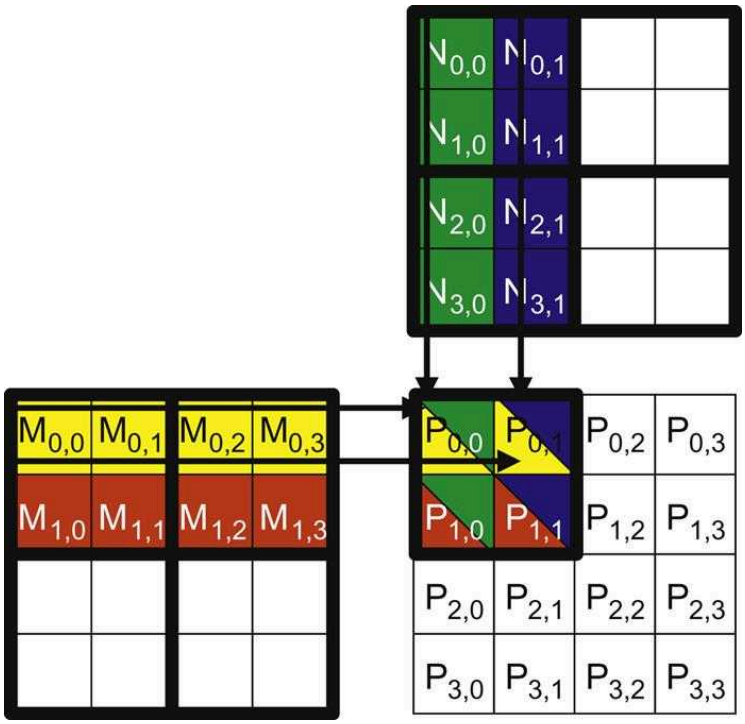
Single Source Shortest Path using CUDA

Kanad Panini Telang
ID: 114194853

Problem statement:
Find and Show the Single Source Shortest Path between any 2 points on a graph where the cost of the edges in each direction is dictated by the value of that point.

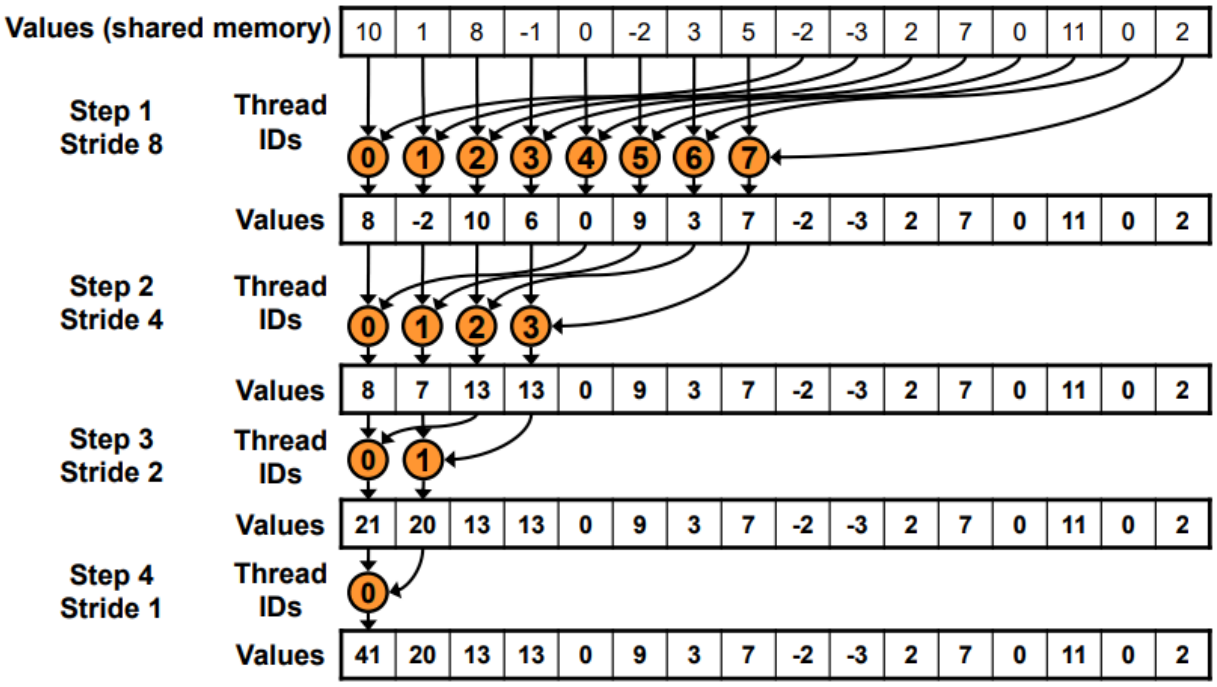
Procedure:
The matrix/array that is responsible for providing us values of the weights is the output of the programming assignments where two input matrices are multiplied using a tiled approach using GPUs and CUDA and the find the 2 global minimums using iterative block reduction.

