GPU-accelerated RDP Algorithm for Data Segmentation

P. Cebrian[1] and J.C. Moure[1]

Computer Architecture and Operating Systems Department
Universitat Autonoma de Barcelona, Spain
{pau.cebrian,juancarlos.moure}@uab.cat

Abstract. The Ramer-Douglas-Peucker (RDP) algorithm applies a recursive split-and-merge strategy, which can generate fast, compact and precise data compression for time-critical systems. The use of GPU parallelism accelerates the execution of RDP, but the recursive behavior and the dynamic size of the generated sub-tasks, requires adapting the algorithm to use the GPU resources efficiently. While previous research approaches propose the exploitation of task-based parallelism, our research advocates a general fine-grained solution, which avoids the dynamic and recursive execution of kernels. The segmentation of depth images, a typical application used on autonomous driving, reaches speeds of almost 1000 frames per second for typical workloads using our massively parallel proposal on low-consumption, embedded GPUs. The GPU-accelerated solution is at least an order of magnitude faster than the execution of the same program on multiple CPU cores with similar energy consumption.

Keywords: GPU acceleration, Data Segmentation, Segmented Scan.

1 Introduction

The line-simplification algorithm defined by Ramer-Douglas-Peucker (RDP) [3] transforms the complex strokes described by a set of points into a set of a few connected segments that capture the inherent structure of the data. This algorithm is defined with a *split-and-merge* strategy, which increases the details of the representation recursively. Although RDP was originally designed to simplify cartographic coastlines, nowadays it applies to a multitude of computer vision problems that need to define or simplify the perimeters of objects within a scene.

The low complexity and the high degree of underlying parallelism of the RDP algorithm make it especially suitable for time-limited segmentation tasks. One usage scenario is autonomous driving systems, which need to provide segmented data within a real-time pipeline and with limited hardware resources. In this context, and compared to a classical segmentation proposal as Stixels [7,6], the RDP algorithm reduces the execution time, generating compressed and accurate representations, as shown in Figure 1. While most of nowadays proposals [18,4,16] follow a global maximization strategy, with quadratic complexity, the RDP proposal allows solving the segmentation problem also globally, but with linear-logarithmic complexity on the practical cases that are addressed. Moreover, the RDP algorithm only relies on a single configuration parameter (ε), which allows adjusting the quality and compression levels of the segmentations.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

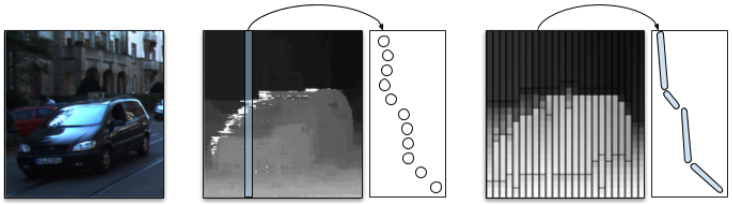

Fig. 1. Column-level segmentation of depth images for autonomous driving tasks.

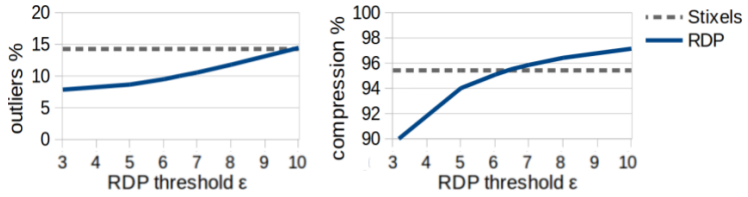

Fig. 2. Variation of the accuracy (left, lower is better) and compression (right, higher is better) of the segmented depth representations depending on the ε parameter.

Our preliminary work on this topic [2] focused mainly on the quality and compression capabilities of the RDP algorithm with respect to the Stixels proposal. We concluded that the split-and-merge strategy allows to compute representations with similar quality and compression to those obtained using expensive global maximization strategies, as shown in Figure 2.

This paper focuses on defining an efficient implementation of the algorithm for GPU acceleration rather than on evaluating the properties of the RDP segmentation, which can be found in the research done by Heckbert *et al.* [5]. Previous approaches propose the exploitation of task-based parallelism for CPU execution [11,20], which represents a natural way to extract parallelism from a recursive algorithm. However, the dynamic generation of tasks and its irregular size do not adapt well to the GPU execution model. Instead, we propose to process the independent tasks generated in the same recursive level together, using a segmented reduction pattern. This approach matches better with the underlying SIMT (Single Instruction Multiple Threads) model of a GPU architecture, but the algorithm must be refactored and the data structures must be redesigned to be statically allocated. Our parallelization strategy transforms dynamic and irregular parallel work into homogeneous work, more appropriate for GPUs.

