GRAPH BASED IMAGE SEGMENTATION ON THE GPU

Akanksha Baranwal, Amory Hoste, Gyorgy Rethy, Kouroche Bouchiat

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Real time segmentation has a wide applicability in many fields such as remote sensing, medical imaging and mobile robotics. Popular region based image segmentation methods based on Minimum Spanning Tree(MST) become prohibitively slow on larger resolutions because of their high computational complexity. GPGPUs are known to perform extremely well for computationally intensive tasks specifically for data parallel algorithms. But graph based image segmentation methods are known to be computationally intensive but also highly irregular which makes it difficult to map them to GPGPUs. Our primary focus in this project is to use CUDA data parallel primitives to accelerate MST based image segmentation methods. In addition, we propose a highly efficient solution based on atomic operations.

1 Introduction

Motivation. Image segmentation is an important subject within computer vision. It is often combined with domain specific analysis steps in order to solve visual-understanding problems. Deep learning models have shown to be very efficient in solving semantic image segmentation problems[1] but they have not eliminated the need for "classical" region-based image segmentation algorithms in all areas. One such area is remote sensing, and more specifically aerial photography, where state-of-the-art algorithms combine these "classical" segmentation algorithms[2, 3] with Convolutional Neural Networks to achieve their goals[4, 5, 6]. These "classical" algorithms are easy to interpret, very robust and are widely and immediately applicable without requiring expensive training steps. Because of the wide applicability, accelerating these classical algorithms provide benefits for many use cases [7, 8].

However, for large image sizes, such as the ones used in aerial photography these techniques are prohibitively compute-expensive. Satellite imagery vendors even offer resolutions up to 30 cm/pixel, for which it becomes interesting to leverage the massive parallelism of General-Purpose Graphics Processing Units (GPGPUs).

Related work. Image segmentation algorithms based on region merging[9, 10], which relies on growing pixel regions into segments[11], are especially useful when the background and region of interest have overlapping pixel intensities[12]. Despite having higher segmentation accuracy, these methods have an irregular nature[12] which make them unsuitable for GPGPU optimization. GPGPUs are more suited for programs which exhibit explicit data parallelism, access memory in a streaming fashion and require little synchronization[13]. Most such region-growing methods use the Kruskal algorithm to construct a minimum spanning tree with each vertex in the tree as a segment component. Haxhimusa *et al.*[14] and Wei *et al.*[15] propose replacing Kruskal's with the Boruvka MST algorithm[16]. There are also works such as Vineet *et al.*[17] based on Boruvka's and Wang *et al.*[18] based on Prim's algorithm which map the respective MST algorithms to CUDA data parallel primitives increasing its suitability for GPGPUs.

2 Background

In this section, we introduce the image segmentation algorithms we decided to accelerate on the GPU along with some necessary background. The implemented algorithms can be categorized in to regular image segmentation algorithms and hierarchical image segmentation algorithms.

2.1 Regular image segmentation

In image segmentation, the goal is to divide an image into groups of mutually exclusive segments or superpixels that share common properties. These common properties often try to capture human perceptual features such as intensity, gradients and edges. Along with clustering based methods, graph based methods are one of the main methods to compute image segmentations. In a graph based method, the segmentation problem is formulated as a graph partitioning problem where pixels are represented as vertices and weights capture dissimilarity between neighbouring pixels. After partitioning the graph, the segments are defined by the resulting mutually exclusive groups of vertices.

Felzenszwalb Segmentation [10] is a widely used graph based method available in many computer vision libraries which solves the graph partitioning problem in $O(n \log n)$ time, with n denoting the amount of pixels. It accomplishes this by using a method based on Kruskal's MST [19]. Graph

weights are computed using the L2 distance between RGB values of neighbouring pixels and an adaptive merging criterion is used to determine whether there is evidence for a boundary between neighbouring components. In this merging criterion, a parameter k is used to influence the amount of segments in the generated segmentation. The segmentation is computed by iterating over the sorted edges in increasing order and evaluating the merging criterion on each edge to determine whether two components should be joined. Once all edges have been evaluated, the resulting components represent the disjoint segments of the image. In addition, a Gaussian filter is also applied to the image in order to compensate for digitization artefacts.

