# ACCELERATING LINE OF SIGHT ANALYSIS ALGORITHMS WITH PARALLEL PROGRAMMING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

GÖKHAN YILMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF GAME TECHNOLOGIES

MAY 2017

**ACCELERATING LINE OF SIGHT ANALYSIS ALGORITHMS WITH PARALLEL PROGRAMMING**

Submitted by Gökhan Yılmaz in partial fulfillment of the requirements for the degree of **Master of Science in The Department of Modelling and Simulation Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Director, **Graduate School of Informatics** ____________________

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Head of Department, **Modelling and Simulation** ____________________

Assoc. Prof. Dr. Alptekin Temizel
Supervisor, **Modelling and Simulation** ____________________

Assist. Prof. Dr. Elif Sürer
Co-Supervisor, **Modelling and Simulation** ____________________

**Examining Committee Members:**

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Modelling and Simulation, Middle East Technical University ____________________

Assoc. Prof. Dr. Alptekin Temizel
Modelling and Simulation, Middle East Technical University ____________________

Assist. Prof. Dr. Elif Sürer
Modelling and Simulation, Middle East Technical University ____________________

Assist. Prof. Dr. Aysu Betin Can
Information Systems, Middle East Technical University ____________________

Assist. Prof. Dr. Adnan Özsoy
Computer Engineering, Hacettepe University ____________________

**Date:** __________

**I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.**

**Name, Last name :** GÖKHAN YILMAZ

**Signature :** ____________________

## ABSTRACT

ACCELERATING OF LINE OF SIGHT ANALYSIS ALGORITHMS WITH PARALLEL PROGRAMMING

Yılmaz, Gökhan
MSc., Department of Modelling and Simulation
Supervisor: Assoc. Prof. Dr. Alptekin Temizel
Co-supervisor: Assist. Prof. Dr. Elif Sürer

May 2017, 64 pages

Line of sight (LOS) analysis is a set of methods and algorithms to determine the visible points in a terrain with reference to a specific observer point. This analysis is used in simulations, Geographic Information System (GIS) applications and games. For this reason, it is important to have a capability to get results quickly and facilitate analysis in such a way that the interaction with the changing reference points is possible. Van Kreveld, R2 and R3 are the most frequently used algorithms in line of sight analysis. The purpose of this research is to develop parallel adaptations of these particular algorithms by making use of the capabilities of a modern Graphics Processing Unit (GPU) and to evaluate these adaptations in terms of performance and memory usage. By analyzing which algorithm is more suitable to be implemented on the GPU, the algorithm that will provide the most appropriate and quick solution to the probing problem can be determined. In this research, Van Kreveld's algorithm, which is basically a sequential algorithm, was developed partly in parallel, and the speed-up was 1.5x compared to the sequential version of the algorithm. Speed-up rates increase up to 10.5x for R2 and 160x for R3 algorithms, respectively. The results can be used to combine CPU / GPU approaches in order to perform hybrid or full parallelization of Van Kreveld’s Algorithm on the GPU. The results presented in the thesis will serve as a guide for the selection of the appropriate algorithm by evaluating the strengths and weaknesses of different algorithms.

Keywords: geographic information systems, line of sight analysis, GPGPU, parallel programming

