green4 green5

1 Instructions

1. The parallel algorithms described in the book are fairly standard but probably they will not work if directly implemented in CUDA. The point of the project is precisely for you to experience the difficulty of using/modifying/adapting ideas that work in theory such that they work in practice. You might have to deviate substantially from the algorithms proposed in the book.

2. Write your report in your file and do not mess others' sections. My file is "bento.tex". Each one of you has its own Latex source file to edit. You should create your own folder for figures. Name the folder according to your name.

3. Identify all the files you create with your name. For example, by figures are name "bentoSomething.pdf". Also, make sure that the Latex labels you use are unique. So if you label one figure as "\label{fig1}", no one else should use this same label. The best thing is to append two of your initials to all of your labels.

4. Use only vector format figures in your report, i.e. figures with infinite resolution. Put all the figures inside the "figuresYourname" folder. I made my figures in PowerPoint, the selected them, right-click, chose "Save as Picture", and then saved them as Pdf.

5. Regarding quality of your report, your report should be much better and more detailed than mine. Mine is a report for a simple problem with a simple solution. I wrote it just to set the "fail line" and to give you some illustration of how to use Latex. Your problem will most likely be more complicated. Or maybe you will solve many small problems. In any case, the total work will be more than the work it took for to me write my section.

   Please pay attention to the way I wrote my sample report. I explain the general idea first. Then I give increasingly more details. I try to break explanations and code in small parts. I put pictures explaining what I have going on in my mind. I leave details to the end.

6. In your project you should explore as many approaches as possible and report in tables and plots their runtime.

7. Please include the most important portions of your code in the report. Specially the kernels. Explain them step by step.

8. Be aware of plagiarisms. If you use or adapt someone else's code, you should cite them. You will not be penalized if you do this. Although I cannot give you points for creativity either.

9. Start early! Come to lectures, which are mandatory, and ask questions. You have 2 months to do this. It might seem like a long time, but it is not. You probably also need to spend some time outside class in working on your project so that you make significant progress.

10. Have fun and learn a bunch!

2 By José Bento, on finding the distances of nodes to roots

Given a forest as input, i.e., a list of trees, we want to output a vector with the distance from every node in the tree to the root of the corresponding tree. See Figure 1 for a representation.

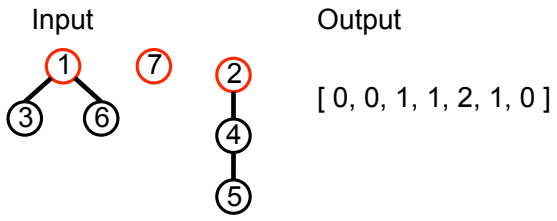


Figure 1: The red circles represent the root of each tree. The output is an array where the ith entry is the distance of the ith node to the root of its corresponding tree.

The technique we will use is called pointer-jumping and to illustrate our use of it, we consider a very closely related but different problem first.

2.1 Root finding problem

This problem is the following. Given a forest as input, we want to output an array where the ith element is the root of the tree that the ith element belongs to. Hence, for the input in Figure 1, the output should be $[1, 2, 1, 2, 2, 1, 7]$.

The pointer jumping idea consists of having each node in the forest sequentially update its ancestral link so that it points to the link of its parent's parent. To avoid having to treat root nodes differently, we assume that the root nodes ancestral link points to themselves, they are their own parent.

This is illustrated in Figure 2, which only shows part of a forest.

If a node sequentially makes such an update, it will eventually point to the root. To determine the number of steps required to finish this algorithm, it is convenient to assume that the all nodes make their updates in synchronism, i.e., the algorithm proceeds in rounds, and in each round all the nodes make one update based on the links in the previous round. If this is the case, and if D_i is the hop-distance from the node that the ancestral link of node i points to,

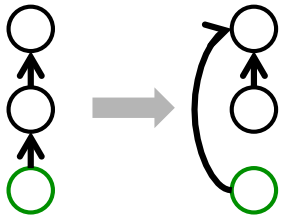


Figure 2: The green node updates its ancestral link to point to its parent's parent. The figure does not illustrate what the other nodes are doing. They are doing similar things.

then, in each round, D_i is halved. If the forest has N nodes, then the maximum depth of any tree is N so the number of steps for each node to point to the root is

$$\mathcal{O}(\log N). \tag{1}$$

Assuming all threads execute in parallel, the run time of the parallel algorithm is also (1).

2.1.1 Code for finding the root of each node in parallel

Since the run time is expected to be very short, it is convenient to work with large inputs to measure the run time accurately. Hence, we define `typedef long int number;` . Let us assume that we represent our forest by an array `number* forest`, where `forest[i]` is the parent of node `i` and, if i is a root, then `forest[i]` is equal to `i`. Sometimes we call the values of `forest` pointers, because they "point" to a another node, although they are not pointers in the strict sense of the word in C. For a single block, inside of which we can synchronize all the threads, this idea can be implemented by the following Cuda Kernel.

```
__global__ void findrootofforests(number * forest, number n){
    long int ix =  threadIdx.x + blockIdx.x*blockDim.x;
    if (ix < n){ // ignore useless threads
        while(forest[ix] != forest[forest[ix]]){ //stop condition
            forest[ix] = forest[forest[ix]]; //pointer jumping
            __syncthreads();//wait for other threads in the block
        }
    }
}
```

Note that we can check the stop condition for each thread by checking if a nodes next pointer will point to the same node it is currently pointing, i.e., look for a fixed-point.

A priori, we could think that to go beyond a single block, we would have to make multiple kernel calls, the traditional way to synchronize threads across a whole grid. However, it is **very important** to note that, regardless of whether

other threads have updated their pointer or not, when thread `ix` runs the command `forest[ix] = forest[forest[ix]];`, its associated pointer either remains equally far from the target root, in case thread `forest[ix]` has not updated its value, or its distance to the target root improves. The target root being the root of the tree that contains node `ix`.

Having this in mind, we conclude that exactly the same code will work if we launch a kernel with multiple blocks and do not synchronize any set of threads.

```
__global__ void findrootofforests(number * forest, number n){
    long int ix =  threadIdx.x + blockIdx.x*blockDim.x;
    if (ix < n){ // ignore useless threads
        while(forest[ix] != forest[forest[ix]]){ //stop condition
            forest[ix] = forest[forest[ix]]; //pointer jumping
        }
    }
}
```

For an input size of `n`, we launch our kernel as

```
findrootofforests<<<(n+1023)/1024,1024>>>(forest, n);
```

, where `forest` is our input defined in global memory.

2.1.2 Code for finding the root of each node serially

Our competitor serial code is very simple. Its input is the same as above, plus an adjacency list representation of the forest. This adjacency list is an array of arrays `number ** tree` such that, for each node `i`, `tree[i][0]` is the number of children of node `i`, and `tree[i][1]`, `tree[i][2]`,... are the Id's of the children of node `i`.

Assuming that we know that node `i` is a root of some tree, we can use a simple Depth First Search (DFS) to set the output array to `i` in all the entries corresponding to nodes that are a descendant of node `i`. We do so using the following code.

```
void DFS(number ** tree, number absroot, number temproot, number *
    output){
    output[temproot] = absroot;//mark node descendant of  absroot
    number numneigh = tree[temproot][0]; //get number of children
    for (number i = 1; i <= numneigh; i++){//go over all children
        DFS(tree, absroot, tree[temproot][i], output); //recurse
    }
}
```

Using this code, the overall algorithm is as follows. First we go over all the entries in `forest`. If `forest[i] == i`, then we know that `i` is a root. If `i` is a root, we call the previous function `DFS` to set all the output elements that are a descent of `i` equal to `i`. Note that the `DFS` function changes the entries of `forest` each time it is called. However, the nodes such that `forest[i] == i` are not changed, which explains we can operate directly on `forest`.

```
for (number i = 0; i < n; i++){//go over all the nodes
        if (forest[i] == i){ //check if the node is a root node
            DFS(tree, i, i, forest); //mark all descents of i as i
        }
    }
```

Since DFS run time is linear and since we only scan each tree once, the over all run time of the serial algorithm is

$$\mathcal{O}(N)$$

, where N is the number of nodes in the forest.

We note that we can create the adjacency list `tree` from `forest` in $\mathcal{O}(N)$ steps. For each entry `i` in `forest`, we append to the list of children of node `forest[i]` the element `i` and increment the number of children of node `forest[i]`. The following code does that.

```
number ** get_adjacency_from_list_of_parents(number* forest,  number n
    ){
    long int** tree;
    tree = (long int **) malloc(n*sizeof(long int *));
    // each children's list starts empty
    for (number i = 0; i < n; i++){
        tree[i] = (number *) malloc(1*sizeof(number));//allocate space
    for the list of children of each node
        tree[i][0] = 0;
    }

    // go over all the nodes
    for (number i = 0; i < n; i++){
        if (forest[i] != i){//check if I am someone's children
          tree[forest[i]] = (long int *) realloc( tree[forest[i]], (
    tree[forest[i]][0] + 2)*sizeof(long int) );//create space for new
    children
            tree[forest[i]][0] = tree[forest[i]][0] + 1;
            tree[forest[i]][ tree[forest[i]][0] ] = i;//add children
        }
    }

    return tree;
}
```

2.2 Run time for finding root of nodes

On a TeslaK40, the run time to find the root of each node for different input sizes is summarized in Table 8. We generate random forest for the input.

2.3 Finding the distance from every node to the root

We are now ready to address the original problem. The main idea is still the same, use the pointer jumping technique. However, now we want each node not

$\log_2$ **of input size**	27	26	25	24
Build adjacency list	7.662429	3.120494	1.332283	0.739289
Find roots serial	6.741057	3.104944	0.935574	0.555759
Copy input to GPU	0.176305	0.060571	0.026734	0.021566
Find roots kernel	0.006143	0.001824	0.000759	0.000463
Copy output to CPU	0.181282	0.077533	0.035911	0.014048
Speedup kernel vs serial	1097	1702	1232	1200

Table 1: Run time for finding root. All times are in seconds.

only to update its pointer to point to its parent's parent, but each node should also update a distance from itself to its corresponding root.

The distances of each node to its corresponding root are updated as follows. We keep track of the distance of each node to the root in an array of distances. Let us call it `dist` for now. At each round of the algorithm, and for each node `i`, we want to keep the following property.

Property X: the value of `dist[i]` should be the hop-distance between `i` and the ancestor node it points to.

If we are able to keep this property throughout the algorithm, then, at the end, when all the nodes point to their corresponding roots, we know that `dist[i]` will have the distance from `i` to its corresponding root. To keep this property, each time node `i` pointer is updated, its associated distance `dist[i]` should be incremented by the distance from its parent to its parent's parent, i.e., if its parent is `j` then, `dist[i]` should be incremented by `dist[forest[j]`. See Figure 3 for an illustration.

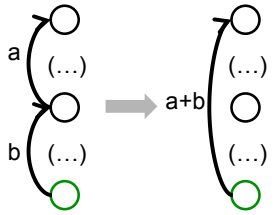


Figure 3: The green node updates its ancestral link to point to its parent's parent. It also updates its corresponding entry in the dist array from a to a+b, where b is the distance from its parent to its parent's parent.

For this idea to work, we initialize `dist` such that `dist[i]` is 1 if `i` is not a root, and 0 if it is a root.

2.3.1 Code to compute the distance of every node to the root

In an ideal world, we would make two kernel calls. The first kernel initializes the array `dist` in parallel.

```
__global__ void setzerodistances(number * forest, number n, number *
    distances){
    long int ix = threadIdx.x + blockIdx.x*blockDim.x;
    if (ix < n){
        if (forest[ix] != ix){//check if root
            distances[ix] = 1;
        }else{
            distances[ix] = 0;
        }
    }
}
```

The second kernel is very similar to that of Section 2.1.

```
__global__ void finddistancetoroot(number * forest,number * distances,
    number n){
    long int ix =  threadIdx.x + blockIdx.x*blockDim.x;
    if (ix < n){ // ignore useless threads
        while(forest[ix] != forest[forest[ix]]){ //stop condition
                distances[ix] += distances[forest[ix]]; //update the
    distances
            forest[ix] = forest[forest[ix]]; //pointer jumping
            __syncthreads();//wait for other threads
        }
    }
}
```

It is important to note that, unlike in the previous simpler problem of Section 2.1, we do need to make sure that all threads proceed in synchronism. Two problems can occur when we do not proceed in synchronism.

Problem 1: If thread `i` runs the inside of the `while` loop twice times before any other thread does something, then, although the `forest` array will not have misleading values, the `dist` array will have added to entry `i` twice the same value and will break Property X.

Problem 2: Similarly, if, while one thread has executed the line `distances[ix] += distances[forest[ix]];` and has not executed the line `forest[ix] = forest[forest[ix]];`, another thread updates both of its values `dist` and `forest`, then Property X will break.

Unfortunately, classically at least, CUDA does not allow the synchronization of threads from different blocks, so the above code only works for very small inputs, and achieves very small speedups. Therefore, a practical final solution must recourse to multiple kernel calls. Each kernel call makes a single update to each entry of `forest` and to each entry of `dist`, this is necessary to avoid Problem 1. Also, each kernel must operate on an old copy of `forest` and `dist` and output a new copy of `dist` and `forest`. This is necessary to avoid Problem 2.

```
__global__ void finddistancetorootIO(number * in_forest,number *
    out_forest, number n, number * in_dist,number * out_dist){

    number ix =  threadIdx.x + blockIdx.x*blockDim.x;

```

```
    if (ix < n){
        number f = in_forest[ix];
        number ff = in_forest[f];
        int b = (f != ff);
        out_dist[ix] = in_dist[ix]  + b*in_dist[f];
        out_forest[ix] = ff;
    }
}
```

Since each kernel only makes one update, the stop condition of the algorithm needs to be determined outside the GPU. In particular, we control how often we launch this kernels as follows.

```
    number rep = 1;
    while (rep <= n/2){
      finddistancetorootIO<<<(n+1023)/1024,1024>>>(forest,forest_2, n,
    dist,dist_2);
      finddistancetorootIO<<<(n+1023)/1024,1024>>>(forest_2,forest, n,
    dist_2,dist);
        rep = 2*rep;
    }
```

In the code above, we have an old and a new copy of the distance array and an old and a new copy of the forest array. From the argument that lead to (1), we know that we do not need to execute the above loop more than $\log_2(N/2)$ times, where N is the number of nodes in the forest and the $N/2$ comes from the fact that we are calling the main kernel twice inside the `while` loop. Note also that, instead of computing the log, we multiply `rep` by two until it reaches the value of $N/2$.

2.3.2 Serial code to compute the distance from nodes to the root

The competing serial code to compute the distances is very similar to the serial code to compute the root of each node. We also proceed in two steps. The only different is that the DFS, as it scans the tree, keeps track of the distance between each node and the root.

```
void DFS_with_dist(number ** tree, number start_dist, number temproot,
    number * dist){
    dist[temproot] = start_dist;
    number numneigh = tree[temproot][0];
    for (number i = 1; i <= numneigh; i++){
        DFS_with_dist(tree, start_dist+1, tree[temproot][i], dist);
    }
}
```

The whole serial code also runs in $\mathcal{O}(N)$ time.

2.3.3 Run time for finding the distance from each node to the root

The final run times are summarized in Table 2.

$\log_2$ **of input size**	22	23	24	25
Build adjacency list	0.670526	1.449542	3.141498	6.579136
Find roots serial	0.421897	1.336503	2.547802	5.136518
Find distances serial	0.441468	1.070775	2.46223	7.533981
Copy input to GPU	0.021523	0.027352	0.055171	0.13419
Find roots kernel	0.000461	0.000765	0.001836	0.006243
Copy output to CPU	0.025713	0.068875	0.071857	0.192935
Copy input to GPU	0.032275	0.044772	0.087318	0.264766
Find distances kernel	0.025739	0.053441	0.115212	0.25224
Copy output to CPU	0.032334	0.054391	0.121949	0.256908
Speedup for finding root	915	1747	1387	822
speedup for finding distances	17	20	21	29

Table 2: Run time of both the root find and distance find problems. All times are in seconds.

3 By Student X, on Sudoku puzzle

This project is about Solving Sudoku using a GPU.

$\log_2$ **of input size**	27	26	25	24
Build adjacency list	7.662429	3.120494	1.332283	0.739289
Find roots serial	6.741057	3.104944	0.935574	0.555759
Copy input to GPU	0.176305	0.060571	0.026734	0.021566
Find roots kernel	0.006143	0.001824	0.000759	0.000463
Copy output to CPU	0.181282	0.077533	0.035911	0.014048
Speedup kernel vs serial	1097	1702	1232	1200

Table 3: Run time for finding root. All times are in seconds.

4 By Michael Duan, on List Ranking Algorithm

4.1 Introduction: List Ranking Problems

Generally, in List Ranking problems, we need to find the distances of each node to the end of the linked list. In the following sections, I am going to introduce one naive way, and a corresponding way to solve it in parallel. Furthermore, I will introduce ways to improve the original algorithms, namely independent set. All code are in my github:

`https://github.com/SpicyGoose/3394ParallelComputing_ListRanking`

4.2 Basic: Pointer Jumping Algorithm

In general, we rely on the pointer jumping algorithm to figure out the distance between certain node to the end of the list.

4.2.1 The assistant lists: Successor and Result

In order to apply pointer jumping algorithm, firstly we have to initialize two extra lists to help us maintaining the structure of the single linked list and recording the final result. Namely, we use one list, Successor, and another list, Result, to do the job.

Successor, as its name suggests, is a list which contains current node's successor. For instance, imagine there is a successor list: $2- > 5- > 4- > 0- > 6- > 3$. Then, the first 2 in the list suggests that 2 is node 1's successor; 5 is node 2's success and so on. In another word, if we translate the successor list to a normal linked list, then we will have: $1- > 2- > 5- > 6- > 3- > 4$. Since node 4's successor is zero, then we can tell that node 4 is the end of the linked list.

Meanwhile, we need another helper list to record the results, which are the distances from one certain node to the end of the linked list. For instance, in our previous example, the result list will be: $5- > 4- > 1- > 0- > 3- > 2$, because node 1 is 5 nodes away from node 4, and node 2 is 4 nodes from node 4 and so on. Initially, the result list is slightly different. We initiate our result list by assigning 0 to the end node, and 1 to the rest. So, the initial result for the example above should be like: $1- > 1- > 1- > 0- > 1- > 1$

4.2.2 Basics of Pointer Jumping

In short, the pseudo code for pointer jumping algorithm is straightforward. Here is the original pseudo code from Jaja's work

Algorithm 4.2.2:

```
Input: 1. initial result list (Using R to represent it);
       2. Successor list (Using S to represent it);
Output: final updated result, R;
Begin:
    for 1 <= i <= n, pardo
        set Q(i) = S(i);
        while (Q(i) != 0 and Q(Q(i)) != 0):
            Set R(i) = R(i) + R(Q(I));
            Set Q(i) = Q(Q(i));
End
```

Implementation: In Linear In linear is straight forward.

```
int *updateRes(int *r, int *s, int len){
  //              result, successor, lenth of array
  int *res = r;
  int *q = s;
  //can use q[i] to find the element
  for(int i =0; i< len; i++){
    while(q[i] != 0 && q[q[i]] != 0){
```

```
      res[i] = res[i] + res[q[i]];
      q[i] = q[q[i]];
    }
  }
  return res;
}
```

the input r is the initial result list, which is a bunch of "1", input s is the initial list of successor, and input len is the length of the list. I implemented initial functions to set the initial values for r and s. To use updateRes, we just put in the initial r and s list.

