3D Plane Detection via Hough Transform

Anthony Duca

Plane Detection

Problem: You have a point cloud containing many points and want to detect planes in this point cloud to find features (walls, floors, ceilings, sides of buildings, fences, rows of objects, etc.).

Hough Transform: Have every point in the point cloud vote for every possible plane it is a part of then pick the planes with the most votes.

Works for any parameterizable shape. Planes can be parameterized by their normal vector in spherical coordinates. (rho,theta,phi).

Algorithm

Step 1: Perform hough transform to build accumulator of votes.

Step 2: Perform non-maximum suppression to eliminate all planes that are not local-maxima in the accumulator

Step 3: Pick planes that have the highest number of votes.

Suitability for GPU acceleration.

In Step 1 each point is voting independently of each other meaning that the votes could be cast in parallel.

Steps 2 and 3 involve operations on large 3d arrays and in step 2 each element of the output is dependent on a small “contiguous” section of the array.

The number of comparisons in step 3 can be drastically reduced via a reduction.

Only part of algorithm not done on GPU is converting indices back to planes.

Converting indices back to planes is <1% of the execution time

Step 1: Hough Transform

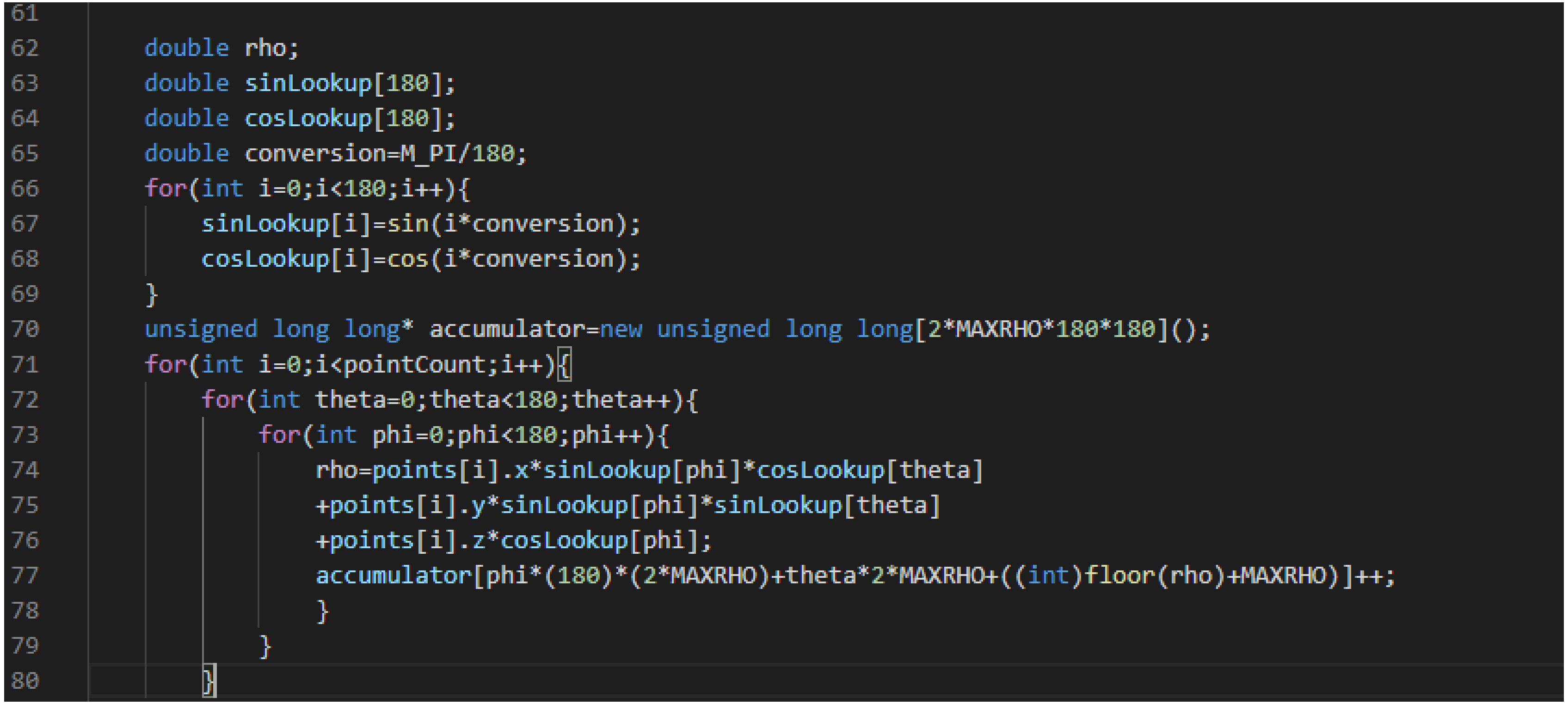
With values x,y,z,theta,and phi the value of rho can be calculated with this formula:

rho= x*sin(phi)*cos(theta) + y*sin(phi)*sin(theta) + z*cos(phi)

Basic idea: Iterate through all points and for each point iterate through all theta/phi values, calculate rho, and increment bin in accumulator.

