Research Paper

Volume : 2 | Issue : 10 | October 2013 • ISSN No 2277 - 8179

Parallel Algorithm Using Opencl for Depth Estimation of Image

Engineering

KEYWORDS : Boruvka's MST algorithm,, Image Tiling, Split, Scan and Compact, Reduce

R Arokia Priya	Asst Professor, D.Y.Patil college of Engineering, Akurdi, Pune
Shreedhar Gyandeo Pawar	Student, D.Y.Patil college of Engineering, Akurdi, Pune

ABSTRACT

Depth maps have a tremendous importance in Robot Navigation, Real-time tracking of scenes, 2D-3D conversions needed for gaming as well as in animated movies and much more. A fairly good implementation of Segment based Depth Estimation chain using GPU(Graphic Processing Unit) computing has been presented in the work by G. Visentini and A. Gupta (2012), but it lacks in the segmentation step to be implemented in parallel and is a bit naive in other steps like bilateral filtering and Stereo Matching in terms of harnessing the GPU power optimally. We present a novel parallel implementation of graph based segmentation using the Boruvka's Minimum Spanning tree(MST) algorithm in Open Compute Language(OpenCL). We have optimized the GPU performance by making a significant use of shared or local memory of the GPU.

Introduction

The use of parallel processing has increased enormously in the recent years in various fields like computer vision, image processing, data processing, machine learning etc.). Depth maps estimations need to be real time for 2D-3D conversions, for real-time tracking of people at crime scenes and also for robot navigation. The massive multi-threaded model of GPUs make it possible to adopt data-parallel approach even to irregular algorithms like graph algorithms.

We choose OpenCL for parallel computing rather than CUDA due to it's portability on heterogeneous systems (multi-core CPU-GPU integrated) and it's open source environment which allows it to be used on GPUs and CPUs of any brand. The OpenCL program can also be run on handheld embedded systems and FPGAs (Altera has recently launched it's OpenCL SDK for FPGAs). OpenCL also has inbuilt vector data-types, the use of which makes the processing the R,G and B values of an image simultaneous.

We have divided the paper in following sections: In section I we describe the steps for Depth Estimation chain, in section II we give a brief description about the limitations of GPU computing due to it's hardware and how we try to solve it in OpenCL, in section III we simplify the Parallel Implementation of the Graph based Image Segmentation and Stereo Matching step, in section IV we have presented the conclusions and in section V we have discussed about future work.

Segment Based Depth estimation chain

For estimating depth of a given image, state of the art chain considers two stereo images captured from a left view and a right view in order to do stereo matching on them. Disparity (absolute difference of pixels) between each corresponding segment of the images is calculated and depth *Z* being inversely proportional to disparity *d* can be easily calculated with the following formula:

$$Z = (f * B)/d \quad (1)$$

where B is the baseline between two cameras and f is their focal length.

Image Rectification

Rectification of an image which is based on epipolar geometry is done to align corresponding pixel positions of the two stereo images in same horizontal line. Rectification nullifies the vertical variation between corresponding pixels if the extrinsic and intrinsic parameters of the camera are known. This minimizes the errors during the matching stage. All the following stages assume rectified images as input.

Bilateral Filtering

In order to obtain good segmentation results, filtering the noise in an image is required without blurring the edges in the image. Bilateral Filter proposed by C. Tomasi and R. Manduchi (1998) is one such non-iterative, local, nonlinear filter and combines domain (spatial closeness) and range (color similarity) filtering. It smoothens the low frequency or less variable pixel intensity region and maintains the region where there is high variation in pixel intensity. We have optimized the algorithm for parallel computing by making use of Image Tiling and localization of pixel intensity values which give significant speed-ups when the kernel or window size is increased.