Multiplication:
The tiling approach is used to perform the matrix multiplication operation using CUDA. This helps us to make better use of the shared memory thereby having a positive impact on the performance. The submatrices/Tiles of both the matrices to be multiplied are loaded in the shared memory and the product for the corresponding tiles are calculated. The following image is a visual representation of the approach.



Minimum:

In order to find the 2 global minimums along with the tiling approach we follow an iterative block reduction approach. The minimum from each tile is found using the block reduction approach and is stored in the zeroth index of the block. Then the block of all these local minimas is reduced iteratively until we have the first minimum. After we find the first minimum, we send that to the host and store it in a variable called min1. Then we set the value of that index where we found min1 to FLT_MAX and we do the iterative block reduction again to find the second minimum. This value is sent to the host and stored in a variable called min2. After we find the second minimum, the value at the index of the first minimum is restored to its original value. The following image visually represents the block reduction process.



Parallel Implementation of Dijkstra's Algorithm:

The following pseudocodes were used to implement the parallel form of the Dijkstra's Algorithm in order to evaluate the shortest weighted path between the source and the destination which are the first and second global minimums of the output of the tiled matrix multiplication operation respectively. The following pictures show the pseudocode used to implement the two kernels used in the cuda code and the code snippets of the kernels from the cuda code.

Kernel 1: Pseudocode

Algorithm 6 KERNEL1_SSSP

```
1: tid ← getThreadID
2: if M_a[tid] then
3:    M_a [tid] ← false
4:    for all neighbors nid of tid do
5:       Begin Atomic
6:       if C_ua[nid] > C_a[tid] +W_a[nid]  then
7:          C_ua[nid] ← C_a[tid]+W_a[nid]
8:       end if
9:       End Atomic
10:   end for
11: end if
```

Kernel 1 : Snippet

```
__global__ void kernel1(minVal* d_initialMinBlock, int *vertex, int *edge, float
*finalCost, path *costs, bool *mask , int size){
// host functions to compute the intermediate costs
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int tid = row * size + col;
    if(mask[tid]== true){
        mask[tid] = false;
        for (int i = vertex[tid]; i<vertex[tid+1]; i++){
            path cpt; // cost + thread Ca+ Wa step in the pseudocode.
            cpt.val = finalCost[tid] + d_initialMinBlock[tid].val;//edge[i]
            cpt.index = tid;
            //__syncthreads();
            atomicMin(&costs[edge[i]], &cpt);
        }
    }
}
```

In the above kernel we check if the mask for a given thread is set to be true, if true, that means that the thread's weight needs to be computed. If true, then we set the flag to a false as the weight is going to be computed again and for a given set of values dictated by the array vertex we get the new values of the weights using the `finalCost` array and the weight of the edge dictated by the output of the multiplication which in our case is called `d_initialMinBlock`. We then compare the new variable called cpt and the current value at that given location of the thread using the atomic min function and the minimum of the two values is assigned to that index in the costs array. The atomicMin function is described later in the report.

Kernel 2: pseudocode

Algorithm 7 KERNEL2_SSSP

1: $tid \leftarrow$ getThreadID
2: **if** $C_a[tid] > C_{ua}[tid]$ **then**
3: $C_a[tid] \leftarrow C_{ua}[tid]$
4: $M_a[tid] \leftarrow$ true
5: $Terminate \leftarrow$ false
6: **end if**
7: $C_{ua}[tid] \leftarrow C_a[tid]$

Kernel 2: Snippet

```
__global__ void kernel2(bool *d_done, int *vertex, int *edge, float*finalCost,
path *costs, bool *mask, int size){ // we don't need a mask for this one.
// device functions to comp
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    //printf("The value of d_done before being updated is %d\n",d_done);
    int tid = row * size + col;
    if(finalCost[tid] > costs[tid].val){
        finalCost[tid] = costs[tid].val; // assign the min value to cost.
        mask[tid] = true;
        *d_done = false;
    }
    //printf("The value of d_done after the if statement and being updated is
%d\n",d_done);

    costs[tid].val = finalCost[tid];

}
```

This kernel compares the value in the `finalCosts` matrix and the `costs` matrix and checks if the values of the weights at the corresponding thread indices in the `finalCosts` matrix is greater than the value at the corresponding index in the `costs` matrix. If so the value is replaced and that thread is set to true as that value got changed and other values associated with that thread need to be recomputed. As we don't have a stable matrix yet, that is, all the elements have gotten their minimum weight values, we set the done pointer to false indicating that we

need to keep running both the kernels. Before we exit the kernel we update the value of `costs` with the value of `finalCost` for the corresponding indices and then we exit the kernel.