This paper describes in detail our massively-parallel proposal and analyzes the optimization strategies applied to minimize its execution time. Also, we present: a theoretical evaluation of the computational behavior and the number of recursion levels expected for our executions; an experimental evaluation of the performance in GPUs; and a performance comparison with the recursive algorithm executed in CPUs. We believe that the methods and the analysis presented on this work can be very useful to accelerate the execution on GPUs of similar recursive split-and-merge problems.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

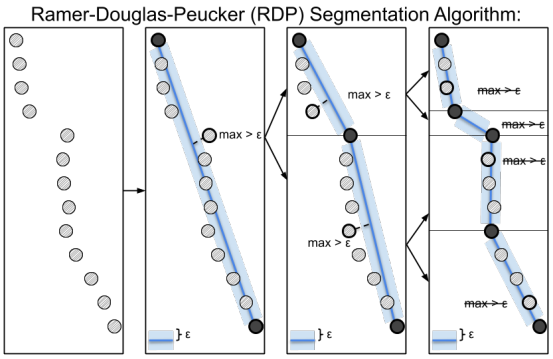

Fig. 3. Segmentation process: the granularity of the representation is recursively refined until the actual set of segments accurately represents the lineal structure contained in the data.

2 Analysis of the RDP Algorithm

This section describes the Ramer-Douglas-Peucker algorithm [3] and presents a qualitative and quantitative analysis of its computational complexity.

The algorithm approximates the one-dimensional data input, which contains n values, into a set of m connected linear segments. The cut-off points between segments are calculated with a split-and-merge strategy, refining the details of the representation recursively. A user-defined constant (ε) determines the accuracy and the compression level of the final segmented description.

Figure 3 illustrates the dynamics of the algorithm. The input data points on the left are substituted by a line drawn from the first to the last point. Then, a certain user-defined distance with respect to the line is computed for each input data point, and the point at the maximum distance is identified. If this maximum distance is greater than a given ε value, the corresponding point is selected as a cut-off point to partition the initial line into two connected segments. Then, the same process is performed recursively on the two new segments created from the cut-off point. The algorithm stops when the maximum distance in all segments is less than ε, which provides control over the level of detail of the segmentation. The higher the value of ε, the lower the number of segments required to represent the data (higher compression), but the lower the accuracy of the segmentation of the data.

Algorithm 1 presents a recursive definition of RDP, and Figure 4 shows the binary tree representing the recursively-generated tasks of a segmentation example. The bulk of the computation work is done at lines 3 and 4, which is the calculation of a distance and the comparison with the maximum distance, and is proportional to the number of points involved in the current segment. The total computation work is the addition of the work done on each task, and depends on the input vector size n, and on the topology of the recursion tree. The number of levels in the tree and the size of each node in the tree (see fig. 4) depends on the combination of the actual data values (i.e., the underlying

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

structure of the data) and the tolerance ε. The total number of generated segments, m, is not enough to fully determine the algorithm's complexity.

Algorithm 1 Segmentation($data^*$, ε, $first$, $last$, $segments^*$)

```
1: maxId = 0, maxDist = 0
2: for i := first → last do
3:     dist = computeDist(data*, first, last, i)
4:     maxDist, maxId = maxArg(dist, maxDist, i, maxId)
5: if maxDist > ε then
6:     Segmentation(data*, ε, first, maxId, segments*)
7:     Segmentation(data*, ε, maxId, last, segments*)
8: else
9:     segments*.add([first, last])
```

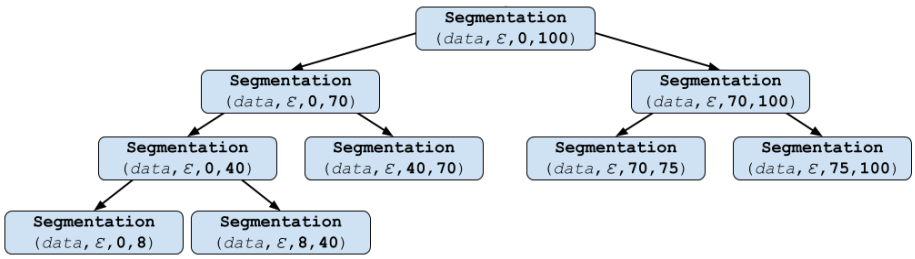

Fig. 4. Recursion tree of an example of execution of RDP.