Graph Based Image Segmentation

Image Segmentation plays an important role in reducing the computations during the stereo matching step, since rather doing matching on a pixel-by-pixel basis by considering windows of various sizes, we match segments. It is assumed that disparity can't change within a segment. A very efficient graph based image segmentation algorithm is presented in paper by P. F. Felzenszwalb and D. P. Huttenlocher (2004). In this method, all the pixels in the image graph G=(V,E) are the vertices V of the graph and the edge weights E are the pixel dissimilarity(R,G,B pixel intensity difference). Every segment formed is given a representative id which is assigned to each pixel in that segment. This makes the matching and the refinement steps suitable for parallel hardware. But rather than implementing the sequential Kruskal's algorithm as by Giovanni Visentini and Amit Gupta (2012) for building minimum spanning forests of the image, we present a novel way of implementing the Boruvka's MST algorithm which is parallel in nature. The Boruvka's algorithm is irregular in terms of parallelism, so we have divided it in steps for each iteration. We make use of techniques like scan and compact, reduce, split and image tiling for its implementation.

Every MST that has been formed as a segment in a particular iteration keeps on merging with other MST with lesser dissimilarity in the next iteration and the iterations continue till the segments in previous and present iterations are constant.

Stereo Matching

In the stereo matching step one of the image is considered as reference image. Its segmentation map is considered for creating an embedded mask for every region. This mask is decided by every segment's length and their ids. Then, we look for a best match of each segment in the other image and calculate the disparity values for segments that have matched the best. This disparity value is assigned to each pixel in the image. Finally depth map is created using these disparity values which is a grey scale representation of the depths of the objects, the objects that are away are darker and the ones that are nearer are brighter.

Disparity to Depth Conversion
Depth *Z* of an image object is inversely proportional to it's disparity between the stereo images and their relation can be given by the above mentioned formula. Finally, as proposed by Microsoft , depth data can be stored as an 8-bit gray scale image in which a pixel's gray value I is calculated as

$$I = [255 * \frac{z_{near}}{z} * \frac{z_{far} - z}{z_{far} - z_{near}} + 0.5] \quad (2)$$

where *and* are near and far clipping planes.

Refinement
The estimated disparity maps are usually affected by lots of mistakes due to ambiguities, occlusions, repetitive patterns and untextured regions. To reduce these errors, as proposed by A. La Martina, D. Aliprandi and E. Piccinelli (2010) we apply a 3X3 median filter on the depth map and then do a cross-check with the disparity values of the left and right image. If they are same then they remain matched, otherwise we declare them as unmatched.

GPU hardware constraints and OpenCL Introduction
GPU hardware doesn't allow dynamic memory allocation in OpenCL kernel code. The sizes of the 1-D or 2-D buffer arrays used in the kernel code have to be fixed and defined in the host or CPU code. Hence this limits us from using data structures like linked lists, adjacency lists, heaps, stacks or queues. Also the algorithms available are mostly sequential in nature, so we need to break them and bring in some concurrency in them step by step. Once the algorithm is divided in parallel steps GPU attacks with its data-parallel model and processes about millions of elements in few milliseconds.

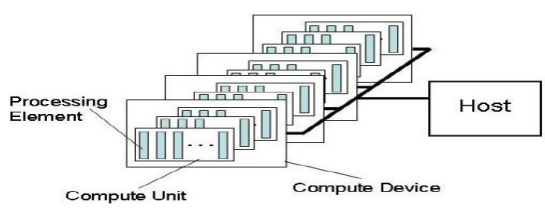

Figure 1 : *OpenCL Platform*

The Typical OpenCL platform is shown in Figure 1. where one host is connected to multiple devices, each multiple device has multiple compute units and each compute unit has multiple processing elements(PE). OpenCL application runs on Host and submits kernel (commands to execute computation on PEs) equivalent to a function in C, PEs execute single stream of instructions, thereby exploiting data-level parallelism.

