Pavani Majety | uniqname: majety | Parallel Computing | Project Report

Parallel Computing: Project Report

Multi-Resolution Scene Cache Generation for Videos using Parallel QuadTree based implementation

PROJECT USE-CASE AND SERIAL IMPLEMENTATION

Background for the project:

Video is an important data source for real-world tasks such as autonomous driving, surveillance analysis, biomedical applications etc. These tasks require detecting objects in an accurate and efficient manner. Video processing techniques like- compression, scene cache generation are difficult for two reasons - the high computational requirements and need for higher picture quality at lower data rates. Scene cache generation can be thought of as a subset of the video compression algorithms. Here, a quadtree structure that best represents the object in every frame of a video is maintained as the object changes its positions through the temporal domain . The serial programs for this use case is considerably slow, therefore parallel computing helps to explore advanced algorithms to speedup the algorithms. With GPU support available in smallest embedded systems like Raspberry pi, etc, parallel computing helps in utilizing the hardware to its maximum potential.

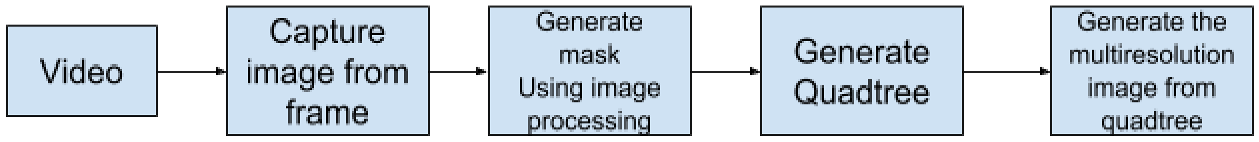
Implementation:

Video scene cache generation is performed with the following objectives in mind -

i. high quality of the foreground objects in video
ii. small delay
iii. high degree of compression of the background objects.
iv. low power requirement(not scope of this project)

The application of scene cache generation is in real-time video applications like autonomous driving or machine vision. However, for benchmarking the parallel algorithm with serial implementation, the a recorded video will be analysed. The algorithm for construction of quadtree and reconstruction of image from quadtree is described below -

Algorithm:


Video
Capture image from frame
Generate mask Using image processing
Generate Quadtree
Generate the multiresolution image from quadtree

A frame is captured at a regular interval from the video and is used for processing. Once an image is obtained from the frame, a mask is generated to detect the object in frame. The object in frame is detected using color thresholding techniques for this project. Other object or face detection algorithms can be deployed at this step to generate its corresponding binary mask. The next step in the process is quadtree generation. This step is agnostic to the method used for generating the mask in the previous step. The quadtree algorithm uses this mask to keep higher resolution of the object in the frame. The algorithm for generating quadtree is described in the next step. For the purpose of comparison, the quadtree is maintained as a linear data structure using vectors and also as a pointer based data structure. Once the quadtree structure is obtained, a multiresolution, compressed scene cache image is reconstructed. There are multiple ways to generate the quadtrees as described in Hanan Samet 1990.

Recursive algorithm for the generation of Quadtree:

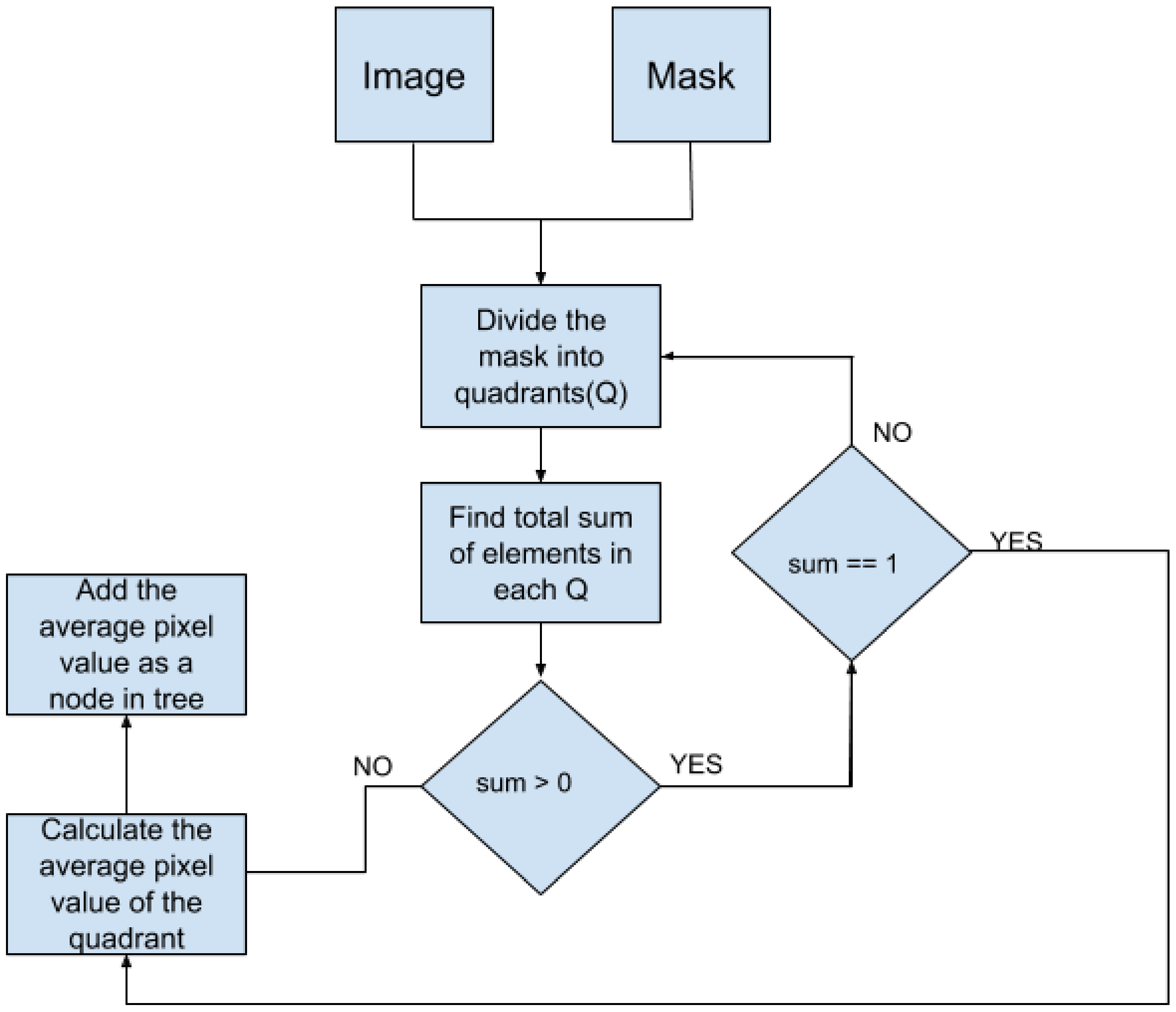
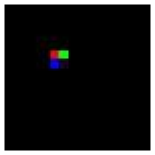


Image
Mask
Divide the mask into quadrants(Q)
Find total sum of elements in each Q
sum > 0
NO
YES
sum == 1
NO
YES
Calculate the average pixel value of the quadrant
Add the average pixel value as a node in tree

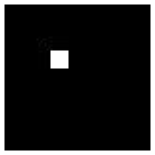
Initially the image is resized to 512 x 512 by applying interpolation techniques to make it compatible to construct a quadtree structure. Following this step, to construct the quadtree, the mask is divided into 4 equal quadrants. Total number of non zero elements is calculated in each of the quadrants. If the total number of non-zero elements in any of the quadrants is greater than zero, then each of such quadrants is further subdivided until the number of rows and cols in the quadrants is equal to one. When the number of rows and cols in a quadrant is equal to one, it means that quadrant is subdivided to the smallest achievable size, i.e, one pixel. The pixel value is added as a leaf node into the quadtree. These are achieved at the maximum possible depth in the tree. The leaf nodes with quadrant size equal to that of a pixel, are of highest possible resolution. When the sum of elements of a non-zero sized quadrant of the mask is equal to zero, then the average pixel value of the corresponding quadrant in the image is calculated and added as a leaf node in the quadtree. This compromise allows us to obtain high compression ratio in the background. These leaf nodes are of lower resolutions. If a node in the tree is identified as a leaf node, it is not further subdivided into smaller quadrants. Hence, this algorithm generates a multi resolution image with higher resolution at the detected object(s) and lower resolution elsewhere.