Bin widths of 1 were used for the accumulator.

Hough Transform: CPU



```

    double rho;
    double sinLookup[180];
    double cosLookup[180];
    double conversion=M_PI/180;
    for(int i=0;i<180;i++){
        sinLookup[i]=sin(i*conversion);
        cosLookup[i]=cos(i*conversion);
    }
    unsigned long long* accumulator=new unsigned long long[2*MAXRHO*180*180]();
    for(int i=0;i<pointCount;i++){
        for(int theta=0;theta<180;theta++){
            for(int phi=0;phi<180;phi++){
                rho=points[i].x*sinLookup[phi]*cosLookup[theta]
                +points[i].y*sinLookup[phi]*sinLookup[theta]
                +points[i].z*cosLookup[phi];
                accumulator[phi*(180)*(2*MAXRHO)+theta*2*MAXRHO+((int)floor(rho)+MAXRHO)]++;
                }
            }
        }
```

Hough Transform: Naive GPU

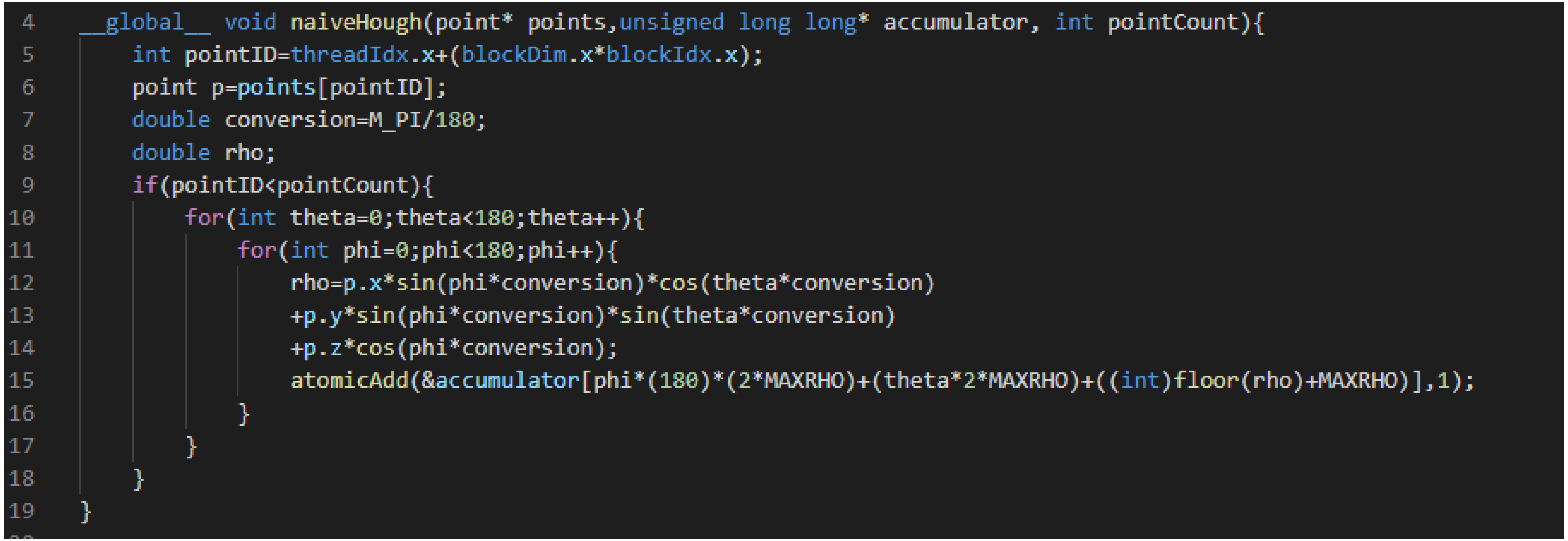
Directly port CPU version to GPU, have each thread cast all votes for one point.

AtomicAdd must be used to prevent collisions in accumulator.

