MATTER: International Journal of Science and Technology
ISSN 2454-5880

Adeel Muhammad, 2016

Volume 2 Issue 2, pp.65-85

Date of Publication: 15th November, 2016

DOI- http://dx.doi.org/10.20319/mijst.2016.22.6585

This paper can be cited as: Muhammad, A. (2016). An Evaluation of Connected Components Labelling Using GPGPU. Matter: International Journal of Science and Technology, 2(2), 65-85.

AN EVALUATION OF CONNECTED COMPONENTS LABELLING USING GPGPU

Adeel Muhammad
University of Stuttgart, Germany
adeel.1989@yahoo.com

Abstract

Connected components labelling (CCL) is one of the basic steps in various image-processing applications such as image segmentation or recognition while performing surveillance or medical imaging. Due to increased demand for real-time processing; fast and efficient connected components labelling and analysis has become significant. It is a resource and time intensive process but if parallelized, it can be done efficiently with much higher performance. The implementation presented makes use of two-pass algorithm by exploiting parallelism provided by graphics card using Open Computing Language (OpenCL). Performance of GPU with varying CPU loads is examined. At the end, performance results are compared with different images and serial implementation on CPU due to its serial nature of execution.

Keywords

Connected Components Labelling, CPU, GPGPU, OpenCL, Image Processing

1. Introduction

1.1 Background

The notation of pixel connectivity describes the relation of a pixel with its immediate adjacent neighboring pixels. In order for two pixels to be considered connected, their pixel values must be from the same set of values V. For a binary image, V comprises of two values

- either 0 or 1. Two pixels are considered connected if and only if they are adjacent and they share the same value from set V (Rosenfeld A. & Pfaltz J., 1966). This means that we are considering two properties of a pixel during our operation. During connected components labelling (CCL) algorithm, we traverse through image in raster fashion and we can skip the neighboring pixels which we have not traversed yet. This means, immediate right pixel and next row pixels are ignored. Therefore, if 8-connectivity is considered, for each object pixel that is to be examined, the 4 neighbors that have already been processed are examined. The examined pixels are shown in Fig 1.

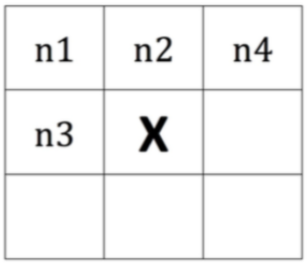


Figure 1: *8-connected neighbors to current pixel X are shown. n1 to n4 are pixels to be examined during this article*

As of second property, a set S of pixels is a connected component if there is at least one path in S that joins every pair p, q of pixels in S, the path must contain only pixels in S. In order to do analysis of connected components in later stage, there is a four-stage process. First the input image is preprocessed through filtering and threshes holding to segment the objects from the background. Secondly, the connected components labeling is used to assign each region a unique label, enabling the individual objects to be distinguished. In the third stage, each region is processed to extract a set of features of the object represented by the region. In the final stage, these features are used to classify a region. In our implementation, we only look at stage two where we assign unique labels to distinguished regions.

1.2 Platform Choice

In order to improve performance of such process, parallelism provided by modern day General Purpose Graphics Processing Units (GPGPUs) can be exploited. To program GPGPUs, standards like CUDA, OpenGL or OpenCL can be employed (Karimi N.G.D.K & Hamze F., 2010). In presented implementation, OpenCL has been proposed which is a cross-vendor standard that holds potential to exploit the massive parallelism. OpenCL provides platform-independent, parallel execution model to target heterogeneous systems including multiple central processing units (CPUs), graphics processing units (GPUs) and digital signal processors (DSPs). Scientific computing using OpenGL involves mapping the problem to graphics context in terms of textures and geometric primitives like triangles, whereas in case of CUDA and OpenCL, memory buffers like variables can be declared and computations can be carried out directly on them. CUDA and OpenCL are obvious choices in this implementation over OpenGL due to nature of problem statement. Choice between CUDA and OpenCL is made due to platform independence option and open source nature of later.

1.3 Previous Work

There exist many algorithms and strategies for effective and efficient fast connected components labelling. Benkrid and Crookes in (Bailey D. G., & Johnston C. T., 2002) have implemented an efficient algorithm for CCL on FPGA but the multi pass algorithm presented requires buffering of more than one intermediate images. Although with present day advance electronics, memory storage is not a significant issue but multiple passes make it unsuitable for real real-time processing. The process discuss here also performs two scans over the image which means two processing cycles per pixel but intermediate image is saved only once. There are other CCL techniques which require only one pass over the image for proper labelling which means no buffer required storing intermediate images (Benkrid K. et al., 2002) (Montes M C.,2010) but such algorithms perform efficiently where input images are streamed into the processor. For such an input, a high bandwidth memory transfer is required which might become bottleneck.