# ÖZ

GÖRÜŞ HATTI ANALİZİ ALGORİTMALARININ PARALEL PROGRAMLAMA İLE HIZLANDIRILMASI

Yılmaz, Gökhan

Yüksek Lisans, Modelleme ve Simülasyon Bölümü

Tez Yöneticisi: Doç. Dr Alptekin Temizel

Eş Tez Yöneticisi: Yrd. Doç. Dr Elif Sürer

Mayıs 2017, 64 sayfa

Görüş hattı analizi, belirli bir nokta referans alınarak bir arazi içerisindeki görülebilir noktaların belirlenmesini amaçlayan yöntemler ve algoritmalar bütünüdür. Bu analiz, simülasyonlarda, Coğrafi Bilgi Sistemi (CBS) uygulamalarında ve oyunlarda kullanılmaktadır. Bu nedenle hızlı bir şekilde sonuç alabilmek, değişen referans noktalarına göre etkileşimi mümkün kılabilecek şekilde analiz yapabilmek önem taşımaktadır. Görüş hattı analizi algoritmalarından sıklıkla kullanılanlar Van Kreveld, R2 ve R3 algoritmalarıdır. Bu araştırmanın amacı, modern bir Grafik İşleme Ünitesi'nin (GPU) kabiliyetlerini kullanarak belirtilen algoritmaların paralel uyarlamalarını geliştirip, bu uyarlamaların performans ve hafıza kullanımları açısından değerlendirilmesini yapmaktır. Böylelikle hangi algoritmanın GPU üzerinde gerçeklenmeye daha uygun olduğu analiz edilerek, ilgili probleme en uygun ve hızlı çözümü sağlayacak algoritma belirlenebilecektir. Bu araştırmada temel olarak sıralı ilerleyişi olan Van Kreveld'in Algoritması kısmi şekilde paralel olarak geliştirilmiştir ve yapılan gerçeklemede algoritmanın sıralı haline göre 1.5x kata kadar daha hızlı sonuç elde edilmiştir. CPU/GPU hız artışı R2 için 5 kata kadar, R3 için ise 160 kata kadar ulaşmıştır. Çalışma sonucu, Van Kreveld Algoritması için CPU/GPU yaklaşımlarını birleştirerek karma ya da GPU üzerinde tam bir paralelleştirme yapmak için kullanılabilir. Tezde sunulan sonuçlar farklı algoritmaların güçlü ve zayıf noktalarını değerlendirerek ihtiyaca uygun algoritmanın seçilmesi sırasında yön gösterici olacaktır.

Anahtar Sözcükler: coğrafi bilgi sistemleri, görüş hattı analizi, GPGPU, paralel programlama

dedicated to all the people who fight for equal and fair life

ACKNOWLEDGMENTS

First of all, I would like to thank my thesis advisor Assoc. Prof. Dr. Alptekin Temizel and co-supervisor Assit. Prof. Dr. Elif Sürer. I wish I could be a more hardworking student to meet their good intentions, I could not complete this thesis without them.

I also would like to thank my family, especially to my sister, Esin Yılmaz, for her support and breakfast duets before my thesis work.

And finally, I would like to thank my friends who drank beer with me and listened to my complaints about this thesis, and always said “You can do it!”.

.

# TABLE OF CONTENTS

ABSTRACT ........ iv
ÖZ ........ v
DEDICATION ........ vi
ACKNOWLEDGMENTS ........ vii
TABLE OF CONTENTS ........ viii
LIST OF TABLES ........ x
LIST OF FIGURES ........ xi
LIST OF ABBREVIATIONS ........ xii
CHAPTERS
1 INTRODUCTION ........ 13
1.1 Motivation ........ 14
1.2 Objective of the Study and Contribution ........ 15
1.3 Outline ........ 16
2 BACKGROUND AND PREVIOUS WORK ........ 17
2.1 Terrain Structure ........ 17
2.2 Line of Sight and Viewshed ........ 17
2.3 Viewshed Algorithms ........ 20
2.3.1 R3 Algorithm ........ 20
2.3.2 R2 Algorithm ........ 21
2.3.3 Van Kreveld’s Algorithm ........ 23
2.3.4 Parallel Algorithms for Viewshed Calculation ........ 27
2.4 GPGPU ........ 28
2.4.1 CUDA ........ 29
2.4.2 Thrust ........ 31
3 GPU IMPLEMENTATIONS OF THE ALGORITHMS ........ 33
3.1 GPU Implementation of R3 ........ 33
3.2 GPU Implementation of R2 ........ 37

3.3 GPU Implementation of Van Kreveld's Algorithm ........................................40
4 RESULTS AND DISCUSSION ................................................................43
4.1 Specifications of the Hardware and Software ..........................................43
4.2 Results And Disccusion........................................................................43
4.2.1 Time Results .........................................................................44
4.2.2 Memory Usage .......................................................................48
4.2.3 Viewshed Comparison ..............................................................50
5 CONCLUSIONS AND FUTURE WORK ...............................................53
5.1 Conclusions ...............................................................................53
5.2 Future Work ..............................................................................53
REFERENCES....................................................................................55
APPENDICES ....................................................................................59
APPENDIX A .....................................................................................59
APPENDIX B .....................................................................................60
APPENDIX C .....................................................................................61

# LIST OF TABLES

Table 1 : Results of R3 algorithm ........................................................................44
Table 2 : Results of R2 algorithm ........................................................................45
Table 3 : Results of Van Kreveld's algorithm.....................................................................47
Table 4 : Memory usage of R3 algorithm ........................................................................48
Table 5 : Memory usage of R2 algorithm. ........................................................................49
Table 6 : Memory usage of Van Kreveld's algorithm. .....................................................49
Table 7 : Different pixels of R2 ........................................................................51
Table 8 : Different pixels of Van Kreveld's algorithm ....................................................51