The best-case complexity, $\Theta(n)$, occurs when the input data fits as single segment, *i.e*, m=1. The worst-case complexity, $\Theta(n^2)$, occurs for very noisy input data that is converted into $m{\approx}n$ segments, and with a configuration that requires $m-1$ levels of recursive calls. In the average case the recursive tree should be quite balanced and the computational complexity is $\Theta(n \log m)$.

We focus our analysis on the data sets used in the segmentation of depth image columns; like KITTI [14], which contains pairs of stereo images of autonomous-driving scenes and their corresponding depth maps, captured with stereo cameras and LIDAR. In this context, from hundreds to a thousand data columns must be processed, with vector sizes ranging from 256 to 1024 values. The raw depth images obtained from stereo matching algorithms usually contain outlier values, which are typically rectified using median filters. This preprocessing step reduces the average amount of segments generated by the RDP algorithm.

We next present an analysis of the actual execution complexity achieved when using a typical autonomous driving dataset. Figure 5 shows, for different tolerance values (ε), the average and standard deviation of the number of recursion levels required to process the data columns.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

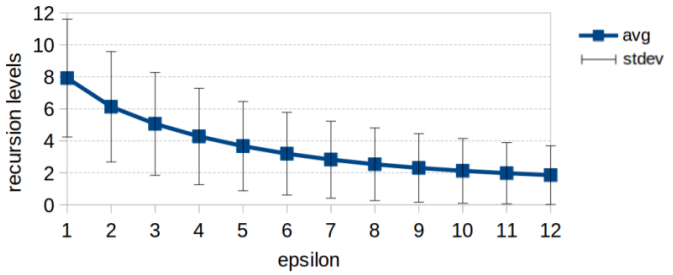

Fig. 5. Experimental analysis on the KITTI data set [14] (described later), with depth images scaled to $n = 1024$ pixels/column, and using median filters. Complexity is measured indirectly as the levels or recursion of the RDP algorithm.

As expected, smaller values of ε generate more segments and then more recursion levels. In the practical range of the application for computer vision, with ε between 4 and 8, most of the cases require less than 5 or 6 recursion levels, generating between 20 and 50 segments, while some exceptional cases may need at most 8 recursion levels. The overall result is that the computation complexity for the data inputs considered are far from the worst case. This result is key to design an efficient massively-parallel algorithm that can perform all the tasks of the same recursion level simultaneously, even when some tasks in the level are not required.

3 Background

The RDP algorithm has been revised several times, either to apply the algorithm as a subroutine of some scene-understanding task, such as Romadi or Hu investigations [17,8]; or with the aim of selecting the best points of start and end for closed segments, such as the proposals of Li or Mahmoudi [10,12].

Researches focused on parallelization and computation efficiency, such as Jingsong *et al.* [11], proposes a task-based parallelization strategy for multicore CPU systems, where processors receive the segments on which to apply RDP from a list of tasks in which they add segmentation sub-tasks dynamically. On the other hand, Scherger *et al.* [20] transform the split-and merge or top-down strategy of the RDP algorithm to an only-merge or bottom-up strategy, initializing as many segments as pixels in the column, and defining their own multiple associative computing (MASC) model to make iterative merging of the segments.

Other approaches have been suggested for solving the split-and-merge strategy, but not directly related with the RDP algorithm.

Paravecino's research [15] studies the nested parallelism of GPU systems, and analyzes the execution of several well-known algorithms with recursive kernels.

Argüello *et al.* [1] treat the split-and-merge problem with a bottom-up strategy by calling a single GPU kernel. However, his research is more related to convolutional filters and the amount of boundary data (thus, repeated) that they require for the blocks

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

to be independent. Applying bottom up strategies in our problem, as they did, can lead us to obtain non-global solutions, and then losing accuracy. Also, our problem benefits from needing only 2 boundary points per segment regardless of the recursion level, and from being able to store the boundary points in local registers.

Mei *et al.* [13] proposes the parallelization of the *QuickHull* algorithm, another split-and-merge strategy similar to RDP, which looks for the furthest points for a series of segments. They focus on the use of a single GPU kernel, applying the collaborative and segmented patterns defined by Shubhabrata *et al.* [19] to nest recursion. In addition, they propose rearranging the data so that the resolved points are packed at the end of the block of blocks, and intra-warp divergence is minimized.

Our work focuses on the use of collaborative strategies at the segment level to solve segmentation with a single GPU kernel. Our strategy allows to reduce the collaborative work of the computer patterns proposed by Shubhabrata *et al.* [19], and benefits from operating on a fixed number of columns containing open segments, performing dynamic and recursive work in a homogeneous and iterative manner.

