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

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

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 $G = (V, E)$, while 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.

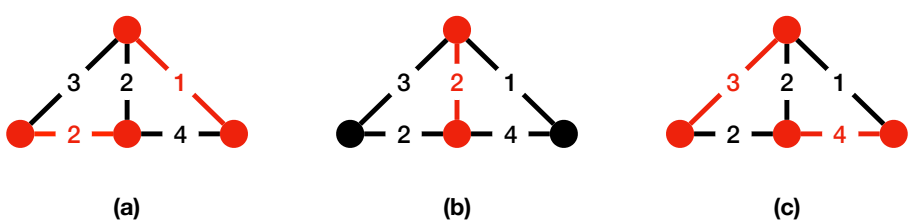


Figure 1: Graph Matching Examples

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 be adopting handshaking-based algorithm that is amenable to parallelization and can be a good fit for GPUs.

In the handshaking-based algorithm, a vertex v extends a hand to one of its neighbours and the neighbor must be sitting 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.

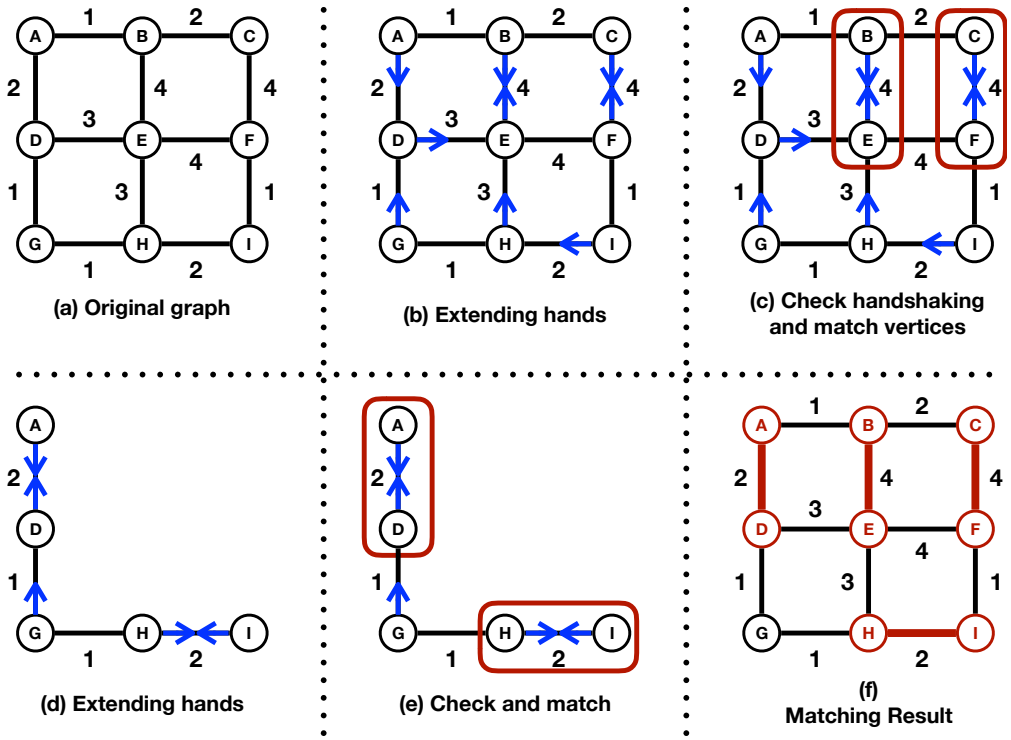


Figure 2: One-Way Handshaking Matching Example

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 need 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

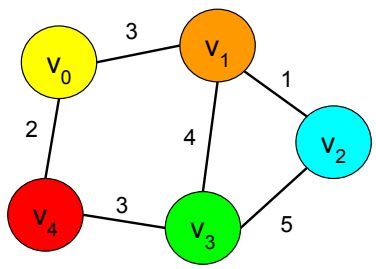
edges can be added to the matching. In Fig. 2, we show two passes of handshaking.

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 but no writes. It is a greedy algorithm that attempts to maximize the total weight in the matching.

2 Finding Maximum-weight Neighbor

In the handshaking algorithm, the component of finding maximum-weight neighbor is non-trivial to parallelize. In this project, you will implement two GPU kernel functions for identifying maximum-weight neighbor of each node in the graph. You can implement other components of the handshaking algorithm as well, however, it is not required. See Section 6 for the extra-credit functionalities if you are interested.