2.2 Hierarchical image segmentation

Next to regular image segmentation methods, there are also hierarchical methods. These methods eliminate the need of a parameter to influences the amount of segments, but instead generate multiple segmentation levels with varying amounts of segments. Higher levels contain less segments and are a union of lower levels which contain more segments. This approach eliminates parameter tuning and allows for image analysis at different levels without having to re-execute the segmentation with different parameters.

Segmentation Graph Hierarchies [14] provides some insights *et al.*[14] to transform the Felzenszwalb Segmentation algorithm into a hierarchical segmentation algorithm that still runs in $O(n \log n)$ time. The most important insight is that lower segmentation levels in the hierarchy can be reused to produce higher levels, allowing the segmentation levels to be generated on the fly using only a single pass over the data. The second insight is the usage of Boruvka's MST [16] algorithm and the usage of supervertices to reduce the run time in further iterations. Supervertices are produced by replacing all vertices to be merged with one single vertex and only keeping the lightest edge out of all duplicate edges when creating the supervertex. The usage of supervertices reduces the amount of edges and vertices in subsequent iterations, which lowers the overall cost of the algorithm. In addition, the paper also makes the observation that removing the Felzenszwalb merging predicate doesn't significantly change the result for most segmentation levels, which significantly simplifies the computations.

Superpixel Hierarchy [15] is another hierarchical image segmentation algorithm based on Boruvka's MST which changes the way edge weights are computed. Instead of computing the weights once and using the cheapest edge between the boundaries of to component, it dynamically adjusts weights during the algorithm based on the absolute difference in average color between two components. In addition, edge detection is also incorporated by weighing the weights by the edge strength, which is obtained using an edge detection algorithm. For edge detection, the paper uses fast edge detection using structured forests [20], a random forest based method that uses a pretrained model to detect edges. Compared to Felzenszwalb segmentation, it claims to achieve lower undersegmentation error resulting in better image segmentations.

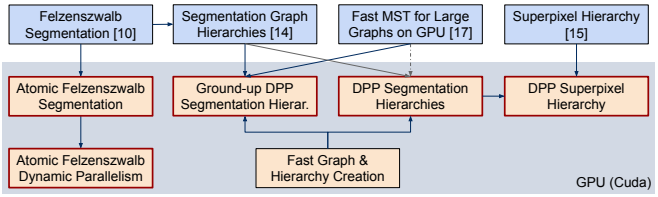


Fig. 1: Overview GPU implementations and used papers.

3 Proposed Method

In this paper for implementing Felzenszwalb *et al.*[10] Boruvka's MST algorithm has been used, because it lends itself nicely to GPU execution and all other segmentation algorithms require it. After reviewing the current state of the art, it seemed that CUDA has developed faster than research could keep up. Our best choice for an implementation candidate[17] relied on parallel primitives from the CUDPP library, which is not supported and usable anymore. Therefore we had two choices, replace/implement the aforementioned primitives or port the algorithm ourselves and see if the new features of CUDA lead to higher performance. Both routes will be discussed in the following subsections with an overview depicted in Figure 1.

3.1 "Atomics-Based" Boruvka Felzenszwalb Segmentation

This is our own implementation, which went through multiple iterations and optimisations with varying success. The first working implementation ran for 77 minutes on a 1000x2000 image and the latest ran for 40ms on the same machine and image.

During development the following principles/techniques proved to be the most effective for GPUs:

1. Reduce memory operations as much as possible. 2. If critical sections cannot be avoided reduce them to atomic operations. 3. Reducing memory operations is often the most important overtaking reducing branch divergence. 4. Reduce work and consequently the amount of threads, which should always be kept linear (This was done by trading space for time).

Almost all optimisations performed on this version sought to achieve one or more of the above goals. In the end our atomics-based Boruvka's MST variant is as follows.

1. Find the minimum outgoing edge from each vertex and write it to a separate array. This step reduces atomic global memory transactions in the next step

by the number of neighbours, since the min-scan can use registers. This can run in parallel for each vertex.

