMap array, with (numEdges+1) elements, contains the results of the exclusive prefix sum from the previous step. The newSrc, newDst, and newWeight arrays are the output of this kernel function.

After this kernel function is complete, for every edge x that we want to keep, it will be the case that newSrc[edgeMap[x]] = oldSrc[x]. The dest and weight arrays are handled similarly.

2.6 GPU Memory

You are expected to handle allocation, initialization, and de-allocation of GPU memory yourself. You must use the function **cudaMalloc** for to allocate GPU memory. For initialization, you can use **cudaMemcpy** to copy memory between GPU and CPU. Finally, you must use **cudaFree** for de-allocation.

You can find examples of how to use these functions inside the Lecture 19 and Lecture 20 slides from class.

2.7 Thread Assignment

It is important to map tasks to threads in a reasonable manner. Furthermore, although GPU kernels can be launched with a large number of threads, we may not want to do so because it incurs significant kernel launch overhead. **In this project, we do not guarantee the**

number of threads allocated for each kernel is the same as the number of tasks to perform. Your code within a kernel needs to account for that.

For example, suppose we are working on an array with 1024 elements, but only have 256 threads. Each thread should work on four elements. To ensure memory coalescing, the first thread should handle array[0], array[256], array[512], and array[768]. Meanwhile the second thread should handle array[1], array[257], array[513], and array[769], and etc. This mapping distributes the work evenly across threads, while offering opportunity for the GPU device to combine adjacent memory requests performed by adjacent threads.

We use one-dimensional grids and one-dimensional thread-blocks, making it straightforward for you to calculate both the thread's ID, and the total number of threads launched by the kernel function. Each thread can query the total number of threads launched using the expression blockDim.x * gridDim.x.

Also note that the number of tasks to process is not necessarily a multiple of the number of threads. In your code, you will need to add condition checks to account for that. For the three kernels you're required need to implement, the number of tasks is the same as the size of the output array, which is given as an argument.

3 Grading

Your programs will be graded primarily on functionality. Grading will be done on *ilab* machines. You will receive 0 credit if we cannot compile and run your code on *ilab* machines.

Although we have provided you with some testcases, we will use secret test cases when we perform grading. Additionally, we will run your code with varying numbers of threads.

During grading, your submission will be run with a (generous) time limit, to ensure the program terminates even if it gets stuck. We may run your kernel functions separately (alongside our own implementations of the other functions), or altogether.

4 How to Get Started

The code package for you to start with is provided on **Sakai**. Create your own directory on the ilab cluster, and copy the entire provided project folder to your home directory or any other one of your directories. Make sure that the read, write, and execute permissions for groups and others are disabled (`chmod go-rwx <directory_name>`).

DO NOT change any files except onewaywrapper.cu, the existing files in the gpu_required directory, and the file in the gpu_extra_credit directory. No other code files will be considered when we grade your submission.

The various parts of the code you're expected to modify are prepended and post-pended by comments stating "YOUR CODE GOES BELOW" and "YOUR CODE GOES ABOVE", respectively. This includes the contents of kernel functions, as well as allocation and deallocation of GPU memory.

4.1 Compilation and Execution

Compilation: We have provided a **Makefile** in the sample code package.

- **make** should compile all code into an executables called **match**.
- **make submit** should generate a tar file containing your current code, which you must manually submit to Sakai.
- **make clean** should remove all files that were generated by Make, including object files, executables, and the tar file.
- **make debug** compiles the executable with flags that disable optimizations and add debugging information. Note that you may need to run **make clean** prior to switching between the debug and default versions.

Execution: The match program performs one-way matching on a given graph. It allows the following arguments:

- **-input <file>** Use the specified graph file as input.
- **-output <file>** Write matching results to specified file.
- **-threads <number>** Launch approximately the specified number of threads when calling a GPU kernel.
- **-device <number>** Run all GPU kernels on the specified GPU device.

The -input flag is mandatory; if the other flags are not used then the program will use default settings, such as writing the results to a file named *out.txt*.

Program Output: The program will output the matching results to the specified output file, and print the approximate time it spent. We have handled the functionality of generating the output file.

4.2 Test Cases

We have provided input and expected output for several randomly generated testcases. Each of the input files has a filename “inputX.mtx" for some value X. The corresponding solution has filename “outputX.txt".

For example, if you fully and correctly implement the required parts of this project, then the command **./match -input testcases/input5.mtx -output out.txt -threads 1024** would produce a file named *out.txt* that's identical to *testcases/output5.txt*. (You can also use a different number of threads than 1024, and should get the same output.)

Note that you can use the **diff** command to compare two text files. For example, **diff out.txt testcases/output9.txt** will either display a list of differences between those two files, or display nothing if they are identical.

4.3 Debugging

You can use **cuda-gdb** to help detect and debug errors in your GPU kernel functions. Make sure to use the command **set cuda memcheck on** when you first launch cuda-gdb.

Before debugging, make sure to compile your code with the **make debug** command. This might make your code run slower, but will allow cuda-gdb to retrieve more high-level information about any memory errors it detects.

4.4 GPUs are a finite resource!

You can run the program **nvidia-smi** to view the GPUs on your current machine, including some information about which ones are currently in use. If the project executable tries to run on a GPU device that is already used by many users, then it may be unable to run correctly, getting stuck or throwing unusual errors (e.g. "out of memory").

We've added an optional argument to the program that lets you select the GPU device to use. For example, if nvidia-smi tells you that there are four GPU devices but devices 0, 1, and 2 are all in use, then you might run the **match** program with the flag "-device 3" so that it will use device #3. However, be aware that some of the computers in ilab that you can use have only one GPU.

If this isn't sufficient to get the program working (i.e. there are no available GPU devices), then you may need to switch to a different ilab machine. You can check which machines are idle here: report.cs.rutgers.edu/nagiosnotes/iLab-machines.html .

5 What to Submit

You should invoke **make submit** to generate the file **submit_me.tar** containing your GPU code. This only generates the file; it does not automatically submit it. You need to submit **submit_me.tar** to Sakai. If you change your code after generating this tar file, don't forget to regenerate (and resubmit) the tar afterward.

If you complete the extra credit, then you should additionally submit your 3-page report in PDF format.

Please **DO NOT** submit any other code, executable, mtx files, matching results, etc.

6 Extra-credit

The extra credit part of this project concerns the parallel implementation of finding the strongest neighbors (i.e. producing the strongNeighbor array in Fig. 1).

If you choose to complete the extra-credit, you must design the parallel algorithm for finding the maximal neighbor of each node. There are several factors to consider include, but are not limited to:

- Load balancing. Each node has a different number of neighbors, so partitioning the work according to node is a bad idea.

- Thread divergence. Each thread in each warp (i.e. each group of 32 consecutive threads) should perform the same operations, or else parallelism is reduced.
- Memory coalescing. A memory operation is more efficient when consecutive threads are accessing consecutive memory locations.

You can use as many GPU kernel functions as you feel are necessary, which should all be placed inside the extra.cu file. You will call these kernels at the appropriate place inside onewaywrapper.cu. If necessary, you are permitted to define additional GPU arrays inside onewaywrapper.cu. Your implementation will be invoked iff you use the -extra flag on the command-line when you run the **match** program.

For full points on the extra credit, **you need need to write a three-page report** including your algorithm, the implementation results, and evaluation.

7 Questions

Questions regarding this project can be posted on Sakai forum. Good luck!