Small example

Image(8x8):



Mask(8x8):



Generated Tree:

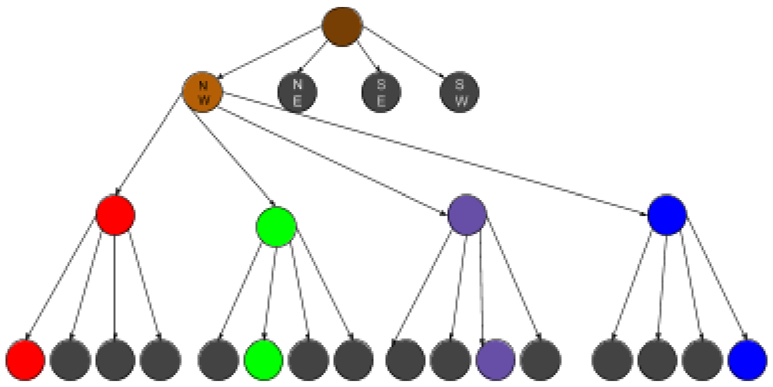
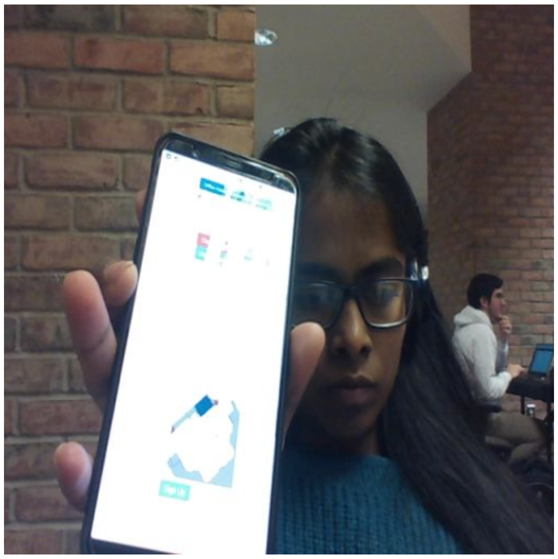
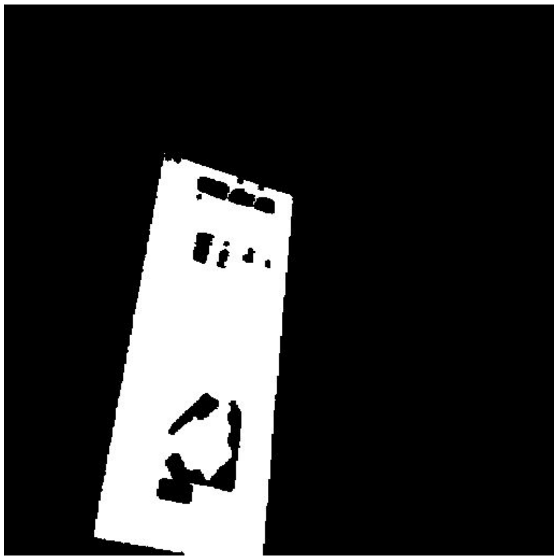


Image example:

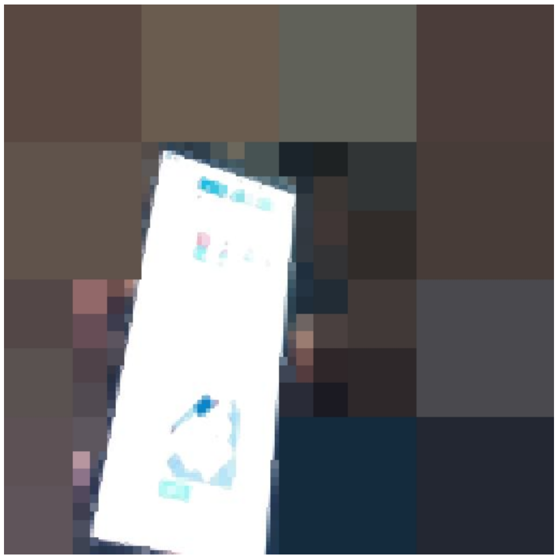
Image:



Mask for the visible white color in image:



Reconstructed Image:



As described before, here the mask for white pixels, is being used for demonstration purpose. The mask can be representing any part of the image and the algorithm is agnostic to the image itself.

