GPU Parallel Visibility Algorithm for a Set of Segments Using Merge Path

Kevin Zúñiga Gárate
Escuela Profesional de Ingeniería de Sistemas
Universidad Nacional de San Agustín
Arequipa, Perú
Email: kzunigaga@unsa.edu.pe

***Abstract*—In this paper, we present an efficient parallel algorithm for computing the visibility region for a point in a plane among a non-intersecting set of segments. The algorithm is based on the cascading divide-and-conquer technique and uses merge path to evenly distribute the workload between processors. We implemented the algorithm on NVIDIA's CUDA platform where it performed with a speedup up to 76x with respect to the serial CPU version.**

I. Introduction

Visibility is one of the most important problems in computational geometry, and it is a subproblem of many others, such as finding the shortest path in a plane with obstacles or the hidden line elimination problem. Diverse applications like video games and robotic motion planning have to deal with visibility. In this paper, we will focus on visibility from a point into a set of segments. Our goal is to provide an efficient parallel algorithm for modern parallel architectures such as the ones presented in current GPUs.

CPUs were getting more powerful in according to Moore's Law until around 2003 where they started to decline on their clock speed growth. This was due to elevated energy consumption, heat dissipation issues, and in general, various physical limits were being reached; exponential growth cannot go on forever after all [1]. CPU manufacturers like Intel and AMD started to shift towards hyperthreading and multicore CPUs to keep the processing power raising. In 2007 NVIDIA released their CUDA parallel computing platform which allowed us to use manycore GPUs for general purpose processing. Today many fields of computer science take advantage of these platforms, however, efficient parallel computing is a requirement not so easy to achieve.

The visibility problem on GPUs has already been seen by Shoja and Ghodsi [2]. In their work, they give a parallel algorithm to solve the point visibility problem on a simple polygon in $O(\log n)$ time with $O(n)$ processors. Our algorithm will solve the problem for a less restricted set of obstacles, a set of non-intersecting segments, taking the same computational time, which is optimal for the visibility of line segments [3].

Our algorithm is based on the technique given by Atallah et al. [4] for solving computational geometry problems on a divide and conquer paradigm for the CREW-PRAM computational model. We also take several visibility concepts from the work of Asano et al. [5] where they build a data structure to find a visibility graph in $O(n^2)$.

The remainder of this paper is organized as follows. In the next section, we give our definition of visibility region and describe one way to find it. Section III gives an overview of the cascading divide and conquer technique. In section IV we propose the algorithm itself, we give a simple analysis of its time complexity and an explanation of how can it be parallelized using merge path. Section V gives some details for its implementation on NVIDIA's CUDA platform. Section VI presents the experimental results of our implementation. Finally, conclusions and future works are discussed in section VII.

II. Visibility Region

Let S be a set of n arbitrarily oriented segments on the plane P allowed to intersect only at their endpoints, and let q be an arbitrary query point. The visibility region $\mathcal{V}_S(q)$ is the set of all points on P that are visible from q [6]. A point p is visible from q if the segment $\overline{pq}$ does not properly intersect any of the segments in S. We say two segments properly intersect if they share exactly one point and this point lies in the interior of both segments.

Figure 1 shows a line segment arrangement S, a query point q and the visibility region $\mathcal{V}_S(q)$. The visibility region $\mathcal{V}_S(q)$ might be a star-shaped polygon with q belonging to its kernel or it might be an unbounded region [6].

A. Finding the visibility region

To ease the calculations we are going to translate the coordinate system so that q becomes the origin.

Before finding the visibility region we are going to remove from S segments whose endpoints are collinear with q. Collinear segments cannot properly intersect any segment that has q as one of its endpoints so they do not affect the visibility region at all. We are also going to split segments that properly intersect the positive x-axis. If a segment with endpoints a and b crosses the positive x-axis on point c it is going to be divided into the segments $\overline{ac}$ and $\overline{cb}$. We will call this new set of segments S_q and let $n_q = |S_q|$.