4.2.3 Implementation: In Parallel

Similar to the functions in subsection 4.2.2, we just transform our algorithm in to cuda language.

```
__global__ void updateResAll(int *r, int *s, int *q, int n){
  int i = threadIdx.x;
  if (i < n){
    q[i] = s[i];
    __syncthreads();

    //try to update all
    while(q[i] != 0 && q[q[i]] != 0){
      r[i] = r[i] + r[q[i]];
      q[i] = q[q[i]];
      __syncthreads();
    }
  }
}
```

This kernel can deal with the data up to the volume of 1024, because it only calls one block to complete the work. Due to cuda language, we can easily synchronize our threads in one block, but cuda doesn't support sychronization between different blocks.Thus, we have to figure a way to solve this problem.

4.2.4 Implementation In Parallel, Synchronization between Blocks

Firstly, here is the kernel:

```
__global__ void updateOnceBetweenBlocks(int *r1, int *r2, int *s1, int
    *s2, int n){
  //update only one round. Use if to control it;
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i <= n){
    int next = s1[i];
    int nextNext = s1[next];
    int check = (next != nextNext);
    r2[i] = r1[i] + check*r1[next];
    s2[i] = nextNext;

    // if(s1[i] != 0 && s1[s1[i]] != 0){
    //   res2[i] = res1[i] + res1[s1[i]];
```

```
    //   s2[i] = s1[s1[i]];
    // }
  }
}
```

this kernel can make sure that blocks and blocks are synchronized by accessing global memory each time it finished doing work. The down side is that it only achieves one updates in a cycle. Thus, the main function will decide how many cycles are going to happen. Generally, the point jumping algorithm can finish in O(Log n) time, which means ideally, the main function only need to call updateOnceBetweenBlocks logN times. Therefore, I come up with the following main function:

```
  int counter = 1;
  int div = 2;
  while(counter <= arrayLen/2){
    //1->2, 2->1
     //<<<block,thread>>>
    updateOnceBetweenBlocks<<<div,arrayLen/div>>>(devRes1, devRes2,
    devArr1,devArr2,arrayLen);
    updateOnceBetweenBlocks<<<div,arrayLen/div>>>(devRes2, devRes1,
    devArr2,devArr1,arrayLen);
     counter = counter * 2;
  }
```

In my main function, the while loop decides how many updates are going to occur. I set the boundary as half of the input length, just to make sure that the algorithm won't miss any nodes. However, theoretically, Log(N) is enough.

4.3 Optimization: Independent Sets

This section will introduce one way to optimize our previous algorithm, which is "Independent Sets". The ultimate goal of independent set is temporarily shrinking our list to a smaller size, and retrieve it later. Here is the original pseudo code from the book:

```
input: A linked list of nodes, and corresponding list S(i) for each node's successor
output: distance R(i) of node i from the end of the list

for 1<=i<=n pardo:
    set Q(i) = S(i)
    while (Q(i) != 0 and Q(Q(i)) != 0):
        R(i) = R(i) + R(Q(i))
        Q(i) = Q(Q(i))
```

The general idea for this optimization has following steps:

1. Prepare several arrays to help us record the necessary data: First array, r, standing for result, stores the data of distances.
Second array, s, standing for successor, stores the data for successors.
Third array, p, standing for predecessor, stores the data for predecessor.

Last array, u, is an empty 2-D array. We are going to store corresponding information about independent sets in u.
2. Use N and U to keep necessary data, and we are going to use these data to retrieve the final result.

However, the original pseudo code does not have a clear way to preserve the data we need in later step. It uses a very confusing array to maintain the data and I spent a huge amount of time to realize his methods and still failed. Thus, I made some adjustments on the original pesudo code. Here is the kernel I design to extract the independent set from the original set. In step 2, instead using N list and U list, I only applied an Nx4 2-D array, where N is the length of our original input list. For each row, we need to store the node's value, predecessor, successor and distance.

```
__global__ void indepSet(int *r1, int *r2, int *s1, int *s2,int *p1,int
     *p2, int n, int *u){
//use 1 as prototype, and store data to 2
  int id = threadIdx.x + blockDim.x * blockIdx.x;
  if( id <= n){
    if(id % 2 == 0){
      r2[id-1] = r1[id-1]+ r1[id];
      p2[s2[id]] = p1[id];
      s2[p2[id]] = s1[id];
//// ---------store data--------
      u[4*id+0] = id;
      u[4*id+1] = p1[id];
      u[4*id+2] = s1[id];
      u[4*id+3] = r1[id];
      r2[id] = 0;
      s2[id] = 0;
      p2[id] = 0;
    }
  }
}
```

For each single list, I create two alike ones (for instance, r1 and r2) to prevent threads interfering each other. Also, in my demo, the input list is going to be a pre-design ascending list, from 1 to N. Therefore, I just sift nodes with even values out and keep the odd node in place.
In detail: firstly, I assign thread id for each thread, and mark those threads with even id. Then, I start updating the nodes by putting the data in secondary lists(r2,s2,p2). Imagine the original list is$1->2->3->4->5->6$. When we pick even nodes out, the new list would look like this: $1->3->5$. So we need to change the successors, predecessor, and distance of the odd nodes. Then, we update the u list. For each deleted even nodes, we load its node's value, predecessor, successor and distance in to the u list. One important thing here is that I did not find a feasible way to transfer 2-D list between CPU and GPU, so I transfer the 2-D list into a 1-D list. After we settle it down, we mark even nodes' successors and predecessors to 0, meaning that we don't need them anymore, for now.

Let's make a simple example here: we delete 2 from the list, so in u list, $< 2, 1, 3, 1 >$, standing for node 2, predecessor is 1, successor is 3 and distance is 1 (by default).
What happens to the odd nodes? Now they have changed successor, predecessor and distances. Successor and predecessor are easy parts. For distance, I add the odd node's distance value to the one before it. For instance:$1- > 2- > 3- > 4- > 5- > 6$. When we delete 2, I add 2's distance to 1; 4's distance to 3 and so on.
We use the kernel in this way:

```
  indepSet<<<div,arrayLen/div>>>
   (devRes1,devRes2,devSuc1,devSuc2,devPre1,devPre2,arrayLen,devStorage
    );
```

After GPU done its work, devRes2, devSuc2, devPre2 and devStorage have the data we need, and we use updateindepSet to put them back in the array and generate the final result.

```
   int counter = 1;

   while(counter <= arrayLen/2){
     //1->2, 2->1
      //<<<block,thread>>>
     updateOnceBetweenBlocks<<<div,arrayLen/div>>>(devRes2, devRes1,
    devSuc2,devSuc1,arrayLen);
     updateOnceBetweenBlocks<<<div,arrayLen/div>>>(devRes1, devRes2,
    devSuc1,devSuc2,arrayLen);
     counter = counter * 2;
   }
//-------------------------------------------------
//retrive original data

   updateIndepSet<<<div,arrayLen/div>>>(devRes2, devSuc2, devPre2,
    arrayLen, devStorage);
```

4.3.1 Optimization: Optimal List Ranking

Finally, we can put everything together, and get the optimal list rnaking algorithm:
1. Generate successor and predecessor lists. I simply used an ascending and an descending list to represent the list: $1- > 2- > 3- > 4- > ...- > N$.
2. Initialize the result list.
3. Use kernel "indepSet" to contract the list.
4. Use kernel "updateOnceBetweenBlocks" to update the distances
5. Using the data stored in U list to restore the predecessor list and true distance.
The sample code is on my github: https://github.com/SpicyGoose/3394ParallelComputingListRanking

4.4 Analyze: Running Time

I ran the program on CPU and GPU, respectively. 4.

List size(Node number)	1024	1024*256	1024*512	1024*1024
Linear algorithm	3.595	169406.8	Too long	Too long
Parallel Without Independent sets	0.20	1.134	1.75	3.03
Parallel With Independent sets	0.19	0.003	0.0039	0.003

Table 4: Run time in Ms, both parallel algorithm has 2 blocks

Block size	2	16	64	256
Parallel Without Independent sets	3.03	2.999648	3.004608	3.009216
Parallel With Independent sets	0.003	0.003936	0.003904	0.003904

Table 5: Run time in Ms, the input list size is fixed at 1024*1024

I tested the performance of my algorithm by assigning them different number of nodes and different number of blocks. In the first table, I compared the performance between linear algorithm, naive parallel algorithm and the parallel algorithm with independent sets. I only tested for four different numbers of input. According to the result, it is clear to see that parallel algorithms take shorter time, and our independent sets actually work as we expected, further decreasing the performance time.
In table two, I focused on testing the block number's influence on running time performance. However, even though I adjusted the block number numerous times, the running time does not change a lot at all.
In conclusion, I suggest that my algorithm works in general, and parallel algorithm can significantly boost the performance as we expected in the beginning.

4.5 Future Possible Improvement

1. The parallel algorithm has some wield output when I increased the node number to 1024*3. I need to fix it if I have time.
2. I only implemented and tested the algorithm based on pre-made input lists, namely a successor list containing ascending numbers ($1->2->...->n$) and another predecessor list containing descending numbers ($n->n-1->...->2->1$). As needed, I can write some helper functions to generate randomized successor and predecessor lists. After that, I may adjust my algorithm to fit the randomized input lists.
3. MY "Independent Set" function will only contract the input list once, which means it only decrease the list size by half. If I want to further decrease the list size, I need to manually adjust the variable to achieve so. I may improve this function by making it a recursive call, which can automatically decrease the input list size to O(1).

4.6 Update on May 12

1. I added a "Check" function to check the validity of the answer. That is, I can compare the results from linear way and parallel way, and see if the outputs

are the same.
2. I have fixed the second problem in section 4.5. I wrote two functions which can generate a pair of successor and predecessor (the function is in genRandom.h). Both linear and parallel methods give out the same correct answers.
3. I just figure out a proper way to make independent set is "Vertex Coloring". Without knowing this before, my optimization progress might be wrong. 4. Continue with 3, I wrote kernels to perform 2-coloring algorithm, just uploaded to the github.

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p312-339. Print.

[2] Parallel List Ranking Advanced Algorithms Data Structures, Prof. Dr. Th. Ottmann, Lecture Theme 17

[3] Wikipedia, Wikimedia Foundation, 12 Feb. 2018, en.wikipedia.org/wiki/Pointer$_j$*umping*.

[4] Stack Overflow, stackoverflow.com/questions/14093692/whats-the-difference-between-cuda-shared-and-global-memory.

[5] Stack Overflow, stackoverflow.com/questions/20518605/two-dimensional-array-error-subscripted-value-is-neither-array-nor-pointer-nor?rq=1.

5 By Anthony Chu on Pathfinding in a Graph/-Grid with A*

Code: https://github.com/anthonychu00/Parallel-A-

The problem we are trying to solve is, given a graph, a source node, and a query, how to find the best way from point A to point B. There exists multiple implementations of pathfinding algorithms, we will be looking at the A* algorithm.

5.1 Serial A* on a CPU

Like other pathfinding algorithms, A* works with weighted graphs (there is also an unweighted version when working with grids with obstacles, where each weight is basically 1). The traditional way A* works is with a queue of some sort. Given a source node and a destination node, A* creates a list of possible paths to take from the start node, and at each iteration chooses which of these paths to take. To do so we pop the node with the smallest distance from the source node, and then add its neighbors to the end of the queue, keeping track of their distance as well. What we described so far is just basic shortest path; what sets A* apart is that it uses additional heuristics to choose paths. There are actually multiple possible ways to implement the heuristics function, the most popular being measuring your current distance from the destination node at the current node you are on. We can define the whole algorithm as:

f(n) = g(n) + h(n)

In the heuristic, g(n) is defined as the amount of effort (the cost) used to get from the starting space in the grid to the current space (which should be stored somewhere in some portion of memory/object relating to the current space), and h(n) is the actual heuristic which calculates some path from the current space to the end space. We call the equation admissible if h(n) never overshoots the distance from the current space to the goal. Any number of methods can be used to implement this, such as Manhattan distance or the straight line distance.

The following are parts of the code for the serial algorithm:

```
1
2  struct direction {
3
4      int x;
5      int y;
6      double d;
7  };
8
9  int main() {
10     int i, j, k, l, b, found;
11     int p_len = 0;
12     int * path = NULL;
13     int c_len = 0;//closed
14     int * closed = NULL;
15     int o_len = 1;//open
16     int * open = (int*)calloc(o_len, sizeof(int));
17     double min, tempg;
18     int s;
19     int e;
20     int current;
21     int s_len = 0;
22     struct node * nodes = NULL;
23     int r_len = 0;
24     struct direction * directions = NULL;
25
26     for (i = 1; i < map_size_rows - 1; i++) {
27         for (j = 1; j < map_size_cols - 1; j++) {
28             if (!map[i][j]) {
29                 ++s_len;
30                 nodes = (struct node *)realloc(nodes, s_len * sizeof(
    struct node));
31                 int t = s_len - 1;
32                 nodes[t].col = j;
33                 nodes[t].row = i;
34                 nodes[t].from = -1;
35                 nodes[t].g = DBL_MAX;
36                 nodes[t].n_len = 0;
37                 nodes[t].n = NULL;
38                 ind[i][j] = t;
39             }
40         }
41     }
42     //constructs a map to relate nodes
43     //start node
44     s = 0;
45     //end node
```

```
46      e = s_len - 1;
47
48      for (i = 0; i < s_len; i++) {
49          nodes[i].h = sqrt(pow(nodes[e].row - nodes[i].row, 2) + pow(
        nodes[e].col - nodes[i].col, 2));
50      }
51
52      for (i = 1; i < map_size_rows - 1; i++) {
53          for (j = 1; j < map_size_cols - 1; j++) {
54              if (ind[i][j] >= 0) {
55                  for (k = i - 1; k <= i + 1; k++) {
56                      for (l = j - 1; l <= j + 1; l++) {
57                          if ((k == i) and (l == j)) {
58                              continue;
59                          }
60                          if (ind[k][l] >= 0) {
61                              ++r_len;
62                              directions = (struct direction *)realloc(
        directions, r_len * sizeof(struct direction));
63                              int t = r_len - 1;
64                              directions[t].x = ind[i][j];
65                              directions[t].y = ind[k][l];
66                              directions[t].d = sqrt(pow(nodes[directions
        [t].y].row - nodes[directions[t].x].row, 2) + pow(nodes[directions[
        t].y].col - nodes[directions[t].x].col, 2));
67                              ++nodes[directions[t].x].n_len;
68                              nodes[directions[t].x].n = (int*)realloc(
        nodes[directions[t].x].n, nodes[directions[t].x].n_len * sizeof(int
        ));
69                              nodes[directions[t].x].n[nodes[directions[t
        ].x].n_len - 1] = t;
70                          }
71                      }
72                  }
73              }
74          }
75      }
76  //traversal
77      open[0] = s;
78      nodes[s].g = 0;
79      nodes[s].f = nodes[s].g + nodes[s].h;
80      found = 0;
81
82      while (o_len and not found) {
83          min = DBL_MAX;
84
85          for (i = 0; i < o_len; i++) {
86              if (nodes[open[i]].f < min) {
87                  current = open[i];
88                  min = nodes[open[i]].f;
89              }
90          }
91          //looks through open nodes
92          if (current == e) {
93              found = 1;
94
95              ++p_len;
```

```
            path = (int*)realloc(path, p_len * sizeof(int));
            path[p_len - 1] = current;
            while (nodes[current].from >= 0) {
                current = nodes[current].from;
                ++p_len;
                path = (int*)realloc(path, p_len * sizeof(int));
                path[p_len - 1] = current;
            }
        }
        //path found!!

        for (i = 0; i < o_len; i++) {
            if (open[i] == current) {
                if (i not_eq (o_len - 1)) {
                    for (j = i; j < (o_len - 1); j++) {
                        open[j] = open[j + 1];
                    }
                }
                --o_len;
                open = (int*)realloc(open, o_len * sizeof(int));
                break;
            }
        }
    //else back to the traversal
        ++c_len;
        closed = (int*)realloc(closed, c_len * sizeof(int));
        closed[c_len - 1] = current;

        for (i = 0; i < nodes[current].n_len; i++) {
            b = 0;

            for (j = 0; j < c_len; j++) {
                if (directions[nodes[current].n[i]].y == closed[j]) {
                    b = 1;
                }
            }

            if (b) {
                continue;
            }

            tempg = nodes[current].g + directions[nodes[current].n[i]].
    d;

            b = 1;


    }

}
```

The runtime of the serial algorithm large depends on the quality of the heuristic. Worst case it can go up to $O(n^d)$, where n is the average number of new nodes you get from each traversal, and d the depth from the start node to end node.

5.2 The Algorithm in Parallel

Now, A* actually does lend itself to being parallelized more than some other algorithms, since in each iteration of the serial algorithm it is more or less doing the same thing (popping a node off a priority queue and reading the state). The key to parallelizing A* is making it so that multiple spaces in the grid and their states can be analyzed in sync. The solution to this is very intuitive. Since with one priority queue we can only read one state, we can read multiple states by having multiple priority queues. And by doing so, we expand the rate at which we look at each state. The parallel version of this algorithm is performing close to the exact algorithm of the serial version, except that a few parts are parallelized. The parts that are parallelized are:

- Popping a node from each priority queue
- Finding open nodes to put back into the priority queues (through searching adjacent spaces)
- computing heuristics and pushing these new nodes back to the priority queue

In particular, parallelizing the heuristics is simply having multiple threads calculating the heuristics of each state independently since they the states themselves are independent from one another. The following is code for the parallel algorithm:

```
1  time_t t;
2
3  int dim = 6;
4  int *grid = (int *) malloc(dim*dim*sizeof(int));
5        srand((unsigned) time(&t));
6  double p = 0.5;
7
8  for (int i = 0; i < dim*dim; i++){
9    grid[i] = p < (double)rand()/(double)(RAND_MAX);
10   //old_game[i]=1;
11 }
12 int * dev_grid;
13 cudaMalloc((void**)&dev_grid, dim*dim*sizeof(int));
14 cudaMemcpy(dev_grid,grid,dim*dim*sizeof(int),cudaMemcpyHostToDevice);
15
16 print_board(grid, dim);
17 //1s are spaces you can walk on
18 int startPoint = -1;
19 while (startPoint == -1){
20   int rando = rand()%(dim*dim);//dim*dim - 1 is highest number( 0 to
     dim*dim-1)
21   if (grid[rando] ==1)
22     startPoint = rando;
23 }
24
25 int endPoint = -1;
26 while (endPoint == -1){
27   int rando = rand()%(dim*dim);
28   if (grid[rando] ==1&&rando!=startPoint)
```

```
29        endPoint = rando;
30    }
31
32    printf("%d %d\n",startPoint,endPoint);
33
34    traverse<<<1,5>>>(dev_grid,startPoint,endPoint,dim,5);//last value is
          # of p-queues
35
36    cudaDeviceSynchronize();
37    return 0;
```