The naive version is a scatter implementation, each thread must have access to the entire accumulator because it is impossible to know where a point will send its votes before it does.

O(n) threads do O(1) work each.

Hough Transform: Naive GPU



```
__global__ void naiveHough(point* points,unsigned long long* accumulator, int pointCount){
    int pointID=threadIdx.x+(blockDim.x*blockIdx.x);
    point p=points[pointID];
    double conversion=M_PI/180;
    double rho;
    if(pointID<pointCount){
        for(int theta=0;theta<180;theta++){
            for(int phi=0;phi<180;phi++){
                rho=p.x*sin(phi*conversion)*cos(theta*conversion)
                +p.y*sin(phi*conversion)*sin(theta*conversion)
                +p.z*cos(phi*conversion);
                atomicAdd(&accumulator[phi*(180)*(2*MAXRHO)+(theta*2*MAXRHO)+((int)floor(rho)+MAXRHO)],1);
            }
        }
    }
}
```

Hough Transform: Optimized GPU

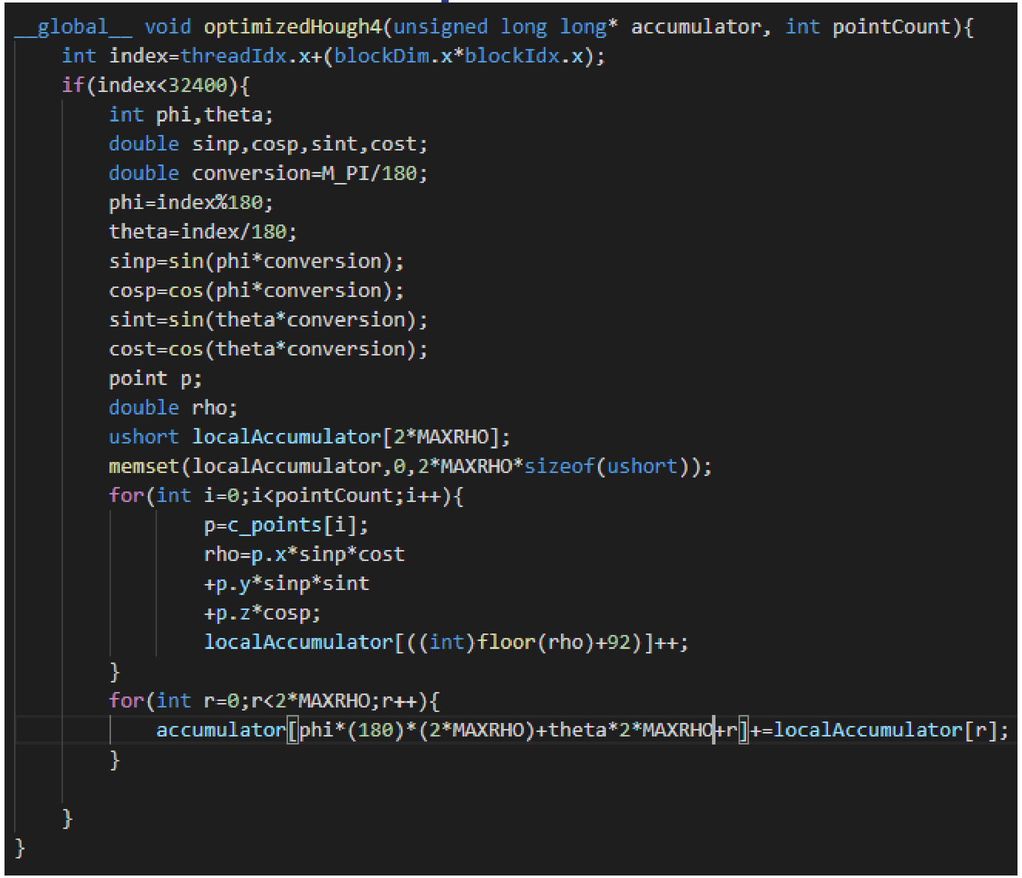
Switch from scatter to gather: Each thread is given a pair (Theta,Phi) and calculates the vote each point makes with those values. Each thread has its own row for rho in the accumulator.

This allows:

- ○ Removal of atomic add
- ○ Storage of sin/cos values in registers.
- ○ Storage of points in constant memory.
- ○ Giving each thread a local copy of its row to reduce global memory accesses.