To explain the previous modification to S we are going do a simple transformation of the coordinate system. Let p be any point on the plane except for the origin. Let us denote $\theta(p)$

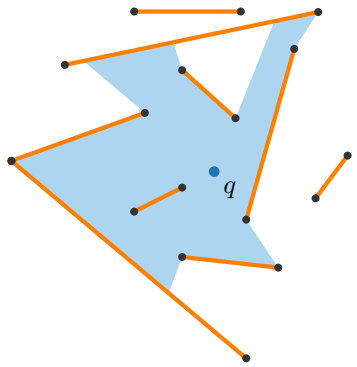


Fig. 1. Query point q in a line segment arrangement S with its visibility region $\mathcal{V}_S(q)$. The visibility region is a star-shaped polygon.

as the counterclockwise angle from the x-axis at which the point p lies. Let $d(p)$ be the distance between the point p and the origin. We are going to map p to the point $(\theta(p), d(p))$. Figure 2 shows an example of this transform. Observe that a collinear segment with q would be a vertical segment on the transformed system and as we mentioned before those do not affect visibility. Also, note that a segment that crosses the positive x-axis would be split in the transformed system since the polar angles of its points would abruptly change from 0 to 2π.

In [5], Asano et al. establish a binary relation $\prec_q$ on the set S_q. For two segments s and s' in S_q we say that $s \prec_q s'$ if there exists a ray originated in q that intersects both segments and hits s before s'. In other words, if $s \prec_q s'$ this means that s' is partially blocked by s and therefore is not completely visible from q. The part blocked in s' is the one that can be intersected by rays starting at q that also intersect s. See segments c and d in figure 2. This relation is a partial order on the set S_q [7].

To continue we are going to divide the original plane into angular sectors. Let p_i $(i = 1, \ldots, N)$ be all the endpoints of the segments in S_q, where $N \leq 2n_q$. Let φ be the linearly ordered set of $\{\theta(p_i) \ / \ (i = 1, \ldots, N)\} \cup \{0, 2\pi\}$, let $n_\varphi = |\varphi|$ so that $\varphi_1 = 0$ and $\varphi_{n_\varphi} = 2\pi$. Let us also define $\overrightarrow{r_i}$ as the ray emanating from q in the direction of φ_i. We denote as Λ_i $(i = 1, \ldots, n_\varphi - 1)$ the infinite angular sector defined between the angles φ_i and φ_{i+1}.

Let $\mathcal{Z}_i$ be the set of segments in S_q that properly intersect the region Λ_i and let z_i be any segment in $\mathcal{Z}_i$ such that $z_i \preceq_q s$ for all s in $\mathcal{Z}_i$. We define R_i as the region Λ_i clipped by the segment z_i, i.e., the triangular area formed by q and the intersection points of z_i with the rays $\overrightarrow{r_i}$ and $\overrightarrow{r_{i+1}}$. In case $\mathcal{Z}_i$ is empty let $R_i = \Lambda_i$.

On figure 2a you can see the plane divided into 7 regions. Observe that $R_3 = \Lambda_3$ since $\mathcal{Z}_3$ is empty, and that Λ_5 is properly intersected by segments c and d with $d \prec c$ so the region R_5 is limited by d.

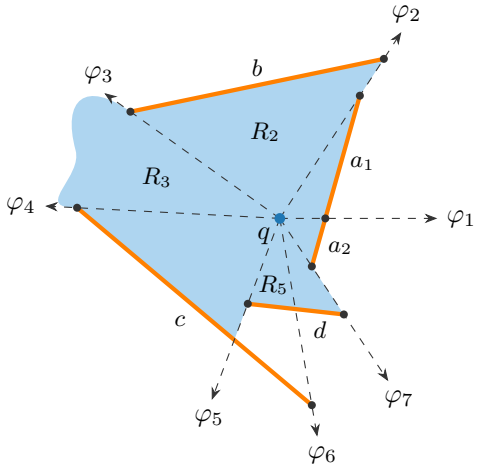
The visibility region will be the union of all R_i regions, i.e.:

$$\mathcal{V}_S(q) = \bigcup_{i=1}^{n_\varphi - 1} R_i$$

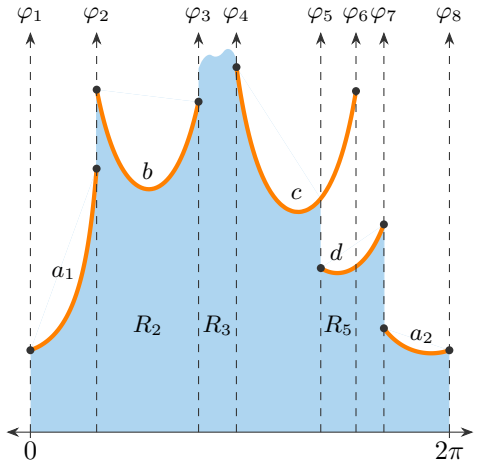
III. Cascading Divide and Conquer

Cascading divide and conquer is a technique for designing parallel divide and conquer algorithms by Atallah et al. [4]. This technique can be used to solve many geometric problems, including point visibility. The algorithms run in $O(\log n)$ time with $O(n)$ processors in the CREW PRAM model. The technique is based on Cole's parallel merge sort algorithm [8].

In the cascading divide and conquer technique we model the solution process as a binary tree. Each node represents a



(a) Normal



(b) Transform

Fig. 2. (a) Query point q, a set of segments S_q and its visibility region. Segment a was divided into a_1 and a_2. Dashed rays represent the angles in φ. Regions R_2, R_3 and R_5 have been labeled. Observe that R_3 is the infinite region Λ_3 and that R_5 is intersected by c and d but bounded by d since $d \prec c$. (b) Transform of elements in (a). The horizontal axis is the polar angle $\theta(p)$ and the vertical axis is the distance $d(p)$. Note that segments a_1 and a_2 are no longer next to each other.

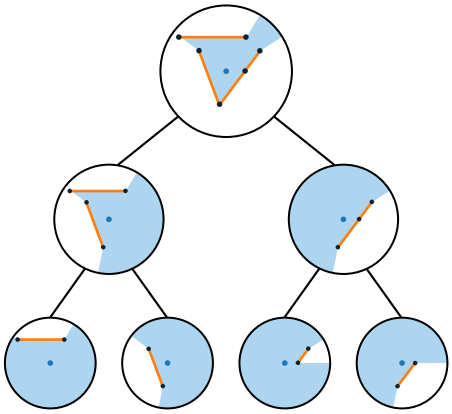


Fig. 3. Cascading divide and conquer tree model for the point visibility problem. Leaf nodes represent the visibility region of a single segment. Inner nodes represent the intersection or merge of the visibility regions of their two children. Root node is the final visibility region of all segments.

sorted list of some type. For the inner nodes, the list is the sorted merge of their two children in a rather complex way. The starting lists of the leaf nodes depend on the nature of the problem. We find the solution of the problem by merging the nodes in a bottom-up fashion. The root of the tree represents the final solution.

For the visibility problem, the lists represent a visibility region and store the endpoints of its obstacles, sorted by their polar angle. We also store additional information to identify the closest segment for two consecutive angles, i.e., z_i. Figure 3 shows what the tree model for the visibility problem might look like.

IV. Visibility Merge Algorithm

In this section, we propose an algorithm based on the cascading divide and conquer technique for the point visibility problem. For simplicity we will assume that the segments have already been translated so the query point is the origin, the collinear segments with the query point have already been removed, and the segments that cross the x-positive axis have already been broken in two.

A. Visibility Merge Algorithm

Each node of the cascading divide and conquer tree model represents a visibility region for some subset of S_q. To represent this visibility region we will use a list of v-rays. These v-rays represent the rays dividing the plane we saw in section II. A v-ray is a 4-tuple $(\varphi,\ \vec{v},\ r,\ l)$ where

- φ is the polar angle,
- $\vec{v}$ is a unit vector in the direction of φ,
- r and l are the distance from q to the endpoints of the segments limiting the angular sector on the right and left side respectively. They take the value of ∞ if there is no segment.