## LIST OF FIGURES

Figure 1: Example viewshed visualization ....................13
Figure 2: DEM with geographical coordinate unit ....................13
Figure 3: LOS vertical cross section ....................18
Figure 4: Line rasterization ....................18
Figure 5: Top-down view of viewshed. ....................19
Figure 6: R3 calculation ....................20
Figure 7: R2 calculation ....................22
Figure 8: Event types ....................23
Figure 9: The active tree ....................24
Figure 10: Event angles ....................26
Figure 11: CUDA architecture ....................29
Figure 12: Thread grouping ....................30
Figure 13: Elevation data on global memory ....................33
Figure 14: Area of interest in elevation data. ....................34
Figure 15: R3 cell to thread mapping ....................35
Figure 16: Local and global indices ....................37
Figure 17: R2 cell to thread mapping ....................38
Figure 18: Event list in global memory ....................41
Figure 19: R2 GPU speed up compared to R3 GPU ....................46
Figure 20: R2 CPU speed up compared to R3 CPU ....................46
Figure 21: Van Kreveld’s CPU speed up compared to R3 CPU ....................47
Figure 22: Sample viewshed outputs of different algorithms (10000x10000) ....................50

## LIST OF ABBREVIATIONS

| | |
|---|---|
| **LOS** | Line of Sight |
| **GPU** | Graphical Processing Unit |
| **DEM** | Digital Elevation Model |
| **CPU** | Central Processing Unit |
| **GPGPU** | General Purpose Computing on Graphical Processing Unit |
| **TIN** | Triangulated Irregular Network |

# CHAPTER 1

# INTRODUCTION

Line-of-Sight analysis is an analysis method that creates a visibility flag for every point on a given DEM (Digital Elevation Model). The visibility flag indicates whether or not that particular point can be seen from a certain observation point and in a fixed area. The output area is called a viewshed. A sample viewshed visualization is displayed in Figure 1.

Figure 1: Example viewshed visualization (Green: Visible, Red: Not Visible)

DEM is a data format that holds elevation values of a terrain with a fixed sampling period. Sampling period can be represented as a geographical coordinate unit, a custom Cartesian coordinate unit or a distance unit (See Figure 2).

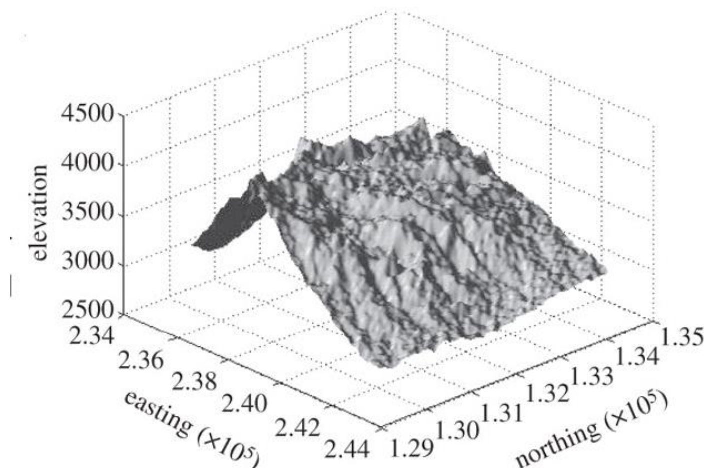



Figure 2: DEM with geographical coordinate unit(Stefanescu et al., 2012)

Visibility of a *point A* means that, the line-of-sight from an observer point to *point A* is not blocked by the terrain, which proves that *point A* can be clearly seen by the observer.

Viewshed can be used in many applications such as the following:

- A Zone of Visual Influence: To determine general visibility effect of a newly constructed building at a terrain. For example, while building a wind turbine farm, one can determine the visibility of a wind turbine at a specific place.

- Placement Problems: Mobile phone base station placement, security camera placement, light placement and safety spot determination are some examples where the placement problems can be solved by viewsheds (Ten Eyck, 2011).

- Visibility Analysis: To determine and mark points where a moving object can be seen by an observer in an area of interest.

- Path planning: In a game or a simulation, the visibility output can be used by AI to find the best path to travel without being seen by a certain object.