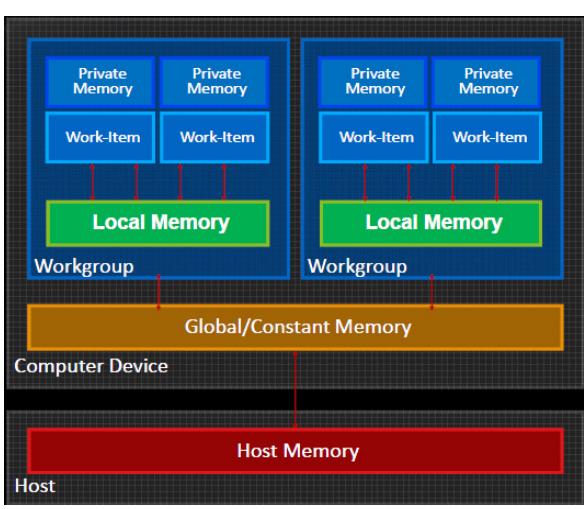

Figure 2 : *OpenCL Memory Model*

Figure 2 shows the OpenCl memory model. The global memory is common to all work-groups and is hence slower to access. The global_size defines the range of computation for a kernel and the local_size defines the size of work-groups. Every work-group has its own local memory and hence this is a faster memory but is limited in size. The work-items working inside a work-group can access the local memory of their respective work-groups only and not of any other work-group. This is the memory that we try to harness while image tiling and localization, by placing a fixed number of pixels in the local memory. This makes the further processing on them faster, since the elements are now accessed from the local memory rather than being accessed from global memory every time.

This also helps while making use of atomic operation (prohibit a work-item's access to a memory location when another work-item is accessing it). Instead of using global atomics, we use local atomics on the tiled local elements and after processing them locally we accumulate them into the global memory as per requirement of the algorithm.

The private memory is private to respective work-items and can't be shared with other work-items even within the work-group.

Parallel OpenCL Implementation
We have used the OpenCL specification 1.2 as a part of the API and Nvidia Quadro FX 880(48 cores) as the GPU hardware.

A) Bilateral Filtering
As proposed by C. Tomasi and R. Manduchi (1998) the general formula for calculating the pixel value at point p is given below. and are the spatial and the range spread of the Gaussian blur.

Where, *p* and *q* are the spatial location of the pixels in the neighbourhood and *I* gives the corresponding intensity values. In simple words, weights in the window keep on changing every time as the pixel intensity () and their positions (p,q) change. This formula can be easily extended for coloured images as well.

The implementation has a complexity $O(Nr^2)$ where N is the number of pixels and r is the radius of the window.

But, as this r increases, the OpenCL implementation that accesses global memory each time starts taking drastically more and more time. Hence we divide the image into tiles i.e. store fixed amount of pixels in the local memory. So rather than accessing global memory every time for every computation, local memory is accessed.

Sizing the tiles is an important point to be considered. We have to decide the size of the tile in such a way that no compute unit sits idle and even there is maximum use of the local memory. Making use of all compute unit demands for making more number of tiles for a given global-size. And making maximum use of local memory demands for making the size of a tile greater. Hence we maintain a trade-off between number of tiles and tile size and optimize the GPU computations.

B) Image Segmentation
We have proposed a novel way of implementing graph based image segmentation using the Boruvka's Minimum Spanning Tree algorithm. As proposed in the paper by P. F. Felzenszwalb and D. P. Huttenlocher (2004) segments are formed by connecting similar pixels by spanning the whole image.

- At the very start, when there are no MSTs, using the Boruvka's algorithm which selects the minimum edge for every vertex to be in the MST, pixels get connected with other pixels one by one in a parallel way if their pixel intensity difference is smaller than a starting threshold value *k*.
- As the pixels start getting connected the segments begin to grow.
- Now when there are more than one pixel in a segment, it connects with the other segment only when the minimum external edge of those segments (external edges are all the edges that connect the two segments to be merged; they can

Volume : 2 | Issue : 10 | October 2013 • ISSN No 2277 - 8179

be more than one, out of which the minimum is selected) is less than sum of all internal edge(internal edges are those edges that connect the MST of a segment internally) and starting threshold *k* divided by the number of pixels in a segment. This we call the *segmentation criteria* henceforth.