4 RDP Parallelization

This section summarizes the working behaviour of GPU architectures, for those readers who are not familiar with these systems, and describes our proposal to parallelize the RDP algorithm, as well as the optimization strategies applied to minimize its execution time.

4.1 Summary of the GPU Parallel Architecture

GPU systems are composed by tens of streaming multiprocessors (SMs), where each SM is made up of shared memory areas, schedulers for massively parallel execution, and thousands of registers, to be distributed among the execution threads. These systems are specialized in performing homogeneous computing, executing the instructions in a vectorized way (SIMD).

The CUDA programming model was designed as a logical layer over the GPU hardware to ease the use of parallel resources. CUDA allows to define kernels, C/C++ functions for which you can specify the number of parallel instances of the code that will be executed as threads. In addition, CUDA allows the creation of CTAs (Cooperative Thread Arrays), blocks of threads that can perform collaborative work, and with areas of low-latency shared memory. Each thread has its own local memory space, consisting of tens of registers, a shared memory area with quick access at the CTA level, and a global memory area, public for all the threads in the kernel, but with a large latency. Finally, unlike CPU threads, GPU threads are not completely independent; CUDA executes the threads of a block in warps, sub-groups of almost-synchronous threads that share execution instructions in SIMD fashion.

The efficiency of a parallel implementation is directly related to the level of local or shared memory usage versus global memory usage, the level of computation dependencies and divergences between threads, and the amount of hardware used to do the job. The objective of this work is to define an efficient GPU version of data segmentation, adapting the RDP algorithm to massively parallel systems.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

4.2 Parallelism Granularity of Our Proposal

Our proposal defines a single kernel to generate the segmentation of all the columns of the image, and assigns a single execution thread to each pixel of the image. Each thread calculates (iteratively) if a pixel is a cut-off point, and the threads corresponding to the pixels of the same data column are grouped into CTAs. This allows all the segmentation tasks on the same iteration to be performed in parallel, limiting the cooperative work at the segment level within a CTA.

The usage of a single kernel allows maximizing parallelism at the column level, and minimizing global memory transactions between kernels. The task-level parallelization alternative entails dynamic kernel creation and, therefore, the serialization of the kernels, which requires many data transactions over the long-latency global memory.

Another way to parallelize at the task level could be to assign to each CTA the calculation of a single segment. However, this requires the dynamic creation of CTAs of different sizes, which can only be done by calling new kernels. Our proposal avoids the dynamic creation of blocks. Treating the segmentation as a single function, instead of a task-level function, minimizes the amount of memory transactions and maximizes parallelism.

Our proposal redefines the recursive segmentation as a global and iterative process, detecting the boundaries between segments instead of creating new segments. Algorithm 2 shows a high-level view of our proposed segmentation strategy.

Algorithm 2 IterativeSegmentation

```
1: Set initial segment
2: repeat
3:     Compute the distance of each data value w.r.t. their segments.
4:     Find the position of the data at maximum distance on its corresponding segment
5:     Update list of segments for those segments whose maximum distance if larger than ε
6: until no new segments
```

4.3 Description of Our Parallel Proposal

We redefine the recursive algorithm iteratively, to avoid dynamic task generation. The calculation of the maximum distance performed on lines 2 to 4 of Algorithm 1, is now performed in parallel by all the threads, for each input data point. Each thread computes the distance to its corresponding segment using copies of the segment ends, stored in local registers, as shown in lines 6 to 9 of the Algorithm 3. By applying a parallel *reduce* computing pattern at the segment level, the last thread of each segment obtains the position of the data point at maximum distance in $log\, n$ steps. Then, each thread reads the position of the maximum distance of its segment, and updates the local information of its ends.

The exit condition from the convergence loop is controlled at the CTA level, using a single register in shared memory, which is updated in parallel for each new cut-off point found. The loop stops when no modification is done in this convergence register.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

Algorithm 3 ParallelSegmentation($data, cuts, \varepsilon$)

```
1: i = threadIdx, first = 0, last = colSize − 1, localCut = 1
2: if i == first or i == last then localCut = 0
3: repeat
4:     cutPointsConvergence[0] = yes
5:
6:     idx = i − first
7:     slope = (data[last] − data[first])/(last − first)
8:     maxCost[i] = |(slope ∗ idx + data[first]) − data[i]|
9:     if localCut == 0 or maxCost[i] < ε then maxCost[i] = 0
10:    argMax[i] = i
11:    __syncthreads()
12:
13:    for stride := colSize/2 → 1; stride = stride/2 do
14:        if idx > stride and maxCost[i] <= maxCost[i − stride] then
15:            maxCost[i] = maxCost[i − stride])
16:            argMax[i] = argMax[i − stride]
17:        __syncthreads()
18:
19:    if argMax[last] == i then
20:        cutPointsConvergence[0] = no
21:        localCut = 0
22:    else if argMax[last] < i then first = argMax[last]
23:    else if argMax[last] > i then last = argMax[last]
24:    __syncthreads()
25: until cutsConvergence[0] == yes
26: cuts[i] = localCut
```