2. Reduce the previously generated array into another separate array, at locations based on the component id. This can run in parallel for each vertex with the help of `atomicMin()` and would in the worst case serialise to run in parallel for each component.
3. With `atomicAdd()` to determine positions compact minimum edges into a continuous block. This can be done in parallel for each minimum edge (number of components).
4. Remove circular merges (e.g. 1→2 and 2→1) This can be done in parallel for each minimum edge (number of components).
5. Evaluate each edge in parallel and mark those that satisfy the predicate.
6. If no edges were marked perform post-processing and return.
7. Update parents (vertices that have the same component id as vertex id) to point to their new component if marked for merging. This can run in parallel for each parent vertex or component.
8. Flatten the component tree and update the component size and internal difference with the help of atomics. This can run in parallel for each vertex.
9. Go to 1

Dynamic parallelism. One seemingly large overhead when using a GPU for Boruvka's MST is the outer loop and the need for synchronization at the end of each iteration. This leads to the host having to synchronize with the GPU and also to perform copies in each iteration. There are works, where this issue was alleviated by introducing some restrictions[21] with great success.

Unfortunately, for image segmentation there was no clear way to introduce such restrictions. However, there is an alternative: dynamic parallelism, which allows device threads to launch kernels. With dynamic parallelism it is possible to have a single "orchestration" kernel removing the need for host-to-device synchronization and memory copies. Leading to two separate implementations: 1. ab conventional, which uses normal CPU to GPU synchronization at the end of each iteration. 2. ab dynamic, which uses Dynamic Parallelism to avoid syncing with the host device. Later a performance comparison and analysis of the two follows in section 4.

3.2 Parallel primitive based hierarchical segmentation

The following sections describe the steps involved in implementing an accelerated version of Segmentation Hierarchies and Superpixel Hierarchy based on Data Parallel Primitives (DPP) [17].
The first step before converting the image into graph representation, is to apply a gaussian filter using OpenCV CUDA filters.

3.2.1 Fast Graph Creation

In order to take advantage of the massive parallelism of the GPU, we decided to implement graph creation on the GPU instead of the CPU. The graph representation needed in the segmentation step is in compressed adjacency list format. To create the graph, each kernel thread adds the edges of a single pixel to the compressed adjacency list[17]. The position to add the edges and weights in the compressed adjacency list arrays is computed using the position of the pixel the thread is modifying. In order to avoid branching, separate kernels are used to transform the outer and inner parts of the image into the compressed adjacency list format. Since all kernels modify distinct data, the kernels are launched concurrently in different CUDA streams.

3.2.2 Fast Minimum Spanning Tree

The MST based component merging step in the parallel primitive based approach is based on Vineet *et al.*[17]. In the implementation to which we will refer to as "Ground-Up Data-Parallel Primitives (DPP) Segmentation Hierarchies", we implemented and extended the algorithm mentioned in [17] from scratch and used the ideas from the Segmentation graph hierarchies paper [14] to convert the minimum spanning tree algorithm to an algorithm that generates a hierarchical image segmentation. In parallel, we also worked on making the official implementation provided by the Fast MST paper [17] work on modern GPUs and extended it to hierarchical image segmentation. We will refer to this approach as "DPP Segmentation Hierarchies". Both approaches will be discussed in the following paragraphs.

Ground up DPP Segmentation Hierarchies. Using the recursive MST formulation in [17], we mapped steps of *Algorithm 1* to CUDA thrust parallel primitives: scan, scan_by_key, sort and parallel kernel calls. For computing the minimum outgoing edge, [17] proposes using segmented reduction. We initially implemented this using the `SegReduceCsr` from Modern GPU library [22]. But this wasn't able to support segmented reduction for edgelists of images larger than 3840X2160. As a result, we switched to implementing the segmented scan functionality using CUDA thrust parallel primitives. Since the thrust library sort functions take a single array as input, we took the same approach as the Fast MST paper [17] and used bit concatenation to concatenate the two endpoint IDs of an edge along with its weight into a single 64 bit integer.

DPP Segmentation Hierarchies. In order to make the official Fast MST [17] implementation work on modern GPUs and adjust it to image segmentation, we mainly used Thrust primitives instead of CUDPP primitives. In addition, we

also discovered and resolved a few bugs in the official implementation that resulted in incorrect segmentations.

We also ran into a trade-off between using more bits for weights, which results in better segmentation quality versus using more pixels for IDs which increases the maximum number of pixels our algorithm supports. We decided to settle with 26 bits for the IDs of the two endpoint pixel IDs and 12 bits for the weights, which allows the algorithm to handle images of up to 2^{26} pixels, which is roughly the size of two 8K images.