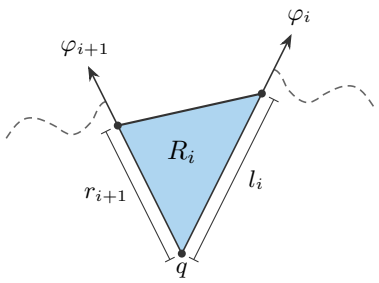


Fig. 4. Region defined by two consecutive v-rays. Dashed lines represent the r_i and l_{i+1} values, they do not affect the region R_i at all.

A pair of consecutive v-rays $(\varphi_i,\ \vec{v}_i,\ r_i,\ l_i)$ and $(\varphi_{i+1},\ \vec{v}_{i+1},\ r_{i+1},\ l_{i+1})$ with $\varphi_{i+1} > \varphi_i$ define the region R_i as the triangular area formed by the points q, $\vec{v}_i \cdot l_i$ and $\vec{v}_{i+1} \cdot r_{i+1}$. See figure 4. In case l_i and r_{i+1} are infinite, the region R_i would be the infinite region between the angles φ_i and φ_{i+1}.

The first step of the algorithm would be to find the visibility region of every single segment. Algorithm 1 finds the v-rays for the two endpoints of a segment. Notice that the endpoints are sorted by their angle, and we handle the special case when the b v-ray has an angle of 0. The v-rays for the angles $\varphi = 0$ and $\varphi = 2\pi$ are implicit and have the values of $(0,\ \vec{i},\ \infty,\ \infty)$ and $(2\pi,\ \vec{i},\ \infty,\ \infty)$ respectively.

Algorithm 1: Visibility region of a single segment.

Input: A segment $s = (a,\ b)$.

Output: A list of two v-rays representing $\mathcal{V}_{\{s\}}(q)$.

if $|a \times b| < 0$ **then** $\text{swap}(a, b)$

$\varphi_a \leftarrow \theta(a)$

if $\theta(b) = 0$ **then** $\varphi_b \leftarrow 2\pi$ **else** $\varphi_b \leftarrow \theta(b)$

$vr_a \leftarrow (\varphi_a,\ \vec{i}\cos\varphi_a + \vec{j}\sin\varphi_a,\ \infty,\ d(a))$

$vr_b \leftarrow (\varphi_b,\ \vec{i}\cos\varphi_b + \vec{j}\sin\varphi_b,\ d(b),\ \infty)$

return $[vr_a,\ vr_b]$

Once we have the initial visibility regions we need to merge them together, two at a time. The merge is identical to the merge from mergesort with the exception that we need to update the values r and l for each processed v-ray. The r and l values can only decrease, and this happens if there is a segment that limits the region on the right and left side of the v-ray respectively. Let us say that we are merging two v-ray lists A and B, and we are adding the v-ray a_i to the merge result, there is only one segment that might limit its right and left regions, the one limiting the region R_{j-1} in the visibility region defined by list B, see figure 5. If the segment does not exist, e.g. if the region between b_j and b_{j-1} is infinite, a_i would keep its current r and l values. Since there is only one segment we need to check, we can update the r and l values

Algorithm 2: Merge two lists of v-rays.

Input: L_1 and L_2 lists of v-rays.
Output: Merged list L.

```
n ← |L1| + |L2|
i ← 1, i1 ← 1, i2 ← 1
while i ≤ n do
    if i2 > |L2| then k = 1, t = 2
    else if i1 > |L1| then k = 2, t = 1
    else if L1[i1].φ ≤ L2[i2].φ then k = 1, t = 2
    else k = 2, t = 1
    L[i] ← Lk[ik]
A   if 1 < it ≤ |Lt| and Lt[it].r < ∞ then
        s ← segment(Lt[it − 1].v⃗ · Lt[it − 1].l,
          Lt[it].v⃗ · Lt[it].r)
        p ← intersection of s with the infinite ray
          originated at the origin on the direction of
          L[i].φ
        L[i].l ← min(L[i].l, d(p))
        L[i].r ← min(L[i].r, d(p))
    end
    ik ← ik + 1
    i ← i + 1
end
```

in constant time.