4.4 Optimization Strategies Applied in Our Proposal

The parallel *reduce* computing pattern is a basic subroutine for many of the tasks performed with GPUs, and has been studied in depth by the community to take advantage of hardware features efficiently. The *stride* strategy shown in Figure 6 has become a *de facto* standard; since it allows maximizing the bandwidth of memory transactions, the warp occupancy, and the thread parallelism.

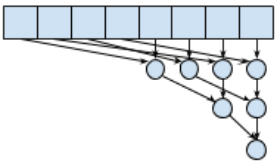

Fig. 6. Strided Reduction Pattern

On the one hand, every time a thread makes a memory access, the system loads a chunk of memory that contains the required data and some of its adjacent ones. When

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

adjacent threads access adjacent memory locations, then the number of chunks loads is minimized. On the other hand, as the threads are executed in warps (adjacent thread groups), the *stride* strategy allows to maximize the use of intra-warp operations, and reduces synchronization latencies.

Our segmentation proposal applies the *reduce* computing pattern for each segment of the same block in parallel. In addition, we limit the cooperation of segments with copies of the indexes of the ends for each thread of the segment. As segments are independent between them, all reductions in the same column can be calculated in parallel, but some strategy must be defined so that the reductions do not use elements from other segments. This problem was previously discussed by Shubhabrata *et al.* [19], whose proposal uses variables with the ID of the segment to limit *segmented-scans* within a block. However, in an algorithm with recursive behavior like RDP, the ID of the segment needs to be recalculated for each level of recursion, and the calculation of the IDs requires $log\ n$ steps. Our strategy allows to limit the cooperative work with the information of the ends of the segment stored locally in each thread, in this way we can avoid the calculation of the IDs of segment to limit the cooperative work.

The main downside of our proposal is that the hardware efficiency decreases as the segmentation of Algorithm 3 progresses; as some segments are solved, they do not require more *useful* computation, as shown in Figure 7. We have found that including specific conditions for these cases can decrease the number of floating-point operations and memory communication processes, but it represents a significant increase in control flow instructions and an increase in the execution time.

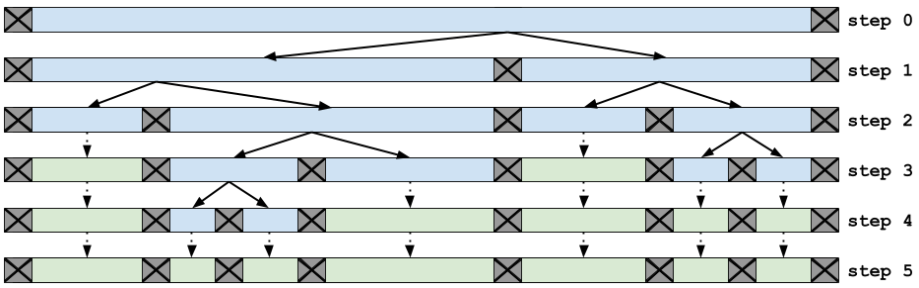

Fig. 7. Computation utility as segmentation progresses. Blue stands for useful computation, and green stands for threads whose segment is already computed.

5 Experimental Study and Results

This section evaluates the performance of our proposal for RDP parallelization. Our evaluation focuses on the performance of the proposal defined in Algorithm 3 (unrolling the last 5 iterations of the loop of lines 13 to 17); and in the performance gained on the parallel implementation with respect to the recursive implementation from Algorithm 1 executed in CPU. Our GPU proposal has been evaluated using *nvprof*, which

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

allows to capture metrics of the kernels, such as the number of instructions, memory transactions, or percentage of hardware usage. The CPU version has been evaluated using *perf* and the C++11 library *chrono*, in order to capture only the execution time of the recursive segmentation algorithm.

Since the underlying structure of the data determines the behavior of the RDP proposal, the input used for the experiments has been obtained by applying the SGM algorithm [9] to the stereo images of the KITTI [14] dataset. This dataset is composed by 200 images with 1242 columns of 375 pixels, with realistic structures on which to analyze the average performance of our proposal. Depth images generated with SGM have been scaled vertically so that each column has 1024 pixels, the maximum number of threads per block in Jetson's Xavier GPUs, in order to evaluate the maximum performance of our GPU algorithm.