Algorithms for parallelizing different parts of the serial code:

The image is cached in the GPU as shared memory.

Conversion of RGB Image into HSV image:
Each of many CUDA kernels work on a single pixel to convert the RGB values into corresponding HSV Values.

Color Thresholding and Mask thresholding:
Each of many CUDA kernels work on a single pixel for the color thresholding and generate a destination matrix, which is later transferred back into CPU memory. Given below is an example of the kernel function that is being called. Here the lower and higher are the HSV color ranges that the kernels are thresholding the pixels to -

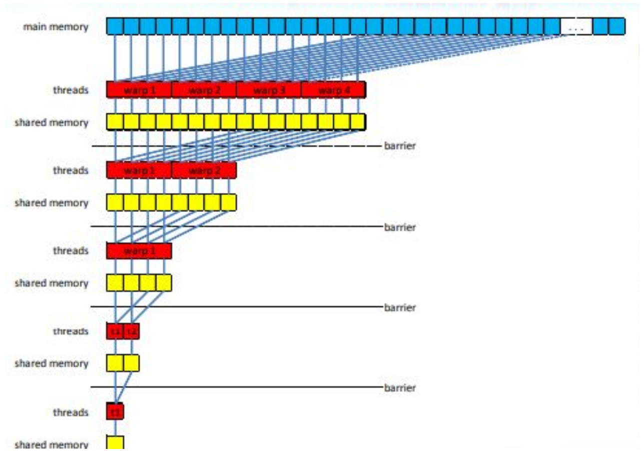
```
void inRange_gpu(cuda::GpuMat &src, cuda::GpuMat &dst, const Scalar
&lower,
   const Scalar &upper);

__global__ void inRange_kernel(const cv::cuda::PtrStepSz<uchar3> src,
cv::cuda::PtrStepSzb dst,
                   int lbc0, int ubc0, int lbc1, int ubc1, int lbc2, int
ubc2);
```

Quadtree Construction and Image reconstruction:
By incorporating the notion of adjacency preserving at each level of a quadtree, parallel algorithms have been devised for different architectures, including mesh computers, transputers, hypercubes, and shared memory abstract machines. Most of the works use the linear quadtree representation. There are methods that rely on vectorized quadtrees. Unlike the scalar

implementation of trees where the traversal is performed node by node, the vectorized tree algorithms traverse the tree level by level. My algorithm stores the nodes in both scalar format and the vectorized format for the sake of comparison. The quadtree construction is performed as a top-down recursive, breadth first order. The tree descent for the computation of the node vector and the hashes is the part that consumes the most amount of time after the computation of sums for averaging the values of pixels. There are few papers that talk about of vectorizing all aspects of tree traversals by performing tree descents on many nodes parallely on different threads or cores. A unique spatial-key which is a combination of depth, x,y position is transformed into an external hash index that fetches the values of the averaged pixels and summed mask values in the gpu arrays and the cpu arrays. The same hash logic is used for accessing the sums of different quadrants of mask and images. For this reason, no has z-ordering or hilbert ordering was particularly useful for storing values. For the reconstruction of the multiresolution image, a similar approach is used, where the traversal is performed parallely at different levels and different quadrants. Each of the spawned threads can fill multiple parts of the image. If two threads are overlapping trying to modify the same part of image, higher priority is assumed by the thread performing at higher depths of the tree. A check is also performed if a parent node overlaps with a leaf, and precedence is given to the leaf node. This also reasons from the fact that leaf nodes are at deeper levels than parent nodes.