O(1) threads with O(n) work each

Hough Transform: Optimized GPU



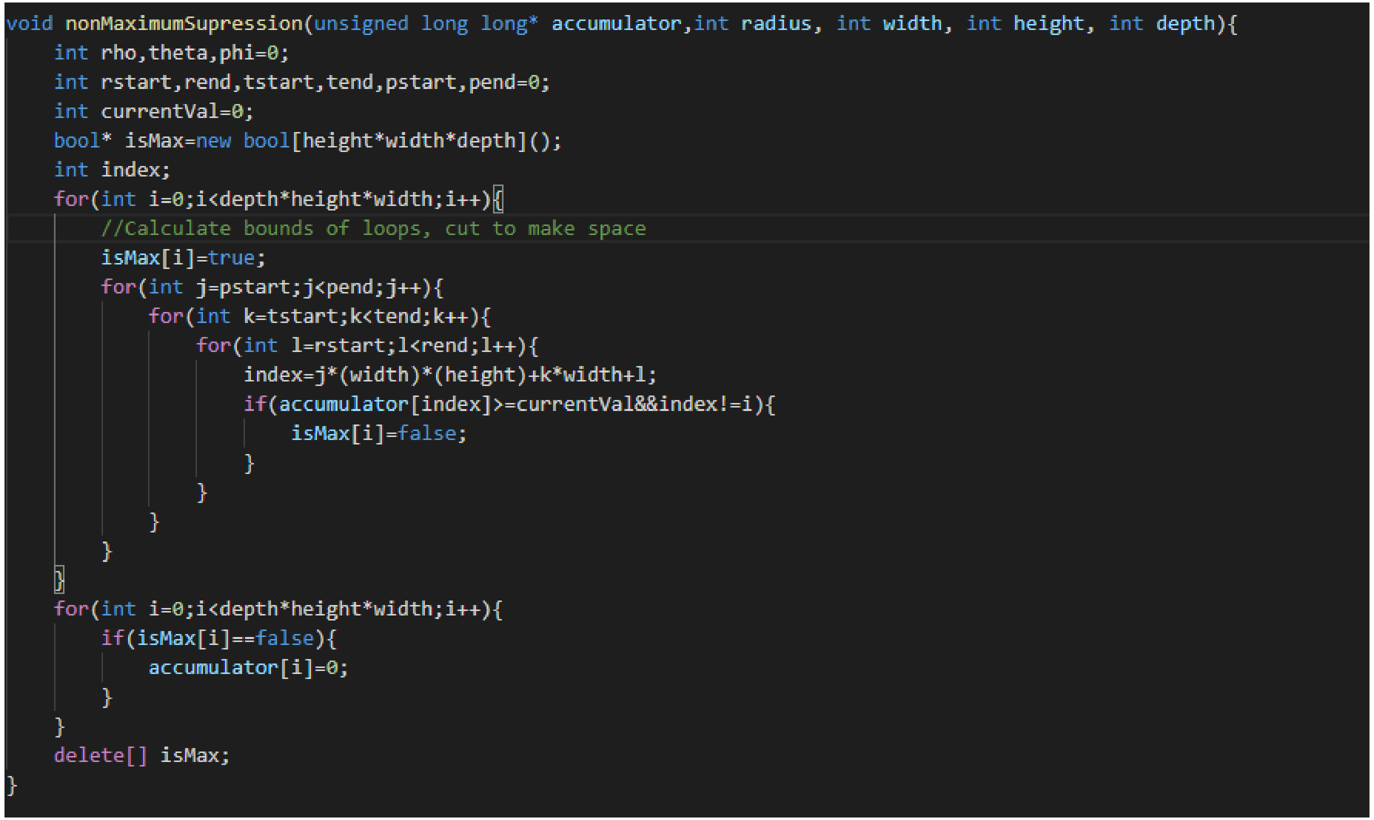
```
__global__ void optimizedHough4(unsigned long long* accumulator, int pointCount){
    int index=threadIdx.x+(blockDim.x*blockIdx.x);
    if(index<32400){
        int phi,theta;
        double sinp,cosp,sint,cost;
        double conversion=M_PI/180;
        phi=index%180;
        theta=index/180;
        sinp=sin(phi*conversion);
        cosp=cos(phi*conversion);
        sint=sin(theta*conversion);
        cost=cos(theta*conversion);
        point p;
        double rho;
        ushort localAccumulator[2*MAXRHO];
        memset(localAccumulator,0,2*MAXRHO*sizeof(ushort));
        for(int i=0;i<pointCount;i++){
                p=c_points[i];
                rho=p.x*sinp*cost
                +p.y*sinp*sint
                +p.z*cosp;
                localAccumulator[((int)floor(rho)+92)]++;
        }
        for(int r=0;r<2*MAXRHO;r++){
            accumulator[phi*(180)*(2*MAXRHO)+theta*2*MAXRHO+r]+=localAccumulator[r];
        }

    }
}
```

Step 2: Non-maximum Suppression

Compare each bin in the accumulator to its local neighborhood, if it is the maximum in its local neighborhood keep it, if it is not the maximum set it to 0.