## 1.1 Motivation

Viewsheds have many applications; nevertheless, viewshed calculation is a very time-consuming process especially for large areas. As a result, in most cases it is a pre-calculated static data due to the infeasible calculation time. Recent advances in the GPUs and their programming tools made the online calculation of viewsheds possible by parallel programming (Gao et al., 2011), facilitating more interactive applications.

GPU (Graphical Processor Unit) is a specific hardware that is used for computer graphics rendering. It rapidly manipulates and creates image data for frame buffer and sends this output to display devices. Because of the characteristics of these operations, GPUs have been developed as high-performance many-core processors which are capable of very high computation and data throughput (Harris, 2002). At the early stages of GPU development, it was not easy to take advantage of the hardware and the available processing power because developers needed to go through a tedious development process and write assembly code to achieve the desired outcome. The development of C-like GLSL and HLSL shading languages allowed easy manipulation of the behavior of graphical pipeline. To use GPU for general purpose programming and exploit its high performance processing power, higher-level models like CUDA and OpenCL have been introduced, which allow programmers to write C code to solve their non-graphical problems on GPU (Wu, 2008). This approach is called General Purpose Computing on Graphical Processing Unit (GPGPU).

With the capability of the modern GPUs, real-time viewshed calculation is now an achievable goal, and the output can be used in real-time applications such as games and simulations even for large areas.

Massive terrain rendering is widely used in GIS and simulation applications. It can also be used in games with the capabilities of modern hardware.

By using dynamic rendering techniques, terrains up to 16385x16385 grid size can be used in real-time applications (Ying, 2016). There are other studies showing more detailed static terrains, having a grid size of 1024x1024 equivalent to 819x1638.5 meters of total surface, can be rendered (González, Pérez, & Orduña, 2016).

Paged or chunk-based streaming techniques can render virtual globes and maintain whole earth data (Porwal, 2013). Even when the data is not visualized in 3D, a 2D application can use a very large terrain and may operate with this large terrain data. If a LOS analysis needs to be done with this kind of terrain data, parallel approaches can dramatically reduce the calculation time by running as a background process. If a real-time calculation is a requirement, parallel approaches become more important.

A viewshed can be calculated with different algorithms and in this thesis three different algorithms will be analyzed both for CPU/GPU, and the results will be discussed. These algorithms are as follows:

- R3 (Franklin, Ray, & Mehta, 1994)
- R2 (Franklin et al., 1994)
- Van Kreveld’s Algorithm (Kreveld, 1996)

## 1.2 Objective of the Study and Contribution

In this thesis, three different LOS algorithms will be implemented for CPU and GPU and their output viewshed will be compared in order to find which method is faster and more reliable for problem resolution.

R2 and R3 are well-studied algorithms and there are several GPU implementations. These algorithms are very suitable for GPU parallelization, but there is no detailed comparison of parallel version of these algorithms to best of our knowledge. Van Kreveld’s algorithm is a sequential algorithm and, to best of our knowledge there is only a CPU parallelization attempt for it (Ferreira, Andrade, Magalhes, Franklin, & Pena, 2013). By its nature, it is not suitable for a complete GPU parallelization but parts of it can be calculated effectively with GPU. A new approach for parallelization of this algorithm will be explained, and the results will be compared with the base methods.

In this thesis, we evaluate the performance of the algorithms considering their speed-up and their memory use. The speed-up is an important criterion for performance evaluation of the LOS analysis, albeit it is not the only performance criterion for real-world applications. In real world applications, LOS is expected to run together with other operations and the memory cannot be exclusively reserved for LOS and needs to be shared. An algorithm using less memory than its counterparts is more useful especially when the memory size is limited.

To compare the reliability of the algorithms, R3 is used as a base algorithm because it is a non-approximate method for LOS analysis (Franklin et al., 1994).

Real-time usage of LOS analysis will also be discussed, especially for graphics applications. Update period of analysis (once per each frame or fixed time period) and maximum analysis area for optimum rendering are defined.

A paper has been written as an output of this thesis and submitted to USMOS 2017 Conference with name "Acceleration of Line of Sight Analysis Algorithms With Parallel Programming", abstract is accepted and full paper is under review[1].

## 1.3 Outline

Outline of the thesis document is as follows:

**Chapter 2** describes the previous work, the problem definition, the general algorithm overview, and the description of the technology used in the implementation.