The Jetson's Xavier architecture used for the experiments integrates a 512-core Volta GPU with 64 tensor cores, and a 8-core Carmel ARM CPU. This architecture allows to operate with different energy modes by specifying the clock frequencies for the internal GPU cores and CPU cores, the number of usable CPUs, and the level of power usage. Table 1 summarizes an evaluation of the execution time required for modes with maximum and minimum power consumption.

	Low Energy	Default Energy	Max Energy
GPU Max. Frequency (MHz)	520	670	1337
CPU Max. Frequency (MHz)	1200	1200	2265.5
Number of Online CPUs	2	4	8
Memory Max. Frequency (MHz)	1066	1333	2133
Total Power Budget (W)	10	15	30+
GPU Time (ms)	5.66	2.33	1.09
CPU Single-Core Time (ms)	43.9	40.10	21.15
CPU Multi-Core Time (ms)	23.3	14.31	4.97

Table 1. Energy mode configurations for the GPU and CPU cores (top). Execution time in miliseconds of the average input for columns of n=1024 pixels for GPU execution and for single-core and multi-core CPU execution (bottom).

Table 1 shows that our proposal can be executed in real time in a GPU regardless the energy mode. Assuming a target frame rate for real-time of 20 fps, a maximum slot of 50 ms per image, our proposal leaves, as average, from 28.5 to 23 ms of margin to the rest of the pipeline tasks, i.e. it occupies from 3.35% to 17.4% of the pipeline's time.

Figure 8 shows the selected input cases with the best and worst performance, and a full-noise input such as those that can be generated by unexpected SGM errors. We have found that even our proposal is able to ignore unknown depth points, unknown points are usually surrounded by errors. Smoothing depth pixels near unknown values could increase our proposal performance.

Figure 9 includes the average execution time (ms) of our proposal for different data sizes, and the execution times for the best and worst case of the SGM evaluation data.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

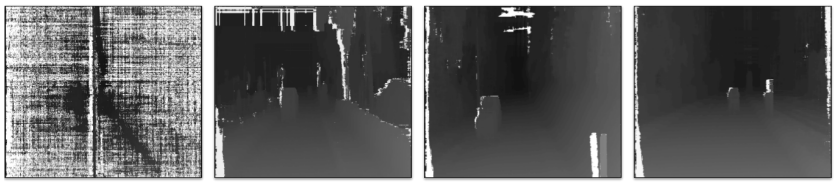

Fig. 8. Selected input cases used in our evaluations. From left to right: a corrupt input case, the worst input case from our dataset, one average input case, and the best input case.

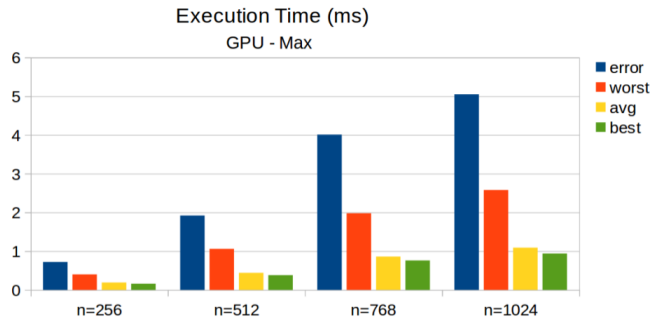

Fig. 9. Execution times of our GPU proposal for different input cases, and images with fixed width (1242px) and variable height (n); tested with max energy mode.

The columns sizes chosen for this evaluation ranges from n=256 to n=1024 pixels, the expected input size for autonomous driving systems. We aware that images with column sizes of 256px may include data smoothness given to the reduction of the original size (378px). This figure shows that by doubling the size of the column from 256 to 512, the execution time increases x2.5; while doubling the size from 512 to 1024, increases the execution time by x6. Also, it shows that the content of the depth image that is segmented has a great influence on the execution time: the higher the quality of the input, the shorter the execution time. The most remarkable result is that the execution times required by our proposal ($<$ 5ms) are good enough for real-time applications, even in our worst case scenarios.