2. Algorithm, Label Selection and Storing Data

The algorithm makes two passes over the image: Each pixel of the image is processed in three stages. In the first stage, a temporary label is assigned to each object pixel and equivalence table is generated if multiple labels correspond to same region. In the second stage, the representative label of its equivalence class replaces the label assigned to each pixel. In third stage, all the unnecessary labels are removed from the equivalence table. During this process, an intermediate image is generated after the first pass. This intermediate image is scanned and temporary labels are replaced by final labels in second pass. The major disadvantage of this implementation is the cost of application in real time processors because of latency introduced and memory required to store intermediate image. With present day electronics, we have sufficient memory required to buffer the image. Hence if latency is reduced or time of operation on a pixel is reduced, better performance can be achieved. The processed image is available for third stage after the whole image has been processed.

Implementation discussed below includes execution of proposed algorithm at two different platforms. Next section describes the algorithm in general which is followed by description of special cases for label marking and merging of table entries. An efficient way of storing labels in appropriate data-structure is also discussed. Implementation of algorithm is discussed on CPU and GPU, which is followed by results, profiling data and overall performance.

2.1 Label Selection

Label selection is performed only if current pixel is not a background pixel. It can be summarized as: The label for interested pixel is selected depending upon its neighbors which include n3 from current row and n1, n2, n4 from previous row. When a region is encountered for first time, a new label is assigned to it. If there is one label in the neighborhood, it is assigned to current pixel. If there are two different labels within the neighborhood (for example at the bottom of a U shaped region) as shown in Fig. 2, then merger table needs to be updated. During the second pass, information from merger table and labeled image with temporary labels is used to decide final label. In a case of two different labels in neighborhood and n2 is a background pixel, these two labels have to be same and must be merged. For such a case to occur, there are two special cases where n2 is the background pixel and mergers only need to be considered between the neighbor pairs of n3-n4 and n1-n4. Both the cases are represented in Fig. 3. In such cases, n3 and n1 are automatically selected without being compared respectively.

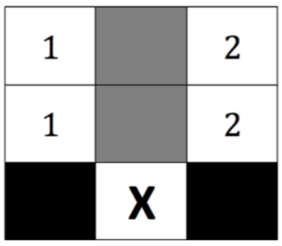


Figure 2: *Two different labels in neighborhood. X is pixel of interest while two different labels exist in previous row 8-neighborhood. 1 and 2 are arbitrary values of labels*

2.2 Algorithm for Label Selection

The algorithm devised to select appropriate label for an interested pixel X is given in Listing 1 in appendix where lnx describes the location of pixel X and nx describes the value of pixel x in original image. For binary images, this value shall be either 1 or 0. For every interested pixel, first the neighbor at location ln1 is checked. If the value of this pixel in image is greater then 0 then it is understood that it has been assigned a label. Therefore, the label of ln1 is copied to pixel X. After label assignment, merger is check for pixels at ln2 and ln3 if they are non-zero in original image. This step is followed by a special case of merger, which is considered as Case 1. This special case is discussed in Fig 3. After successful assignment of pixel, rest of conditions can be skipped and next pixel is considered. This enables to avoid considerable number of unnecessary conditions being checked. If n1 is a background pixel then label value can be ignored, which will be zero. If n2 and n3 are also background pixels then n4 is checked. If n4 is not a background pixel, its label is copied to current pixel. Otherwise it can be inferred that pixel X is not connected to any of probable 8 neighbors. Therefore, a new label value is assigned to X. Rest of conditions are skipped as discussed earlier. In a case where one of n2 or n3 is not a background pixel, n3 is given priority on arbitrary basis. If n3 is not background, label of n3 is copied as label of X. Here a probable merger is checked with ln4. This is a special merger case 2, which is discussed in Fig 3 in detail. In case n2 and n3 both are labeled, then label of n3 is copied to label of X and they are checked for equivalence. Whenever there is a merger, this must be looked up in the merger table to correct the label. As each pixel is processed, the label table is updated to reflect the current state of each region. When regions are merged, the corresponding label table entries are also merged to reflect the combined object.