Calculation of sum of elements in mask and averaging pixels of different sized quadrants in mask and image respectively

The different quadrants in the source image frame are considered as Image tiles. The whole 2D image frame with multiple channels and the single channel mask is present in the shared memory among different threads and blocks. Each quadrant's tile size is defined based on the depth of the node. The thread defines the depth of the node, and the blocks work on the same sized image tiles. This way, there won't be overlap of the shared image quadrants between different threads in shared memory. Each of the threads spawn more



child threads for calculating the sum of each column of the image quadrant. The threads are synced using the commands provided by CUDA before summing the values of sums of each column. Because the source and destination are different, the threads are not modifying same data at any point. This reduces any necessity for critical sections in the code. The calculated sums are written to a destination array index given by the depth and the quadrant, hence writing to different memory locations.

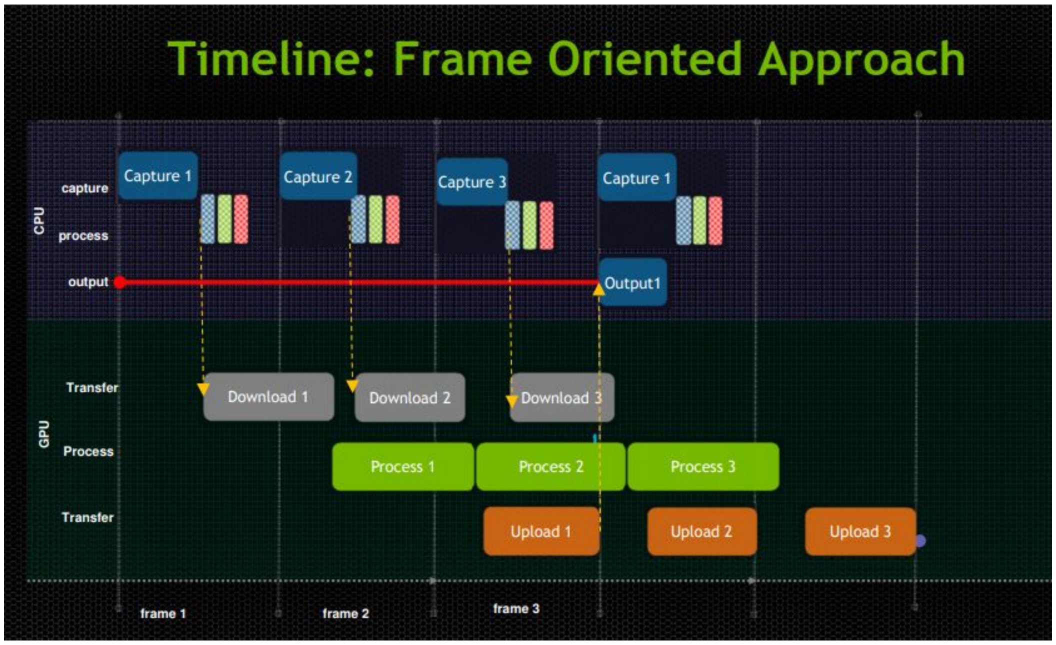
Below is the example of the function prototype that calculates the average sums:

```
void calculate_mask_qsums_gpu(cuda::GpuMat &src, cuda::GpuMat &dst, long
int & lenArray, int & depth);

__global__ void calculate_mask_qsums_kernel(const
cv::cuda::PtrStepSz<uchar> src, cv::cuda::PtrStepSzb dst,int qheight, int
qwidth, int offset, int wid);

__global__ void calculate_mask_qsums_kernel_child(const
cv::cuda::PtrStepSz<uchar> src, cv::cuda::PtrStepSzb dst,int x, int
qheight, int * colsum);
```

The first global function is being called for every depth iteration. This call generates as many kernels as the number of quadrants in that specific depth. Each of those kernels spawns more child threads that calculate the column sum. The parent kernels wait for all the child threads to calculate their column sum, use __sync_threads() function before calculating the cumulative sum to synchronize the sum of all the columns and then finally exit the parent kernels. This is a tradeoff between more kernels and parallelization.