Finally, Figure 10 summarizes and ranks the execution time required by the CPU and GPU proposals for different energy modes. This figure shows that the GPU implementation is 4.6 times faster than the best CPU option, in maximum power mode and for 1242x1024px images; and that this speedup can increase to 6.8 for 1242x256px images. This can be a great advantage in autonomous driving systems, where images have few rows (short columns), or are horizontally cropped. The small speed-up of the GPU execution with respect to the multi-core execution is due to the fact that number of operations per second of the CPUs, and therefore, the CPU energy consumption, is considerably higher. The energy efficiency of GPU cores implies that a GPU-accelerated solution is at least an order of magnitude faster than the execution of the same pro-

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

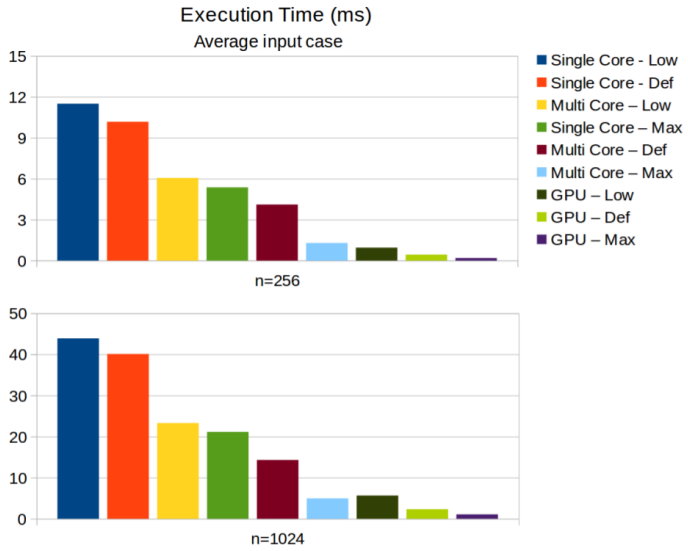

Fig. 10. Ranked execution times on CPUs and GPUs with different energy modes. At the top for 1242x256px images; at the bottom images with 1242x1024px.

gram on multiple CPU cores with similar energy consumption. This is another great advantage for autonomous driving systems.

6 Conclusions

In this paper we have presented a fine-grained parallelization of the RDP algorithm for the segmentation the multiple data vectors, a common task on many computer vision applications. Our research adapts the dynamic and recursive execution of the RDP algorithm to massively parallel environments, minimizing the execution of kernels, and using collaborative computing strategies to reduce its execution time. The great advantage of our proposal is the capability to manage the segmented-collaboration patterns with local registers, and avoiding the computation of the segment ids.

Our GPU proposal is from 3.8x to 6.8x times faster than an standard RDP recursive implementation executed on a multi-core CPU, and also consumes much less energy. Its low energy consumption, and its high execution speed (up to 1000 frames per second in 1024x1242px images), make it especially suitable for latency-limited environments such as autonomous driving systems. Using more CPU-cores allows to reach the same speed as our GPU solution, but this implies a significant increase in energy consumption, which might be critical in autonomous systems.

The experimental results also show that our proposal is sensitive to the contents of the input data, i.e. the execution time reduces as the structure of the input data is simpler and smoother. This can be taken as an advantage, by defining filters with which to pre-

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

process the segmentation inputs. For example, we can add filters to detect undefined outliers and delete them from the input data.

Future work could also focus on extending the size of the input data to values higher than n = 1024. This will require assigning more than one data point to each thread and carefully redistributing the larger amount of program's data among the different memory spaces (private, shared and global). In the same line of research, we can include specific conditions for the worst-case input cases, which generate many recursive levels. In this situation, most of the threads are doing redundant work, since their associated data points have already been fitted into a segment. When the RDP algorithm reaches a deep recursion level, detecting those threads doing redundant work and setting them idle could save resources that will benefit the execution of other thread blocks executing on the same SM.

We finally hope that our research will serve as base for studies on increasing the dimensionality of parallel split-and-merge strategies, so it can also be applied over 2D and 3D data.