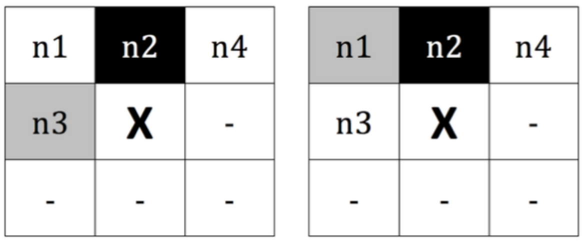


Figure 3: *For such special cases to occur, n2 has to be a back- ground pixel. Merger has to be considered between n1-n4 in first case and n3-n4 in second case.*

2.3 Storing Merger Table using Dis-Joint Set

A disjoint-set data structure maintains a collection S = S1, S2, . . . ,Sk of disjoint dynamic sets. A representative identifies each set, which is some member of the set. Each tree is a disjoint set in a universe of multiple sets, meaning that no item is in more than one set. If two elements are in the same tree, then they are in the same disjoint set. In other words, two sets are disjoint if their intersection is null. The root node (or the topmost node) of each tree is called the representative of the set (Munshi E. A., 2008). There is always a single unique representative of each set. It supports two operations:

- Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset
- Union: Join two subsets into a single subset

In order to optimize, path compression is employed. Whenever a find operation is performed this technique attempts to flatten the tree structure as much as possible. As parent pointers to the root node are traversed during find operation, all elements found along the way share the same root. Parent of each node encountered during traversal to root node has same root as child node. Thus one traversal is performed to find out the root node followed by second traversal during which each node's parent pointer is set to the root node. This has the effect of majorly flattening the structure and therefore decreasing the time required for future operations to fetch root node. The optimized union-find data structure makes the connected components labeling algorithm more efficient. The first pass of the algorithm performs label propagation to propagate a pixel's label to its connected neighbors. At the end of the first pass, each equivalence class has been completely determined and has a unique label, which is the root of its tree in the union-find structure. A second pass through the image then performs a translation, assigning to each pixel the root label of its equivalence class.

3. Implementation

In order to compare the execution time and calculate performance gain of parallel execution with respect to sequential execution, same program was implemented on CPU and GPGPU. As a test image,a randomly generated image of size 8192x8192 pixels is used. Generated image was passed through pre-processing and filtering. As an input to CCL algorithm described, a black and white image with pixel values of ones or zeroes is available.

3.1 Input Image

Although a real life image could be taken as an input but it can result in content specific results. To consider a generic case, a random image can be generated. In order to consider extreme cases, images with zero and hundred percent object pixels can be generated. Images were also generated with 25%, 35%, 50%, 75% and 85% background values which mean non-object pixels. Such images were generated with random value generator seeded by current time. Each image was generated three times and performance results were computed with averaged out results. By having randomly generated images with different amount of object pixels, overall performance distribution curve can be generated to enable prediction of performance values when real life images are used. Such a distribution curve will also depict the overall usage of GPGPU resources.

3.2 Memory Requirements, Management and Transfers

In order to store all the data and successfully execute the algorithm, it is necessary to have efficient and fast memory accesses in practice. For merger table, data structure used will be Dis-joint set. With close examination, it can be deduced that size required for merger table is equal to maximum number of labels. In a worst-case scenario image, where every alternate pixel is an interested pixel or non-background pixel, maximum number of labels is half of image width or length. Therefore, in worst-case scenario of 2D image, maximum number of regions encountered is one fourth of total number of pixels (Khanna P. G. V. & Hwang C., 2002) (Lumia L. S. R. & Zuniga O.,1983). This can be shown in Equation 1,

$$\text{total\_number\_of\_labels} = (\text{countX}/2 * \text{countY}/2) \qquad (1)$$

Where count X and count Y are number of image pixels in x-dimension and y-dimension respectively. The input image is read in an array at host side (CPU), which is then copied into device (GPU) memory. At GPU, the image is copied into global memory space, which has high latency when accessed. If we examine the algorithm, it can be seen that whenever a pixel is accessed, pixels that are adjacent to it, are also required. Due to the fact that array elements are accessed in spatial locality, if image data is cached, a significant gain in performance can be achieved, as global memory is not accessed again and again. This can be seen in memory model as shown in Fig. 4. There- fore, OpenCL images are used to store the input image data, which shall provide data caching.