**Chapter 3** explains the details of GPU implementation of R2 and R3 and suggests a way to implement Van Kreveld's Algorithm on GPU.

**Chapter 4** presents the results of the implementation, memory usage and output viewsheds, and discusses the limitations of each algorithm.

**Chapter 5** provides the concluding remarks and proposes ideas for future researches

**Appendix A** includes the data format of the utilized DEM.

**Appendix B** includes a guide for source code compilation and briefly explains the code structure.

**Appendix C** includes GPU Profiler results of the algorithms.

---

[1] Yılmaz, G., Sürer, E., Temizel, A. (2017). Acceleration of Line of Sight Analysis Algorithms with Parallel Programming. Submitted to USMOS 2017 Conference

# CHAPTER 2

# BACKGROUND AND PREVIOUS WORK

In this chapter, background information on viewshed calculation is provided and alternative computational platforms that can be used for calculation, more particularly CPU, multi-core CPU and GPU solutions, are discussed.

## 2.1 Terrain Structure

Terrain is represented as a 2-dimensional flat-surface. This surface has sampling points which are intersects with the terrain, and each sampling point has elevation data (elevation of the terrain at that exact point).

The terrain elevation is usually stored as a Triangulated Irregular Network (TIN) or a Digital Elevation Model (DEM) (Li, Zhu, & Gold, 2005). A TIN divides a terrain into planar triangles. To query elevation of a point *p*, triangle onto which point *p* is projected is found and then elevations of corner points of the triangle are interpolated. DEM is a simple matrix storage structure. Each grid is regularly spaced onto terrain, and each grid has an elevation. Spacing can be in fixed coordinate unit, distance or geographical unit (See Figure 2).

In this thesis, DEM is used since it simplifies the calculations and there are many publicly available DEM data. The sampling period of the DEM data used is in a Cartesian coordinate unit, not in a geographical or distance unit.

## 2.2 Line of Sight and Viewshed

The line of sight (LOS) is the procedure of marking points which are visible to the observer on a given DEM, along the sightline originating from the observer point with a direction and a radius.

We can express this problem in terms of angles, or slopes; a target point *T* is visible if and only if the vertical slope of the line from the observer position to target *T* is higher than other vertical slopes of the lines from the observer position to prior points along the LOS line (Blelloch, 1990). The problem can be extended to calculate not only visibility of a point, but also the minimum height required for a point *T* to be visible from an observer point. Franklin et al. (1994) used this result to compare different algorithms, but this is out of the scope of the this thesis where only binary visibility results are compared.

Figure 3 displays the representation of the LOS calculation. Each point represents a cell in DEM. The observer point can see $P_1$ as LOS line does not intersect with any prior points. The observer can see $P_2$ because $\alpha_2$ is greater than $\alpha_1$ but cannot see $P_3$ because $\alpha_3$ is lower than $\alpha_2$. Each angle is compared with the prior angles since the order of the points is important. Order is decided by following the line starting from observer to target point. In Figure 3, the line starting from observer to target point P3 passes over P1 first and after than P2 and finally P3, that gives the order of the points.