Algorithm 2 merges two lists of v-rays updating the r and l values as described. Observe that, except for the *if* condition in the line A, the algorithm is pretty much a regular merge.

Since the additional operations take constant time to compute, the time complexity for the Visibility Merge algorithm is the same as a regular merge, $O(n)$. As in mergesort, in order to merge all the elements we need to do $O(\log n)$ passes, each pass taking a total of $O(n)$ time. The final time complexity to find the visibility region is $O(n \log n)$.

B. Parallel Merge

Given that this algorithm is so akin to mergesort we can use parallel merge [8] to parallelize it. The way parallel merge works is by splitting the lists to be merged into non-overlapping sublists. Each processor then does a serial merge of the two sublists assigned to it.

The problem with parallel merge is that the workload is not evenly divided and processors have a different amount of elements to merge. This is not efficient for GPU architecture because threads are grouped in warps that must execute the same instructions, and it could lead to idle threads while others are working. This problem is solved by GPU merge path [9].

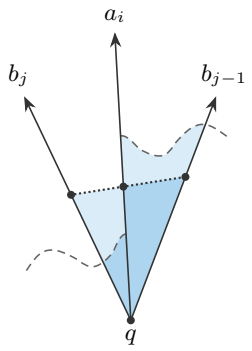


Fig. 5. Updating the r and l values of a v-ray. a_i is the chosen v-ray to add to the output list. b_j and b_{j-1} are the next and previous v-rays on the other list. The dotted line is the bounding segment of the region R_{j-1} in the B list. The dashed gray lines represent the bounding segments for the regions R_{i-1} and R_i in the A list. In this example the dotted segment is limiting the R_{i-1} region, but not the R_i region.

Merge path divides the lists into sublists so that the length of the merge result is the same across all threads.

In the next section, we will give some details of our parallel visibility merge implementation on NVIDIA's CUDA platform.

V. CUDA Implementation

The algorithm was implemented in C++ with NVIDIA's CUDA toolkit 9.1[1]. We targeted a specific GPU, the Tesla K80, with compute capability 3.7.

The CUDA programming model is described in [10]. Basically, we can invoke C functions on the GPU, named kernels, that are called N times on N threads in parallel. The threads are organized in blocks. One block is a group of threads that have access to a shared memory. Multiple blocks reside in a Stream Multiprocessor (SM), which has hardware limitations in the amount of shared memory and number of threads that can be executed concurrently.

For the compute capability 3.7 the limits are as follows [10]:

- 1024 threads per block,
- 16 blocks per SM,
- 112 KB shared memory per SM, and
- 48 KB shared memory per block.

Those limits are important because we need them in order maximize the GPU occupancy. Let us say that we process vt segments per thread, and we have nt threads per block. That means we process $nv = vt \cdot nt$ segments per block. The visibility region of one segment needs two v-rays to be represented. In C code, the v-ray list is represented as 3 arrays of numbers: one for the angles and two for the right and left scalars (we do not need to store the unit vector since we already store the angle), therefore the size, sz, of a v-ray is 12 bytes if we use single-precision floating points or 24 bytes if we use double-precision. We also use 3 additional arrays as

[1]Source code can be found at: `https://github.com/kevinzg/visimerge.git`.

TABLE I
GPU OCCUPANCY FOR SINGLE AND DOUBLE PRECISION FLOATING POINTS NUMBERS.

					Single-precision		Double-precision	
nt	vt	nv	blocks per SM	threads per SM	memory per block	memory per SM	memory per block	memory per SM
32	1	32	16	512	1,536	24,576	3,072	49,152
64	1	64	16	1024	3,072	49,152	6,144	98,304
64	2	128	16	1024	6,144	98,304	12,288	*196,608
128	1	128	16	2048	6,144	98,304	12,288	*196,608
128	2	256	16	2048	12,288	*196,608	24,576	*393,216
256	1	256	8	2048	12,288	*196,608	24,576	*393,216

output buffers. Consequently, we require of 4 $(nv \times sz)$ bytes to process nv segments in a block.