This is the main function that calls the kernel, traverse, which is where the bulk of the code is. We create the grid here as well, which is a bunch of 0s and 1s. 1s are traversable, and 0s are not. One thread is used per priority queue that we are using.

```
1
2      __shared__ int prevNode [36];//previous Node to print out route
3     __shared__ int lowestCost [36];//current lowest cost to get to that
        Node
4     __shared__ int array [30];//simulates priority queues, k marker at
        each step (dim*k)=(6*5)
5     __shared__ int heuristics[30]; //will store the heuristics
        corresponding to a
6     __shared__ int sizes [5];//stores current sizes of priority queues
7     __shared__ int expandedNodes[20];//k*4 for 4 directions
8  extracted=array[dim*threadIdx.x];//set the extracted Node to the front
                          of this P.Queue
9        shiftqueue(array,dim,threadIdx.x);
10       sizes[threadIdx.x]--;
11       //extraction and other stuff here
12
13
14       if(extracted == end||flag ==1)
15       {
16         flag = 1;
17         break;
18       }
19
20
21       //add to expanded nodes here
22       int top = extracted-dim;
23       int bottom = extracted+dim;
24       int left = extracted-1;
25       int right = extracted+1;
26       if(extracted%dim==0)
27         left = -1;
28       if(extracted%dim==dim-1)
29         right = -1;
30
31
32       //dumps adjacent squares into array
33       expandedNodes[threadIdx.x*4]=top;
34       expandedNodes[threadIdx.x*4+1]=bottom;
35       expandedNodes[threadIdx.x*4+2]=left;
36       expandedNodes[threadIdx.x*4+3]=right;
```

The "grid" we are dealing with is an n by n matrix numbered in row-major order. We represent it in the program via a 1d array, and use a dimension variable to keep track the size of the grid. This portion of the code is in a while loop that runs until every single queue is empty or the goal is found. Once it pops from each priority queue, it gets the locations for the tiles adjacent to it from the top, bottom, left, and right. On another note, we have one continuous shared memory array to simulate multiple priority queues. The size of each queue is static and depends on the size of the grid. We also have another shared memory array to correspond the heuristic. We also have an array to keep track of the "sizes" of each priority queue to make insertion easier.

```
for(int i =0; i<4;i++)


      //checks adjacent squares
      //deduplicates list
      for(int i =0;i<4;i++)
      {
        int curNum = expandedNodes[threadIdx.x*4+i];

            if(curNum<0 || curNum>dim*dim|| grid[curNum]==0||lowestCost
    [extracted]+1>lowestCost[curNum])
        {
          expandedNodes[threadIdx.x*4+i]=-1;
          continue;
        }//checks for invalid indices


        if(expandedNodes[threadIdx.x*4+i]!=-1)
        {

          //route is shorter, therefore update cost and previous node
          lowestCost[curNum]=lowestCost[extracted]+1;
          prevNode[curNum]=extracted;
        }



      }
```

This part checks the the adjacent spaces for invalid indices that are not within the grid or have a longer path needed than the current lowest cost. If we find a space that hasn't been explored yet or a more efficient way to get to that space, we update it's cost and set a new pointer to the previous space as the current space. If we do so, then we aim to place the node back in the priority queues.

```
for(int i=0;i<4;i++)
      {


        if(expandedNodes[i]!=-1)
```

```
        {
          int h = lowestCost[expandedNodes[i]]+
    heuristic(dim,expandedNodes[i],end);
          int check =0;

          while(check==0)
          {
            int targetLocation=findNextInsertionPoint(sizes, k);
            printf("T: %d\n", targetLocation);
            printf("Sizes: %d\n", sizes[targetLocation]);
        printf("Loc = %d\n", array[targetLocation*dim+sizes[
    targetLocation]]);
            if(atomicCAS(&array[targetLocation*dim+
              sizes[targetLocation]],               -1,expandedNodes[i
    ])==-1)
            {

              heuristics[targetLocation*dim+sizes[targetLocation]]= h;
              check = 1;
              sizes[targetLocation]++;
            }
          }

        }

      }
```

And finally this portion updates the priority queues in parallel, sort of. To make sure no other threads messed with the priority queues when another was working on it (they are in the same shared memory array), we use an atomic function to control access to it, which granted does serialize things a little bit here.

```
__device__ int findNextInsertionPoint(int * sizes, int k){

  int smallestQueue = -100;
  int curSmallest = 5000;
  for(int i = 0; i<k;i++)
  {
    if(sizes[i]<curSmallest)
    {
      smallestQueue = i;
      curSmallest = sizes[i];
    }
  }

  return smallestQueue;

}

__device__ __host__ int heuristic(int dim, int current, int end){//
    heurstics (Manhattan distance)

  int curX = current % dim;
  int curY = current / dim;
  int endX = end % dim;
```

```
  int endY = end / dim;

  return 3*((int)(fabsf(curX-endX) + fabsf(curY-endY)));

}

 __device__  void shiftqueue(int* array_sh,int dim,int k)//simulates
    popping
{

  for(int i=0;i<dim;i++)
  {
    if(i==dim-1)
      array_sh[dim*k+i] = -1;
    else
      array_sh[dim*k+i]= array_sh[dim*k+i+1];

  }
}

__device__  int checkIfQueueEmpty(int* array_sh, int dim, int k)
{
  int check = 0;//0 if all queues empty, 1 otherwise
  for(int i=0;i<k;i++)
  {
    if(array_sh[dim*i] != -1){
      check = 1;
      break;
    }
  }
  return check;
}
```

The following are various helper functions used in the code. Since we can only move in 4 directions, it makes sense to use Manhattan distance as our heuristic since it measures vertical and horizontal straight line distance. The first helper function finds the most optimal priority queue to place the expanded node in. It checks for the priority queue with the smallest amount of elements. The other 2 helpers are fairly self explanatory, checking for an empty queue or popping from the specified queue. At the end of this code, we have a chain of previous nodes starting from the goal node (if it was reached), and we can simply follow the shared memory array of previous nodes until we get to the starting node.

5.3 Runtime and Improvements

The runtime of the algorithm again, depends on the heuristic. The heuristic implemented here was fairly simple, but it is not very computationally intensive. The real slowdown from not having an optimized heuristic is that it might take more analyses of states to get to the goal. Extracting the nodes themselves is O(1), and finding the adjacent nodes is as well only requires 4 calculations. Pruning for duplicates also only took 4 iterations per thread, since you just ran

through multiple conditions for the adjacent squares. Computing the heuristic for each state was proportional to the number of threads and to our heuristic function.

The code provided is not perfect, it runs into an infinite loop 20 percent of the time, mostly when a node is presented many spaces to expand to. Therefore, it was difficult to time. It is likely an issue with the atomic function or the priority queues not properly being updated. So as a start, that is definitely something to improve on. An attempt was made to use texture memory for this problem, since the 2d grid lends itself well to the spatial memory utilized by textures, but that code would not run. Having dynamically resizing (external) shared memory would also be nice for the purpose of space, considering a lot of shared memory was used, but the inherent limitations of shared memory made attempts in doing that quite difficult.

References

[1] Zhou, Yichao, *Massively Parallel A* Search on a GPU*, AAAI

[2] "A* Search Algorithm" Wikipedia, The Free Encyclopedia, 25 Mar. 2019. Web. 12 May. 2019.

[3] Rios, Luis Henrique Oliveira "PNBA*: A Parallel Bidirectional Heuristic Search Algorithm" , Departamento de Ciencia da Computacio

6 By Edsel Rudy, on Range Minimum Query

Link to entire code for RMQ query: github.com/EdselR/RMQ

The Range-Minima problem or the Range Minimum Query is a useful and equivalent problem to the Lowest Common Ancestor. Solving the Range-Minima problem is faster than solving the LCA Problem in theory. I will aim to provide an optimal $\mathcal{O}(\log N)$ pre-processing algorithm which will allow one to solve queries in $\mathcal{O}(1)$ sequential time.

6.1 Introduction: The Range-Minima Problem

The problem states that given an input array B of size $n = 2^l$, we want to solve for queries in the following form: given two indices i and j, where $0 \leq i < j < n$, we can determine the minimum element in the sub-array $\{b_i, b_{i+1}, \ldots, b_j\}$ in $\mathcal{O}(1)$ sequential time. For example, given the input array $B = \{5, 10, 3, 4, 7, 1, 8, 2\}$ the minimum element from the query $i = 1$ and $j = 4$ is the minimum element from the sub-array $\{10, 3, 4, 7\}$ which results in the RMQ of 3.

6.2 Range-Minima Algorithm

Given B we can construct a complete binary tree, with the leaves of the tree as the elements from the input array. Some additional information is needed to answer the query in $\mathcal{O}(1)$ sequential time. Let u and w be, respectively, the left and right children of an arbitrary node v. Then, the sub-arrays B_u and B_w associated with u and w partition B_v into $B_u = \{b_r, \ldots, b_i, \ldots, b_p\}$ and $B_w = \{b_{p+1}, \ldots, b_j, \ldots, b_s\}$ where $r \leq i < j \leq s$. The minimum element p, then lies between $i \leq p \leq j$. The element we seek could be found from the following two elements: the minimum of the suffix $\{b_r, \ldots, b_i, \ldots, b_p\}$ of the left child B_u, and the minimum of the prefix $\{b_{p+1}, \ldots, b_j\}$ of B_w. Therefore, for each node v, we could store the suffix minima and the prefix minima of the sub-array corresponding to the leaves of v. See Figure 4 for clarity. The prefix minima of B are the elements of the array $(a_1, a_2, \ldots, a_n)$ such that $a_i = min\{b_1, \ldots, b_i\}$ for $1 \leq i \leq n$. The suffix minima of B are then the elements of the array $(a_n, a_{n-1}, \ldots, a_0)$ such that $a_i = min\{b_i, \ldots, b_n\}$ for $1 \leq i \leq n$.

The objective of the preprocessing algorithm is to construct a complete binary tree whose leaves are the elements of B, and each node v is associated with it two array, P and S, corresponding to the prefix and suffix minima of the subarray, respectively, defined by the leaves of the subtree rooted at v.

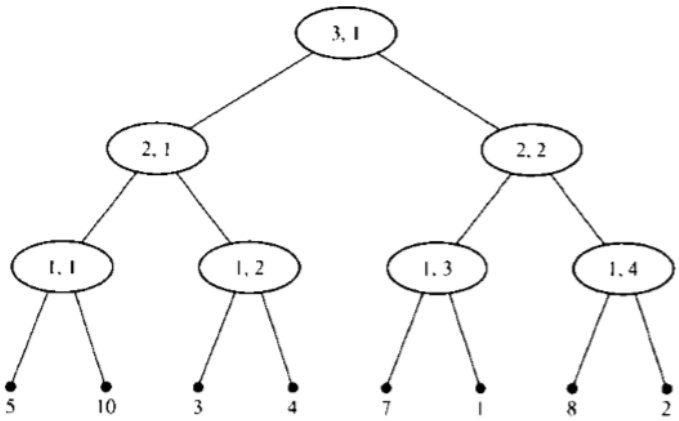


Figure 4: Complete Binary Tree formed from the input array B as the leaves. Node (h, j) holds the suffix and prefix minima of the subarray associated with the leaves rooted at (h, j)

6.2.1 Pseudocode and Example

The Range Minima preprocessing algorithm could be summarized as follows: loop through the leaves of the tree and set each node equal to the input array,

then loop through each level of the tree, for each node, merge the prefix/suffix minima for the left and right child.

Algorithm 1 Basic Range Minima Algorithm

```
for 1 ≤ i ≤ n do
    Set P(height, i) = B(i)
    Set S(height, i) = B(i)
end for
for height − 1 ≥ i ≥ 0 do
    for 1 ≤ j ≤ num of nodes for level i do
        Set P(i, j) = merge P(i + 1, 2j − 1) and P(i − 1, 2j)
        Set S(i, j) = merge S(i + 1, 2j − 1) and S(i − 1, 2j)
    end for
end for
```

Consider the previously mentioned array $B = (5, 10, 3, 4, 7, 1, 8, 2)$. The complete binary tree is shown in Figure 4. The P and S sub-arrays corresponding to nodes $(2, 1), (2, 2)$, and $(3, 1)$ are the following:

$P(2, 1) = (5, 5, 3, 3)$ $P(2, 2) = (7, 1, 1, 1)$

$S(2, 1) = (4, 3, 3, 3)$ $S(2, 2) = (2, 2, 1, 1)$

$P(3, 1) = (5, 5, 3, 3, 3, 1, 1, 1)$ $S(3, 1) = (2, 2, 1, 1, 1, 1, 1, 1)$

We create array $P(3, 1)$ by copying $P(2, 1)$ into the first half of $P(3, 1)$, then we copy $P(2, 2)$ into the second half. However, each element i of the the second half is replaced by the min of i and the last element of $P(2, 1)$ - which is $min(i, 3)$ in this example. The same principle is applied to find S arrays except $S(2, 2)$ is copied onto the first half of the array and $S(2, 1)$ is then copied onto the second half.

Let us take the example of a minimum query over the interval $[i, j]$, since this is a complete binary tree, the $LCA(i, j)$ could be found mathematically in constant time.

Let $v = LCA(i, j)$, $i = 1$, $j = 4$. Therefore, $B(1) = 10$ and $B(4) = 7$, and the $LCA(1, 4) = node(3, 1)$. The children are nodes $(2, 1)$ and $(2, 2)$, which corresponds to $S(2, 1) = (4, 3, 3, 3)$ and $P(2, 2) = (7, 1, 1, 1)$ respectively. Hence, the answer to the RMQ is the minimum of the second element of $S(2, 1)$, which corresponds to the leaf $B(1) = 10$, and the first element of $P(2, 2)$, which corresponds to leaf $B(4) = 7$. The result could then be computed as $RMQ = min(3, 7) = 3$.

6.2.2 Runtime Analysis

In order to achieve $\mathcal{O}(\log N)$ time, we need to specify how to merge lists effectively. If we are merging $P(i+1, 2j-1)$ and $P(i-1, 2j)$ into $P(i, j)$, we could launch a thread for each element of $P(i, j)$. Since we are only copying $P(i+1, 2j-1)$ into the first half of $P(i, j)$, we could perform this in $\mathcal{O}(1)$ time. The second half is similar to this approach except we are comparing the min of each element of $P(i-1, 2j)$ and the last element of $P(i+1, 2j-1)$, comparing and copying takes $\mathcal{O}(1)$ time. As such, in total, merging suffix and prefix lists takes constant time. Therefore, for each level of the tree, it takes $\mathcal{O}(1)$ to merge lists, since there are $\log n$ levels the total running time for parallel RMQ is $\mathcal{O}(\log n)$. Without parallelization, merging takes $\mathcal{O}(n)$ time, since there are n nodes, this results in an $\mathcal{O}(n^2)$ running time.

6.3 Serial Implementation

6.3.1 Tree Representation

The initial representation of the tree is with an adjacency list, however, this representation is presented with a few of problems. One should be able to traverse the tree in level order from down up, effectively. Furthermore, an adjacency list adds a layer of complexity as the input array needs to be copied onto the leaves in an efficient manner. It is easier and much more efficient to represent the tree as a tree array, since it is a complete binary tree, we could jump towards specific nodes using mathematical calculations. A tree array is also indexed in level order by default. Refer to Figure 5 for details. For the purpose of this implementation, all arrays will be indexed at 1.

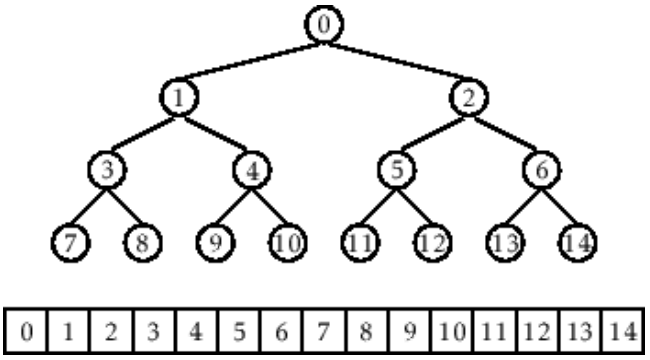


Figure 5: A tree represented by an array, if the tree is a complete binary tree, one could make specific jumps with mathematical calculations. In this project, the array will be indexed at 1.

Given B as the size of the input array, the number of nodes, n, could be calculated as:

$$n = 2B - 1$$

As such, we create a tree array of size $n+1$, with the root starting at index 1.

6.3.2 Code for Level Order Traversal

In order to perform the two for-loops mentioned in the RMQ pseudocode, we need to select a specific node in the tree array given the level, l, and the node number, num. Let o be the offset from the first index of the tree array and B be the size of the array. We are assuming that the root starts at level 0, and node 1,2 is level 1, so on and so forth.

$$O = \frac{B}{2^{height-level}} + num$$

To calculate the height of a tree given B nodes, we use the following:

$$height = \lceil \log_2 B \rceil$$

The code to choose a node given the specific level and node number is then as follows:

```
1 //returns the tree array offset for a specific level and node
2 int chooseNode(int numNodes,int level, int node){
3
4   double height = ceil(log2(numNodes));
5
6   int offset = (numNodes/pow(2,height-level)) + node;
7
8   return offset;
9 }
```

The Level Order traversal comes in two parts, one for loop to fill in the leaves, and two for-loops to fill in the rest of the prefix/suffix lists. The leaves of the tree is always located in the last level, i.e. $height - 1$. The other two for loops goes from level $height - 2$ to 0, and each level contains 2^{level} nodes. The code for level order traversal is then as follows:

```
1
2 void fillInLists(int* inputArray, int numNodes, int numInputs, int**
      prefixTree, int** suffixTree){
3
4   double height = ceil(log2(numNodes));
5
6   //printf("\nTesting fill leaves\n");
7   for(int i = 1; i <= pow(2,height -1); i++){
8     int index = chooseNode(numNodes,height-1,i);
9
10  }
11
12  for(int i = height-2; i>=0; i--){
13    for(int x = 1; x <= pow(2,i);x++){
14
15      int index = chooseNode(numNodes,i,x);
```

```
      //int *prefixList = prefixTree[index];
      //int *suffixList = suffixTree[index];

      int left_index = get_left(index, numNodes);
      int right_index = get_right(index, numNodes);
    }
  }
}
```

The runtime to go through each node is $\mathcal{O}(2n-1) = \mathcal{O}(n)$. The get_left and get_right function returns the left and right children of each node, which could be seen from the following code:

```
//function to get right child
int get_right(int index, int NUM_NODES){

  if(index != -1 && ((2*index) + 1)<= NUM_NODES){
    return (2*index) + 1;
  }

  return -1;
}

  //function to get left child
int get_left(int index, int NUM_NODES){

  if(index != -1 && ((2*index)<= NUM_NODES)){
    return (2*index);
  }

  return -1;
}
```

6.3.3 Code to fill Prefix and Suffix Lists

The Prefix and Suffix Lists is represented by a list of lists, all of which are indexed at 1. The index of the tree array corresponds to each index of the prefix/suffix lists. For example, the prefix list of node 2, correspond to Prefix[2]. The third element of that list then corresponds to Prefix[2][3], so on and so forth. The main problem with creating this list of lists is the size of the inner list. The number of elements for the prefix/suffix list changes depending on which node we are referring to. For example, node 2 needs to have a list of size 4, node 4 needs to have a list of size 2, so on and so forth. The size we need for the prefix/suffix lists correspond to the number of children that level has, the calculation is done with the following code:

```
//num of children for each level
int numOfChildren(int node, int numInputs){

  int level = floor(log2(node));
```

```
  int numChildren = numInputs/ pow(2,level);

  //printf("Node: %d Num Children: %d\n", node, numChildren);

  return numChildren;
}
```

With this information in mind, we could then build the suffix and prefix lists.

```

//Build prefix and suffix lists
int** prefixTree = (int**)malloc((numNodes+1) * sizeof(int*));

  for(int i = 1; i <= numNodes;i++){
    prefixTree[i] = (int*) malloc((numOfChildren(i,numInputs)+1) *
     sizeof(int*));
  }

  int** suffixTree = (int**)malloc((numNodes+1) * sizeof(int*));

  for(int i = 1; i <= numNodes;i++){
    suffixTree[i] = (int*) malloc((numOfChildren(i,numInputs)+1) *
     sizeof(int*));
  }
```

The code for merging two prefix/suffix lists is straightforward and follows the principle stated in the RMQ algorithm.

```
void merge_list(int* dest_list, int dest_list_length, int* array1, int
    array1_length, int* array2){
  int x = 1;
  for(int i = 1; i <= array1_length;i++){
    dest_list[x] = array1[i];
    x++;
  }

  x = 1;
  for(int i = array1_length+1; i <= dest_list_length;i++){

    int last_elem = array1[array1_length];

    if(last_elem <= array2[x]){
      dest_list[i] = last_elem;
    }else{
      dest_list[i] = array2[x];
    }

    x++;
  }
}
```

Finally, we can add these two functions to complete the level order traversal of the tree array, filling in the prefix/suffix lists in the process.

```
void fillInLists(int* inputArray, int numNodes, int numInputs, int**
    prefixTree, int** suffixTree){

  double height = ceil(log2(numNodes));

  //printf("\nTesting fill leaves\n");


  for(int i = 1; i <= pow(2,height -1); i++){
    int index = chooseNode(numNodes,height-1,i);

    int * prefixList = prefixTree[index];
    prefixList[1] = inputArray[i-1];

    int* suffixList = suffixTree[index];
    suffixList[1] = inputArray[i-1];
  }

  for(int i = height-2; i>=0; i--){
    for(int x = 1; x <= pow(2,i);x++){

      int index = chooseNode(numNodes,i,x);
      //int *prefixList = prefixTree[index];
      //int *suffixList = suffixTree[index];

      int left_index = get_left(index, numNodes);
      int right_index = get_right(index, numNodes);

      merge_list(prefixTree[index], numOfChildren(index,numInputs),
    prefixTree[left_index],
        numOfChildren(left_index,numInputs),prefixTree[right_index]);

      merge_list(suffixTree[index], numOfChildren(index,numInputs),
    suffixTree[right_index],
       numOfChildren(right_index,numInputs), suffixTree[left_index]);
    }
  }
}
```

List merging takes $\mathcal{O}(n)$ time, and there are n nodes . In total, the sequential pre-processing algorithm takes $\mathcal{O}(n^2)$ time.