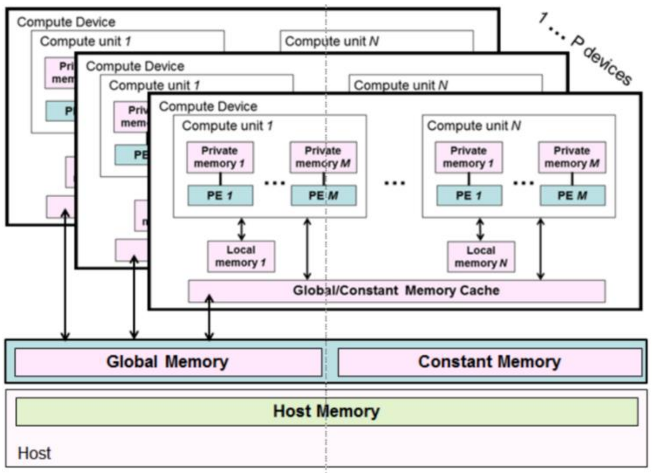


Figure 4: *The named address spaces exposed in an OpenCL Platform. Global and Constant memories are shared between one or more devices within a context, while local and private memories are associated with a single device. Each device may include an optional cache to support efficient access to their view of the global and constant address spaces [6]*

As we can see in Fig. 4, every work item has its own private memory. Access to this kind of memory is fastest along with local memory access. Once a work item has read the required image data, data is stored in its private memory. With the help of caching of data and private storage, memory access latency is reduced to minimum. In order to write the final image, any adjacent pixel values are not required, thus we don't require caching. A simple array can be used to store the values. Each work item writes at its own pixel location. An array of image size is used to store the output image, which is read back to host memory from GPGPU memory buffer. During the whole execution of algorithm, we have to transfer the input image to device and copy back the output image. Merger table can be generated at device in order to reduce memory transfer time.

3.3 CPU Implementation

As only one CPU is available in a system, execution code had to run sequentially and execution time was calculated. In CPU execution, it needs to be taken in account that CPU has to perform a lot of other miscellaneous tasks for example input output, memory management, interrupt handling and user defined tasks. This means that execution time on CPU is highly dependent on processing load at any given time. For CPU implementation, problem can be divided into two parts. First initial labels are assigned which are constantly updated and stored as temporary labels in merger table as described in Section 2.1. In second part, final labels using information stored in merger table replaces all temporary labels. No explicit parallelism was introduced between the cores. CPU was allowed to parallelize any part of code or divide thetask in multiple threads on its own. Thus original CPU execution model of multi-core CPU was utilized.

Advantages of such multi-core CPUs is naturally the proximity of multiple CPU cores on the same die. With presence on same die, the cache coherency circuitry can operate at a much higher clock rate than if the signals had to travel off-chip. Combining equivalent CPUs on a single die significantly improves the performance of cache snoop (alternative: Bus snooping) operations.

3.4 GPU Implementation

For GPU implementation, problem can be divided into three separate parts. First part divides the image into slices of 4x4 pixels and assigns initial labels. Second part merges all the tiles and updates the merger table in multiple steps, thus gradually increasing the tile size. Third part of execution assigns the final label and generates output image. On GPU, one can have theoretically unlimited number of threads (work items) that can execute in parallel. If every work item is given one pixel, a maximum of 8192x8192 pixels can execute concurrently. Although such a tremendous parallelism is achievable where every pixel resolves its own unique label but it is not practical as every interested pixel requires its neighboring pixel values as described in Section 2. During initial labeling of any given pixel, at least its four neighboring pixels as shown n1, n2, n3 and n4 in Fig. 1 are required.

3.5 Formation of Tiles

As the image is a 2D image, the image can be partitioned into smaller slices to increase parallelism. This way, the work done by individual work items can be reduced and all work items can run algorithm concurrently on separate parts of image. For discussed example image of 8192x8192 images, by dividing the image in horizontal slices or vertical slices, a maximum of 2048 slices can be achieved. One work item shall work on each slice and thus we require a maxi- mum of 2048 work items. In case of vertical slices, each work item shall work on four pixels wide and 8192 pixels long slice. Although it shall provide significant parallelism but it can be further improved by using tiles rather slices.

A slice of 32x32 pixels allows us to use a grid of 256 work items along x-axis and y-axis. Thus a total number of 65536 parallel work items. This arrangement allows 32 times more parallelism but due to more number of merging operations and subsequently more kernel invocations, performance gain is not with same factor. Although reducing the slice size shall result in fine-grained parallelism with better performance but it also increases over head for more merging operations and kernel invocations which reduce the overall performance.

Therefore, performance is improved but not with same factor with which slice size is reduced. Smallest tile size possible for stated algorithm is 4x4 pixels. With use of OpenCL images, work items in both X and Y dimension can be used. Such a grid of work items allows us to have 2048x2048 work items concurrently at first stage. Any work item can later be accessed using its local id and global id.