Table I shows the number of threads per SM and the shared memory in bytes needed for various values of nt and vt. Values marked with a $*$ exceed the hardware limits. Having less than 128 threads per block means that the maximum number of threads per SM will not be reached. The best values for single precision floating point numbers is 128 threads per block, and process 1 segment per thread, the reason for this is that we get to use all the threads in a Stream Multiprocessor, and do not exceed the shared memory limit. For double-precision, there are not optimal values to maximize occupancy, but we obtained the best results with 64 threads per block, and 1 segment per thread.

The first kernel call computes the initial v-rays for every single segment, this task is trivial to parallelize since there is no need for cooperation between threads.

Visibility regions are then computed at two levels. First at a block level where we independently compute the visibility region of the nv segments assigned to each block. We use GPU merge path to partition the lists and assign them to the nt threads. After $\log_2 nv$ passes we get the visibility region of the nv segments.

At the next level we compute the visibility region of all segments. We merge the n/nv visibility regions cooperating between blocks. Just as at the block level we use merge path to partition the lists and assign the sublists to the blocks. The blocks then again divide their sublists and assign them to their threads. We do this for $\log_2 (n/nv)$ passes and finally obtain our result.

Next section shows the results of this implementation.

VI. EXPERIMENTAL RESULTS

For the experiments, we have implemented a serial version of the algorithm in C++ in addition to the parallel CUDA version. We have tested these two implementations with single-precision and double-precision floating point numbers, and with problem sizes up to 8 million segments.

The serial version of the algorithm ran on a 3.00GHz Intel Xeon Platinum 8124M CPU with 16GiB RAM. The parallel version ran on a single Tesla K80 GPU with 2496 CUDA cores at 562MHz and 12 GiB VRAM.

TABLE II
TIME TO FIND THE VISIBILITY REGION OF n SEGMENTS.

	Single-precision		Double-precision	
n	CPU (ms)	GPU (ms)	CPU (ms)	GPU (ms)
128K	93.559	4.284	109.109	6.199
256K	200.959	5.699	233.126	9.563
512K	423.761	8.555	494.696	16.927
1M	894.754	14.529	1041.260	31.362
2M	1877.940	27.180	2182.070	62.058
4M	3927.400	53.784	4599.920	126.723
8M	8215.790	108.532	9544.570	269.777

CUDA code was compiled with CUDA toolkit 9.1 and C++ compiler g++ 5.4.0. The compute capability version set for compilation was 3.7 to target the Tesla K80 GPU. Additional flags were `-std=c++14`, `-O2` and `-use_fast_math`. The serial version was compiled with just the `-std=c++14` and `-O2` flags.

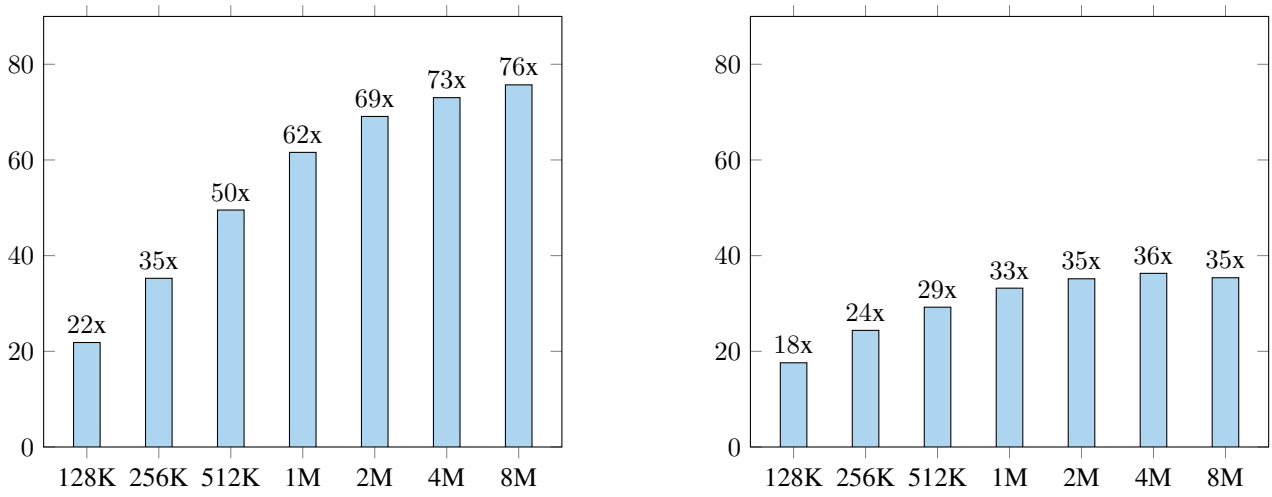
The CUDA kernel launch parameters for the single-precision floating point tests were 128 threads per block and as many blocks as it was needed to process the entire input. The configuration for the double-precision tests was 64 threads per block. Those values were chosen in order to maximize the GPU occupancy.