Non-maximum Suppression: CPU



```
void nonMaximumSupression(unsigned long long* accumulator,int radius, int width, int height, int depth){
    int rho,theta,phi=0;
    int rstart,rend,tstart,tend,pstart,pend=0;
    int currentVal=0;
    bool* isMax=new bool[height*width*depth]();
    int index;
    for(int i=0;i<depth*height*width;i++){
        //Calculate bounds of loops, cut to make space
        isMax[i]=true;
        for(int j=pstart;j<pend;j++){
            for(int k=tstart;k<tend;k++){
                for(int l=rstart;l<rend;l++){
                    index=j*(width)*(height)+k*width+l;
                    if(accumulator[index]>=currentVal&&index!=i){
                        isMax[i]=false;
                    }
                }
            }
        }
    }
    for(int i=0;i<depth*height*width;i++){
        if(isMax[i]==false){
            accumulator[i]=0;
        }
    }
    delete[] isMax;
}
```

Non-maximum Suppression: Naive GPU

Directly port to CPU version.

One thread per bin, each thread calculate output for that bin.

O(r^3) work per for each thread.

r=radius of neighborhood

Non-maximum Suppression: Optimized GPU

Intention: Tile accumulator in shared memory, have one thread read in one value from the accumulator and write one (or zero) values to the output.

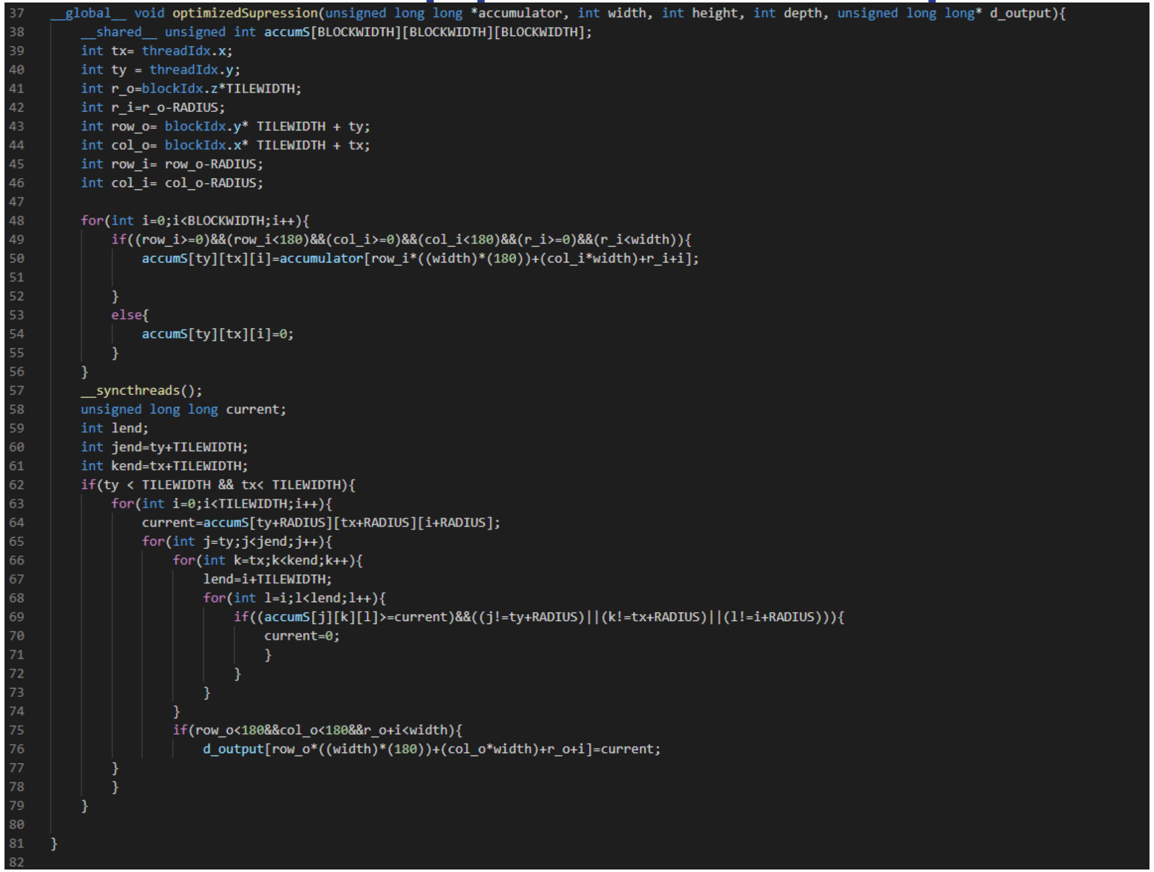
Two problems: Not enough threads, and not enough shared memory.