Load imbalance and global queue: Processing multiple video frames

With the use of GPU and CUDA, load Imbalance is greatly simplified because of the shared memory and writing to different location, with no real restriction on the number of blocks and threads. The Video HW captures a frame and transfers to the GPU. The GPU stores the frames in shared memory as described above. The GPU performs operations using the image and tranfers the processed result or image back to the CPU. Opencv's GPU::Mat is used for declaring the matrices and upload and download functions are used to utilize the rendering using copy engines. The copying of the frames back and forth between GPU and CPU causes a latency of about two frames. Memory buffers can also be used(I haven't), which enable video transfers through a shareable system, by eliminating memcopy. The display thread can be executed in the GPU itself to eliminate latency.



Because of the simultaneous downloading, processing and uploading of data to/fro and in the GPU, this problem becomes a mismatched load problem. Another imbalance comes from the issue that the quadtree that represents an image is not a balanced tree. When a quadtree is being processed, the different quadrants of the quadtree can result in a leaf node or another parent node. Hence, more computational resources are required for a parent node in comparison to a leaf node. But because CUDA can spawn more threads from parent threads, all these small threads can be mapped to a global queue maintained by the OS.

Time Complexities(Parallel vs Serial):

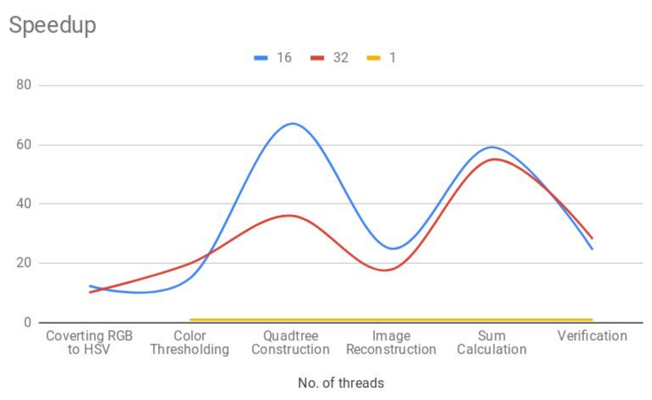
Method	Serial (Worst case	Parallel (worst case

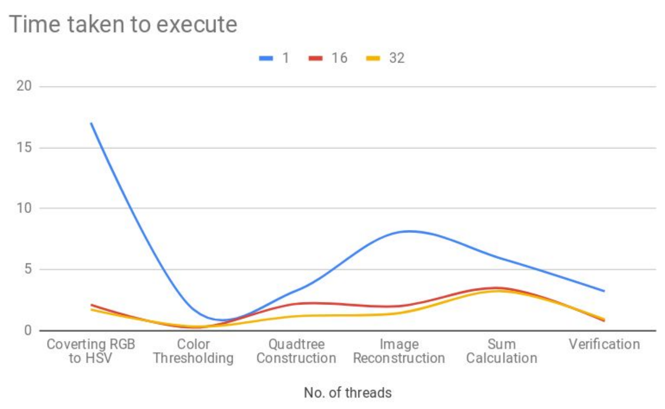
	complexity)	complexity)
Converting into HSV image	O(mn)	O(1)
Color Thresholding	O(mn)	O(1)
Quadtree construction	O(4^(depth))	O(depth)
Image Reconstruction	O(m*n*depth)[linear] quadtree]	O(depth)
Sum Calculation	5.8942 O(mn*depth)	3.24218 O(depth)
Verification through RMSE	3.23021 O(mn)	0.91062 O(1)

Method	Serial (frame)(ms)	ColorParallel(frame) (ms)(32 x 32 threads)
Converting into HSV image	17.0327	1.73369
Color Thresholding	1.4212	0.34551
Quadtree construction	3.27576	1.18226
Image Reconstruction	8.0753	1.44673
Sum Calculation	5.8942	3.24218
Verification through RMSE	3.23021	0.91062

Method	Serial (video: 100 frames)	Parallel(video: 100 frames)
Total Computation time	3940 ms	1040.92 ms

Discussion of Results:





No. of threads	Coverting RGB to HSV	Color Thresholding	Quadtree Construction	Image Reconstruction	Sum Calculation	Verification
16	12.45290664	14.96689658	67.05314187	24.95052815	59.16154864	24.54475096
32	10.17859764	19.92606519	36.09116663	17.9154954	55.00627736	28.19073683

Based on my testing with different number of threads per block, I found that my code performed the best when using 256 threads per block - blockDim.x = 16 and blockDim.y = 16. This is probably because the maximum physical number of cores executable per block is 128, beyond which the real parallelism ends and time-sharing happens. Also with the increased number of blocks and threads, the time taken to synchronize between the threads and blocks also increases. With greater number of threads, the communication overhead also increases, effectively increasing the total time taken.