6.3.4 Code to Perform the RMQ query

To perform the RMQ query in constant time, recall the example mentioned in section 6.2.1, we need to calculate the $LCA(i, j)$ where i, j are starting/ending indices of the query. After calculating the LCA, we need to compare the i^{th} element which corresponds to the suffix list of the left child, and the j^{th} element which corresponds to the prefix list of the right child. In order to perform these two operations, we need to perform to bit-wise operations.

Code to find the LCA of two nodes: The lowest common ancestor of two nodes in a complete binary tree could be calculated in 'constant' time through

bit operations. By converting the level-order indexes to their binary representation, we could calculate the LCA by comparing the binary positions from left to right. For example, the LCA(8,10) is 2 in the tree array. Convert 8 and 10 into their binary representations, 1000 and 1010 respectively. Comparing their bits from the most significant bit to their least significant (left to right), keep track of the bits until the first difference is found. Therefore, the first bits which coincide with one another is 10, which translates to 2. We are assuming that bit operations takes constant time to perform.

```
int LevelOrder_LCA(int num1, int num2){

  int binaryLength = log2(num1);
  int pos = binaryLength;
  int mask = 1 << binaryLength;

  //printf("\nMask: %d, Binary Length: %d\n", mask, binaryLength);

  int num1Offset = mask & num1;
  int num2Offset = mask & num2;

  //compare bits from left to right untill first bit does not match
  while((num2Offset ^ num1Offset) == 0){
    // printf("Index %d: same, Mask: %d,Num 1 Off: %d, Num 2 Off: %d\n"
    //  ,pos, mask, num1Offset, num2Offset);

    pos--;
    mask = mask >> 1;

    num1Offset = mask & num1;
    num2Offset = mask & num2;
  }
  pos++;

  //printf("Pos of change: %d\n Calculating LCA: \n", pos);
  int result = 0;
  int power = 0;
  mask = 1 << pos;

    //calculate the bits that coincides
  for(int i = pos; i <= binaryLength;i++){

    if((mask & num1) != 0){
      //printf("incrementing result\n");
      result += pow(2,power);
    }

    // printf("Index: %d, Mask: %d,Power: %d, Pos and Num: %d,Result: %
    d\n"
    //  ,i, mask, power,(mask & num1),result);

    mask = mask << 1;
    power++;
  }

  return result;
```

```
}
```

Code to convert query index to prefix/suffix offset:
The query index needs to be converted to the prefix/suffix index in order to perform the RMQ query correctly. For example, if the query provided is $RMQ(4, 7)$ which translates to $LCA(12, 15)$ in the tree array. We would have to compare the first element of the suffix list of the left child, and the second element of the prefix list of the right child. As such, this level order to prefix/suffix offset is essential. To calculate this conversion, the query index needs to be divided by 2 depending on what level the prefix/suffix list is located. This translates to eliminating the first n significant bits, with n according to the amount of levels the prefix/suffix list is located. For example, 7 translates to 111, and since node 7's prefix/suffix list is located in level 2, we eliminate the first 2 most significant bits of 7. This results in 1, since the arrays are indexed at 1, we add 1 to the end result.

```
int shiftBits(int num, int offset){

  if(num == 0){
    return 1;
  }

  int binaryLength = log2(num);
  int length = binaryLength - offset;

  int mask = 1;
  int result = 0;

  //printf("Length: %d, Offset: %d", binaryLength,length);

  for(int i = 0; i <= length;i++){

    if((mask & num) != 0){
      result += pow(2,i);
    }

    mask = mask << 1;
  }

  //printf("\nAfter Bit Shift: %d by %d bits\n", result,offset);

  result++;

  return result;
}
```

With these two shift bit operations in mind, we are able to perform the RMQ query in constant time.

```
int RMQ(int index1, int index2, int numInputs,int** prefixTree, int**
    suffixTree){

```

```
  int minQuery = 0;
  int numNodes = 2* numInputs - 1;
  double height = ceil(log2(numNodes));

  //if index1 and 2 is right next to each other, only compare the two
    numbers
  if(abs(index2 - index1) == 1){

    int num1 = prefixTree[chooseNode(numNodes,height-1,index1 + 1)][1];
    int num2 = prefixTree[chooseNode(numNodes,height-1,index2 + 1)][1];

    if(num1 <= num2){
      return num1;
    }else{
      return num2;
    }
  }

  if(index1 == index2){
    return prefixTree[chooseNode(numNodes,height-1,index1 + 1)][1];
  }


  int leftIndex = chooseNode(numNodes, height-1, index1 + 1);
  int rightIndex = chooseNode(numNodes, height-1, index2 + 1);

  int LCA_index = LevelOrder_LCA(leftIndex,rightIndex);
  int leftChild = get_left(LCA_index,numNodes);
  int rightChild = get_right(LCA_index,numNodes);

  int level_left = floor(log2(leftChild));
  int index1_suffix = shiftBits(index1,level_left);
  //printf("Left Child Index: %d ",index1_suffix);

  int level_right = floor(log2(rightChild));
  int index2_prefix = shiftBits(index2,level_right);
  //printf("Right Child Index: %d\n", index2_prefix);

  int minSuffix = suffixTree[leftChild][index1_suffix];
  int minPrefix = prefixTree[rightChild][index2_prefix];

  printf("Min Suffix: %d from Node %d, Min Prefix: %d from Node %d\n"
    ,minSuffix,leftChild,minPrefix,rightChild);

  if(minPrefix <= minSuffix){
    minQuery = minPrefix;
  }
  else{
    minQuery = minSuffix;
  }

  return minQuery;
}
```

6.4 Runtime for Serial Range Minimum Query

The runtime to find the range minimum query for different input sizes is summarized in Table 6

$\log_2$ **of input size**	20	18	16	14
Build Prefix/Suffix list	708.205994	176.568008	44.159000	11.073000
List Merging Serial	4.423000	1.104000	0.277000	0.067000
Fill in Prefix/Suffix Lists	8.859000	2.228000	0.569000	0.143000
Perform RMQ Query	0.005000	0.004000	0.005000	0.00500

Table 6: Run time for RMQ in serial. All times are in seconds.

6.5 Parallel Implementation

Several immediate improvements could be added to the serial implementation. In order to achieve $\mathcal{O}(\log n)$ pre-processing time, we would have to merge the prefix/suffix lists and traverse each level in $\mathcal{O}(1)$ time. For this particular project, I was not able to implement an $\mathcal{O}(1)$ level traversal in time. However, I was able to implement a parallel version of the list merging algorithm.

6.5.1 Merging prefix/suffix lists in Parallel

In order to merge the lists in parallel, we would launch a thread for each element in prefix/suffix array in the left and right child. Since we are only comparing/-copying each element into the bigger list, we could achieve constant time for an arbitrary array of size n. We implement the kernel with a grid-stride loop, ensuring maximum memory coalescing by going through the elements per unit-warp.

6.5.2 Code to Merge prefix/suffix lists in Parallel

```
1
2  __global__ void parallelMerge(int* dest_list, int dest_list_length, int
       * arrayLeft, int arrayLeft_length, int* arrayRight){
3
4    int index = blockIdx.x * blockDim.x+ threadIdx.x + 1;
5    int stride = blockDim.x * gridDim.x;
6    int arrayRightIndex = 0;
7
8    for(int i = index; i <= dest_list_length; i+= stride){
9
10     if(i <= arrayLeft_length){
11       dest_list[i] = arrayLeft[i];
12     }
13     else{
14       arrayRightIndex = i - arrayLeft_length;
15       int minNumLeft = arrayLeft[arrayLeft_length];
16       int arrayRightNum = arrayRight[arrayRightIndex];
```

```
17
18        if(minNumLeft <= arrayRightNum){
19          dest_list[i] = minNumLeft;
20        }else{
21          dest_list[i] = arrayRightNum;
22        }
23      }
24      __syncthreads();
25
26    }
27  }
```

With this parallel merge, we are able to call this kernel in place of the original sequential merge list. We are assuming there are 512 threads per block, and called the kernel with the minimum number of blocks needed to span the size of the input array. The older "fillInLists" function could then be updated with the parallel merge as follows:

```
1
2  void fillInLists(int* inputArray, int numNodes, int numInputs, int**
       prefixTree, int** suffixTree){
3
4    double height = ceil(log2(numNodes));
5
6    //printf("\nTesting fill leaves\n");
7    for(int i = 1; i <= pow(2,height -1); i++){
8      int index = chooseNode(numNodes,height-1,i);
9
10     int * prefixList = prefixTree[index];
11     prefixList[1] = inputArray[i-1];
12
13     int* suffixList = suffixTree[index];
14     suffixList[1] = inputArray[i-1];
15   }
16
17   for(int i = height-2; i>=0; i--){
18     for(int x = 1; x <= pow(2,i);x++){
19
20       int index = chooseNode(numNodes,i,x);
21       int left_index = get_left(index, numNodes);
22       int right_index = get_right(index, numNodes);
23
24       // testing parallel version
25       //for prefix tree
26       int dest_length_host = numOfChildren(index,numInputs);
27       int arrayLeft_length_host = numOfChildren(left_index,numInputs);
28
29       int *destinationDevice;
30       int *arrayLeft_device;
31       int *arrayRight_device;
32
33       cudaMalloc((void**) &destinationDevice, (dest_length_host+1) *
       sizeof(int));
34       cudaMalloc((void**) &arrayLeft_device, (arrayLeft_length_host+1)
       * sizeof(int));
35       cudaMalloc((void**) &arrayRight_device, (arrayLeft_length_host+1)
        * sizeof(int));
```

```
36
37        cudaMemcpy(arrayLeft_device, prefixTree[left_index],
38          (arrayLeft_length_host+1) * sizeof(int), cudaMemcpyHostToDevice
      );
39        cudaMemcpy(arrayRight_device, prefixTree[right_index],
40          (arrayLeft_length_host+1) * sizeof(int), cudaMemcpyHostToDevice
      );
41
42        int threads_per_block = 512;
43        int numBlocks = (dest_length_host + threads_per_block - 1) /
      threads_per_block;
44
45        parallelMerge<<<numBlocks,threads_per_block>>>(destinationDevice,
       dest_length_host,
46          arrayLeft_device, arrayLeft_length_host, arrayRight_device);
47
48        cudaMemcpy(prefixTree[index], destinationDevice,
49          (dest_length_host+1) * sizeof(int), cudaMemcpyDeviceToHost);
50
51
52        //for suffix tree
53        cudaMemcpy(arrayLeft_device, suffixTree[left_index],
54          (arrayLeft_length_host+1) * sizeof(int), cudaMemcpyHostToDevice
      );
55        cudaMemcpy(arrayRight_device, suffixTree[right_index],
56          (arrayLeft_length_host+1) * sizeof(int), cudaMemcpyHostToDevice
      );
57
58        parallelMerge<<<numBlocks,threads_per_block>>>(destinationDevice,
       dest_length_host,
59          arrayRight_device, arrayLeft_length_host, arrayLeft_device);
60
61        cudaMemcpy(suffixTree[index], destinationDevice,
62          (dest_length_host+1) * sizeof(int), cudaMemcpyDeviceToHost);
63      }
64    }
65  }
```

6.6 Runtime for Parallel Range Minimum Query

I was not able to achieve $\mathcal{O}(\log n)$ pre-processing time as originally intended due to only implementing one of the two parallel improvements. Nevertheless, parallel list merging would achieve a speedup of order n, increasing the pre-processing time to $\mathcal{O}(n)$. The summary of runtimes for my Parallel implementation could be found in Table 7

6.7 Future Improvements

For the scope of this paper, I was not able to fully implement an optimal parallel algorithm for the Range Minima Query. Assuming that I was able to implement the aforementioned level order traversal in constant time, it would bring the runtime to $\mathcal{O}(\log n)$. However, there are several future improvements which

$\log_2$ **of input size**	20	18	16	14
Build Prefix/Suffix list	708.2059	176.5680	44.1590	11.0730
List Merging Serial	4.423000	1.104000	0.27700	0.06700
List Merging Parallel	0.013000	0.011000	0.00800	0.00600
Fill in Prefix/Suffix Lists Serial	8.859000	2.228000	0.56900	0.14300
Fill in Prefix/Suffix Lists Parallel	5.941000	1.612000	0.43900	0.14700
Perform RMQ Query Serial	0.005000	0.004000	0.00500	0.00500
Perform RMQ Query Parallel	0.006000	0.004000	0.00500	0.00500
Speedup for List Merging	340.231	100.364	34.625	11.167
Speedup for Pre-processing	1.491	1.382	1.296	0.973

Table 7: Run time for RMQ in Parallel. All times are in seconds.

could be added to improve the runtime to $\mathcal{O}(\log \log n)$. These are the steps which could be taken for an array B of an arbitrary size of 2^n :

1. Partition B into equal-sized block B_i, each of size $\log n$.

2. Preprocess each B_i block separately for the range-minima algorithm.

3. For each B_i block, compute the minimum element x_i, and its subsequent preifx/suffix minima.

4. Preprocess the array $(x_1, x_2, \ldots, x_{n/\log n})$ using the range-minima algorithm.

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p131-136. Print.

7 By Marina Doric, on the String Matching Algorithm

Given two series of characters, a text and a pattern, the object of the String Matching Algorithm is to find the indices at which the pattern occurs in the text. For example, if given the text "the dog is the best" and the pattern "the", the algorithm would return the indices 0 and 11.

t	h	e		d	o	g		i	s		t	h	e		b	e	s	t
0	**1**	**2**	3	4	5	6	7	8	9	10	**11**	**12**	**13**	14	15	16	17	18

7.1 The Serial Implementation

The string-matching problem is traditionally solved in a serial fashion. The naive way to solve this problem would be to check for the entire pattern by comparison at each index of the text, also known as the brute force method. Let the pattern have length m and the text length n. Then the algorithm will have a $O(m*n)$ run time. We will first implement this naive search method below.

```
__host__ void serialSearch_naive(char* p, int psize, char* t, int tsize
    , int* match){
  int iT = 0; //Index in text
  int iP = 0; //Index in pattern
  int iR = 1; //Index in match
  while (iT <= tsize-psize){
    if (t[iT] == p[iP]){
      int found = 1;
      while (iP < psize){
        if (t[iT + iP] != p[iP]){
          found = 0;
          break;
        }
        iP += 1;
      }
      if (found){
        match[iR] = iT;
        iR += 1;
      }
    }
    iP = 0;
    iT += 1;
  }
  match[0] = iR-1;
}
```

There is also an optimal version of this search which has an $O(n)$ run time, the Knuth-Morris-Pratt Algorithm. However, I was unable to get this working correctly. I attempted to implement it, but it is not functioning. (I mistakenly said it was in my presentation.) Thus, it is not included in the analyses.

7.2 The Parallel Version

We will next endeavor to implement the algorithm in parallel based on the scheme described in Chapter 7 of "Introduction to Parallel Algorithms" by Joseph Jaja.[1] The basic form of the algorithm is first to perform *pattern analysis* and then *text analysis*. Theoretically, the run time for the parallel version should be *O(log(m))*. We will focus on the non-periodic case. To analyze the pattern using a parallel algorithm, it will utilize what is called the witness function as a form of pattern pre-processing.

7.2.1 The Witness Function, The Witness Array, Their Usefulness

The witness function's goal is to provide an index where the pattern mismatches itself. These will be used to create the witness array, which we will use in the algorithm. Let P represent the pattern, and T the text. Let i indicate the index of each letter in the pattern. Let *psize* indicate the length of the pattern. Then, the witness function is defined as follows:

$$\phi_{wit}(i) = \begin{cases} i = 0 & 0 \\ 0 < i < psize & k, where P(k) \neq P(i+k) \end{cases}$$

The witness array, W, will contain at each index the result of the witness function at that index in P. This array can be used to eliminate one of each pair of indices i or j where P occurs in T, by providing indices at which to compare the pattern vs. the text. The size of this witness array is half the pattern size, rounded up. This witness array can be implemented either on the CPU or the GPU. We will compare the speed of the two versions.