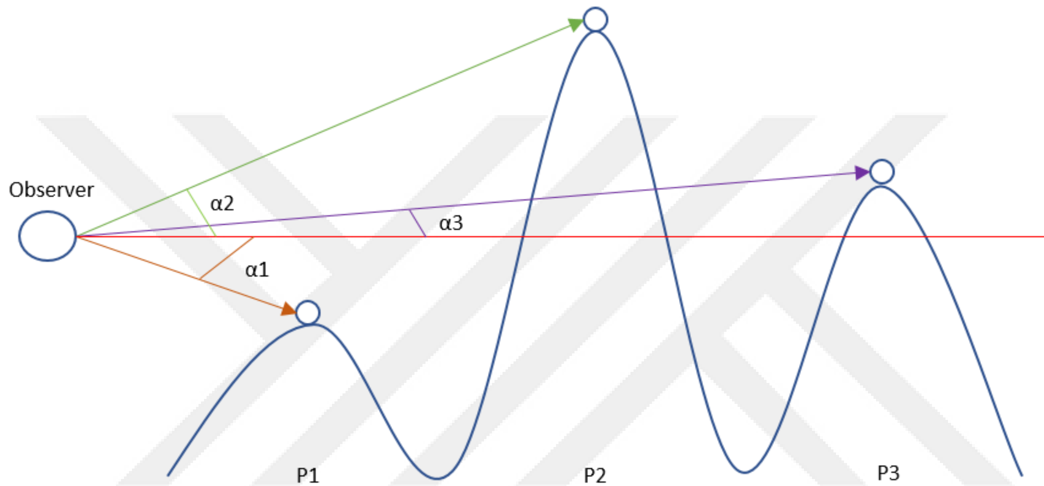



Figure 3: LOS vertical cross section

Since we are using grid based elevation data, the line should be rasterized. The rasterization process is done with Bresenham's Line Algorithm (Bresenham, 1965). Since the start and end points of the line are known, by checking differences of x and y coordinates of these two points, we can mark cells through which the line passes, starting from the start point. All the calculations after rasterization are done for grid cells (See Figure 4).

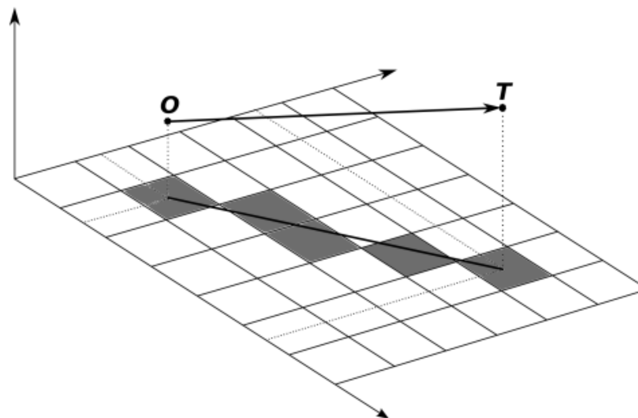



Figure 4: Line rasterization (Ferreira et al., 2013)

Cells which are the rasterized versions of line $O$ to $T$ are called $c_o, c_1, c_2, c_3 \ldots c_t$ and the slope of a line which connects $O$ to any $c_i$ is called $\alpha_i$ :

$$\alpha_i = \frac{\text{elev}(c_i) - (\text{elev}(c_0) + h_o)}{\text{dist}(c_0, c_i)}$$

where:

- $c_0$ is the observer cell
- $h_o$ is the height above the terrain of the observer
- $\text{elev}(c_i)$ is the elevation of cell $c_i$
- $\text{dist}(c_0, c_i)$ is the distance between the observer of the cell and cell $c_i$

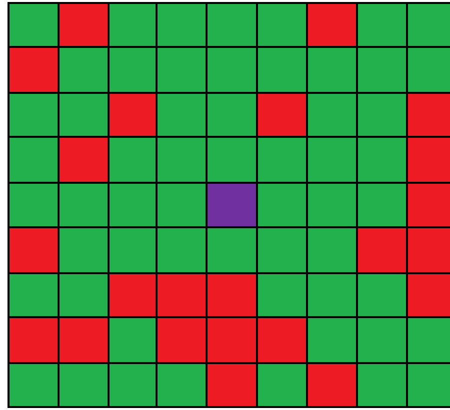


Figure 5: Top-down view of viewshed (Purple: Observer, Green: Visible, Red: Not Visible).

We can calculate the slope of a target as follows:

$$\alpha_t = \frac{\text{elev}(c_t) + h_t - (\text{elev}(c_0) + h_o)}{\text{dist}(c_0, c_t)}$$

$h_t$ is height above the terrain of the target cell. If $\alpha_t$ is greater than every $\alpha_i$, $i \in \{0, 1, .., t\}$, then point $T$ is visible from the observer point.

Viewshed is a map that shows which points are visible by the observer in a given radius. To calculate the viewshed, LOS needs to be calculated to mark every point in a given area Algorithms use different techniques to cover all of the cells (See Figure 5).

### 2.3 Viewshed Algorithms

Different algorithms can be used to process different terrain structures. A TIN model can be processed with algorithms studied by Cole & Sharir (1989) and De Floriani & Magillo (2003). Since this thesis use DEM format, the algorithms that uses DEM will be our focus. R3 is a well-known non-approximate method for the DEM format (Shapira, 1990). There are also approximate methods for DEM formats called R2 (or RFVS) (Franklin et al., 1994) and Van Kreveld's algorithm (Kreveld, 1996).

In R3 and Van Kreveld's algorithm a cell is visible only if the center of the cell is visible, while in R2 a cell could be visible if some part of the cell is visible to the observer. If an application needs high accuracy rather than efficiency, R3 can be used.