The timing for the GPU does not include input/output operations between the host and the device.

Table II shows the timing for the computation of the visibility region of n segments in milliseconds for the CPU and GPU, and for single and double precision floating point numbers. Figure 6 shows the relative speedup of GPU vs CPU for each of the problem size tested.

VII. CONCLUSION

We have shown an efficient parallel algorithm to find visibility region of a point in a set of segments. The algorithm is heavily inspired by mergesort and therefore we have implemented it on NVIDIA's CUDA platform using current techniques for GPU mergesort, namely GPU Merge Path.



(a) Single-precision floating-point speedup

(b) Double-precision floating-point speedup

Fig. 6. GPU vs CPU execution time speedup for different number of segments.

The implementation uses the GPU hardware efficiently in both processing power and memory system, outperforming the sequential version by a factor of up to 76x for single precision floating point numbers and 36x for double precision.

Future research includes the efficient parallelization of other visibility algorithms and their implementation on modern parallel platforms such as a GPU. These algorithms include those that solve problems like the finding the visibility graph of a set of obstacles, the art gallery problem and their 3D variations as well.

ACKNOWLEDGMENT

This work was partially supported by the CiTeSoft research group at the Universidad Nacional de San Agustín de Arequipa, Peru.

REFERENCES

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005.

[2] Ehsan Shoja and Mohammad Ghodsi, "GPU-based Parallel Algorithm for Computing Point Visibility Inside Simple Polygons," *Comput. Graph.*, vol. 49, no. C, pp. 1–9, Jun. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.cag.2015.02.010

[6] S. Ghosh, *Visibility Algorithms in the Plane*. New York, NY, USA: Cambridge University Press, 2007.

[3] S. Suri and J. O'Rourke, "Worst-case Optimal Algorithms for Constructing Visibility Polygons with Holes," in *Proceedings of the Second Annual Symposium on Computational Geometry*, ser. SCG '86. New York, NY, USA: ACM, 1986, pp. 14–23. [Online]. Available: http://doi.acm.org/10.1145/10515.10517

[4] M. J. Atallah, R. Cole, and M. T. Goodrich, "Cascading Divide-and-conquer: A Technique for Designing Parallel Algorithms," in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, ser. SFCS '87. Washington, DC, USA: IEEE Computer Society, 1987, pp. 151–160. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1987.12

[5] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai, "Visibility of disjoint polygons," *Algorithmica*, vol. 1, no. 1-4, pp. 49–63, Nov. 1986. [Online]. Available: https://link.springer.com/article/10.1007/BF01840436

[7] B. Chazelle, "Filtering Search: A New Approach to Query-Answering," in *Siam Journal on Computing - SIAMCOMP*, vol. 15, Dec. 1983, pp. 122–132.

[8] R. Cole, "Parallel Merge Sort," *SIAM J. Comput.*, vol. 17, no. 4, pp. 770–785, Aug. 1988. [Online]. Available: http://dx.doi.org/10.1137/0217049

[9] O. Green, R. McColl, and D. A. Bader, "GPU Merge Path: A GPU Merging Algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 331–340. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304621

[10] C. Nvidia, "CUDA C programming guide, version 9.1," *NVIDIA Corp*, 2018.