3.2.3 Fast Hierarchy Creation

In order to be able to reconstruct the hierarchy levels after the segmentation is done, we store the supervertex ids between subsequent iterations. The final step in the algorithm is the hierarchy creation, which is again shared by the Segmentation Hierarchies and the Fast MST hierarchies implementations and is also done on the GPU. First, cuRAND is used to create a mapping from component IDs to random colors. To build the hierarchy, the previously stored supervertex IDs can be used. These provide a mapping from the previous level component IDs (or the original pixels in case of the top level) to the new component IDs. We create the hierarchy levels by iteratively mapping the previous component IDs to their corresponding new component IDs and then mapping these component IDs to colors using our previously generated color mapping. This is all done on the GPU where each thread is responsible for mapping the component ID of a single pixel to its new component ID.

3.2.4 Extending the Segmentation Hierarchy: Fast Superpixel Hierarchy

We also implemented the Superpixel Hierarchy algorithm on the GPU, which is mentioned as future work in the corresponding paper [15]. Since the Superpixel Hierarchy algorithm is also based on Boruvka's MST, we implemented this by extending our Fast MST Hierarchies implementation. For the edge detection, using structured forests [20] turned out to take around 20 times longer than our whole segmentation algorithm with no GPU implementation available. Because this would completely shift the bottleneck from the segmentation to the edge detection, we opted to use a simple Sobel filter which we applied along with the gaussian filter using OpenCV CUDA filters. The only component that needed changes to support the Superpixel Hierarchy approach was the segmentation step. Instead of calculating the weights during the graph creation, the weight is recalculated in each iteration by multiplying the edge strength with the average weight of each component using a CUDA kernel where each thread calculates the weight of a single edge. The average color of the components is maintained during the algorithm by storing the component size and current average color of each component which are updated during the algorithm using some CUDA kernels and the thrust inclusive_scan_by_key primitive.

4 Experimental Results

In this section, we evaluate the implementations discussed in the previous sections. We benchmarked the performance improvement in terms of run times. We also ran benchmarks for segmentation quality to analyse the impact of these optimisations. Both aspects will be discussed in the coming subsections.

Experimental Setup. The experiments were run on a Ubuntu 18.04 private cloud instance with 4 Intel Xeon Silver VCPUs @ 2.10 - 3.00 GHz, a NVIDIA 1080 Ti GPU and 12 GiB of DDR4-2666 RAM. The executables were build using GCC 7.5.0 with release flags, CUDA 11 and OpenCV 4.5.0. All datasets, results, scripts and implementations are available in our GitLab repository[1].

Baseline. For our experiments, we used two different baselines. The first baseline is the official C Felzenszwalb segmentation CPU implementation[10][23]. The second baseline is the GPU graph based image segmentation implementation provided by NVIDIA as part of the CUDA samples in the CUDA toolkit[24]. Since NVIDIA knows its hardware best, this is the best available GPU implementation for performance comparison. These implementations give a good idea of the current best performance for the Felzenszwalb algorithm on both the CPU and GPU.

4.1 Performance Results

We analyse the performance for multiple input sizes, by measuring the run time for image resolutions ranging from 960x540 to 7680x4320 with the total number of pixels increasing exponentially. Run times were measured using C++ high resolution clocks over 20 iterations on the same input, which turned out to be enough for the 95% to be within 5% of the mean. These time measurements only include work done and synchronization costs and exclude all irrelevant operations such as disk reads/writes. The mean of the run time measurements, along with the standard deviation indicated by translucent error bars is shown in figure 2.

Total run time. We observe in figure 2.a, that all of our implementations have lower total run times for all image resolutions than both of the baselines.

Surprisingly NVIDIA's GPU based segmentation is the slowest of all of them. As shown in figure 2.b this is due to their inefficient graph creation/output and image filtering. Upon further inspection of their source code we found that all of these were implemented on the CPU. By moving these steps to the GPU we were able to exploit it's massive parallelism. Since the images are a more compact representation than the produced graphs we were also able to reduce memory transfer sizes.