References

1. Argüello, F., Heras, D., Bóo, M., Lamas-Rodríguez, J.: The split-and-merge method in general purpose computation on gpus. Parallel Computing **38**(6), 277 – 288 (2012). https://doi.org/https://doi.org/10.1016/j.parco.2012.03.003, http://www.sciencedirect.com/science/article/pii/S0167819112000208
2. Cebrian, P., Hernandez-Juarez, D., Moure, J.C.: Column-level segmentation of depth images for autonomous driving. Tech. rep., Autonomous University of Barcelona, Department of Computer Architecture and Operative Systems, Barcelona, Spain (Feb 2020)
3. Douglas, D.H., Peucker, T.K.: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartographica: the international journal for geographic information and geovisualization **10**(2), 112–122 (1973)
4. Dunham, J.G.: Optimum uniform piecewise linear approximation of planar curves. IEEE Trans. Pattern Anal. Mach. Intell. **8**(1), 67–75 (Jan 1986). https://doi.org/10.1109/TPAMI.1986.4767753, https://doi.org/10.1109/TPAMI.1986.4767753
5. Heckbert, P.S., Garland, M.: Survey of polygonal surface simplification algorithms (1997)
6. Hernandez-Juarez, D., Espinosa, A., Moure, J.C., Vázquez, D., López, A.M.: Gpu-accelerated real-time stixel computation. In: 2017 IEEE Winter Conference on Applications of Computer Vision (WACV). pp. 1054–1062 (March 2017). https://doi.org/10.1109/WACV.2017.122
7. Hernandez-Juarez, D., Schneider, L., Espinosa, A., Vazquez, D., Lopez, A.M., Franke, U., Pollefeys, M., Moure, J.C.: Slanted stixels: Representing san francisco's steepest streets. In: British Machine Vision Conference (BMVC), 2017 (2017)
8. Hu, X., Ye, L.: A fast and simple method of building detection from lidar data based on scan line analysis. ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences **II-3/W1**, 7–13 (05 2013). https://doi.org/10.5194/isprsannals-II-3-W1-7-2013
9. Juárez, D.H., Chacón, A., Espinosa, A., Vázquez, D., Moure, J.C., Peña, A.M.L.: Embedded real-time stereo estimation via semi-global matching on the GPU. CoRR **abs/1610.04121** (2016), http://arxiv.org/abs/1610.04121
10. Li, L., Jiang, W.: An improved douglas-peucker algorithm for fast curve approximation. In: 2010 3rd International Congress on Image and Signal Processing. vol. 4, pp. 1797–1802 (Oct 2010). https://doi.org/10.1109/CISP.2010.5647972

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17

11. Ma, J., Xu, S., Pu, Y., Chen, G.: A real-time parallel implementation of douglas-peucker polyline simplification algorithm on shared memory multi-core processor computers. In: 2010 International Conference on Computer Application and System Modeling (ICCASM 2010). vol. 4, pp. V4–647–V4–652 (Oct 2010). https://doi.org/10.1109/ICCASM.2010.5620612
12. Mahmoudi, S.A., Lecron, F., Manneback, P., Benjelloun, M., Mahmoudi, S.: Gpu-based segmentation of cervical vertebra in x-ray images. In: 2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS). pp. 1–8 (Sep 2010). https://doi.org/10.1109/CLUSTERWKSP.2010.5613102
13. Mei, G., Zhang, J., Xu, N., Zhao, K.: A sample implementation for parallelizing divide-and-conquer algorithms on the gpu. Heliyon **4**(1), e00512 (2018). https://doi.org/https://doi.org/10.1016/j.heliyon.2018.e00512, http://www.sciencedirect.com/science/article/pii/S2405844016326032
14. Menze, M., Geiger, A.: Object scene flow for autonomous vehicles. In: Conference on Computer Vision and Pattern Recognition (CVPR) (2015)
15. Paravecino, F.N.: Characterization and exploitation of nested parallelism and concurrent kernel execution to accelerate high performance applications. Ph.D. thesis, Northeastern University (2017)
16. Pikaz, A., Dinstein, I.: Optimal polygonal approximation of digital curves. In: Proceedings of 12th International Conference on Pattern Recognition. vol. 1, pp. 619–621 vol.1 (Oct 1994). https://doi.org/10.1109/ICPR.1994.576378
17. Romadi, M., Faizi, R., Chiheb, R., Romadi, R.: A shape-based approach for detecting and recognizing traffic signs in a video stream. In: 2017 4th International Conference on Control, Decision and Information Technologies (CoDIT). pp. 0254–0258 (April 2017). https://doi.org/10.1109/CoDIT.2017.8102600
18. Schneider, L., Cordts, M., Rehfeld, T., Pfeiffer, D., Enzweiler, M., Franke, U., Pollefeys, M., Roth, S.: Semantic stixels: Depth is not enough. In: 2016 IEEE Intelligent Vehicles Symposium (IV). pp. 110–117 (June 2016). https://doi.org/10.1109/IVS.2016.7535373
19. Sengupta, S., Harris, M., Garland, M., et al.: Efficient parallel scan algorithms for gpus. NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003 **1**(1), 1–17 (2008)
20. Tran, H., Scherger, M.: A massively parallel algorithm for polyline simplification using an associative computing model. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). p. 1. The Steering Committee of The World Congress in Computer Science (2011)

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_17