CPU Version

```
__host__ void witnessCPU(char* P, int psize, int* W, int wsize){
  W[0] = 0;
  for (int i=1; i<wsize; i++){
    for (int k=1; i+k<psize; k++){
      if (P[k] != P[i+k]){
        W[i] = k;
        break;
      }
    }
  }
}
```

GPU Version

```
__global__ void witnessGPU(char* P, int psize, int* W, int wsize){
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  if (i < wsize){
    for (int k=1; i+k<psize; k++){
```

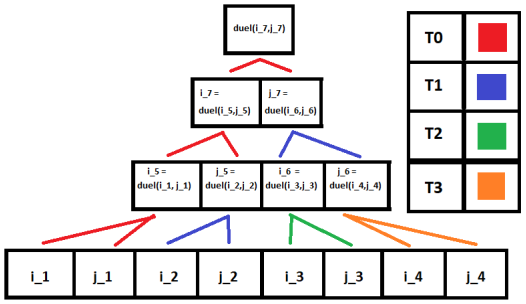
```
5           if (P[k] != P[i+k]){
6             W[i] = k;
7             break;
8           }
9       }
10    }
11    W[0] = 0;
12  }
```

Thus, the witness array contains locations where the pattern differs from itself at the current index. We can use these locations, to determine possible locations where the pattern may start. Given two indices that are possible locations of P, we can eliminate one based on where the witness array says the pattern mismatches itself. The following algorithm, called *duel*, returns the index where the occurrence of the pattern is possible:

```
1  int duel(char* T, char* P, int* W, int i, int j){
2      int k = W[j-i];
3      if (T[j+k] != P[k]){
4          return i;
5      } else {
6          return j;
7      }
8  }
```

7.2.2 The Synced Implementation

Once the text is split into blocks of the same size as the witness array, one can find a single index where the pattern can occur within this block by creating a tree from the results of the *duel* function, where the leaves are all the indices in that particular block of text, and each interior node is the result of the *duel* function called on its children. We will use a reduction scheme where each thread places the duel function results for every two leaves onto the next tree level. For example, if the witness size is 8, we will reduce like so with four threads:



In this implementation of the algorithm, instead of actually creating this tree, the leaves of the tree were assembled in an array, and then reduced in

parallel by calling the *duel* function repeated until one result is found. Then, the resulting index of each tree from each block will be tested using the brute force (serial) algorithm to determine if the pattern occurs in the block. The thread zero of each block will complete this. Let *wsize* represent the size of the witness function, *psize* the size of the pattern, and *tsize* the size of the text. Let *tree* represent the array of the leaves of the described tree.

```
__global__ void search_synced(int* W, int wsize, char* P, int psize,
    char* T, int tsize, int* match, int* tree){

  int ID = threadIdx.x + blockIdx.x*wsize;
  int blockStart = blockIdx.x*wsize;
  if (ID < tsize){
    tree[ID] = ID;
  }

  __syncthreads();

    //The reduction scheme
  for (int s = 1; s < wsize; s = 2*s){
    if ((2*s*threadIdx.x+s < wsize)&&(2*s*threadIdx.x+s+blockStart <
    tsize)){
      int i = tree[2*s*threadIdx.x + blockStart];
      int j = tree[2*s*threadIdx.x + s + blockStart];
      tree[2*s*threadIdx.x + blockStart] = duel(T, P, W, i, j, wsize,
    psize, tsize);
    }
    __syncthreads();
  }

  //Now for each block, check the one location where the pattern may
    possibly occur using brute force. Use thread 0 to finish.
  if (threadIdx.x == 0){
    int i = 0;
    int m = tree[blockStart];
    while (i < psize){
      if (i + m >= tsize){
        m = -1; //Out of bounds
        break;
      }
      if (T[i+m] != P[i]){
        m = -1;
        break;
      }
      i++;
    }

    //Store the result
    match[blockIdx.x] = m;
  }
}
```

This version, although functional, can be improved. One issue is the repeated use of global memory, which is slow to access. This can be improved by instead using shared memory to build the tree. We will see later what speedup this

gives us.

```
__global__ void search_synced_shared(int* W, int wsize, char* P, int
    psize, char* T, int tsize, int* match){

  __shared__ int tree[1024];
  int ID = threadIdx.x + blockIdx.x*wsize;
  if (ID < tsize){
    tree[threadIdx.x] = ID;
  }

  __syncthreads();

    //The reduction scheme
  for (int s = 1; s < wsize; s = 2*s){
    if ((2*s*threadIdx.x + s < wsize)&&(2*s*threadIdx.x + s + blockIdx.
    x*wsize < tsize)){
      int i = tree[2*s*threadIdx.x];
      int j = tree[2*s*threadIdx.x + s];
      tree[2*s*threadIdx.x] = duel(T, P, W, i, j, wsize, psize, tsize);
    }
    __syncthreads();
  }

  //Now for each block, check the one location where the pattern may
    possibly occur using brute force. Use thread 0 to finish.
  if (threadIdx.x == 0){
    int i = 0;
    int m = tree[0];
    while (i < psize){
      if (i + m >= tsize){
        m = -1; //Out of bounds
        break;
      }
      if (T[i+m] != P[i]){
        m = -1;
        break;
      }
      i++;
    }

    //Store the result
    match[blockIdx.x] = m;
  }
}
```

7.2.3 The Multiple Kernel Version

As you can see in the synced version, this reduction scheme requires that threads be synced after each new layer of the tree is created. Since threads can only be synced within blocks, this requires that a specific amount of threads and blocks be used for each kernel call that is dependent on the size of the witness array. To avoid the necessity of this, we will instead call multiple kernels that each reduce the tree only one level within a similar loop to that above, but rather on

the CPU. Then, a new kernel will be called where each thread will complete the brute force algorithm to check the only location where the pattern may occur in each block. Please note that the example code below is faulty and only works for pattern sizes up to 65,537. It requires further debugging to be used for larger sizes. (This holds for both the 'new' and 'old' GPUs.) First, we will create the initial tree leaves, as described before:

```
__global__ void setup_Tree(int* tree, int tsize){
  int ID = threadIdx.x + blockIdx.x*blockDim.x;
  if (ID < tsize){
    tree[ID] = ID;
  }
}
```

We will then call the following kernel in the described loop in the same manner that it was in the synced version, to assemble the tree:

```
//The looped code snippet
for (int s = 1; s < wsize; s = 2*s){
    createTreeLevel<<<block, div>>>(W, wsize, P, psize, T, tsize, tree,
     s);
}

//The kernel
__global__ void createTreeLevel(int* W, int wsize, char*P, int psize,
    char* T, int tsize, int* tree, int s){
  //Setup
  int ID = threadIdx.x + blockIdx.x*blockDim.x;
  int tID = ID % wsize;
  int bID = ID / wsize;
  int blockStart = bID*wsize;
  int cap = tsize - blockStart;

  //Create current level
  int ind = 2*s*tID + blockStart;
  if ((2*s*tID + s < wsize)&&(cap > wsize)){
    int i = tree[ind];
    int j = tree[ind + s];
    int temp = duel(T, P, W, i, j, wsize, psize, tsize);
    tree[ind] = temp;
  }
}
```

Finally, instead of finishing off each block with thread one like we did in the synced version, we will have each individual thread finish these blocks:

```
__global__ void search_Finish(int wsize, char* P, int psize, char* T,
    int tsize, int* match, int* tree){
  //Setup
  int ID = threadIdx.x + blockIdx.x*blockDim.x;

  //Check for pattern
  if (ID*wsize < tsize){
    int i = 0;
    int m = tree[ID*wsize];
```

```
    while (i < psize){
      if (i + m >= tsize){
        m = -1; //Out of bounds
        break;
      }
      if (T[i+m] != P[i]){
        m = -1;
        break;
      }
      i++;
    }

    //Store result
    match[ID] = m;
  }
}
```

Like with the first synced version, this uses global memory. We cannot use shared memory here since we are creating the tree across blocks. However, we can instead access the text from constant memory and see what kind of speedup this gets us. The code is exactly the same as above except for instead retrieving the text from constant memory. To copy the memory to constant memory, the following is added and called before running. (If running all in one file, there is no need to create a function like this.)

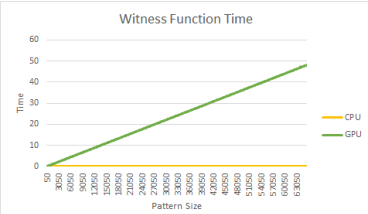
```
__constant__ char TC[2048*32]; //Max constant memory allowed

__host__ void setConstantMem(char* text, int tsize){
  cudaMemcpyToSymbol(TC, text, tsize*sizeof(char));
}
```

This itself has limitations though because only a limited amount of constant memory is available. As noted in the code, it only allowed for the allocation of 65,536 characters. This accounts for only approximately 60% of the text that was collected from the bible.txt data file. (This is the same for the 'new' and 'old' GPUs.)

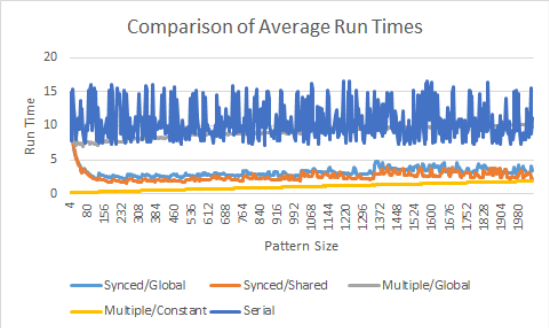
7.3 Comparison of Run Times

Now, we must analyze the run-times of the various versions and see how they compare. We will first compare the creation of the witness array on the CPU versus the GPU. As you can see in the graph below, as the size of the pattern increases, the time to calculate the witness array has nearly a constant relationship with the CPU version, whereas it has a linear relationship with the GPU version. Thus, in all cases it seems that the CPU version is a better choice.



Pattern Size	CPU	GPU	Slowdown
100	0.0009	0.088314	0.087414
500	0.0012	0.380269	0.379069
1000	0.0026	0.74416	0.74156
2000	0.0048	1.480474	1.475674
5000	0.0124	3.675719	3.663319
50000	0.0948	36.643757	36.548957

Next, let's compare the run-times of the four parallel versions, given the limitations of the synced versions, and the constant memory version. Thus, the analysis is compared while limiting the pattern size to 2048, and the text size to 65,536 characters. Each run time is an average of 10 runs on the same pattern size.

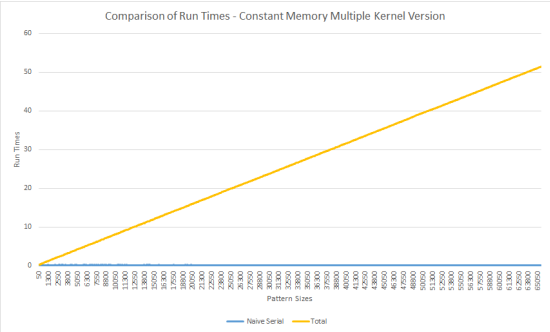


Pattern Size	Serial	Synced/Global	Speedup - 1	Synced/Shared	Speedup - 2	Multiple/Global	Speedup - 3	Multiple/Constant	Speedup - 4
4	7.443101	18.424498	-10.981397	16.940392	-9.497291	12.52824	-5.085139	0.21537	7.227731
24	12.4605	6.640951	5.819549	6.822371	5.638129	7.810324	4.650176	0.160298	12.300202
64	8.9513	3.252992	5.698308	3.410525	5.540775	6.98334	1.96796	0.203216	8.748084
128	12.6175	2.095536	10.521964	2.08487	10.53263	7.125229	5.492271	0.271754	12.345746
800	12.409901	2.926935	9.482966	2.292157	10.117744	8.811953	3.597948	0.850797	11.559104
1000	7.7682	2.612074	5.156126	1.982061	5.786139	8.983408	-1.215208	1.010652	6.757548
2048	11.2273	3.434717	7.792583	2.257011	8.970289	9.873536	1.353764	1.868292	9.359008

As the graph visualizes, on this small text size with small pattern sizes, the multiple kernel version that uses constant memory is fastest overall. Let's remove the pattern size restraint and compare the multiple kernel version with constant memory to the serial version below. As you can see, even the naive serial version performs far better overall. Neither of these graphs even include the time required to copy data, only the run time alone. Thus, they are even slower in comparison than pictured here.

7.4 Further Ideas and Improvements

Considering the whole point of the parallel versions to try be faster than the serial, these algorithms are performing poorly. They have not even been compared to an optimal serial algorithm, such as the Knuth Morris Pratt Algorithm, and they still are failing. One major consideration is the use of global memory. The version using constant memory did perform the best of the four, which supports this theory of it being an issue.



When comparing the run time speeds of the tree creation, and the searching afterwards separately in the multiple kernel version, the time to create the tree is comparable to the serial run time. Thus, it is the search_Finish kernel that is taking up so much time. It is serialized in that one thread checks for the whole pattern at each index marked as a possible location. Perhaps a speedup could be gotten here if all the locations in the pattern were checked simultaneously between threads, and then any mismatches could be recorded, and indicate the mismatch. However, this could pose the problem of syncing threads, like in the synced versions of the code. This may be avoided with an atomic expression used as a counter of the mismatches.

Another issue that may have been encountered in the synced version with shared memory, is bank conflicts. These were not accounted for in the reduction process and therefore may cause issues with some pattern sizes. Removing these may also provide a speedup.

Overall, this problem shows the challenges of 'parallelizing' traditionally serial algorithms. Although the process may seem straight forward, when it comes to implementation, there are many issues to consider. Further tinkering and playing with this code may be able to result in a worthwhile parallel algorithm, accounting for the currently encountered issues.

7.5 References

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p311-356. Print

[2] bible.txt, The Canterbury Corpus, corpus.canterbury.ac.nz/descriptions/.

[3] "Knuth–Morris–Pratt Algorithm." Wikipedia, Wikimedia Foundation, 7 May 2019, en.wikipedia.org/wiki/Knuth–Morris–Pratt_algorithm.

[4] Source Code: https://github.com/MDoric18/Final

8 By Jiaxuan(Isabell) Huang, on Perfect Matching

8.1 Problem Description and the Algorithm

Given a graph $G = (V, E)$ such that G has a perfect matching, we want to output a set of edges M that is a perfect matching of G.

Our algorithm is a randomized algorithm that outputs a perfect matching with probability $\geq 1/2$. If we run the algorithm n times, we can increase the probability of finding a perfect matching to be arbitrarily close to 1.

Tutte matrix: Given a graph $G = (V, E)$ with $|V| = n$, let A be the $n \times n$ adjacency matrix of G. For $i, j \in 1, \ldots, n$, suppose we have a set x_{ij} for all $i < j$. We define the Tutte matrix T of G to be the $n \times n$ matrix such that if $i < j$, we substitute x_{ij} for each entry $A(i, j) = 1$; if $i > j$, w substitute $-x_{ji}$ for each entry $A(i, j) = 1$.

Theorem(Tutte): Let $G = (V, E)$ be a graph and T be its Tutte matrix. $det(T) \neq 0$ if and only i G has a perfect matching.
It is relatively easy to test if G is a perfect matching. If not, the algorithm will stop and output that finding a perfect matching is impossible.
Note that if the graph has no perfect matching, it will always stop at this step. However, if the graph has a perfect matching, it is possible that the randomized edge weight produce a determinant 0 matrix, and output a wrong result.
If determinant $d \neq 0$, W can be proven to be the weight of the minimum-weight perfect matching if W is the largest integer satisfying $2^{2W} \mid d$.

Some notation: Let T be an $n \times n$ matrix, T_{ij} is the submatrix of T obtained by deleting the i-th row and the j-th column.
Adjoint matrix: Denoted by $adj(T)$, the adjoint matrix of an $n \times n$ matrix is an $n \times n$ matrix whose (i, j)-th entry is equal to $(-1)^{i+j} \det(T_{ji})$.
Computing determinant for each T_{ij} can take a lot of time. Here, we take advantage of this formula to simplify the calculation:

$$T^{-1} = \frac{1}{|\det(T)|} adj(T)$$

Algorithm for finding the perfect matching:
Input: A graph $G = (V, E)$.
1. Assign random integer weights to all edges in E, such that each weight is independently chosen from $\{1, 2, \ldots, 2|E|\}$.
2. Produce the Tutte matrix of G, denoted by T. Substitute $2^{w_{ij}}$ for each non-zero entry x_{ij}.
3. Compute $\det(T)$. Deduce the value W of the weight of the minimum-weight perfect matching.

4. Compute the adjoint matrix $adj(T)$, where the absolute value of x_{ij} is $\det(T_{ji})$.
5. for each edge $(i, j) \in E$ pardo
if $(2^{w_{ij}} \det(T_{ij})/2^{2W})$ is odd, then include (i, j) in M.
Output: A set of edges M such that M is a perfect matching with probability $\geq 1/2$.

8.2 Serial Implementation

Since most serial algorithms for perfect matching differ a lot from the algorithm above, I implement the parallel algorithm serially as the serial version.
First we implement a helper function that count the number of edges in G. `countEdge` counts the number of 1's in the adjacency matrix, and then divide by 2 to get the number of edges in G.
And then, we assign random integer weights to all edges, as described in the first step of the algorithm.

```
int countEdge(int * graph, int dim){
  int count  = 0;
  for(int i = 0; i<dim*dim; i++){
    if(graph[i]>0){
      count++;
    }
  }
  return count/2;
}

void randomWeight(int * graph, int * newGraph, int dim, int edge){
  srand(time(NULL));
  for(int i = 0; i<dim; i++){
    for(int j = i; j<dim; j++){
      if(graph[i*dim+j]>0){
        newGraph[i*dim+j] = rand()%(2*edge)+1;
        newGraph[j*dim+i] = newGraph[i*dim+j];
      }else{
        newGraph[i*dim+j] = 0;
        newGraph[j*dim+i] = 0;
      }
    }
  }
}
```

Then we implement the 2nd step of the algorithm: Produce the Tutte matrix of the randomly weighted matrix.

```
for(int i = 0; i<dim; i++){
    for(int j = 0; j<dim; j++){
        if(graph[i*dim+j]!=0){
            if(i>j){
            newGraph[i*dim+j] = pow(2.0,(double) graph[i*dim+j]);
          }else{
            newGraph[i*dim+j] = -pow(2.0, (double) graph[i*dim+j]);
```

```
            }
        }else{
            newGraph[i*dim+j] = 0.0;
        }
    }
}
```

To compute the determinant serially, we first compute the LU decomposition of the Tutte matrix, and then multiply together the diagonal entries of the U matrix. All of these computations are done in a blackbox using functions from GNU C library.

The function `gsl_linalg_LU_det` perform a LU decomposition with partial pivoting. Given a matrix A, the function produces an upper-triangular matrix U, a lower-triangular matrix L, and a permutation matrix satisfying $LU = PA$. The matrices L and U are stored in the original input matrix `m`, where the upper triangular part of `m` is L, and the lower triangular part of `m` is U. The diagonal entries of L are all 1's, so these entries are omitted.

In the following code, first we copy the Tutte matrix to a `gsl_matrix`. And then we allocate space for the permutation matrix P. The sign of the permutation is defined to be $(-1)^n$, n is the number of nodes in the graph.

The following code shows how to set up the computation. If determinant for the Tutte matrix is 0, we output "No perfect matching." Otherwise, we use a simple While-loop to compute the weight of the minimum-weight perfect matching.

```
gsl_matrix *m = gsl_matrix_alloc(dim, dim);
for (int i = 0; i<dim; i++){
  for(int j = 0; j<dim; j++){
  gsl_matrix_set(m, i, j, weighted2[i*dim+j]);
  }
}
int* sign = (int *)malloc(sizeof(int));
sign[0] = (int) pow(-1, (double) dim);
gsl_permutation *p = gsl_permutation_alloc(dim);
gsl_linalg_LU_decomp(m, p, sign);
double d = gsl_linalg_LU_det(m, 1);
if(d==0){
  printf("No perfect matching.\n");
  return 0;
}
int d1 = d;
int b = 0;
int k;
while(1){
  k = d1%4;
  if(k!=0)  break;
  d1 = d1/4;
  b++;
}
```

To compute the adjoint matrix efficiently, we use the formula

$$A^{-1} = \frac{1}{|\det(A)|} adj(A)$$

Since we already have the determinant, we only need to compute the inverse of the Tutte matrix, and then multiply by the absolute value of the determinant. Note that the function `gsl_linalg_LU_invert` takes the permutation matrix `p` and the result of LU decomposition `m` as input, and the inverse matrix is stored in `inv`.
In the next step, the expression determining whether an edge is in the perfect matching is $2^{w_{ij}} \det(T_{ij})/2^{2W}$. Replace $\det(T_{ij})$ by $T_{ij}^{-1} \cdot \det(T)$. We know that $\det(T) = d1 \cdot 2^{2W}$, so we can simplify the formula to be $2^{w_{ij}} T_{ij}^{-1} \cdot |d1|$.
Therefore, we set the ij-th entry of T^{-1} to be $T_{ij}^{-1} \cdot |d1|$.

```
gsl_matrix *inv = gsl_matrix_alloc(dim, dim);
int y = gsl_linalg_LU_invert(m, p, inv);
gsl_matrix *madj = gsl_matrix_alloc (dim, dim);
double temp;
for (int i = 0; i<dim; i++){
  for(int j=  0; j<dim; j++){
      temp = (double)gsl_matrix_get(inv, i, j);
    gsl_matrix_set(madj, i, j, temp*abs(d1));
  }
}
```

The last step is to determine which edges are in the perfect matching. For the ij-th entry of T^{-1}, if it's non-zero, we multiply it by $2^{w_{ij}}$ and see whether it's odd. If the result is odd, we add it to the perfect matching. The calculation is simple using a for-loop.

```
int* result = (int *)malloc(dim*dim*sizeof(int));
int t;
double z, z1;
for (int i = 0; i<dim; i++){
  for(int j = 0; j<dim; j++){
    if(graph[i*dim+j]!=0){
      z = pow(2.0, (double) weighted[i*dim+j]);
      temp = (double)gsl_matrix_get(madj, j, i);
      temp = abs(temp)*z;
      t = (int) temp;
      if(t%2 == 1){
        result[i*dim+j] = 1;
      }
    }
  }
}
```

In the end, we print out the adjacency matrix `result[]` as the perfect matching we found.