3.6 Labeling and Boundary Merging

In specific example case discussed, 4x4 pixels slices, a total of 2048 x 2048 work items run algorithm stated in Listing 1. Each work item collects its required data via caching and store in its private memory. A new label is assigned to every new encountered region. In a slice of 4x4, there can be a maximum of four new regions. As labels have to be merged afterwards, the label values should not be overlapping. Each slice has to assign separate labels. This can be made sure using equation 2

$$\text{next\_label} = (8192 * \text{global\_Y\_id}) + (4 * \text{global\_X\_id}) \tag{2}$$

Where next label is the number from which every work item starts labeling and *globalids* are global identification number in X and Y dimension. After successful temporary labels assignment, merging steps start. Every alternate slice boundary in X and Y dimension is resolved in one kernel invocation. Boundaries to be resolved can be computed by each work item using equation 3 and 4

$$\text{i\_bndry} = \text{first\_bndry} + (\text{col\_count} * \text{alt\_bndry}) \tag{3}$$

where i_bndry is the boundary to be resolved in x-dimension, first_bndry is one less than initial offset where first boundary exists, col count are number of slices in x-dimension and alt_bndry is twice the offset because every alternate boundary has to be resolved.

$$\text{j\_bndry} = \text{first\_bndry} + (\text{row\_count} * \text{alt\_bndry}) \tag{4}$$

where j_bndry is the boundary to be resolved in y-dimension, first_bndry is one less than initial offset where first boundary exists, row count are number of slices in y-dimension and alt_bndry is twice the offset because every alternate boundary has to be resolved. The boundary merger algorithm is described in later section. At each boundary merger step, two boundaries are merged in x-dimension and two in y-dimension. This means that in first step, every work item resolves eight pixels in each dimension. After successful completion of first step, slice size will grow from 4x4 to 8x8 pixels. In next merging step invocation, slice size shall increase to 16x16 and number of work items required will be halved. This process can be seen in Fig. 5.

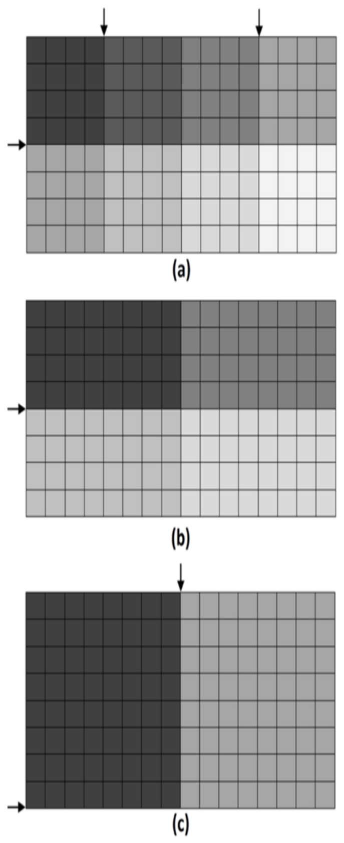


Figure 5: *Arrow marked shows the boundary to be resolved. In (a), both horizontal and vertical boundaries are to be resolved. Two different workitems work on both vertical boundaries. Result can be seen in (b). After vertical, horizontal boundary is resolved and finally four different la- belled regions are combined as shown in (c)*

This process shall continue until there are four big slices of 4096*4096 pixels and we require only one work item to resolve.

3.7 Final Label Selection

After all boundaries are resolved, unique temporary labels are assigned in whole image. Each distinct region in whole image is assigned a unique label which has common root value as described in disjoint sets in Section 2.3. The final labeling step is doing nothing but reading every label and finding its root label from disjoint set and replacing root value as its own label.As there is no dependency of pixels on neighboring pixels, we can perform this in parallel. Each work item is assigned one pixel and they find their root value from merger table. There can be no race condition as merger table is shared resource and data is only read. Therefore, 8192x8192 work items perform read and write operation in parallel. During writing, everyone accesses unique array index.

3.8 Boundary Resolution Algorithm

During the boundary resolution operation, each work item finds its boundary to resolve using equation 3 and 4. Once a work item starts its operation, it goes on to merge two boundaries in x-dimension and two in y-dimension. On arbitrary basis, first we have chosen y-dimension to be resolved followed by x-dimension. X can only be started once all pixels are resolved in Y by one work item. The pixels used during merging boundaries are shown in Fig. 6.