Each algorithm operates on an area of interest. This area is defined with an observer point(center of area) and radius(width and height are equal). Area of interest may cover the whole elevation data or only part of it.

#### 2.3.1 R3 Algorithm

The R3 algorithm is a straight forward algorithm which basically operates LOS calculation for every DEM cell in the area of interest (See Figure 6) . It has high accuracy since it is not an approximate method but runs in $\theta(n^3)$ for n by n raster DEM (Franklin et al., 1994). This algorithm provides high accuracy as it makes full use of the available elevation data (Izraelevitz, 2003).

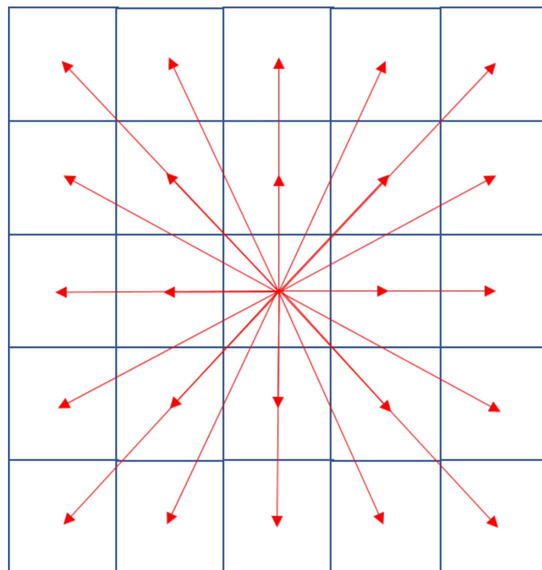


Figure 6: R3 calculation

The pseudo code for the algorithm is given in **Algorithm 1** (for LOS calculations see 2.2):

---

**Algorithm 1 : R3**

---

**Inputs**: $c_1, \dots, c_I$, $\boldsymbol{Elevation\ data\ cells\ in\ area\ of\ interest}$;
$c_O$, $\boldsymbol{Observer\ cell}$;
$h_O$, $\boldsymbol{Observer\ height\ above\ terrain}$;
**Output**: $v_1, \dots, v_I$, $\boldsymbol{Visibility\ flag\ of cells\ in\ area\ of\ interest}$
$\boldsymbol{for\ i = 1, \dots, I\ do}$
    $\boldsymbol{if\ c_i\ not\ equal\ to\ c_O}$
        $\boldsymbol{make\ a\ line\ from\ c_O\ to\ c_i}$
        $\boldsymbol{calculate\ the\ slope\ of\ the\ line\ \alpha_i = \frac{\mathrm{elev}(c_i) - (\mathrm{elev}(c_0) + h_o)}{\mathrm{dist}(c_0, c_i)}}$
        $\boldsymbol{rasterize\ the\ line\ and\ find\ cells\ on\ the\ line\ r_1, \dots, r_T\ (except\ c_O\ and\ c_i)}$
        $\boldsymbol{assign\ maxSlope\ as\ a\ very\ small\ value}$
        $\boldsymbol{for\ t = 1, \dots, T\ do}$
            $\boldsymbol{make\ a\ line\ from\ c_O\ to\ r_t}$
            $\boldsymbol{calculate\ the\ slope\ of\ the\ line\ \beta_t = \frac{\mathrm{elev}(r_t) - (\mathrm{elev}(c_0) + h_o)}{\mathrm{dist}(c_0, r_t)}}$
            $\boldsymbol{if\ maxSlope < \beta_t\ then\ maxSlope = \beta_t}$
        **end for**
        **if** $\boldsymbol{maxSlope < \alpha_i}$
            $\boldsymbol{mark\ cell\ as\ visible\ v_i = true}$
        $\boldsymbol{else}$
            $\boldsymbol{mark\ cell\ as\ not\ visible\ v_i = false}$
        **end if**
    $\boldsymbol{else}$
        $\boldsymbol{mark\ cell\ as\ visible\ v_i = true}$
    **end if**
**end for**

---

For elevation interpolation, different methods can be used. Since the elevation data is only for the center of the cell (or edges of the cell depending on DEM data), either the elevation data for the points that are not at the center of a cell can be calculated or only the center of the cell data can be used. In this thesis, the latter approach is used for every algorithm (no interpolation for elevation).