To make this work on parallel hardware we have proposed the following algorithm:

- Every pixel position is taken as input in a source array, its neighbours' pixel position in a destination array and their intensity difference in a weight array. Only 4 neighbours per pixel are considered, since the other 4 are considered by pixels to the side and above. An example of some random pixels is shown below and their minimum edge weight is marked.

src	17	17	17	17	18	18	18	18	19	19	19	19
dest	18	24	25	26	19	25	26	27	20	26	27	28
weight	40	35	10	67	86	28	96	18	68	82	36	44

Figure 3: *Chunk of InputArrays*

- Minimum edge weight is found for each pixel and placed in min_src, min_dest and min_weight arrays. As the start for each vertex in the array is known(multiple of 4), for each vertex the OpenCL kernel calculates the minimum edge in a data-parallel manner. Following figure shows minimum edges obtained for pixel positions 17 to 29.

min_src	17	18	19	20	21	22	23	24	25	26	27	28	29
min_dest	25	27	27	21	30	31	30	25	26	27	31	31	30
min_weight	10	18	36	15	25	44	12	10	38	6	46	24	51

Figure 4: *Minimum-edge arrays*

- As the minimum arrays are found, a super-vertex_id array is updated which stores the representative id for every segment formed. This step is sequential, since the segmentation is based on a region growing method and at every instance of acquiring a pixel it needs to be checked for the segmentation criteria. Also an internal sum array with the edge-weight and internal count array is updated for the respective super-vertex as every source-destination pair is given its id. This internal edge array and count array are used to check the segmentation criteria.
- As these *ids* are updated, a flag for every pixel position is set, as the pixels get visited. From the flag array we know what *id* is to be updated. If both flags(source and destination) are 0, then a new id is given to both. If one of them is 1, then id of whose flag is 1 is given to other. If both are 1 and ids of both are same, then they are from the same MST, hence this edge is not included in the MST as it will form a cycle. If both are 1 and ids are different, then one with lesser internal count is given the id of the one with greater count.
- Following figure shows how the pixel positions are given their supervertex_id. Using minimum arrays above in figure 4, 17 and 25 are given a fresh id 0 as their flags are 0 initially. Then 18 and 27 are given 1. Now in the next pair (19-27) 27 is visited, hence 19 is given the id of 27 i.e. 1. Next 20 and 21 are given 2, since both their flags are 0. Such a process continues for all positions. The internal sum array and count array are also updated.

positions	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
supervertex_id	0	1	1	2	2	2	2	0	0	1	1	2	2	2	2

Figure 5: *Supervertex_id array*

- But for the pair 27-31, both the *ids* are updated, so as per our last condition 27 should be given the id of 31 i.e. 2. Here we have a problem, that there are other vertices too that are connected to 27(18,19,26) and need to be updated with id 2. Hence the *id* of both 27 and 31 and all such other pairs are stored in an array to be updated later. Such super-vertices are updated only if they fulfil the *segmentation criteria.*
- To make this step parallel, we apply a split technique on the supervertex_id array as proposed by S. Patidar and P.J. Narayan (2009). This technique brings all the same supervertex_ids in a contiguous position. Also the start positions of all the segments are stored, so that the updating process takes place in parallel. We make use of local atomic operations for split. After split the arrays look as below.

supervertex_id	0	0	0	1	1	1	1	2	2	2	2	2	2	2	2
positions	17	24	25	18	19	26	27	20	21	22	23	28	29	30	31
start_position	0	3	7												