[1] https://gitlab.ethz.ch/ahoste/graph-algorithm-image-segmentation

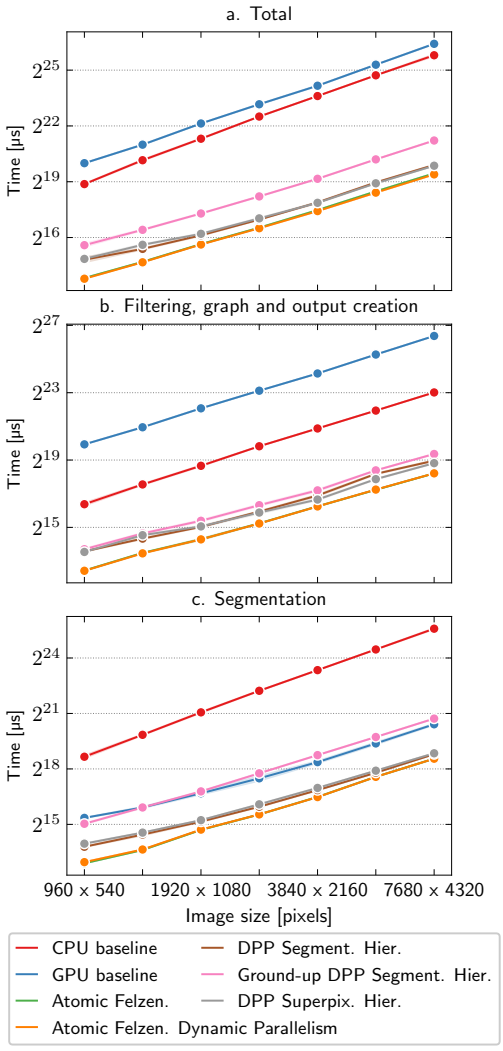


Fig. 2: Comparison of run time measurements across algorithms (log-log scale).

Segmentation run time. We observe in figure 2.c, that all of our implementations have lower segmentation run times for all image resolutions than the CPU baseline. We also observe that except Ground Up DPP Segm. Hierarchies all of our implementations are faster than the GPU baseline.

An overhead of the recursive formulation described in [17] is that it uses a large amount of memory. As visible from figure 2.c, the Ground Up DPP Segm. Hierarchies is slower than the DPP Segm. Hierarchies, which is surprising since both are based on [17]. This can be largely attributed to the use of Unified Memory in Ground Up DPP Segm. By using Unified Memory for multiple large arrays, a large number of page faults are introduced, when these arrays are first accessed on the GPGPU. However, this issue is not present in the original implementation since they allocate and copy these large arrays explicitly on the device. A solution for this would be to analyze which memory segments can be prefetched asynchronously to the device.

Our last observation is that, the performance difference between atomics-based conventional and dynamic are indiscernible in figure 2.c, despite our hypothesis that Dynamic Parallelism could alleviate the device synchronisation overheads.

Dynamic Parallelism. To determine why Dynamic Parallelism is seemingly not providing any performance benefit the segmentation times of both the atomics-based conventional and dynamic implementations are evaluated.

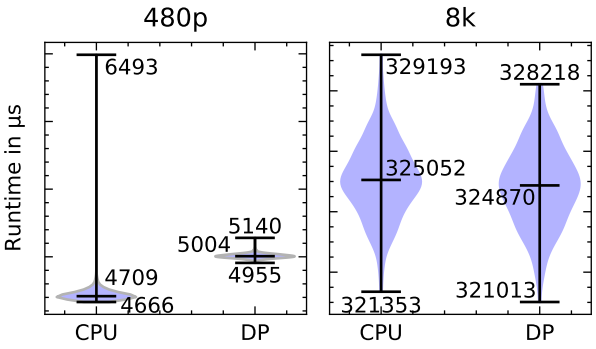


Fig. 3: Performance characteristics of atomics-based conventional and dynamic segmentation on two different popular resolutions (16:9 aspect ratio) over 1000 measurements each

As shown in figure 3 the Dynamic Parallelism version has more predictable performance, because it is able to avoid synchronization with the host and memory copies. However, using the `nvprof` profiler we found that an "orchestration" kernel takes some resources away from all other kernels. Therefore, the conventional version is able to achieve better median performance.