1. *Single pixel operations - Parallelization of conversion of RGB image to HSV Image , Color thresholding, Verification through RMSE*
   a. The speedup for this serial to parallel conversion was much lower than expected with an average speedup of 12. The process should have ideally been much faster

but seems to have very little speedup although the number of threads have been set to give the maximum possible efficiency.

b. The usage of the GPU matrices defined by opencv seems to be an issue, because of multiple overhead functions that are not accessible to me. The opencv's compiled GPU functions are also another reason for slowing down the parallel algorithms.

2. Quadtree construction, Sum Calculations:

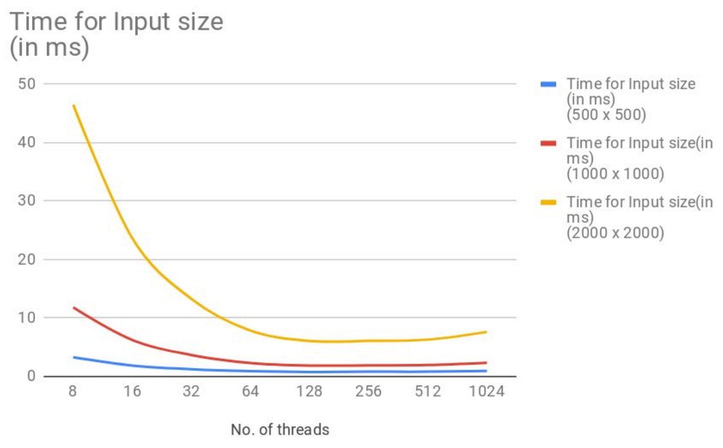
The speedup seems to be about as expected for Quadtree construction. In the serial algorithm, I calculated sum only if the mask was non zero. However, to have a lookup table method for the parallel algorithm, I initially calculated the averages of all quadrants based on the depth and size of quadrants. These unnecessary overhead calculations have proved to add to the time taken to completion.

3. Image Reconstruction

For the image reconstruction process, the image was being written simulatenously by multiple kernels. I initially tried to sort the vector based on depth, however, the sorting the st vector was a little tricky in cuda. Hence, the overhead was higher than expected. A parallel sorting algorithm for the vector or using a priority queue(max) based on the depth of the useful leaf nodes would have been a better way to deal with Reconstruction.

Scaling with Input Size:

Since the images are confined to a 512x512 pixels for easy quadtree construction, the scaling with input size is not extremely important. This is the analysis of the different rescaled version of the images for the calculation of the color thresholding function.



Things that I could have done differently:

1. Initialize the matrices for the three channels of the image and the mask separately by myself, instead of using the cv::GpuMat defined by opencv. Opencv's memory is also not being efficiently handled. Hence the process was slower as the video ran for multiple frames. This seemed to be the biggest issue for slowing down my algorithm.
2. For the purpose of displaying the images, the matrix was being copied back and forth CPU and GPU. I could instead display the image from GPU itself.
3. The video frames were not pipelined for this project. Each frame was being processed one after the other. Pipelining the frames and having global threads in CPU that can manage the pipelining and displaying of frames could have made the program faster.
4. The recursion used in the construction of quadtree was not a tail recursive procedure. Using tail recursion could have made the process faster.

Works Cited

[1] Samet, Hanan. "Hierarchical Spatial Data Structures." *Lecture Notes in Computer Science Design and Implementation of Large Spatial Databases*, 1990, pp. 191–212., doi:10.1007/3-540-52208-5_28.

[2] GPU Programming for Video and image processing by NVIDIA http://on-demand.gputechconf.com/siggraph/2013/presentation/SG3108-GPU-Programming-Video-Image-Processing.pdf