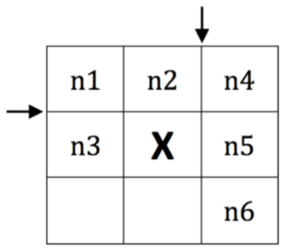


Figure 6: *Pixels required for Boundary Resolution and their Respective Identifiers*

Each work item starts traversing in vertical direction, one pixel before the boundary. This means that during vertical resolution, n4, n5 and n6 belong to next tile. Each traversal starts from zero till the vertical width of tile. In first case, value ranges from zero to three because there are four pixels along boundary. After successful merging operation in both dimensions, the second traversal range will be from zero to seven as there will be eight pixels in each tile. This can also be seen in Fig. 5. After vertical operation, same work item starts in horizontal direction. Now the pixels row to be resolved is taken as one row under the boundary. This means that for a given interested pixel X; n1, n2 and n4 belong to previous tile. Like in vertical case, in horizontal, the boundary range keeps increasing with each successful merging of tiles. This is shown in Fig. 5.

The algorithm for boundary traversing is given in Listing 2 in appendix. Horizontal operations can be applied before vertical or vice versa. During both the operations, if interested pixel X is not background pixel, middle pixel is always checked first for possible merger. This means that n5 and n2 are always checked before the rest of pixels. In case middle pixel successfully merges, the operation is completed because n4 and n6 would have been merged in temporary labelling of tiles as discussed in Listing 1. Every successful work item merges a total of four tiles which means that number of work items in subsequent invocation will be one fourth of previous invocation. This shall continue until we have one work item which merges four big tiles of 4094x4096.

3.9 Results and Verification of Resulting Images

The algorithm discussed above was made to run on different platforms. Different sizes of slices were used on GPU. With repeated executions, slice size of 4x4 was considered the most efficient and thus taken into account. Different types of images were used with variable amount of object and background pixels. Performance results were generated and compared with CPU under different amount of work loads.

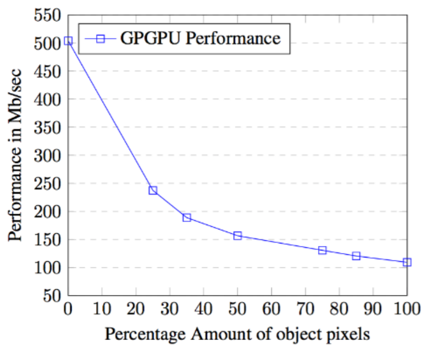
3.10 Image Generation

In order to evaluate the effect of number of object pixels on GPGPUs and CPUs, several images were produced with 0% object pixels to 100%. In order to generate 25% object image, a loop is run till 25% of number of image pixels. Random number generator can be used to select any random coordinate which can then be assigned object pixel value which is one. If selected random coordinate has object pixel already, a new coordinate value is computed until its a background pixel. In such a manner, it is made sure that total number of object pixels is exactly 25% of number of image pixels. This process is repeated again and again to obtain images of 0%, 25%, 35%, 50%, 75%, 85% and 100% object pixels.

4. Performance

4.1 CPU Performance

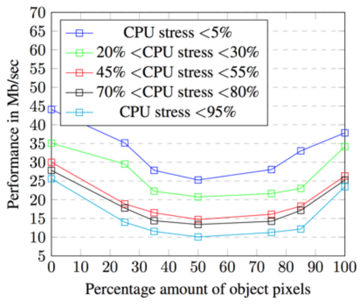
The results of CCL algorithm running sequentially on CPU (Intel Core2Quad Q9650) with allowed instinctive parallelism between its four cores is shown in Graph 1. CPU was stressed with different range of workloads to examine the overall performance. Algorithm was run repeatedly in order to obtain averaged out results. CPU stress of fewer than 5% is only possible when it is reaching its ideal state with almost no other interference from any other process. This can only be achieved if we have a dedicated processor that executes only given algorithm apart from basic operating systems tasks. Under normal usage of CPU, the stress load lies between 20% to 30%. It can be seen from Graph 1 that with little extra load, the performance decreases by significant factor. During the execution on CPU, it can be observed that 50% object pixels in whole image results in worst case performance. This behavior is due to worst case execution image where every alternate pixel is background pixel. Due to same amount of object and background pixels, we require most number of labelling and mergers. There is also more probability of special merge cases as discussed in section 2.



Graph 1: *CPU work load dependence on CCL*

4.2 GPU Performance

GPGPU used to run the algorithm was NvidiaQuadro k5200 with 2304 processing units with clock frequency of 650Mhz. Performance with respect to number of object pixels in image can be seen in Graph 2.