8.3 Parallel Implementation

Producing random numbers using kernels is not trivial. We use cuRAND to create random number generators. After a setup kernel, the function `curand.uniform` generates a float for each thread.

Suppose the graph G has n vertices. The kernel `randomWeight` has n blocks, and n threads in each block. A thread with `blockIdx.x`$= i$ and `threadIdx.x`$= j$ represents the ij-th entry in the adjacency matrix.
Suppose we generate a random float a for ij-th entry above the diagonal in the adjacency matrix. The thread then assign 2^a to the ij-th entry and -2^a to the ji-th entry to produce the Tutte matrix.
This calculation is done only by entries above the diagonal, since the Tutte matrix is skew-symmetric. So, half of the threads are doing the calculation in parallel.

```
__global__ void setup_kernel(curandState *state, unsigned long seed){
  int id = threadIdx.x+blockIdx.x*blockDim.x;
  curand_init(seed, id, 0, &state[id]);
}

__global__ void randomWeight(int *graph, int dim, int edge, curandState
    * globalState){
  int Tutte = 0;
  if(blockIdx.x<threadIdx.x && graph[blockIdx.x*dim+threadIdx.x]!=0){
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    curandState localState = globalState[id];
    float RANDOM = curand_uniform(&localState);
    globalState[id] = localState;
    graph[id] = RANDOM*2*edge+1;
    Tutte = (int) pow(2.0, graph[blockIdx.x*dim+threadIdx.x]);
    graph[id] = Tutte;
    graph[threadIdx.x*dim+blockIdx.x] = -Tutte;
  }
}
```

The following lines are needed in the main function to call the kernels:

```
curandState *devStates;
cudaMalloc(&devStates, sizeof(curandState));
srand(time(0));
int seed  = rand();
setup_kernel<<<dim, dim>>>(devStates,seed);
randomWeight<<<dim, dim>>>(dev_a, dim, edge, devStates);
```

The strategy of computing determinant is the same as the serial version: We first compute LU decomposition of the Tutte matrix, and then we multiply together the diagonal entries of the U matrix. All of these computations are done in a blackbox using cuBLAS functions. Note that all the following code that set up the computation are in the main function.
`cublasSgetrfBatched` is the function that computes LU decomposition. It takes 7 inputs `(cublasHandle_t handle, int n, float *Aarray[], int lda, int *PivotArray, int *infoArray, int batchSize)`
Note that all the pointers must be copied from host to device in order for the function to work.
First we create a `cublasHandle`. The integer n and lda are both dimension of the matrix, so we already have that.

This function can compute LU decomposition for multiple matrices simultaneously, where `Aarray[i]` points to the i-th matrix. In our case, we only need to define `Aarray[0]` to be our Tutte matrix.
We allocate space for `PivotArray`, which is used for partial pivoting.
`infoArray` returns whether the LU decomposition is successful, so we also allocate space for that.
`batchSize` is equal to the number of matrices in `Aarray[]`, so we set it to 1.
This setup is mainly memory allocation and copying from host to device.
`cublasSgetrfBatched` stores the resulting LU decomposition in the same way as the function we used in the serial version. The matrix L is stored as the upper half of the original matrix, and U is the lower half of the original matrix, with diagonal entries of L omitted.

```
cublasHandle_t hdl;
cublasCreate(&hdl);
int *info;
cudaMalloc((void **)&info, sizeof(int));
int *infoH =(int *)malloc(sizeof(int));
int batch = 1;
int *p;
cudaMalloc((void**)&p, dim*sizeof(int));
float **ha = (float **)malloc(sizeof(float *));
ha[0] = dev_b;
float **a;
cudaMalloc((void**)&a, sizeof(float *));
cudaMemcpy(a, ha, sizeof(float *), cudaMemcpyHostToDevice);
cublasSgetrfBatched(hdl, dim, a, dim, p, info, batch);

cudaMemcpy(weightedF, dev_b, dim*dim *sizeof(float),
    cudaMemcpyDeviceToHost);
float d = 1;
for(int i = 0; i<dim; i++){
    d = d*weightedF[i*dim+i];
}
```

If determinant for the Tutte matrix is 0, we output "No perfect matching." Otherwise, we use the same function as the serial version to compute the weight of the minimum-weight perfect matching. (This part of the code is omitted because of repetition)

Similar to the serial implementation, we use the inverse matrix to compute the adjoint matrix. The function that compute the inverse is a cuBLAS function as well, and the input is the LU decomposition of the Tutte matrix, which we just produced. The following code is the part that sets up the function to compute the inverse matrix. Note that this part of the code is also in the main function.
`cublasSgetriBatched` is the function that computes the inverse. It takes 8 inputs: `(cublasHandle_t handle, int n, float *Aarray[], int lda, int *PivotArray, float *Carray[], int ldc, int *infoArray, int batchSize)`

`Carray[]` is used to store the output, which is the inverse of the original matrix.
Except `Carray[]` requires us to allocate new space, we can use the outputs from last step for all other inputs.

```
float **hc = (float **)malloc(sizeof(float *));
float **c, *c1;
cudaMalloc((void **)&c, sizeof(float *));
cudaMalloc((void **)&c1, dim*dim*sizeof(float));
hc[0] = c1;
cudaMemcpy(c, hc, sizeof(float *), cudaMemcpyHostToDevice);
cublasSgetriBatched(hdl, dim, a, dim, p, c, dim, info, batch);
```

The last step is to compute $(2^{w_{ij}} \det(T_{ij})/2^{2W})$ in a parallel way. This is not hard to implement:

```
__global__ void adjoint(int *graph1, float *graph, int *result, int dim
    , float det, int weight){
  int id = blockIdx.x*blockDim.x+threadIdx.x;
  float temp = ((float)graph1[id]) *graph[id]*det;
  int t = temp;
  if(t<0) {t = -t;}
  if(t%2==1){
    result[id] = 1;
  }else{
    result[id] = 0;
  }
}
```

The following lines are needed in the main functions to call the kernel:

```
int *resultH  = (int *)malloc(dim*dim*sizeof(int));
int *result;
cudaMalloc((void **)&result, dim*dim*sizeof(int));
cudaMemcpy(resultH, result, dim*dim*sizeof(int), cudaMemcpyHostToDevice
    );
adjoint<<<dim, dim>>>(dev_a, c1, result, dim, d1, i);
cudaMemcpy(resultH, result, dim*dim*sizeof(int), cudaMemcpyDeviceToHost
    );
```

In the end, we print out the adjacency matrix `resultH[]` as the perfect matching we found.

8.4 Running Time Analysis

The weight of each edge in the Tutte matrix range from 2^1 to $2^{2|E|}$, so the entries for LU decomposition and the inverse matrix get out of range very quickly. I tried to reduce the magnitude by dividing all entries by some number, but the exponential growth is too quick. As all entries for LU decomposition become inf or -inf, it's impossible to compute the determinant.
Therefore, the upper bound for the number of vertices in the graph is quite small.

When the graph has 8 vertices, the result becomes unreliable. A graph with 7 vertices cannot have a perfect matching.

In all the following calculations, the graph has 6 vertices, where I observe a reasonable chance of the result being a perfect matching.
Serial version: Total time is around 0.12-0.15 if it outputs a perfect matching. If there is no perfect matching, the running time is around 0.095.
Parallel version: Total time is around 228 if it outputs a perfect matching. If there's no perfect matching, the running time is mostly around 227.
Since the serial version runs a lot faster than the parallel version, We break the program into parts to see which part takes the most time in the parallel version:

Part 1: Random weight and Tutte matrix.
In the serial version, we time the function `randomWeight` and the for-loop that produces the Tutte matrix.
In the parallel version, we time the kernels `setup.kernel` and `randomWeight`, including the previous lines that allocate spaces and set up random number generators.
Serial version: 0.038
Parallel version: 2.5
If we time only the `randomWeight` kernel, the running time is 0.009, but as we include the first kernel, the running time is already around 2.48.
Thus, in Part 1, setting up the random number generator takes the most time. Allocating space and move from/to host to/from device also takes some time, but not as significant as `setup.kernel`.

Part 2: Determinant.
In the serial version, we time from the line we define a `gsl_matrix`, to before the line we determine whether the determinant is 0.
In the parallel version, we time from the line we define a `cublasHandle_t`, to before the line we determine whether the determinant is 0.
Serial version: 0.04
Parallel version: 225
So this is the part that takes the most time in the parallel version.
If we time only the line with `cublasSgetrfBatched`, the running time is 0.027, so the actual calculation takes very little time.
If we time right after the `cublasHandle` is created, the running time is 0.034, so allocating space and move from/to host to/from device takes some time. The step that takes the most time is to create the `cublasHandle`, which takes almost all of the 225 time.

Part 3: Inverse.
In the serial version, we time from the line we define a `gsl_matrix` to store the inverse matrix, to the line we use `gsl_linalg_LU_invert` to compute the inverse.
In the parallel version, we time from the line we define an array to store the

inverse matrix, to the line we use `cublasSgetriBatched` to compute the inverse.
Serial version: 0
Parallel version: 0.04
If we time only the line with `cublasSgetriBatched`, the running time is 0.016. However, the serial version almost take no time. The parallel version cannot beat that.

Part 4: Final calculation.
In the serial version, we time all lines that are involved with the calculation of $2^{w_{ij}} \det(T_{ij})/2^{2W}$.
In the parallel version, we time the kernel `adjoint`, which produces the result matrix.
Serial version: 0.01
Parallel version: 0.011

By breaking down the program into parts, we find out that in the parallel version, creating a `cublasHandle` takes about 225/228 of total time, and setting up a random number generator takes about 2.5/228 of total time. All the other parts take up very similar running time as their serial counterparts.

8.5 Potential Improvements

First we need to make the upper bound of the graph size significantly larger. A graph with 4 or 6 vertices is almost trivial. However, I do not have a good way to raise the upper bound unless we do not use GNU C library and cuBLAS.
If we do not use these functions as a blackbox to compute LU decomposition and the inverse, we could potentially do addition on the $\log_2$ of entries to compute the determinant. This may need additional running time, but it is less possible for the determinant and entries to get out of range.
To make the parallel version more efficient, we could utilize the fact that `cublasSgetrfBatched` and `cublasSgetriBatched` both can compute multiple matrices simultaneously. For any given graph, if we generate k Tutte matrices using more blocks, we could compute LU decomposition and inverse matrices for k matrices at the same time. Since this is a randomized algorithm, computing multiple matrices simultaneously can significantly reduce the time for finding the final perfect matching.

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p475-484. Print

[2] Stack Overflow, https://stackoverflow.com/questions/18501081/generating-random-number-within-cuda-kernel-in-a-varying-range

[3] Stack Overflow, https://stackoverflow.com/questions/22814040/lu-decomposition-in-cuda

[4] CURAND Library, https://docs.nvidia.com/cuda/curand/host-api-overview.html

[5] CUBLAS Library, https://docs.nvidia.com/cuda/cublas/index.html

[6] 14.1 LU Decomposition, https://www.gnu.org/software/gsl/manual/html_node/LU-Decomposition.html

9 By Kaimin Liu, on String Matching

Project Description: Given a text string $T(1:n)$ and a pattern string $P(1:m)$, the string-matching problem is to find all the occurrences of the pattern in the string, which is, if $P(1:m)$ occurs in T at position i, $P(j) = T(i+j-1)$, for $1 \leq j \leq m$.

9.1 Possible Solutions

9.1.1 Naïve String-search Algorithm

One method that people will intuitively come up with when first reaching this problem would be the brute-force algorithm, or called **Naïve string-search algorithm**. The brute-force algorithm would compare each position of pattern with text from the most left to right. In another word, it tries to match $P(1:m)$ against $T(i:i+m-1)$ for each position i, where $1 \leq i \leq n$. Clearly, $\Theta(m)$ operations are required for each such position from 1 to n, and totally the algorithm would use $O(nm)$ operations.

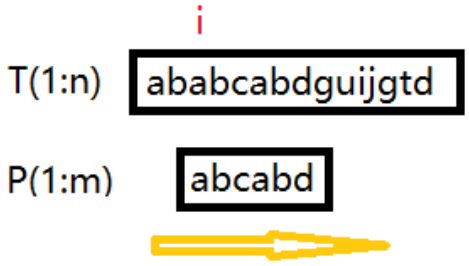


Figure 6: Comparing from T(i) to T(i+m-1) to match with P(1:m)

9.1.2 The Knuth-Morris-Pratt Algorithm—

A classic linear-time algorithm that Joseph Jaja implemented in Introduction to Parallel Algorithm is called **The Knuth-Morris-Pratt algorithm (KMP Algorithm)**. In this algorithm, we use the failure function F defined by $F(j) = j - D(j)$, where $1 \leq j \leq m+1$ and $D(j)$ is the period of the prefix $P(1:j-1)$. Then we still compare each position of pattern $P(j)$ with each position of text $T(k)$ from the most left to right, but differently, this time we don't need to go back to $T(i+1)$ like in Naïve string-search algorithm. We can obtain the next possible candidate position by sliding the pattern $D(j)$ places to the right and setting $j = j - D(j)$. According to Jaja, the algorithm terminates after $O(n)$ sequential steps.

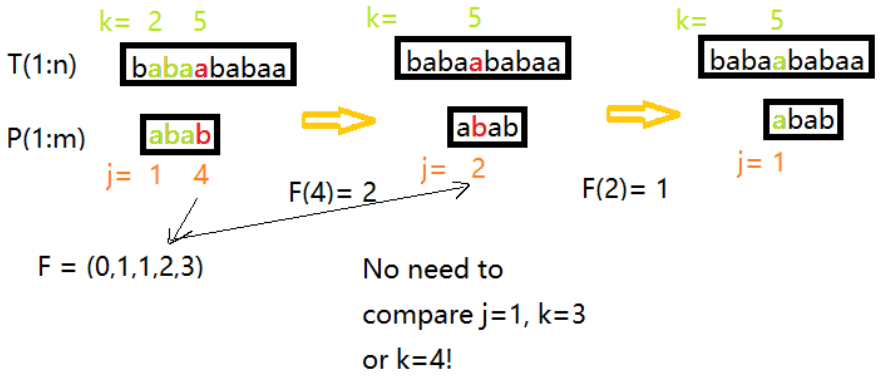


Figure 7: One part of using KMP search

9.1.3 Boyer-Moore Algorithm

Another linear-time algorithm that is briefly mentioned in Introduction to Parallel Algorithm is called **Boyer-Moore algorithm**. BM Algorithm is similar to KMP Algorithm in that it also skips alignments by first examining the information of pattern, but it compares from the end of the pattern, that is, starting at index $P(n)$ and $T(k)$ and moving backwards. The detailed steps will be mentioned in the following chapters.

9.1.4 Parallel Algorithm

Joseph Jaja also gave an alternative strategy that reduces the total number of operations to $O(n)$ by using **parallel algorithm**, which combined the algorithms of solving both nonperiodic and periodic cases.
For nonperiodic pattern, the parallel algorithm will partition T into $\frac{2n}{m}$ blocks Ti, and for each block select one position as a possible candidate, and for each candidate position, verify if P occurs at that position in T using brute-force algorithm.
For periodic pattern, the nonperiodic strategy is first applied finding all the occurrences of $P(1 : 2p - 1)$, and then we keep the occurrences i where u^2v occurs, where $u = P(1 : p)$, $v = P(kp + 1 : m)$, and $k = m/p$, and mark them with $M(i) = 1$. Then we generate p subarray of M consisting of the bits $(M(i), M(i + p), M(i + 2p), ...)$. For each position l of Si that contains a 1, set $C(i, l) = 1$ if there are at least $k - 1$ consecutive 1's starting at this position. For all the remaining positions $C(i, l) = 0$. For each $1 \leq j \leq n - m + 1$, let $j = 1 + lp$, where $1 \leq i \leq p$ and $l \geq 0$. Then we set $MATCH(j) = C(i, l + 1)$, and P occurs at $T(j)$ where $MATCH(j) = 1$.

9.2 Code for Brute-force algorithm

The brute-force serial code would be very simple. We just loop around $T(1 : tsize - psize)$ and $P(1 : psize)$. When one position i is checked to be the occurrence of the pattern, $match(i)$ is set to be 1. At last, an array match would show all the occurrences of the pattern.

```
__host__ void b_f_serial(char* p, int psize, char* t, int tsize, int*
    match){
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i <= tsize - psize) {
        int flag = 1;
        for (int j = 0; j < psize; j++) {
            if (t[i + j] != p[j])
                flag = 0;
        }
        match[i]=flag;
    }
}
```

9.3 Code for KMP algorithm

For KMP algorithm, we will first generate the F(1:psize) array. Then, starting from T[1] and P[1], we move the pointer i and j each time, comparing t[i] and p[j]. If they don't match, j will jump to f[j-1] in this code.

```
__host__ void KMP_serial(char* p, int psize, char* t, int tsize, int*
    match)
{
    int f[M];//calculate f[] array
    Fgenerator(p, psize, f);

    int i = 0;
    int j = 0;
    while (i < tsize) {
        if (p[j] == t[i]) {
            j++;
            i++;
        }

        if (j == psize) {
            match[i]=1;
            j = f[j - 1];
        }

        // mismatch after j matches
        else if (i < psize && p[j] != t[i]) {
            if (j != 0)
                j = f[j-1];
            else
                i = i + 1;
        }
    }
}

```

```
__host__ void Fgenerator(char* p, int psize, int* f)
{
    int j = 0;
    f[0] = 0; // f[0] is always 0
    int i = 1;
    while (i < psize) {
        if (p[i] == p[j]) {
            j++;
            f[i] = j;
            i++;
        }
        else // (p[i] != p[j])
        {
            if (j != 0) {
                j = f[j - 1];

            }
            else // if (j == 0)
            {
                f[i] = 0;
                i++;
            }
        }
    }
}
```

9.4 BM algorithm

9.4.1 Shift Rules

As mentioned before, BM algorithm has its own shift rules that makes it efficient. There are two rules: the bad character rule and the good suffix rule. The actual shifting offset is the maximum of the shifts calculated by these rules.