Setup for running the two kernels:

As seen in the above codes for both the kernels we realize that both these kernels use some arrays like costs, finalCost, mask, vertex etc and in order to successfully use these arrays. The following code shows the setup for these arrays.

```
void setup(int *vertex, int *edge, float *finalCost, path *costs, bool *mask ,
int size, minVal min1){// setup all the arrays so that we can perform the sssp
steps.
    int edgeIdx = 0;
    // this block of code initializes the finalcost matrix, the cost matrix, (Ca
and Cua r  espt), initializes the mask as we tlaked about in class, adn the edges
array to get us the
    //number of edges associated with each vertex and the the vertices array that
store the information about edges at each vertex.
    for (int i = 0; i < size; i ++) {
        for (int j =0; j<size; j++){
            mask[i*size +j] = false;
            finalCost [i*size +j] = FLT_MAX;
            costs[i*size +j].val = FLT_MAX;
            costs[i*size +j].index = -1;

            vertex[i*size +j] = edgeIdx; // This array would tell us where in the
edges array would this point belong. So point 0 would start at 0, point 1 would
start at 2 etc.

            if ((j-1) >= 0) edge[edgeIdx++] = i*size+(j-1); //given this
condition is true, th point has a edge on the left directed to the assigned point
            if ((i-1) >= 0) edge[edgeIdx++] = (i-1)*size +j; // edge on the top
            if ((j+1) < size) edge[edgeIdx++] =  i*size + (j+1); // edge on th
right
            if ((i+1) < size) edge[edgeIdx++] = (i+1)*size + j; // edge on the
bottom.

        }
    }
```

```
        //h_done[0] = false;

        costs[min1.row*size + min1.col].val = 0.0f; // the cost of the source
would be a zero as itwould be a t a distance zero from itself.
        finalCost[min1.row*size + min1.col] = 0.0f;
        mask[min1.row*size + min1.col] = true; // this is wher we start.

}
```

The arrays costs and finalCosts act as the Cu and Cua in the pseudocode. The masks array is needed to know what threads are active and which ones are not and there are two more arrays namely called vertex and edges. The edges array stores the index of the element that is connected to a given vertex, that you're currently on, by an edge. Basically, storing the indices of the neighbours of a given point. And the vertex array stores the information about how many neighbours a vertex has and what element in the edges array would concern a given vertex. The edge array is used for a different implementation of the same algorithm. The edges array makes more sense when we directly use the pseudocode as it mentioned the neighbour id but as the weight of the edge is determined by the value of the point in the array called `d_initialMinBlock` which is the output of the multiplication.

Atomic Min Function:

As mentioned above, we need to find the minimum of the threads using an atomic function. Meaning that the cost is processed in a manner where one thread is scheduled randomly it does a check comparing whether the current cost is greater than the resulting cost if that thread was used, if so then the values are swapped and the new value is stored in the costs array. The following code shows how the atomicMin function is implemented.

```
__device__ __forceinline__ path *atomicMin(path *location, path *cpt){ // the
forceinline directive would make sure that the function is inserted at the
function call which would reduce the overhead. This doesn't guarentee that this
would happen but if the overhead is reduced, the program overall would be faster.
// the function that we are going to use in this case to compare and find the min
is atomic compare and swap. The function only takes int * or unsigned long long
int as parameters
// as we are comparing two elements of the type path, we would have to
reinterpret path as an ulli.

    unsigned long long int loc = PathAsULLI(location);
    while(cpt -> val < (location)-> val){ //ULLIAsPath(&loc)
```

```
        unsigned long long int preCAS = loc;
        loc = atomicCAS((unsigned long long int*) location, preCAS,
PathAsULLI(cpt));
        if (loc == preCAS) break;
    }

    return ULLIAsPath(&loc);
}
```