Figure 6: *Split.*

- Now we can easily find out where exactly the segment containing 27 is and even all such segments that need to be updated. And this updating process for each segment can take place in a parallel way. In the example, we suppose that the segmentation criteria is not fulfilled and let 3 segments exist instead of 2.
- The former updating process which is sequential in nature needs to be divided and conquered. Hence we divide the image into tiles and store the pixels for a tile in the local memory. Now each tile runs in a parallel way and though region growing takes place in a sequential manner within a tile, we achieve speedups due to parallel running of each tile.
- Later, the segments at the border of these tiles are merged using their external edges and segmentation criteria.
- Now from the very first source and destination arrays shown in figure 3, the external edges are separated from the internal edges, by using the supervertex_ids for each pixel. If they are different, then those edges are treated to be external and their positions as well as their supervertices are stored in different arrays. To make this happen in a parallel way and to store the required edges in a contiguous form we need to perform a scan and compact operation on the input array. The positions of required external edges are marked 1 in a different array(marked_array) and an inclusive sum-scan is performed on the marked array which gives us the positions in the new compacted array. This process is shown in figure 7. The marked_array is obtained using supervertex_id array of figure 5 in a straight forward parallel way. Then the scanned array is obtained by simply summing the marked array till the present positions. For this a parallel step-efficient O(nlogn) prefix sum scan is implemented. Figure 7 shows the marked and scanned arrays.

src	17	17	17	17	18	18	18	18	19	19	19	19
dest	18	24	25	26	19	25	26	27	20	26	27	28
marked_array	1	0	0	1	0	1	0	0	1	0	0	1
scanned_positions	1	1	1	2	2	3	3	3	4	4	4	5

Figure 7: *Scan*

- The scanned positions allow us to store the external edges in contiguous positions in the new compacted arrays as shown in *figure 8*. This gives a significant GPU speedup

for further processing, as GPUs work faster on contiguous memory.

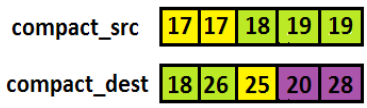

Figure 8: *Compact*

- Compacted supervertex_id arrays are required for actual further processing. Hence while storing the positions in the compacted array, super-vertices are also stored in corresponding positions in different arrays as in *figure 9*.

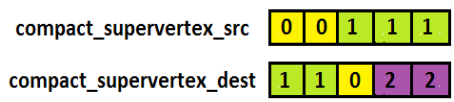

Figure 9: *Compacted superveryex_id arrays*

- There are more than 1 external edges between two segments (i.e. two super-vertices). So there is a need to find the minimum external edge between pairs of segments. But here again the same pairs may not be at one place. So again the split technique is used to bring them in one place and their start positions are calculated, so that every super vertex-pair's minimum edge is found in parallel.
- After finding the minimum edge between all pairs of super-vertices, these super-vertices need to be merged. This merging involves updating of the supervertex_ids of all pixels in the super-vertex. The super-vertex with less number of pixels gets the i*d* of one with more number.
- This completes the 1st iteration. For further iterations again the external edges are separated from the remaining edges and the segments are merged till the number of segment in the previous and present iterations are same.
- Almost all the computation has an *O(V)* work complexity, where V is the number of vertices or pixels to be processed at a particular iteration. Scan operation has a work complexity of *O(VlogV)* as mentioned above.
- The initial iteration takes about 90-95 milliseconds for an image: cones (450X376) used by G. Visentini and A. Gupta (2012) and this goes on reducing for further iterations. Normally 3 to 5 iterations are needed for any kind of an image. The total time taken for all iterations to complete is about 400 milliseconds. The Kruskal's sequential algorithm takes about 1 second with same image size and same hardware.