The bad-character rule considers the character in T at which the comparison process failed (assuming such a failure occurred). The next occurrence of that character to the left in P is found, and a shift which brings that occurrence in line with the mismatched occurrence in T is proposed.

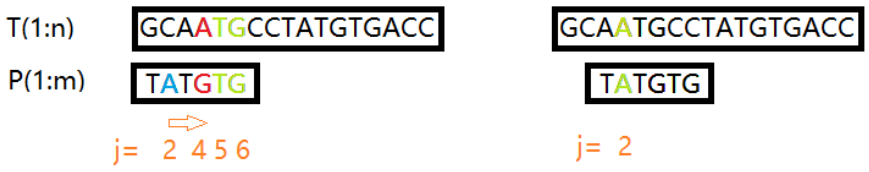


Figure 8: Let mismatch become match

If the mismatched character does not occur to the left in P, a shift is proposed that moves the entirety of P past the point of mismatch.

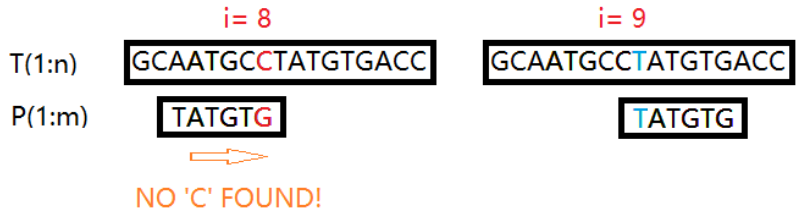


Figure 9: If no match then pass the whole pattern

The good suffix rule is the comparisons begin at the end of the pattern rather than the start.
Let t be the substring of T that matched a suffix of P. Skip alignments until
(a) t matches opposite characters in P, or
(b) a prefix of P matches a suffix of t, or
(c) P moves past t,
whichever happens first.

Figure 10: Case(a), case(b) and case(c)

9.4.2 Code for Preprocessing using Bad-character Rule

Before processing BM algorithm, we will first build a function to produce an array of shift using bad-character rule.

```
__host__ void Badcharacter(char *x, int m, int bmBc[]) {
   int i;
   for (i = 0; i < ASIZE; ++i)
      bmBc[i] = m;
```

```
5     for (i = 0; i < m - 1; ++i)
6        bmBc[x[i]] = m - i - 1;
7  }
```

9.4.3 Code for Preprocessing using Good-suffix Rule

Then, we will build a function to produce an array of shift using good-suffix rule. A suffixes function will return an array that stores the length of suffixes.

```
1  __host__ void suffixes(char *x, int m, int *suff) {
2     int f, g, i;
3     suff[m - 1] = m;
4     g = m - 1;
5     for (i = m - 2; i >= 0; --i) {
6        if (i > g && suff[i + m - 1 - f] < i - g)
7           suff[i] = suff[i + m - 1 - f];
8        else {
9           if (i < g)
10             g = i;
11          f = i;
12          while (g >= 0 && x[g] == x[g + m - 1 - f])
13             --g;
14          suff[i] = f - g;
15       }
16    }
17 }
18
19 __host__ void Goodsuffixes(char *x, int m, int bmGs[]) {
20    int i, j, suff[XSIZE];
21
22    suffixes(x, m, suff);
23
24    for (i = 0; i < m; ++i)
25       bmGs[i] = m;
26    j = 0;
27    for (i = m - 1; i >= 0; --i)
28       if (suff[i] == i + 1)
29          for (; j < m - 1 - i; ++j)
30             if (bmGs[j] == m)
31                bmGs[j] = m - 1 - i;
32    for (i = 0; i <= m - 2; ++i)
33       bmGs[m - 1 - suff[i]] = m - 1 - i;
34 }
```

9.4.4 Code for combing the two rules together

After all the preparing work, when shift is needed, we can choose the greater shift calculated by the two rules and apply it to i.

```
1  __host__ void BM(char *p, int m, char *t, int n, int* match) {
2     int i, j, bmGs[XSIZE], bmBc[ASIZE];
3      //preprocessing
4     Badcharacter(p, m, bmBc);
5     Goodsuffixes(p, m, bmGs);
```

```
6
7     i = 0;
8     while (i <= n - m) {
9        for (j = m - 1; j >= 0 && p[j] == t[i + j]; --j); //compare from
       the end until difference happens
10       if (j < 0) { //pattern is found in text
11          match[i] = 1;
12          i += bmGs[0];
13       }
14       else  //shift is needed
15          i += max(bmGs[j], bmBc[t[i + j]] - m + 1 + j);
16    }
17 }
```

9.5 Further Ideas

This time I focus on the BM algorithm, but I didn't find much relationship to parallel computing. What if parallel computing could be implemented in BM algorithm, would that be more efficient? How could it be realized?

9.6 References

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p311-356. Print

[2] "Boyer–Moore string-search algorithm." Wikipedia, The Free Encyclopedia, 2 Mar. 2019. Web. 13 May. 2019.

[3] "KMP Algorithm for Pattern Searching" GeeksforGeeks, 7 May 2019, geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

[4] "Boyer-Moore algorithm", www-igm.univ-mlv.fr/ lecroq/string/node14.html

10 By Nate Hayes, on the Convex Hull Algorithm

10.1 Problem

: Given a set of points lying on a plane, we want to find the smallest possible convex hull that contains all of the points.

10.2 Definitions & Topic Overview

: A convex hull can be defined as follows: A set $P \subseteq \mathbf{R}^d$ is convex if $\overline{pq} \subseteq P$, for any $p, q \in P$. This means that for any given point, there exists a line segment to every other point that lies within the polygon region.

Another property of a convex hull is that all of the interior angles of the polygon are less than 180°. Therefore, if one were to travel around the edge of the convex hull in a counterclockwise manner, every change in direction in the hull would be a left turn.

Each point located along the exterior of the convex hull is called an extreme point or a vertex of the polygon.

10.3 The Serial Implementation using Graham's Algorithm

:

For the CPU, I opted to go with the Graham Algorithm due to its ability to create a complex hull in O(n*logn) time.

A notable feature of the Graham algorithm is that points that form the convex hull are stored in a stack data structure. After the initial point is determined (often the point with the minimum y-value), it is added to the stack. When a subsequent point is encountered, it is immediately added to the stack. However, if the point is later determined to lie within the convex hull, points can be popped off the stack in order to backtrack.

Additional functions can be used to optimize the algorithm and eliminate points that cannot lie on the convex hull—if two points have the same angle, the point located closer towards the center of the convex hull is an internal point and can be disregarded.

PHASE 1: Sorting Coordinate Points
The first phase of the algorithm is to sort the coordinate points. Since the Graham algorithm sequentially travels around the coordinate plane to ensure that all points lie within the convex hull, the points must be arranged in such a way that equips the algorithm to traverse the points in the right order. The function above details the way in which points are compared to other neighboring points. The function sorts values based on their x-values. Using this function, I was able to run all of the points through a quicksort—qsort()— function to properly

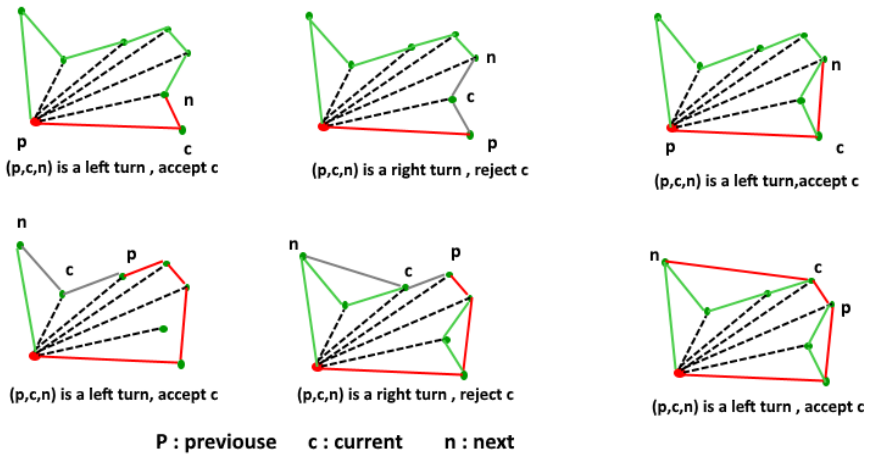


Figure 11: In the diagram, one can observe the algorithm traversing a set of points. Points are pushed and popped onto the stack as the algorithm determines whether they form the convex hull.

arrange the points inside an array. In all, the sorting step of the algorithm takes O(n*logn) since each point must be processed (n) and the quicksort function takes log(n) for each point.

```
bool counterclockwise(const Point *a, const Point *b, const Point *c) {
    return (b->x - a->x) * (c->y - a->y) > (b->y - a->y) * (c->x - a->x
    );
}
```

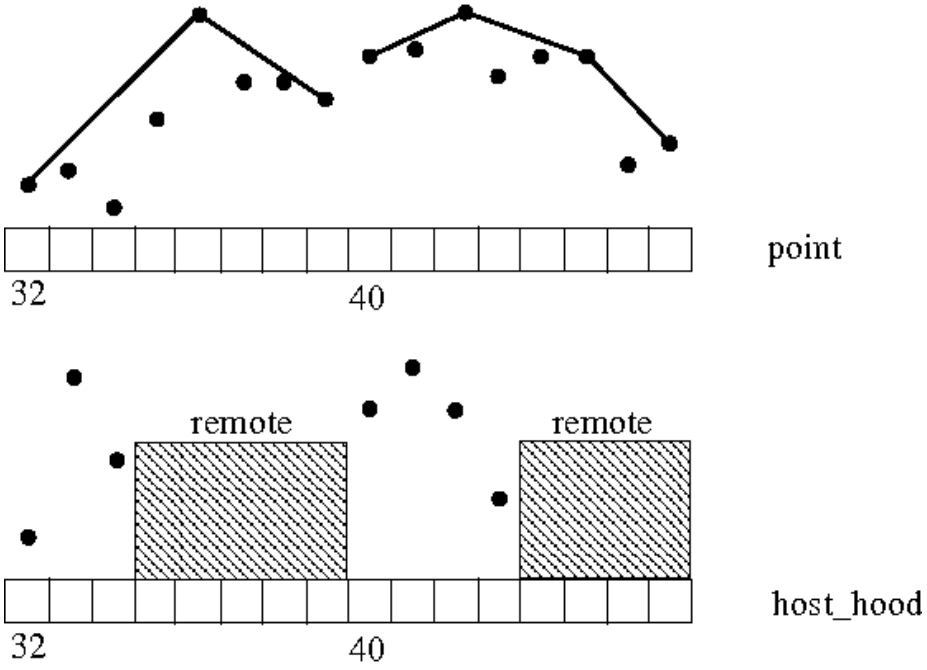
PHASE 2: Accept/Reject Coordinate Points

The above function is also used for determining whether the coordinate point belongs along the exterior of the convex hull. Points that belong on the exterior of the hull are stored in the stack. The function computes the polar angle of the two most recent points on stack and the latest point in the array. If the function returns True, the points form a counterclockwise turn and could potentially lie on the edge of the convex hull. If the function returns False, the newest point is discarded and the next point is tested. In this phase, each coordinate point is either pushed or popped onto the stack at most one time and is never revisited. Since push and pop operations take O(1), and every point is visited, we can conclude that the runtime of the phase takes O(n).

```
int comp(const void *leftHandSide, const void *rightHandSide) {
    Point leftPoint = *((Point *)leftHandSide);
    Point rightPoint = *((Point *)rightHandSide);
    if (leftPoint.x < rightPoint.x){
        return -1;
    }
    if (rightPoint.x < leftPoint.x){
        return 1;
    }
    return 0;
}
```

10.4 The Parallel Version using Wagener's Algorithm

: Since Graham's algorithm is calculated sequentially, I was forced to start over when I attempted to create a parallel version of the algorithm to run on the GPU.



The algorithm functions by creating hoods which blanket interior points of the convex hull. Once the hoods have been created to contain the various coordinate points, the bulk of the algorithm is used to create tangent points between the various hoods. Tangent points between the hoods are guaranteed to

Part 1: Finding Hoods

```
__device__ short f( float2 * hood, short i, short j,short start, short
    d ){
    float2 p, q, p_next, p_prev;
    int atstart, atend;int isleft;
    if ( hood [i] . x > 1 ){
        return HIGH;
    }
    p = hood[i];
    q = hood[j];

    atend = ( i == start + d - 1 || hood[i+1].x > 1 );
    p_next = hood [ i+1-atend ];
    p_next.y -= (float) atend;
    if ( left_of ( p_next, p, q ) ){
        return LOW;
    }
    atstart = ( i == start );
    p_prev = hood[ i + atstart - 1 ];
    p_prev.y -= (float) atstart;
    isleft = left_of ( p_prev, p, q );
    return HIGH * isleft + EQUAL * (1-isleft);}
```

Part 2: Match and Merge

```
j = scratch[start+d+x];
if ( hood[i].x <= 1.0 &&
    f(hood,i,j,start,d) <= EQUAL &&
    ( x == d1-1 ||hood[i+d2].x > 1.0 ||
    f(hood,i+d2,scratch[start + d + x + 1],start,d) == HIGH))
        scratch[start] = i;
__syncthreads();
```

Although just a small portion of the match and merge algorithm, the function above attempts to calculate a tangent line that encompasses hood[i] and hood[i+d2]. The three possible values that can be returned from this section of code are LOW, EQUAL, and HIGH. In this function, LOW is returned if the proposed tangent line connecting hood p to hood q is too low and cuts out at least one point from hood p. HIGH is returned if the proposed tangent line connecting hood p to hood q is too low and cuts out at least one point from hood q. EQUAL is returned if the proposed tangent line connection hood p to hood q properly encompasses points from the two hoods. Only in the case of an EQUAL are the two hoods combined under the common tangent line. Once this has been completed, the process is then continued for the next neighboring hood.

The overall time complexity of this algorithm is O(n*logn). Hoods initially consist of d number of elements. However, during each iteration of the match and merge function, the number of elements in the hood is doubled. Additionally, to calculate the correct tangent line between the two hoods, O(n) number

of operations are run. Therefore, the total time complexity of the algorithm is O(n*logn).

I initially chose to follow Wagener's Algorithm due to its O(n*logn) time complexity. I was especially interested in the algorithm because the O(n*logn) algorithm can even be improved to an O(logn) runtime if the right steps are taken; perhaps I was a little bit too optimistic in this regard because I ran out of time to work on this advanced optimization technique.

Challenges: Storage of data in same location Math behind finding tangent lines

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p475-484. Print

[2] Convex Hull—Set 2 Graham Scan, https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/

3CUDA implementation of Wagener's 2D convex hull PRAM Algorithm, https://www.researchgate.net/publicat

11 By Andrew Willis, on Finding the Lower Envelope

Git url: github.com/12345aaw/Parallel-Computing-Final-Project.git

The premise of the visibility problem that when surfaces or shapes overlay each other, we must figure out which shapes to display first so that the closest shapes appear in front of further shapes. Algorithms that solve the visibility problem have great application in the 3D graphics area. Depth buffering algorithms will display the surfaces that have the least distance from the viewpoint. Similar algorithms have also been implemented in robotics for object-detection purposes. We will look at a solution to a simplified version of the visibility problem in the form of *finding the visibility polygon* on a 2D plane.

11.1 The 2D Visibility Polygon

Given a set S of line segments and a point p in the plane, a point q is visible from p if the line segment pq does not intersect any segments in S. The points visible from p form a (possibly unbounded) polygon called the **visibility polygon** of p.[1]

11.2 Problem

Our problem is a simplified case of finding the visibility polygon of p given that line segments in S do not intersect. To keep the algorithms simple, we will also assume that all points on the grid take integer values. Additionally, all y values will be on the $\mathbb{N}$ set of numbers. This allows us to consider specifically the visibility polygon to be viewed from a point on the line $y = -\infty$.

Figure 12 serves as an example of the aim of our solution to the visibility problem. When viewed from below all of the segments, some segments, like s_1 would remain unaltered, because they do not share the same x-coordinates with any other segments. However when we look at segments s_2, s_3, and s_4, we see that the segments that are underneath other segments (like how s_3 is under s_2) take precedence and are displayed. A portion of a segment that has *any other* segments underneath it will not be displayed.

11.3 The Lower Envelope Algorithm

Given the constraints, the lower envelope algorithm can best be summarized as finding the lowest segment which also intersects the given x value for each x value on the grid. A simple, naive algorithm can be generated simply by checking every segment for the lowest y value, between every sequential pair of integer x values. This brute-force method may not be viable for sets with a

[1] Jaja, Joseph, Introduction to Parallel Algorithm, Addison-Wesley Publishing Company, 1992. p285-291

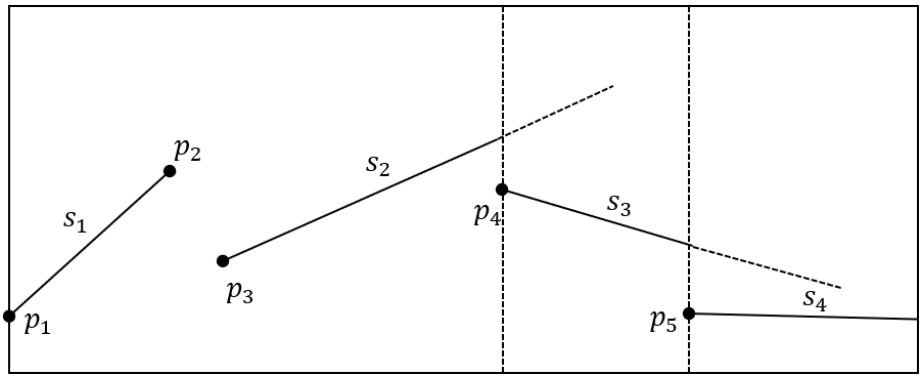


Figure 12: In this example, the lower bound is formed when endpoints p_4 cuts off the rest of s_2 and p_5 cuts off s_3.

large number of segments and grids with a high range of x values.
A faster algorithm can be made by using a similar divide and conquer technique to merge sort.

1. Compute trivial lower envelope of each segment separately
2. Compute lower envelopes of pairs of segments
3. Compute lower envelope of pairs of pairs of segments
4. Continue until lower envelope of all segments is found

Unfortunately, this faster algorithm will not be implemented here, as it relies on a different set of constraints than the ones we'll be using for our algorithm. The differences between the faster algorithm's constraints and ours can be found in the Further Improvements section. For a more in depth perspective on how this algorithm works, refer to Joseph Jaja's *Introduction to Parallel Algorithm.*

11.4 Algorithm Structure

- $S = \{s_1, s_2..., s_n\}$ is a set of non-intersecting line segments
- $vis = (+\infty, s_1, +\infty, s_2, s_3, s_4, +\infty, s_5, +\infty)$ where vis is the sequence of visible segments. $vis[i] = +\infty$ when there is no segment visible in the interval
- p_{i1}, p_{i2} are endpoints of s_i such that $x(p_{i1}) < x(p_{i2})$

11.5 Naive Serial Version

The most basic implementation of the lower envelope algorithm is a simple brute force to find the 2D visibility polygon. The algorithm compares the y values of

every segment for every area (the area between 2 consecutive integer x-values), thereby ensuring that the lower bound y value is found for an area of all the segments that intersect the area. Shown below is an implementation of the algorithm, written in C.

```
int * lower_envelope(int * coordinate_array, int size){
  int number_areas = global_num_areas_serial;
  int * min_y = (int *) malloc(number_areas * sizeof(int));
  for(int i = 0; i < number_areas; i++){
    min_y[i] = 99999;
    for(int j = 0; j < size - 1; j = j + 4){
      if(isBetweenPointsInclusiveSerial(coordinate_array[j],
    coordinate_array[j+2],i,i+1) && coordinate_array[j+1] < min_y[i]){
        min_y[i] = coordinate_array[j+1];
      }
    }
  }
};
```

Note: min_y is initialized to 99999 as a substitute for ∞

11.5.1 Code for checking for a point between two coordinates

The IsBetweenPointsInclusive helper function is a simple algorithm that checks whether a point exists between two x coordinates.

```
bool isBetweenPointsInclusiveSerial(int a, int b, int x1, int x2){
  if( (a <= x1 && b >= x2) || (a >= x1 && a <= x2) || (b >= x1 && b <=
    x2) ){
    return true;
  }
  return false;
};
```

11.5.2 Runtime complexity

Given that each area (the space between two, consecutive integer valued x coordinates) is iterated over by every segment on the graph, the run-time complexity of the native algorithm is:

$$\mathcal{O}(N * M) \tag{2}$$

where N is the number of areas and M is the number of segments on the graph. This can be calculated trivially, as the size of the work is solely dependent on the two nested for loops, one that loops M times and the other looping N times, Everything else in the code is assignment and conditions. The isBetweenTwoPoints helper function is also a simple conditional check that will always run in constant time.