The above function is an atomicMin function which is called from our first kernel that's used to implement the parallel Dijkstra's Algorithm. First we compare the value that the new thread is bringing with the value currently stored at the location. If the value because of the new thread is less than the current value stored in our array we enter the while loop where we call the atomicCAS function that checks if the value `old` stored at the location given by `location` with the `preCAS` parameter which again is `location` reinterpreted as an `unsigned long long int*` value and as they would match the current value would be replaced by the new value as we have already checked whether the new value is less than the old value or not. The `atomicCAS` function can only take either int or an `unsigned long long int` as input parameters and therefore it is extremely important to reinterpret our `struct path` to either one of the data types that are accepted by this function and then convert the output of the `atomicCAS` function to the type `path` in order to make sure that we can store the proper values back in our costs array. As the `struct path` is composed of a `float` and an `int` value it is best to reinterpret it as `unsigned long long int`. The following `__device__` functions are used to reinterpret the `path` type to an `unsigned long long int` and the other way round.

```
__device__ unsigned long long int PathAsULLI(path *path){
    unsigned long long int *ulli = reinterpret_cast<unsigned long long int*>
(path) ;
    return *ulli;

}

__device__ path* ULLIAsPath(unsigned long long int * ulli){
    path *var = reinterpret_cast<path*>(ulli);

    return var;
}
```

The first function reinterprets `path` as an `unsigned long long int` value and the second function reinterprets the `unsigned long long int` value as the type `path`.

Visualization using OpenCV:

The OpenCV Library was used in order to visualize the short path between the first minimum and the second minimum. The following code was used to obtain the desired results:

```
void showPath(path *s_path, int size, minVal min2, minVal min1){
    cv::Mat image = cv::Mat::zeros(size,size, CV_8UC3);
    int next = min2.row * size + min2.col; // get the index for the destination.
    bool last = false;



    while(last == false){ // check if it is the last element. If so then raise
the flag so that we can stop.
        if(s_path[next].index == -1) last = true; // when we find the last
element set the flag.
        if (last == true) next= min1.row * size + min1.col; // as this would be
the last element the index would be -1 as it shouldn't be pointing to anything
cuz it is the source. So we should update the -1 to the index of the source.
        image.at<cv::Vec3b>((int)(next / size), (next % size)) =
cv::Vec3b(0,165,255); // 0 , 165, 255 color for orange
        if (next == min2.row * size + min2.col) image.at<cv::Vec3b>((int)(next /
size), (next % size)) = cv::Vec3b(255,255,255); // white for the destination.
        if (next == min1.row * size + min1.col) image.at<cv::Vec3b>((int)(next /
size), (next % size)) = cv::Vec3b(0,0,255); // blue for source.
        next = s_path[next].index;
    }

    cv::imshow("Shortest Path", image);
    while(true){
    int key = cv::waitKey(0);
    if(key == 27) break;
    }
}
```

The function above helps us to visualize the path between the source and the destination. We start by initializing an array of size matrix_size by matrix_size and we call that image. That's the number of pixels that our image is going to have and every pixel represents a vertex.

When we calculate the weight of the path to get to a particular index, we store the weight to get to that index and the previous index to get to the current index. Therefore, when we start to visualize the path, we start at the destination and trace our way back to the source.

The variable next is telling us where we're currently at and the vertex/pixel at that index should be colored to show that the pixel that we're currently on is a part of the path. We start by checking if the current pixel where we are right now is the source pixel. If it is the source then we raise the last flag indicating that this is the last pixel that we're going to be on and now we would just be exiting the while loop. When we realize that it is the last pixel we also set the variable next to the index of that last pixel so that the last pixel which represents the source is also colored. We change the color of that pixel to the desired color using the line

```
image.at<cv::Vec3b>((int)(next / size), (next % size)) = cv::Vec3b(0,165,255);
```

This makes sure that the pixel at the point next/size and next%size, namely the row and the column numbers in the matrix, are changed to the color orange from the original color of black. Once all the pixels on the desired path are colored we show the image and we make sure that the image is displayed until the escape key is pressed.

The destination point is colored white and the source is colored red in order to separate those two points from the other points on the path.

Results:

The following images show the results of the minimum operation, the weight of the shortest path from the source to the destination and the image of the visualized path.

```
g12@lion:~/ese565/project$ ./proj 1024
The input size is 1024
The minimum Value is 790.078186 at index 407, 976
The 2nd minimum Value is 791.253235 at index 403, 289
The weight of the path to get to the second min value from the first min value is 582969.812500.
```

The following image shows the visualization of the path between the two minimum points listed above.


Shortest Path@lion.ece.stonybrook.edu