Result: Tile accumulator in shared memory, have one thread read in one row for rho from the accumulator and write one row for rho to the result.

A 3d grid of 2d blocks.

Shared memory constraints prevent the radius from being large.

Non-maximum Suppression: Optimized GPU



```
__global__ void optimizedSupression(unsigned long long *accumulator, int width, int height, int depth, unsigned long long* d_output){
    __shared__ unsigned int accumS[BLOCKWIDTH][BLOCKWIDTH][BLOCKWIDTH];
    int tx= threadIdx.x;
    int ty = threadIdx.y;
    int r_o=blockIdx.z*TILEWIDTH;
    int r_i=r_o-RADIUS;
    int row_o= blockIdx.y* TILEWIDTH + ty;
    int col_o= blockIdx.x* TILEWIDTH + tx;
    int row_i= row_o-RADIUS;
    int col_i= col_o-RADIUS;

    for(int i=0;i<BLOCKWIDTH;i++){
        if((row_i>=0)&&(row_i<180)&&(col_i>=0)&&(col_i<180)&&(r_i>=0)&&(r_i<width)){
            accumS[ty][tx][i]=accumulator[row_i*((width)*(180))+(col_i*width)+r_i+i];

        }
        else{
            accumS[ty][tx][i]=0;
        }
    }
    __syncthreads();
    unsigned long long current;
    int lend;
    int jend=ty+TILEWIDTH;
    int kend=tx+TILEWIDTH;
    if(ty < TILEWIDTH && tx< TILEWIDTH){
        for(int i=0;i<TILEWIDTH;i++){
            current=accumS[ty+RADIUS][tx+RADIUS][i+RADIUS];
            for(int j=ty;j<jend;j++){
                for(int k=tx;k<kend;k++){
                    lend=i+TILEWIDTH;
                    for(int l=i;l<lend;l++){
                        if((accumS[j][k][l]>=current)&&((j!=ty+RADIUS)||(k!=tx+RADIUS)||(l!=i+RADIUS))){
                            current=0;
                            }
                        }
                    }
                }
                if(row_o<180&&col_o<180&&r_o+i<width){
                    d_output[row_o*((width)*(180))+(col_o*width)+r_o+i]=current;
        }
        }
    }

}
```

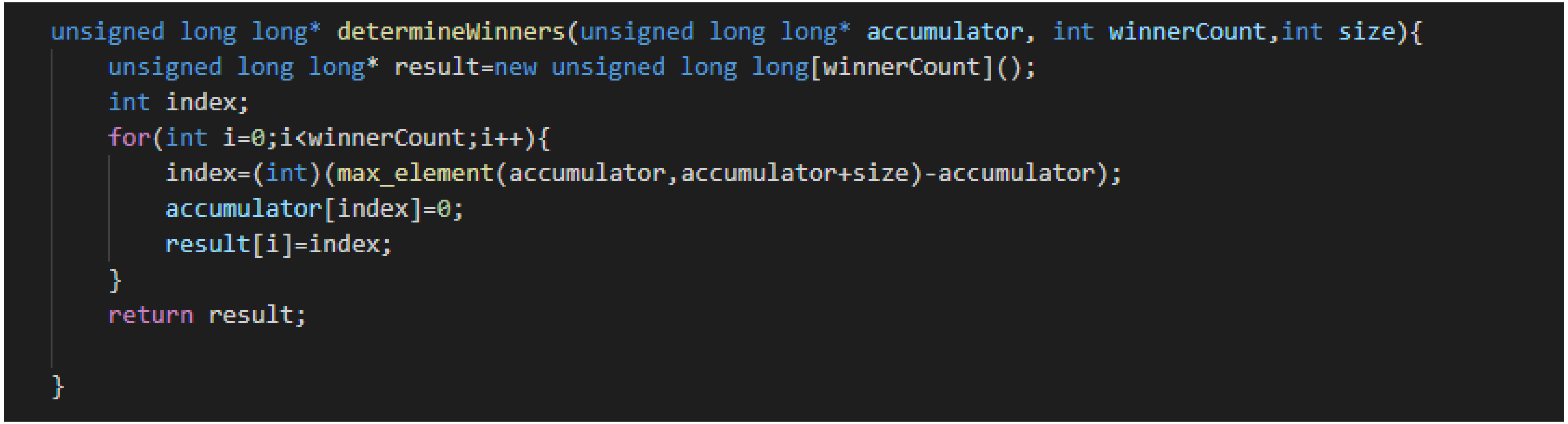
Step 3: Determine Winning Planes

Pick planes that have the highest number of votes.