By profiling we also observed that the outer loop of Boruvka's MST algorithm is executed maximum $10 - 20$ times for all of our images and at the end of each iteration only 4 bytes had to be copied back to the host. Therefore, for the large image(8k) all of the above mentioned effects, such as synchronization and dynamic parallelism costs, stay constant across resolutions and are completely dominated by the time of the segmentation. This explains the negligible difference (%1) in performance between the two implementations for larger resolutions.

4.2 Segmentation Accuracy

Metrics. We adopted two metrics used in Wei 2018 [15] in order to assess the output quality:

• *Achievable Segmentation Accuracy* which measures the maximal area covered by our segments or superpixels when iterating over the ground-truth segments:

$$\mathrm{ASA}(\mathcal{S}) = \frac{\sum_k \max_i |s_k \cap g_i|}{\sum_i |g_i|}, \quad (1)$$

where s_k is the k-th segment or superpixel, and g_i is the i-th segment in the ground-truth.

• *Under-segmentation Error* which measures the segment or superpixel "spill" over the borders of the ground-truth segment they intend to cover:

$$\mathrm{UE}(\mathcal{S}) = \frac{\sum_i \sum_k \min(|s_k \cap g_i|, |s_k - g_i|)}{\sum_i |g_i|}, \quad (2)$$

where s_k is the k-th segment or superpixel, and g_i is the i-th segment in the ground-truth.

Dataset. Our quality assessment was conducted using the Berkeley Segmentation Dataset (BSDS500). It is commonly used for measuring segmentation quality and contains 500 images with several human-labeled ground truths for each. We picked the ground-truth that led to the highest ASA score when taking the metrics for the different implementations. Figure 4 presents the results obtained on the BSDS500.

Analysis. Fundamentally, switching the MST algorithm from Kruskal to Boruvka reduces the segmentation quality. Even though the MST algorithm by itself will always produce the same output, the segmentation algorithm also uses a merging predicate which is dependent on the component size. The combination of Boruvka and the merging predicate results in smaller and less accurate components because of Boruvka's parallel nature. This explains the reduction in accuracy for all GPU based algorithms, which are all based on Boruvka's MST.

Both the GPU baseline and DPP Segmentation Hierarchy algorithms perform very similar in terms of accuracy. This can be explained by the fact that they are both hierarchy based and thus work in a similar fashion. The Ground-Up DPP Segmentation was an additional exploration which turned out to give inferior accuracy results for the BSDS 500 dataset. For higher resolution images, we found that the output looked more accurate so we suspect there still is some issue specific to lower image sizes.

To achieve segmentation scores closer to the CPU baseline, we also explored incorporating edge detection through the Segmentation Hierarchy algorithm. As explained in section 3.2.4, we had to compromise on the edge detection algorithm by using Sobel instead of Structured Forests. Because of the higher sensitivity to noise and inability to produce thin edges, median segmentation quality only improved marginally. As future work, more experimentation with different edge detection methods could be done to explore the trade-off between faster edge detection algorithms and better segmentation scores.

The atomics based Felzenszwalb algorithm exhibits a similar accuracy-performance trade-off. It achieves a lower runtime, but also has higher variance in the metrics compared to the hierarchy based methods. This can be explained by the fact that the hierarchy based methods generate multiple segmentation levels, without the need for specific parameter tuning.

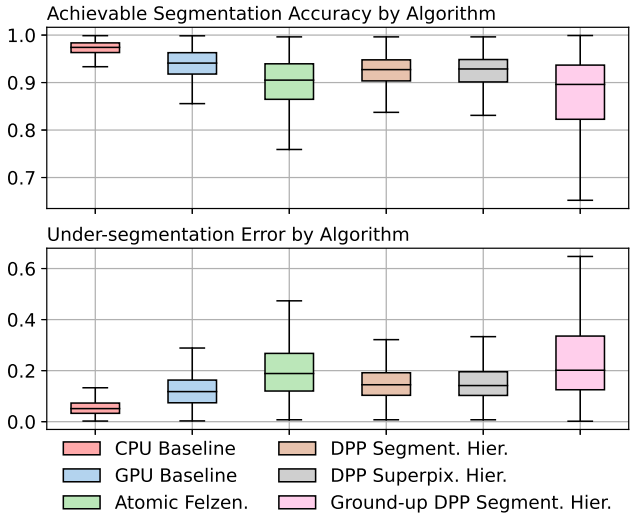


Fig. 4: ASA (higher is better) and UE (lower is better) on the BSDS 500. All measures were taken using $K = 80$ or by taking the 4-th level in the hierarchy for the hierarchical algorithms.