Graph 2: *Performance with respect to number of object pixels in image*

It decreases as the percentage of object pixels in image increases. The significant cause of such behavior is increasing number of temporary labels and merges operations required to assign unique label to similar objects in image. This behavior can be observed in table 1. From table 1, it can be deduced that time to assign final label or to transfer the image to guest (GPGPU) from host (CPU) is almost constant and does not depend on number of object pixels as expected. On the other hand, as number of background pixels' decrease, the time required to do initial labelling increases significantly as more and more la- bels are assigned and mergers are carried out. Likewise time required to merge all the boundaries also increases significantly. This behavior is result of checking and calling union function repeatedly as described in section 2.3. As discussed in both listing 1 and 2, it can be seen that algorithms start with initial check if pixel under consideration is object pixel or background pixel. When more and more object pixels are present, this check becomes successful more frequently. In case of 100% image, this check holds true for each and every pixel which means following tasks are always performed by each work item. Thus time required is most for temporary labelling and merging boundaries is most for 100% as shown in table 1.

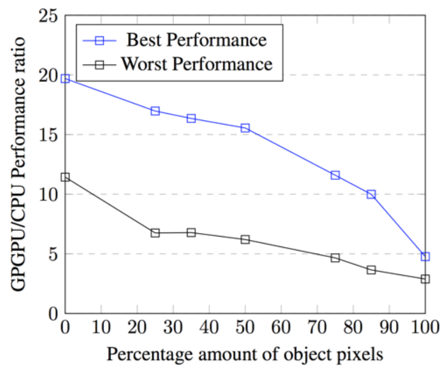
Table 1: *GPGPU execution times of various steps*

Time (seconds) — Object Pixels	0%	25%	35%	50%	75%	85%	100%
Temporary Labelling	0.0810	0.1185	0.1328	0.1306	0.1402	0.1662	0.2092
Merging Boundaries	0.0257	0.1301	0.1872	0.2620	0.3358	0.3508	0.3684
Assigning Final Label	0.0122	0.0206	0.0220	0.0207	0.0227	0.0222	0.0234
Memory Transfers	0.0141	0.0135	0.0135	0.0154	0.0149	0.0178	0.0144
Total (seconds)	0.1332	0.2828	0.3555	0.4286	0.5137	0.5570	0.6154

4.3 Performance Gain of GPGPU over CPU

Graph 3 shows the relative comparison of general purpose graphics processing unit and central processing unit. The best case performance is achieved when the CPU is under complete stress and worst case performance is achieved when CPU is reaching ideal state.

Under normal computations, both cases are not probable but an intermediate performance curve is achievable which shall lie between the two graphs shown in Graph 3. It can also be seen that GPGPU has much better performance if image under test consists of more background pixels and less foreground areas. As object pixels' ratio increases, the overall performance gain decreases. CPU exhibits more steady response as compared to GPGPU with increasing interesting pixels.



Graph 3: GPGPU vs CPU performance comparison

4.4 Verification

In order to verify the consistency and accuracy of obtained results, an associative container can be used to store labels formed by combination of a key label and a mapped label. Key label values can start from one and go till required values while mapped label value is the original value which is too translated to key value. By performing such an operation, higher label values are replaced by consistent sequence of labels starting 1, 2, and 3 ... n. Once both resulting images from CPU and GPU are assigned consistent label values, they can be subtracted. If subtracted image results in all pixels having zero value, then results of both devices are not only consistent but also accurate.

5. Conclusion and Future Work

5.1 Conclusion

From the results described above, it can be concluded that GPGPU does offer performance gain over CPU especially when CPU has average or above average work load. Such a situation is normal if CPU is handling other operating system tasks or interrupts. Due to constant context switching of higher priority or system level tasks, the execution time of CCL can increase significantly on CPUs. Such a situation is not possible on GPGPU because once it starts executing code; it cannot be interrupted by any other task. Execution time on GPGPU can increase if there are other tasks queued to be executed on it. In such a case, tasks have to wait until resource is free or ready. In order to handle this situation, an intermediate layer of scheduling can be introduced. Under normal circumstances, GPGPU executes tasks in FIFO queue (First in first out). If input to FIFO queue is scheduled based on priority and higher priority task updates the queue, wait for resource to get free can be minimized. A CCL execution with higher priority can get GPGPU access earlier and therefore delay before execution is reduced.

Another factor to be considered from above results is that which kind of images is preferred on GPGPU over CPU. From results, it can be seen that images with more background pixels offer much more performance gain when executed on GPGPU. This means that with prior knowledge of image, it can be decided and performance can be improved. With introduction of an intermediate layer that evaluates background pixels' percentage in overall image, it can therefore be decided to schedule and run on central processing unit or general purpose graphics processing unit.