11.6 Naive Parallel Version

This naive version can be parallelized easily by having different threads each find min_y values for different areas.

```
__global__ void parallel_lower_envelope(int * coordinate_array, int
    number_of_threads, int coordinate_array_size, int * min_y){
  // Number of areas between x values
  int number_areas = global_num_areas;
  // For each area between points
  int numElementsPerThread = number_areas / (gridDim.x * blockDim.x);
  int cumulative_thread_id = threadIdx.x + (blockDim.x*blockIdx.x);
  int startPos = cumulative_thread_id * numElementsPerThread;
  // Each thread computes their own min_y index
  for(int i = 0; i < numElementsPerThread; i++){
    int i_thread = i + startPos;
    min_y[i_thread] = 99999;
    // For each segment update the min_y array if betweeen segment
    exists between the points x = i and x = i+1
    for(int j = 0; j < coordinate_array_size - 1; j = j + 4){
      if(isBetweenPointsInclusive(coordinate_array[j],coordinate_array[
    j+2],i_thread,i_thread+1) && coordinate_array[j+1] < min_y[i_thread
    ]){
        min_y[i_thread] = coordinate_array[j+1];
      }
    }
  }
  __syncthreads();
};
```

This parallel version is run on the GPU as opposed to the serial version running on the CPU. The number of computations that each threads makes is dependent on the number of total areas and the number of total threads. Specifically, each thread will make $\#areas/\#threads$ computations. At the end of the function, the threads do not necessarily have to be synced, as none are accessing the same areas, but it is useful to sync if you wish to print the polygon after the *syncthreads()*.

11.7 Runtime Analysis

$\log_2$ **of input size**	23	24	25	26
Serial Version	221.515991	435.276001	903.954956	1720.340942
Parallel Version	0.460800	0.868352	1.724192	3.496416
Speedup kernel vs serial	481	501	524	492

Table 8: Run time for finding the lower envelope. All times are in milliseconds.

Informal tests were also run to decide the best number of threads and blocks to run the Parallel Version on. At $log_2(26)$ input size, 512 blocks at 32 threads per block seemed to provide the best results, and so all formal tests were run on 512 blocks at 32 threads per block. Also for the sake of proofing, the segments

were held constant. They scale at the same magnitude as the number of areas (whose numbers we did manipulate), and by prewriting a set of segments, we could verify that the algorithm was, in fact, generating the lower-bound of the segments.

11.8 Further Improvements

The lower envelope algorithm can be expanded upon primarily in four ways.

1. Firstly, the algorithm's constraints can be made more general in several facets. A far more useful (but necessarily more complicated) algorithm would allow for intersecting line segments and would take float values.

2. The second useful expansion can be to use the algorithm in three dimensions. This is the most common form that visibility algorithms take, because of their necessity in displaying complex diagrams in 3D graphics.

3. The third and most complicated way we can expand the algorithm is to forego our assumption that the segments are viewed from a point on the line $y = \infty$ and to make the algorithm work from any vantage point (even, possibly a moving point). While such an algorithm would be supremely useful for somebody studying 3D graphics and its applications, such an algorithm is beyond the scope of this course and requires a very strong foundation in linear algebra.

4. Finally, the algorithm can be made to be faster into $O(n * log(n))$ time with a divide-and-conquer strategy like the one listed in Jaja's *Introduction to Algorithm* textbook. The reason it could not implemented under our problem's specifications is that our problem finds the lower bound for each x coordinate. Jaja's problem specification finds the lower bounds only at segment endpoints, which can be solved more efficiently with a merge sort.

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p285-291. Print

[2] Stack Overflow, https://stackoverflow.com/questions/14038589/what-is-the-canonical-way-to-check-for-errors-using-the-cuda-runtime-api

12 By Justin Kim, on Parallel Algorithms for Forming Binary Trees

12.1 Introduction: The Problem

Link to GitHub Repository: https://github.com/kimavz/ParallelFinalProjectKim

Many algorithms use trees as their primary data structures. For instance, many search algorithms use binary search trees. Trees are also used in structures like heaps, which are then used to implement priority queues. Some graph search algorithms also exist, such as Breadth-First Search (BFS) and Depth-First Search (DFS). The goal of this project was to find a way to implement trees as a data structure and apply them to parallel algorithms and other CUDA programs. Binary trees are not something that one would initially associate with CUDA, or using a parallel algorithm, so I thought it would be a good idea of trying to run an algorithm for it.

Here, I will describe one of the ways that I used CUDA to implement binary trees. Also, I will be describing some of the challenges and problems that we run into in implementing binary trees in CUDA and parallel coding.

12.2 Serial Implementation

The serial implementation of a binary tree is quite simple and was something that we learned back when we were learning C. If you want to create a binary tree in the CPU, we would create something that looks like this:

```
struct node
{
  int data;
  struct node *left;
  struct node *right;
};
```

In the CPU, a tree would be represented by a pointer to the topmost node (the root) in a tree. If a tree is empty, then the value of the root is NULL. A tree node will contain the following parts: the data, a pointer to the left child, and a pointer to the right child. If you want a tree with more than two children (a n-ary tree instead of a binary tree), then you would have to include more nodes in the structure to represent more children.

It is easy to see how one would execute all of the simple binary tree functions using a structure like this one. Leaves would be categorized by the nodes that have NULL for both of their children nodes. If you want to add a node to this tree, you would have to traverse the tree to find the node you want to add too and then modify the child node as necessary. This is nothing new, and is something that we have done various times in other classes.

Below is a way to create some simple trees using the structure that we have described above:

```
struct node
```

```
{
    int data;
    struct node *left;
    struct node *right;
};


struct node* newNode(int data)  {
  struct node* node = (struct node*)malloc(sizeof(struct node));

  node->data = data;

  node->left = NULL;
  node->right = NULL;
  return(node);
}


int main() {
  struct node *root = newNode(1);
  /* 1 is the root of the tree */

  root->left        = newNode(2);
  root->right       = newNode(3);
  /* 2 and 3 become left and right children of 1 */

  root->left->left  = newNode(4);
  /* 4 becomes left child of 2 */
}
```

12.3 Challenges with Parallel Implementation

Now that we have an idea of what binary trees would look like in a serial implementation, we now want to find a way to implement binary trees in a parallel implementation. The most obvious way that one might approach this is to assign the nodes of our potential tree to threads. The various threads would then act independently and hopefully create something that resembles a binary tree. However, as we will soon see, there will be a few problems with creating a binary tree with this method.

12.3.1 Divergence

The biggest problem of trying to use threads as nodes and then creating a tree that way is the problem of divergence. Divergence is a measure of whether nearby threads are doing the same thing or different things. There are two main types of divergence: execution divergence which means that the threads are executing different code or making different control flow decisions, or data divergence which means that they are reading or writing disparate locations in memory.

This is not a problem in creating trees in the CPU. Since, we are individually creating each node and placing them in the structure of the tree, we don't

run into issues of wanting to put multiple tree nodes in a certain spot. In other words, trees on the CPU are dynamically allocated. However, this is a bit of a problem with the GPU. The most obvious problem with a recursive implementation is high execution divergence. Each thread will be running, trying to get their value to a node in a tree. However, the decision of whether to skip a given node or recurse to its children is made independently by each thread, and there is nothing to guarantee that nearby threads will remain in sync once they have made different decisions.

12.4 Parallel Implementation: The Idea

So, how exactly will we run and create binary trees in parallel, or in the GPU? We do not want to be using dynamic allocation on our GPU kernels because of the issues that we stated above. The idea that we want to use is to assign each integer to a thread (like we have described above). However, the order in which the threads will execute will be beyond our control. So, we must be okay with different tree structure ultimately representing the same data.

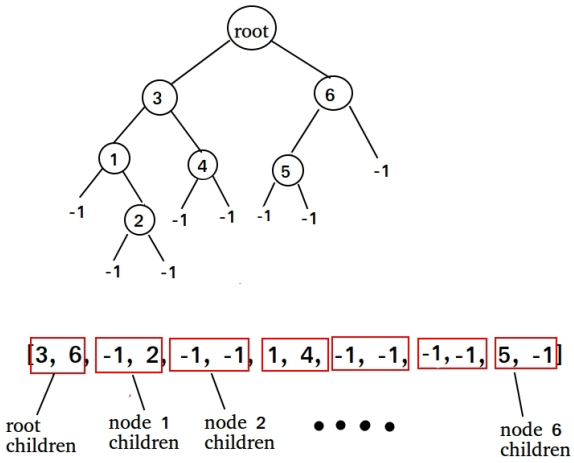
12.4.1 Node Insertion Algorithm

Adding a node to a tree will involve two main steps: 1. Traversing the partially constructed tree until you hit a leaf node, and 2. Adding the new node to the tree. We can do Step 1 in parallel, while we do Step 2 sequentially. So, each thread can traverse the tree in parallel, but the threads must insert their values one at a time. We want to use our known synchronization methods to ensure that our integers are added one at a time.

12.4.2 Binary Tree Structure in CUDA

The idea that we will use is that we will only record the children of all of the nodes. Because every node is itself a child node of its parent, this effectively means that we will store every node except the root in our structure. NULL nodes will be represented with a -1. Every pair in the structure will represent a node. The first element in the pair is the left child and the second element is the right child.

Suppose we had the array of integers $C = [5, 6, 3, 4, 2, 1]$. An example of a binary tree structure of this array can be found below:



As we can see, the array representation of the tree will have 14 elements. This is because the array C has 6 nodes, which gives us a total of 12 elements, but we will need 2 more for the children of the root node.

We can implement this idea and structure with to a tree of any number of nodes. Of course, when we can have different trees that can represent the same array of nodes.

12.5 Parallel Implementation: The Code

My code will create binary trees using the structure that we have described above. It will use a parallel algorithm and the GPU to insert the nodes into the tree. We will go step-by-step at each part of the code to explain what it is doing.

12.5.1 The Main

```
1   int *h_x;
2   int *d_x;
3   int *h_root;
4   int *d_root;
5   int *h_child;
6   int *d_child;
7
8   h_x = (int*)malloc(n*sizeof(int));
9   h_root = (int*)malloc(sizeof(int));
10  h_child = (int*)malloc(2*(n+1)*sizeof(int));
11  cudaMalloc((void**)&d_root, sizeof(int));
12  cudaMalloc((void**)&d_x, n*sizeof(int));
13  cudaMalloc((void**)&d_child, 2*(n+1)*sizeof(int));
14  cudaMemset(d_child, -1, 2*(n+1)*sizeof(int));
```

Here, we are allocating space for the structure of our tree. n is the number of nodes that we have. h_x is the host array and d_x is the device array.

```
for(int i=0;i<n;i++){
    int j = random() % (n-i);
    int temp = h_x[i];
    h_x[i] = h_x[i+j];
    h_x[i+j] = temp;
  }
  *h_root = h_x[0];

  for(int i=0;i<n;i++){
    printf("%d ", h_x[i]);
  }
  printf("\n");
```

This part of the code randomly shuffles the nodes that we have, representing the fact that we are randomly assigning threads to each node.

The rest of the code is call the kernel code on the inputs that we have created here.

12.5.2 The Kernel

```
int childPath;
int temp;
offset = 0;
while((bodyIndex + offset) < n){

  if(newBody){
    newBody = false;

    temp = 0;
    childPath = 0;
    if(x[bodyIndex + offset] > rootValue){
      childPath = 1;
    }
  }
  int childIndex = child[temp*2 + childPath];

  // traverse tree until we hit leaf node
  while(childIndex >= 0){
    temp = childIndex;
    childPath = 0;
    if(x[bodyIndex + offset] > temp){
      childPath = 1;
    }

    childIndex = child[2*temp + childPath];
  }

  if(childIndex != -2){
    int locked = temp*2 + childPath;
    if(atomicCAS(&child[locked], childIndex, -2) == childIndex){
      if(childIndex == -1){
        child[locked] = x[bodyIndex + offset];
      }

```

```
35        offset += stride;
36        newBody = true;
37      }
38    }
39
40    __syncthreads();
41  }
```

This is the main part of our code. The first if-statement is checking each thread and then adding to the tree depending on the root value. The next while-loop places the nodes if the thread encounters a leaf. Finally, the next if-statement accounts for the case when two threads reach the same leaf. The code then locks one of the threads, and the other threads skip the if-statement and then go to *__syncthreads*() and then waits.

12.6 Further Ideas and Improvements

Clearly, this implementation is not perfect, and there is a lot of room for improvement here. For instance, this implementation works best when it takes elements that are integers from 1 to n. We can potentially look to improvements in making the info of our nodes not be integers, and instead be something else, like strings.

Also, we can see that structure of the tree may be inefficient in some ways as well. For instance, suppose that our tree had an abnormal amount of NULL children nodes. Then, the mode that we constructed here would not be very efficient, and perhaps the structure could use some improvements.

Finally, we notice that binary trees are not usually the type of structures that one might implement in CUDA or with parallel computing. What would be the advantages of using a parallel algorithm in this case, other than some time being saved?

Another thing that we can potentially implement is search and traversal problems. The biggest problem that we might have is to implement these functions in a way that is strictly faster than the serial version on the CPU.

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p475-484. Print

[2] NVIDIA,
https://devblogs.nvidia.com/thinking-parallel-part-ii-tree-traversal-gpu/

[3] NVIDIA,
https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/

13 By Brendan Gregory, on the Fast Fourier Transform

13.1 Introduction: The Fast Fourier Transform

This project looks at the ways in which a Fast Fourier Transform can be implemented in CUDA for the purpose of applying different effects to images in the bitmap format. Specifically, it uses the FFT to apply a Gaussian blur to a 1400x1000 picture of marbles.

13.2 The process

The image is blurred using this procedure.

1. 1. The image is separated into 3 different arrays of unsigned char: one for the red pixel value, one for green, and one for blue.
2. 2. The unsigned chars are converted to complex numbers, represented by one double for the real component and one double for the imaginary.
3. 3. The image is zero-padded so its dimensions are a power of 2.
4. 4. A Gaussian filter is created.
5. 5. The filter is zero-padded so that its dimensions match those of the image.
6. 6. The image and filter both undergo FFT on every row, then on every column.
7. 7. The two matrices are multiplied together with pointwise multiplication, not with matrix multiplication.
8. 8. The image undergoes inverse FFT on every column, then on every row.
9. 9. The result is written back into the file.

Any part of this process can be parallelized. The nontrivial ones of these are steps 6 and 8.

13.3 The Serial Solution

The serial FFT is fairly straightforward. Its pseudocode is as follows:

```
complex* fft(array) {
    even = even_indices_of_array;
    odd = odd_indices_of_array;

    fft(even);
    fft(odd);

```

```
    for(int k = 0; k < even.length; k++) {
        e = even[k];
        o = odd[k];
        factor = e^(-2*pi*i*k/(2*even.length));
        o_factor = o * factor;

        result[k] = e + o_factor;
        result[k + even.size] = e - o_factor;
    }

    return result;
}
```

It is recursive, reaching a depth of log N. At each level, there are N/2 sets of required operations. The total runtime is O(NlogN).

13.4 The Parallel Solution

In the serial solution, N/2 different transformations are being performed at every "level." These happen to be totally independent of one another. This allows for the process to be parallelized. There are at least N/2 threads. Each thread calculates its own one transformation and temporarily stores its two results. All the threads are synchronized, and then every thread writes its results into a new array. The threads synchronize again, and the process continues to the next level and repeats.

The process works like this:

```
complex* fft_gpu(array) {
    for (int level = 1; level <= logN; level++ {
        // calculate x1 and x2
        // calculate k1 and k2

        e = array[x1];
        o = array[x2];
        factor = e^(-2*pi*i*k/(2^(level+1)));
        o_factor = o * factor;

        syncthreads();
        result[k1] = e + o_factor;
        result[k2] = e - o_factor;
        syncthreads();
    }
    if (inverse FFT) {
        scale = 1.0 / array.size;
        for(all elements in array) {
            scale by size;
        }
    }
}
```

Here is the code that calculates x and k vaues:

```
int offset_old = (level - 1) * n;
int offset_new = offset_old + n;
```

```
3
4  int oldSize = pow(2, level - 1);
5  int newSize = 2 * oldSize; // the size of the group we are expanding
       into
6  int dx = n / newSize;
7  int base = threadIdx.x % dx;
8  int x = (threadIdx.x / dx) * (2 * dx) + base;
9  int k_old = x / (2 * dx);
10 int k_new = x / dx;
```

The array is read from

```
offset_old + x
```

and

```
offset_old + x + dx
```

and it is written at

```
offset_new + base + k_old * dx
```

and

```
offset_new + base + (k_old + oldSize) * dx
```

This implementation is very fast, at a speed of O(logN). At each level, two thread synchronizations are required. The data is stored in global memory, as in some cases, for larger images, it may not fit in shared memory. The code deals heavily in powers of two, so shared memory would lead to many bank conflicts anyway. Threads are set up to access the array in indices close to one another, so coalescence is naturally attained. There is some warp divergence, as only thread IDs under a certain number will perform a transformation, and the others will do nothing. The difference this makes in performance is likely not significant.

The problem with this implementation is that it requires NlogN memory, as every level must be stored independently of the others. One way to improve upon this is simply to store space for 2*N layers, and continuously swap between the two.

13.5 Analysis and Conclusion

The implementation of the parallel algorithm provides a significant speedup, from a time of 13,000 to 5,000 as measured by cstart() and cend(). This can still be improved upon. Processes like multiplying the image matrix with the filter matrix can be parallelized, and FFTs for different rows or columns can be run in parallel, on different blocks. These are not implemented because they are trivial and do not require the use of different algorithms, and consume far less time than the FFT itself. A significant cause of slowdown is the continuous use of cudaMemcpy, which is invoked 2*N times, with N being the number of rows

and then the number of columns. One way to get around this is to allocate the entire image array, all at once. I chose not to implement it this way, because it would require logN times the entire dimension of the image of space, which does not scale well and can get to the degree of hundreds of megabytes of space needed. For smaller images, however, this would lead to great speedup.

The code itself can be greatly improved in the future. One problem it currently has is that the number of threads per block must be manually set to the largest power of two less than the dimensions of the image. With more careful code, this can be remedied.

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p285-291. Print

[2] Stack Overflow, https://stackoverflow.com/questions/11129138/reading-writing-bmp-files-in-c