5 Conclusion

In this project, we explored optimizing both Felzenszwalb and Hierarchical image segmentation for the GPU. We achieved significant speedup compared to both the original Felzenszwalb and NVIDIA's implementation, while maintaining practical segmentation accuracy. In our atomics-based implementation we were able to minimise memory accesses and achieve higher performance by using atomic operations, which resulted in a speedup of up to 127 times (overall), 5.6 times (on the segmentation step) relative to GPU baseline. For our use case, dynamic parallelism did not yield a significant performance improvement, except for slightly reducing maximum run times. When using data parallel primitives, the same approach used by NVIDIA, we also managed to realize up to 93.2 times (overall) and 3.6 times (on the segmentation step) speedup relative to GPU baseline. We argue that these speedups should mean perceptible improvements for real world applications that use classical image segmentation methods, especially when working with larger images.

6 References

[1] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos, "Image segmentation using deep learning: A survey," *arXiv preprint arXiv:2001.05566*, 2020.

[2] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk, "Slic superpixels compared to state-of-the-art superpixel methods," *IEEE transactions on pattern analysis and machine intelligence*, vol. 34, no. 11, pp. 2274–2282, 2012.

[3] Zhengqin Li and Jiansheng Chen, "Superpixel segmentation using linear spectral clustering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1356–1363.

[4] Martin Längkvist, Andrey Kiselev, Marjan Alirezaie, and Amy Loutfi, "Classification and segmentation of satellite orthoimagery using convolutional neural networks," *Remote Sensing*, vol. 8, no. 4, pp. 329, 2016.

[5] Yang Chen, Rongshuang Fan, Xiucheng Yang, Jingxue Wang, and Aamir Latif, "Extraction of urban water bodies from high-resolution remote-sensing imagery using deep learning," *Water*, vol. 10, no. 5, pp. 585, 2018.

[6] Zeinab Gharibbafghi, Jiaojiao Tian, and Peter Reinartz, "Modified superpixel segmentation for digital surface model refinement and building extraction from satellite stereo imagery," *Remote Sensing*, vol. 10, no. 11, pp. 1824, 2018.

[7] X. Chen and L. Pan, "A survey of graph cuts/graph search based medical image segmentation," *IEEE Reviews in Biomedical Engineering*, vol. 11, pp. 112–124, 2018.

[8] M. Grundmann, V. Kwatra, M. Han, and I. Essa, "Efficient hierarchical graph-based video segmentation," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 2141–2148.

[9] M. Liu, O. Tuzel, S. Ramalingam, and R. Chellappa, "Entropy rate superpixel segmentation," in *CVPR 2011*, 2011, pp. 2097–2104.

[10] Pedro Felzenszwalb and Daniel Huttenlocher, "Efficient graph-based image segmentation," *International Journal of Computer Vision*, vol. 59, pp. 167–181, 09 2004.

[11] A. Humayun, F. Li, and J. M. Rehg, "The middle child problem: Revisiting parametric min-cut and seeds for object proposals," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1600–1608.

[12] Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster, and Frank Lindseth, "Medical image segmentation on gpus – a comprehensive review," *Medical Image Analysis*, vol. 20, no. 1, pp. 1 – 18, 2015.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[14] Yll Haxhimusa and Walter Kropatsch, "Segmentation graph hierarchies," in *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, 2004, pp. 343–351.

[15] Xing Wei, Qingxiong Yang, Yihong Gong, Narendra Ahuja, and Ming-Hsuan Yang, "Superpixel hierarchy," *IEEE Transactions on Image Processing*, vol. 27, no. 10, pp. 4838–4849, 2018.

[16] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová, "Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history," *Discrete Mathematics*, vol. 233, no. 1, pp. 3 – 36, 2001, Czech and Slovak 2.

[17] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the gpu," in *Proceedings of the Conference on High Performance Graphics*. 2009, HPG '09, p. 167–171, Association for Computing Machinery.

[18] W. Wang, S. Guo, F. Yang, and J. Chen, "Gpu-based fast minimum spanning tree using data parallel primitives," in *2010 2nd International Conference on Information Engineering and Computer Science*, 2010, pp. 1–4.