5.2 Future Work

Although the suggested algorithm makes use of graphics card parallel capabilities but it can be seen that repeated launching of kernels on GPU adds load on CPU side. If kernel is invoked many times, the overhead of launching can severely impact the performance of application. In order to reduce the impact of this overhead, attempt can be made to make kernels as coarse-grained as possible, thereby increasing the amount of work performed in each kernel call and reducing the total number of calls. Dividing work items in a work group followed by synchronization and continuing with next kernel task can reduce this overhead. This means that merging two kernels together in one using barrier synchronization where task of second shall start after completion of first. As we know from algorithm execution, every successive kernel launch requires one fourth of work items. Possible drawback is that after every barrier statement, one fourth of work items will be idle. On the other hand, with this strategy in place, there can be two advantages: CPU idle time can be increased and overhead of kernel launch can be reduced. For an image of size 8192x8192, there are eight calls but after joining every two kernels, it can be reduced to half. This can be improved further by using CUDA standard and launching one kernel, which can make recursive kernel launches [10]. Thus reducing overhead to negligible when compared with total execution time. Therefore, it can be ignored.

Another possible improvement can be launching of two command queues. One queue shall take care of data transfer while other can perform connected components labeling. This has to be done using asynchronous command queues and performing explicit synchronization between them using OpenCL events. However, in case of non-synchronization, there can be race conditions, which shall lead to inconsistent results.

References

Rosenfeld A. & Pfaltz J., (1966) “Sequential operations in digital picture processing,” Journal of the ACM, vol. 13, no. 471-494,

.Karimi N.G.D.K & Hamze F., (2010). “A performance comparison of CUDA and OpenCL,” D-Wave Systems Inc., no. 100-4401, p.

Galil Z. & Italiano G. F., “Data structures and algorithms for disjoint set union problems,” CUCS.

Khanna P. G. V. & Hwang C., (2002) “Finding connected components in digital images by aggressive reuse of la- bels,” Image and Vision Computing, no. 557-568,.

Lumia L. S. R. & Zuniga O., (1983). “A new connected components algorithm for virtual memory computers,” Computer Vision, Graphics, and Image Processing, no. 22: 287-300,

Munshi E. A., (2008) “The OpenCL specification,” Khronos OpenCL Working Group, vol. 1.0, no. 29, pp. 23–25

Benkrid V et al., "Towards a general framework for FPGA based image processing using hardware skeletons", Parallel Computing, 28: 1141-1154

Bailey D. G., & Johnston C. T., (2002). "Optimised Single Pass Connected Components Analysis", pp 185 – 192, ICECE Technology, 2008. FPT 2008

Benkrid K. et al., (2002). "Towards a general framework for FPGA based image processing using hardware skeletons", Parallel Computing, 28: 1141-1154

Montes M C, (2010) "CUDA to solve scientific problems", Centro de

InvestigacionesEnerg´eticasMedioambientalesyTecnol´ogicas,.

Montes M C, (2010) "CUDA to solve scientific problems", Centro de

Appendix

Listing 1: Algorithm for label selection

```
if (X = ! (background_pixel) )if(n1 = !(background_pixel) )
    label [X]= label [ ln1];
    check merger with ln2 and ln3
    #Special case 1
    check merger with ln4
    . . continue

elseif ( n2=background and n3=background )
    if(n4 = !(background_pixel) )
            label [X]= label [ln4];
            . . continue
    label [X]=nextLabel++;
    . . continue

elseif (n2=!background or n3=!background)
    if(n3 = !(background pixel) )
            label [X]= label [ ln3 ];
            #Special case 2
            check merger with ln4
    otherwise
            label [X]= label [ ln2 ];
            . . continue

elseif (n2=!background and n3=!background)
    label [X]= label [ ln3];
    check merger with ln2 and ln3
endif
```

Listing 2: Algorithm for boundary traversing

```
if (X =!(background pixel) )
        #VERTICAL RESOLUTION OF BOUNDARY
        for(j=0 to tile_height)
                if (j=0)
                        #n4 not in adjacent tile
                        n4=0;
                if(j=tile_height − 1)
                        #n6 not in adjacent tile
                        n6=0;
                if(n5>0)
                        merge(X and n5);
                        . . continue;
                if(n6>0)
                        merge(X and n6);
                        . . continue;
                if (n4>0)
                        merge(X and n4);
                        . . continue;

        #HORIZONTAL RESOLUTION OF BOUNDARY
        for (i=0 to tile width)
                if ( i =0)
                        #n1 not in adjacent tile
                        n1=0;
                if(i=tile height − 1)
                        #n4 not in adjacent tile
                        n4=0;
                if (n2>0)
                        merge (X and n2);
                        . . continue;
                if (n4>0)
                        merge (X and n4);
                        . . continue;
                if(n1>0)
                        merge (X and n1);
                        . . continue;
endif
```