Important Note: the desired value is not the number of votes but rather the index of the planes with those votes as the index stores the parameters of the plane.

Determine Winning Planes: CPU

Use Built-in C++ function.



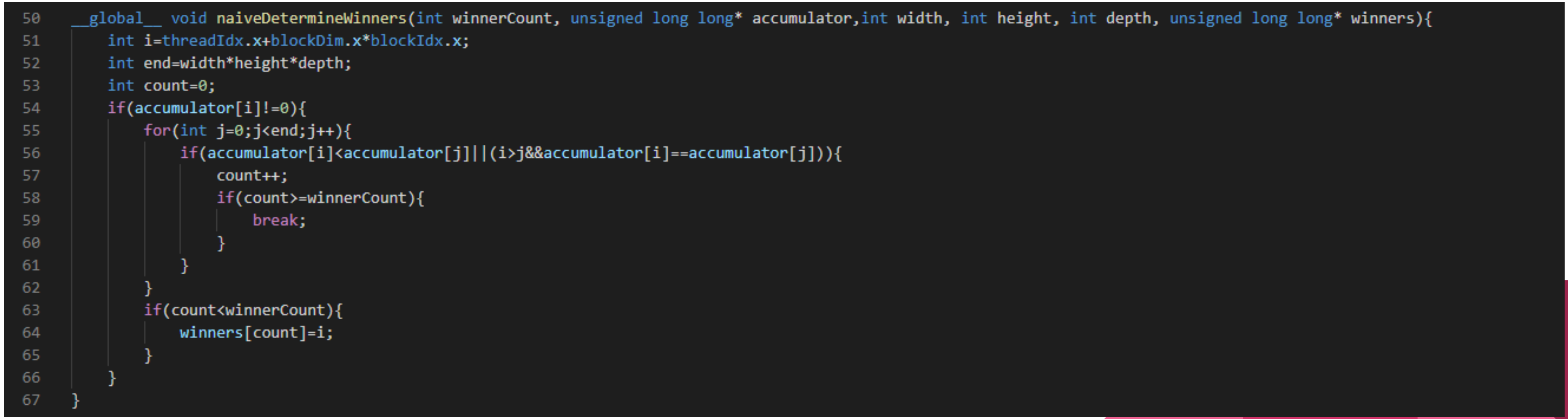
```
unsigned long long* determineWinners(unsigned long long* accumulator, int winnerCount,int size){
    unsigned long long* result=new unsigned long long[winnerCount]();
    int index;
    for(int i=0;i<winnerCount;i++){
        index=(int)(max_element(accumulator,accumulator+size)-accumulator);
        accumulator[index]=0;
        result[i]=index;
    }
    return result;

}
```

Determine Winning Planes: Naive GPU

Use modified rank sort to determine indices with highest values

Very naive and slow (bad), but gave idea for optimized version.



```
__global__ void naiveDetermineWinners(int winnerCount, unsigned long long* accumulator,int width, int height, int depth, unsigned long long* winners){
    int i=threadIdx.x+blockDim.x*blockIdx.x;
    int end=width*height*depth;
    int count=0;
    if(accumulator[i]!=0){
        for(int j=0;j<end;j++){
            if(accumulator[i]<accumulator[j]||(i>j&&accumulator[i]==accumulator[j])){
                count++;
                if(count>=winnerCount){
                    break;
                }
            }
        }
        if(count<winnerCount){
            winners[count]=i;
        }
    }
}
```

Determine Winning Planes: Optimized GPU

Perform “Rank reduction” with shared memory.