[19] Joseph B Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[20] Piotr Dollár and C Lawrence Zitnick, "Fast edge detection using structured forests," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 8, pp. 1558–1570, 2014.

[21] Ahmed Shamsul Arefin, Carlos Riveros, Regina Berretta, and Pablo Moscato, "K nn-borůvka-gpu: a fast and scalable mst construction from k nn graphs on gpu," in *Proceedings of the 12th international conference on Computational Science and Its Applications - Volume Part I*, 06 2012, pp. 71–86.

[22] S Baxter, "Moderngpu 2.0: A productivity library for general-purpose computing on gpus," .

[23] Pedro Felzenszwalb and Daniel Huttenlocher, "Felzenszwalb official source code," http://cs.brown.edu/people/pfelzens/segment/segment.zip.

[24] NVIDIA Corporation, "Felzenszwalb cuda samples gpu source code," https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-segmentation-tree-thrust-library.

Appendix A. Atomics-based Boruvka's MST pseudocode

Algorithm 1: Segmentation using our port of Boruvka's MST on a GPGPU

```
Result: Segmented Graph
while new components are merged do
    Find minimum outgoing edge for each vertex;
    if not first iteration then
        Reduce minimum edges for each component;
        Compact minimum edges to start of array;
    end
    Remove cycles;
    Mark minimum edges for merging;
    Update parents;
    Flatten component tree;
end
```

Algorithm 2: Find minimum outgoing edge for each vertex

```
Result: Minimum edge of each vertex
for each vertex v
    if v has no outgoing edges then
        skip;
    end
    Scan v.edges for minimum by weight and component ID;
    if no outgoing edge found then
        Mark v as having no outgoing edges;
    else
        Save minimum edge to array indexed by v.id;
    end
end
```

Algorithm 3: Find minimum outgoing edge for each component

```
Result: Minimum edge of each component
for each minimum edge e
    id←e.component.id;
    i←concatenate(e.weight, id);
    atomicMin(minimum_edges[id], i);
end
```

Algorithm 4: Remove cycles from minimum outgoing edges

```
Result: Minimum edges with cycles removed
Construct sources array;
Check and correct destinations;
```

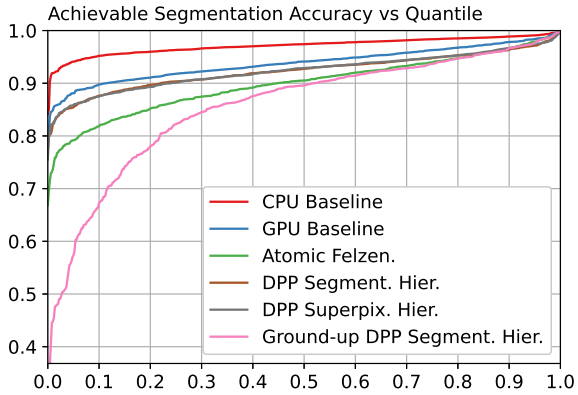
Algorithm 5: Construct sources array

```
for each minimum edge e
    sources[e.src] = e.dest;
```

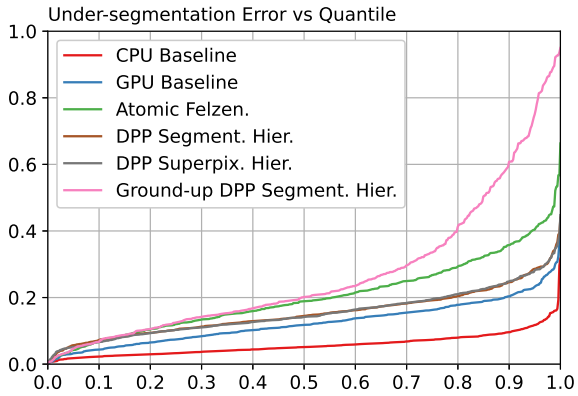
Algorithm 6: Check and correct destinations

```
for each edge e in edges
    if sources[e.dest] == e.src then
        e.dest = e.src;
```

Appendix B. Segmentation Quality Plots



(a) Cumulative Distribution Function of the ASA scores as shown in Figure 4.



(b) Cumulative Distribution Function of the UE scores as shown in Figure 4.