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 encoded 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;
}
```

We use the global arrays strongNeighbor[ ] and strongNeighbor_gpu[ ] to store the results of maximum-weight neighbors. strongNeighbor[i] stores the maximum-weight neighbor of node i on CPU. strongNeighbor_gpu[i] stores the maximum-weight neighbor of node i on GPU. In the end they should be identical, except one is allocated in CPU memory and the other is allocated in GPU memory.

In the main function located in src/mainStrong.cu for running the required component of the project, we call the write_match_result function, which will write the results into the specified output file. The size of the output will be the number of nodes in the graph. For example, if strongNeighbor=[4,2,1,4,0] then it will output those five numbers, in that order, separated by line breaks.

There are two GPU kernel functions that you are required to implement. Their arguments (input and output) are all described in the file *gpuHeaders.cuh*, along with the extra credit functions. Your implementations should go inside the *gpu_required* directory, which currently contains a separate file for each function. The two functions are described respectively in Section 2.2 and Section 2.3.

2.2 Parallel Segment-scan

To find the maximum-weight neighbor, we use parallel **segment-scan**. We first define segment-scan: Let P be an input array of segment IDs, A be an input array of data elements, and O the output array. Given an arbitrary *reduce* function f, the result of segment-scan is

$$O[j] = f(A[i], A[i+1], ..., A[j]) \ such\ that\ i = min\{x | 0 \leq x \leq j, P[i] = P[j]\}$$

For example, if we want to use the **max** function, which returns the larger of two or more numbers, as our reduce function, the output for each element will be the max of all prior numbers in that segment. In other words,

$$O[j] = max(A[i], A[i+1], ..., A[j]) \ such\ that\ i = min\{x | 0 \leq x \leq j, P[i] = P[j]\}$$

For example, suppose P=[0,0,0,1,2,2,3,3,3,3] and A=[6,1,9,2,7,4,2,8,3,9]. Then the output of a segment scan for maximum values is O=[6,6,9,2,7,7,2,8,8,9]. We can see in this segment-scan example, the last element in each segment should be the largest of all that segment.

In this project, we divide an edge list into multiple segments such that each segment contains edges from the same source vertex. In Fig. 3 and Fig. 5 we have given each segment a different color, to make them more visually distinctive. Here index $0-1$ (inclusive) forms the first segment, index $2-4$ forms the second segment, and so on.

Each edge is associated with a weight, and we want to find the edge that has the maximum weight within each segment. In Fig. 5, the strongestDst array shows the result of segment scan on the graph from Fig. 3.

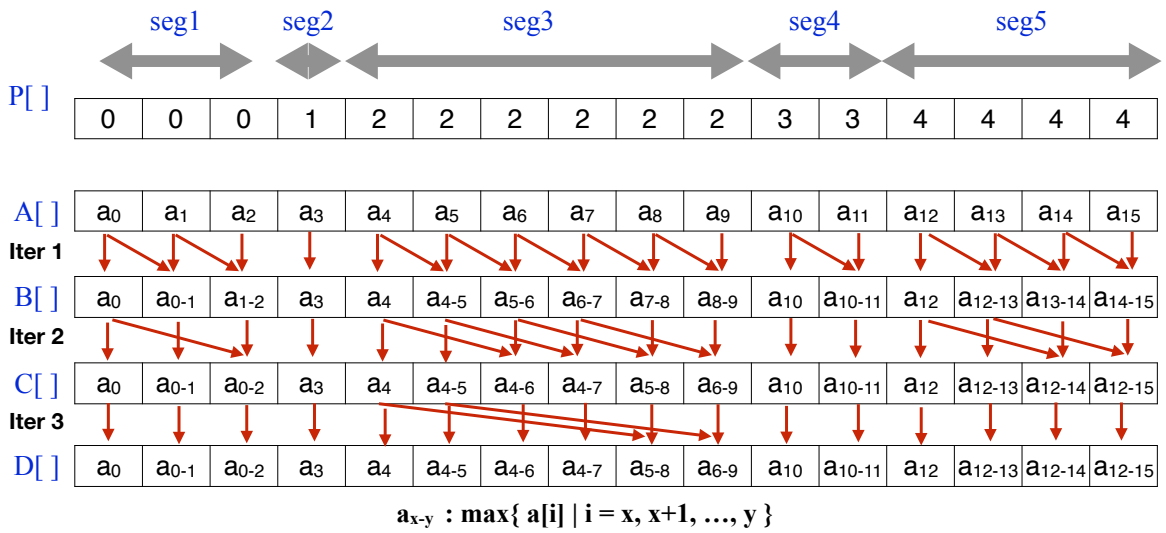


Figure 4: Example of a segment scan

A parallel segment scan can be performed with logarithmic time complexity. Fig. 4 illustrates the the parallel segment scan algorithm. In Fig. 4, array P is used to mark the segment. In Fig. 4, array A contains the input data we operate on. Array B contains the output data for the first iteration. Arrays C and D contain the output data for the second and third iterations respectively.

During each iteration of parallel segment-scan, each (independent) task picks two elements with a stride s, checks if these two elements are in the same segment; if so, it compares the two elements, store the maximum one in the appropriate location in the output array. A parallel segment-scan may involve multiple iterations, the first iteration uses stride $s = 1$ and the stride s doubles at every iteration.

In the example of Fig. 4, one thread is in charge of updating one data element in the output array at once. In the first iteration, if a thread is in charge of output the i-th data element B[i], it will do the following: if P[i-1] = P[i], the thread compares A[i] with A[i-1], write to B[i] such that B[i] = max(A[i], A[i-1]); if P[i] ! = P[i-1], it directly copies A[i] to B[i]. In the second iteration, a thread operates on elements with a stride of 2, if P[i] = P[i-2], a thread compares A[i] with A[i-2], and write to B[i] such that B[i] = max(A[i], A[i-2]); if P[i] ! = $P[i-2]$, B[i] = A[i]. It keeps doubling the stride until the stride s is large enough and any two elements that are compared are in two distinctive segments. The example in Fig. 4

demonstrates the idea, which takes three iterations in total. Be aware that we only describe the idea of the segment-scan algorithm, but we did not talk about how to handle boundary cases such that an element is located as the first element in an array. You have to handle the boundary cases in your code.

The kernel function you need to implement only performs one iteration of segment-scan. It is given a stride (which we refer to as 'distance' in the code) as an argument so it knows which two elements to compare. To avoid race conditions, the input and output will be separate arrays for each given iteration. We use two arrays in total to accomplish it, as we can swap the pointers of these two at the end of each iteration. You do not have to worry about allocating the input or output arrays.

You can let a thread be in charge of updating one element of the output array at one time. A thread might need to handle more than one data elements, thus you might need a loop within a kernel. See Section 2.4 for more details.

Recall that the maximum-weight neighbor is the one with the greatest weight. In the event of a tie, the neighbor with smaller vertex ID should be treated as stronger. E.g. if source node 1 has two possible destinations, nodes 2 and 3, that both have weight 4, then we say node 2 is the stronger neighbor since $2 < 3$. This is accomplished by sorting edge lists with respect to source node first; Within a segment that has the same source node, the edges will be sorted by the destination node, in ascending order.

Note that our implementation requires operating output two data arrays in conjunction: an array where the ID of the maximum-weight neighbor will be stored, and an array the maximum weight will be stored, both as the last element of each segment. Your implementation needs to account for that.

The first function you need to implement, to perform this segment scan, is described below:

```
__global__ void strongestNeighborScan_gpu(int * src, int * oldDst, int * newDst,
int * oldWeight, int * newWeight, int * madeChanges, int distance, int numEdges)
```

In this kernel function the arrays src, oldDst, and oldWeight are the input arrays. And newDst and newWeight are the output arrays. Each of these arrays contain numEdges elements. Additionally, we include an output called madeChanges, which is a pointer to an integer variable. It is used for detecting the termination condition, i.e. when to stop the segment-scan iterations. A thread should set (*madeChanges) to 1 if and only if oldDst[i] ! = newDst[i] for any of the data items i that it handles. The program given to you will use this variable to determine if the segment-scan process should terminate. If this variable is not set properly, it might result in an infinite loop.

For example, suppose we have src=[0,0,1,2,2,2,3,3], oldDst=[2,3,3,0,1,3,0,2], and oldWeight=[3,1,1,3,1,2,1,2], and are using a stride of distance=1. Then the result of the kernel will be that newDst=[2,2,3,0,0,3,0,2], newWeight=[3,3,1,3,3,2,1,2], and *madeChanges=1.

2.3 Pack Maximum-weight Neighbors

After the previous step, the maximum-weight neighbors are placed as the last element within each segment in the output array(s). Most elements in these two arrays do not contain useful information. We want to collate the useful parts of this array, in order to produce an output array that has the same number of elements as the number of segments, and only store the maximum-weight neighbor information.

This kernel function you need to implement is defined as follows:

```
__global__ void collateSegments_gpu(int * src, int * scanResult, int * output, int numEdges)
```

This function should identify the last element in every segment and pack them into the output array. The input src and scanResult, respectively represent the source nodes of the edges list, and the segment-scan result (the index array part). Each thread is as if in charge of one edge in the edge list at one time. A thread needs to tell whether the data element it is handling is the last one in each segment. To do that, let's assume a thread is in charge of the i-th data element, it compares src[i] with src[i+1], if they are not equal, then the i-th data element is the last one in its own segment. Once a thread identifies that it is handling the last element of a segment, assuming the data item is x, it then stores scanResult[x] into output[src[x]].

For example, suppose we have source array src=[0,0,1,2,2,2,3,3] and the strongest destinations scanResult=[2,3,0,0,1,1,0,2]. Then the result of collateSegments would be output=[3,0,1,2].

2.4 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 both kernels you need to implement, the number of tasks should be the same as the number of edges, as indicated by the numEdges argument in both functions.

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/run your code on *ilab* machines.

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

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 outside the gpu_required directories, except the gpu_extra_credit folder (if you want to work on extra-credit functionalities). Any modifications you make outside those two directories will not be considered when we grade your code.

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 **gsn**, which stands for 'get strongest neighbors'.
- **make submit** should generate a tar file containing your current code.
- **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** first, to delete the non-debug version of the code.

Execution: The gsn program finds the maximum-weight neighbor of all nodes in a given graph. It takes the following arguments: input file, output file, and the number of GPU threads to use for each kernel function. You can run the executable without arguments to see usage information. Please **DO NOT** change the usage of the program.

Program Output: The program will output the maximum weight neighbors to the output file you specify in the arguments, 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 for strongest neighbor has filename "strongX.txt".

For example, if you fully and correctly implement the required part of this project, then the command **./gsn testcases/input9.mtx out.txt 1024** would produce a file out.txt that's identical to testcases/strong9.txt. (You can also use a different number of threads than 1024, and should get the same output.)

You can use the **diff** command to compare two text files. For example, **diff out.txt testcases/strong9.txt** will either display a list of differences between those two files, or display nothing if they are identical.

We have told the Makefile to set the test case files to read-only when you compile your program, to ensure you don't accidentally overwrite a reference solution.

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 fourth 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 your code with the command "./gsn input.mtx output.txt 2048 **3**" so that it will run on device #3. However, be aware that most of the computers in ilab that you can use has 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. You need to submit **submit_me.tar**. If you change your code after generating this tar file, don't forget to regenerate (and resubmit) the tar.

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

6 Extra-credit Functionalities

The extra credit component completes the remaining kernel functions for the handshaking algorithm. In Fig. 5 we show an example of the data produced during the first iteration of the handshaking algorithm – which involves both required and extra-credit functions – based on the same graph as in our edge list example. Note that the notations used in Fig. 5 will be used consistently in the following subsection. Those notation include the keep_edges, new_indices, strongestDst, strongestWeight, strongNeighbor, and matches arrays. Please refer to Fig. 5 for specific examples of each of the functions described below.

Figure 5: An Iteration of the Handshaking Algorithm

There are four additional GPU kernel functions that need to be implemented for the matching algorithm. Their arguments (input and output) are described in the file *gpu-Headers.cuh*, along with the required functions. Your implementations should go inside the

gpu_extra_credit directory, which currently contains a separate file for each function.

6.1 Update Matches

With the strongNeighbor results produced by the required kernels, 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, and they should not be changed. An unmatched node is indicated by a matches value of -1.

6.2 Edges Filtering

To filter out the edges we no longer need, we must first decide which edges to keep. 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);
```

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.

6.3 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. Check Lecture 20, page 27 for an example of *inclusive prefix sum*.

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.

6.4 Filter Edges

At this point we know where the edges will go in the filtered edge list, so we can 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 should be the case that newSrc[edgeMap[x]] = oldSrc[x]. The dest and weight arrays should be handled similarly.

In order to determine whether to keep an edge, you must check the adjacent edge in the edgeMap. An edge x is kept if and only if edgeMap[x+1] != edgeMap[x], since that indicates that edge x requires its own index.

7 Compiling & Running

When you run make it also generates a second executable, **match**. This executable performs the complete matching process, using all of the required and extra credit kernel functions, and outputs the matching to the specified file. Its usage is otherwise the same as the gsn executable mentioned earlier.

For each testcase with input inputX.mtx, we have provided the correct match results in file matchX.txt.

8 Questions

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