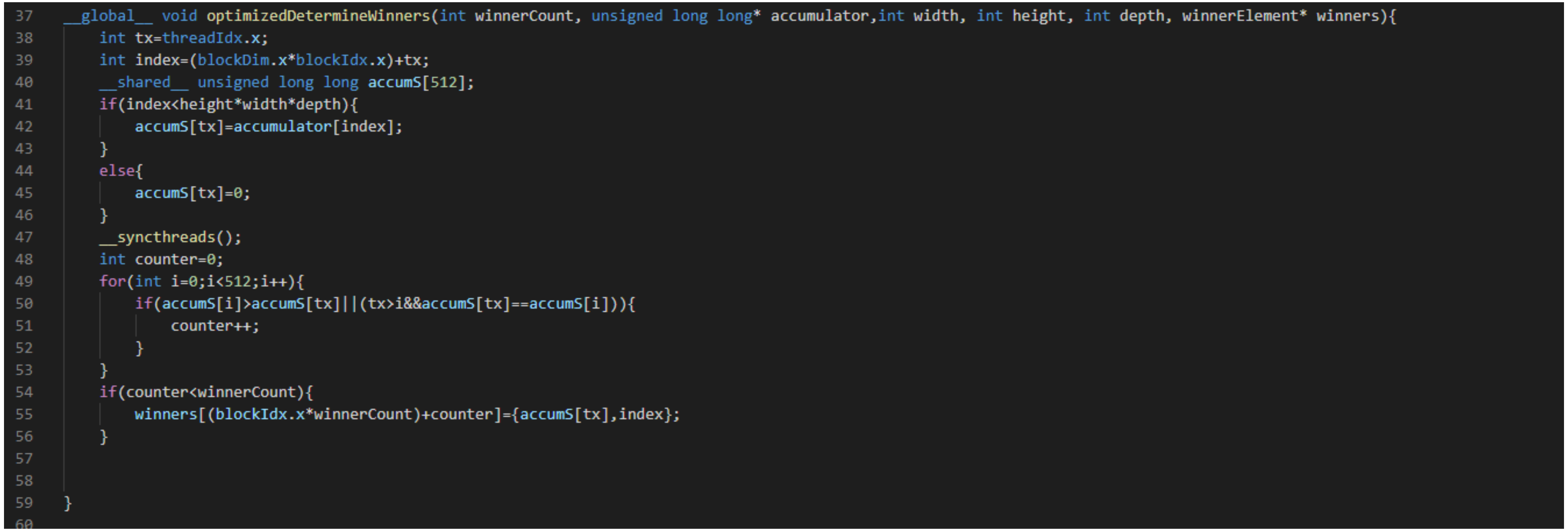
Basically get the n planes with the highest votes in each block, pass these planes onto next grid launch.

Run last round on CPU.

Drastically reduces global memory usage and comparisons compared to naive implementation.

Each thread does one global memory read and one write.

Determine Winning Planes: Optimized GPU



```
__global__ void optimizedDetermineWinners(int winnerCount, unsigned long long* accumulator,int width, int height, int depth, winnerElement* winners){
    int tx=threadIdx.x;
    int index=(blockDim.x*blockIdx.x)+tx;
    __shared__ unsigned long long accumS[512];
    if(index<height*width*depth){
        accumS[tx]=accumulator[index];
    }
    else{
        accumS[tx]=0;
    }
    __syncthreads();
    int counter=0;
    for(int i=0;i<512;i++){
        if(accumS[i]>accumS[tx]||(tx>i&&accumS[tx]==accumS[i])){
            counter++;
        }
    }
    if(counter<winnerCount){
        winners[(blockIdx.x*winnerCount)+counter]={accumS[tx],index};
    }


}
```

Timing: Number of planes

	2	5	10
CPU	16s	16.5s	17s
Naive Scatter	2.7s	2.8s	3s
Naive Gather	2s	2.163s	2.2s
Optimized Gather	2s	2.101s	2.2s
Optimized nms+Gather	2s	2.127s	2.2s
Optimized Final	.05s	.05s	.05s

Timing: Number of points

	1	1000	10,000	100,000	1,000,000	10,000,000	23,000,000
CPU	15s	17s	23s	88s	676s	6589s	[4h]
Naive scatter	.177s	3s	4.2s	8s	36.5s	321s	729s
Naive gather	.053s	2.2s	4.2s	12s	75.8s	718s	1629s
Optimized gather	.061s	2.2s	3.6s	6.3s	11.6s	75s	161s
Optimized nms+gather	.036s	2.2s	2.5s	6.1s	11.5s	75s	161s
Optimized: Final	.047s	.05s	.1s	.7s	6.8s	67s	152s
Overall Speedup	319x	340x	230x	125x	100x	99x	[95x]

Sources

Point cloud from http://semantic3d.net

Based on pseudocode from:

https://robotik.informatik.uni-wuerzburg.de/telematics/download/3dresearch2011.pdf

End

Time for output.