C) Stereo Matching

Stereo Matching consists of matching the segments formed in left and right image using the sum of squared difference method (SSD). When a segment is to be matched with other segment, difference between corresponding pixels in left and right images is taken, then these values are squared and then their sum is taken. The segment for which this sum is the least is declared as matched. The parallel implementation is as follows:

- In the segmentation step itself, all the segments are aligned in a continuous way using the split technique. Even the start positions are calculated for all the segments. This makes the matching step very easy to be implemented on parallel hardware.
- Due to this, segments from the reference image can be matched in a parallel way with segments in the other image.
- Only one OpenCL kernel is launched which matches one segment in the reference image with segments in the other image in a parallel way.
- This requires fairly less time, due to proper implementation and arrangements made in segmentation step. Image: cones takes about 390 milliseconds.

Results and Conclusions

We have successfully proved that our parallel implementation of Graph based Image Segmentation using the Boruvka's MST algorithm achieves significant speedups against the sequential Kruskal's Algorithm and the prior arrangements of segments in a contiguous way along and the storage of start positions give speedups in Matching step. Even the filtering part gains speed due to storage of pixels in local memory for larger window size. Below in Table 1 are the results that we have speeded up compared to naive implementations done before. Naive implementations consists of sequential nature of the algorithm or which are done without considering where the GPU hardware is good at. We have obtained both the results on our hardware for transparency.

	Naive Implementation (milliseconds)	**Speedup Implementation (in milliseconds)**
Bilateral filtering (Window radius=7)	850	210
Image Segmentation	990 (Kruskal's MST)	400 (Boruvka's MST)
Stereo Matching	1220	390

Table 1 *Results for speed ups against naive implementation for the image: cones(450X376) used by G. Visentini and A. Gupta (2012), both obtained on our hardware Nvidia Quadro FX 880.*

We have concentrated more on making the algorithm suit the GPU hardware, rather than on the depth estimation method.

GPU hardware doesn't allow dynamic memory allocation. Hence data structures like queues, stacks, adjacency lists, heaps, linked lists can't be implemented inside the openCL kernel code. Implementation of techniques like reduce, tiling, scan and compact, split overcomes these constraints of GPU hardware.

The more we reduce the sequential part of the algorithm, the more speedups we get. GPU always gives speedups when it processes coalesced memory rather than random memory. Global atomic operations reduce the speed of GPU performance, so local atomic operations should be used to achieve speedups. Local memory should be used optimally, but not at the cost of keeping any compute units idle. We have designed our algorithm considering all these points, whereas the naive implementation doesn't consider these points.

Future Work

For future work following points can be considered :

- Machine Learning techniques like neural networks and support vector machines can be applied to segmentation algorithm for more accuracy. As neural networks are parallel in nature, they will give huge speedups with GPU computing.
- The hardware used by us has a compute power 1 which allows only data-parallelism. The GPU devices that have compute power 2 or more allow task parallelism as well. Efficient algorithms can be designed that harness the power of task-parallel hardware.
- As OpenCL code can be interfaced with Altera FPGAs, we can use this code for robot navigation or in driverless cars with more compactness and more robustness.

Volume : 2 | Issue : 10 | October 2013 • ISSN No 2277 - 8179

REFERENCE

[1] A. Munshi, "OpenCL Programming Guide," Addison_Wesley(2012) | [2] developer.amd.com/opencl | [3] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient graph-based segmentation," International Journal of Computer Vision, vol. 59, n. 2, pp. 167–181, 2004. | [4] A. La Martina, D. Aliprandi, E. Piccinelli. Segment-based depth mapsextraction from stereo video content. 11th International Workshop onImage Analysis for Multimedia Interactive Services (WIAMIS), in April 2010. | [5] S. Patidar and P.J. Narayan, "Scalable Split and Gather Primitives for the GPU" in High Performance Graphics (Poster), April 2009. | [6] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in Proc. of the Sixth International Conference on Computer Vision, Bombay, India, 1998. | [7] G. Visentini and A. Gupta, "Depth Estimation using Open Compute Language(OpenCL)", International Conference on Latest Computational Technologies (ICLCT'2012) March 17-18, 2012 Bangkok | [8] www.khronos.org/opencl/ |