### 2.3.2 R2 Algorithm

The R2 algorithm is very similar to the R3 algorithm. The only difference is that R3 calculates LOS for every cell separately, while R2 calculates LOS for only the boundary cell of area of the interest, and lets the rays from the observer to the boundary cells fill the other cells (Franklin et al., 1994) (See Algorithm 2).

**Algorithm 2 : R2**

**Inputs**: $c_1, \dots, c_I$, $\boldsymbol{Elevation\ data\ cells\ of\ borders\ of\ area\ of\ interest}$;
$c_O$, $\boldsymbol{Observer\ cell}$;
$h_O$, $\boldsymbol{Observer\ height\ above\ terrain}$;
**Output**: $v_1, \dots, v_I$, $\boldsymbol{Visibility\ flag\ of cells\ in\ area\ of\ interest}$
$\boldsymbol{for\ i = 1, \dots, I\ do}$
    $\boldsymbol{if\ c_i\ not\ equal\ to\ c_O}$
        $\boldsymbol{make\ a\ line\ from\ c_O\ to\ c_i}$
        $\boldsymbol{rasterize\ the\ line\ and\ find\ cells\ on\ the\ line\ r_1, \dots, r_T\ (except\ c_O)}$
        $\boldsymbol{assign\ maxSlope\ as\ a\ very\ small\ value}$
        $\boldsymbol{for\ t = 1, \dots, T\ do}$
            $\boldsymbol{make\ a\ line\ from\ c_O\ to\ r_t}$
            $\boldsymbol{calculate\ the\ slope\ of\ the\ line\ \beta_t = \frac{elev(r_t) - (elev(c_O) + h_o)}{dist(c_O, r_t)}}$
            $\boldsymbol{if\ maxSlope < \beta_t}$
                $\boldsymbol{maxSlope = \beta_t}$
                $\boldsymbol{mark\ cell\ as\ visible\ v_{r_t} = true}$
            $\boldsymbol{else}$
                $\boldsymbol{mark\ cell\ as\ not\ visible\ v_{r_t} = false}$
            **end if**
        **end for**
    $\boldsymbol{else}$
        $\boldsymbol{mark\ cell\ as\ visible\ v_i = true}$
    **end if**
**end for**

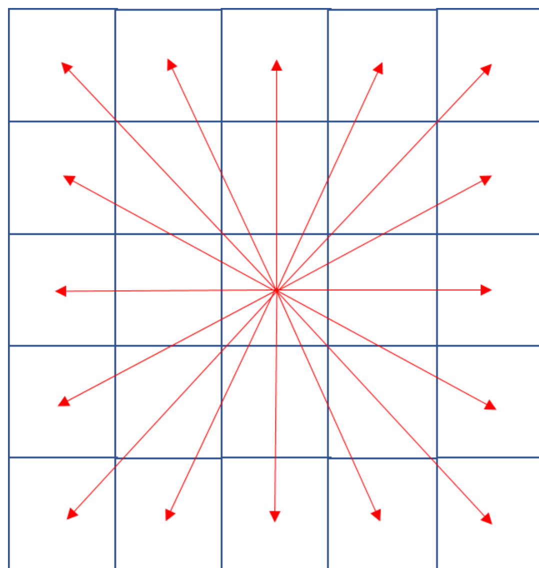


Figure 7: R2 calculation

This leads to another problem; several lines can be passed from the same cell and we need a decision mechanism to decide which line should update the visibility of that cell. Some solutions let every line write to the cell visibility since it will not change from line to line very much and the error is negligible. Franklin et al.(1994) suggest that only the line that passes closest to the center of the cell should write the visibility of that particular cell. In this thesis, we just let every line update the visibility of the cell, and the decision of the last line that updates the cell visibility is assigned as a final value.

The R2 algorithm has the complexity $\theta(n^2)$ and has relatively low approximation error (Franklin et al., 1994). This algorithm can be used if the application needs only an overview of the visibility area but not the exact result (See Figure 7).

### 2.3.3 Van Kreveld's Algorithm

Van Kreveld's Algorithm is a sweep line algorithm that runs in $\theta(n^2 log\ n)$ (Kreveld, 1996). It is also an approximate method, but has virtually the same accuracy as R3 and is faster. (Zhao, Padmanabhan, & Wang, 2013).

The algorithm basically rotates a sweep line over the area of interests and calculates the viewshed according to this. It defines three events for each cell (See Figure 8):

- Enter Event: When the line enters the cell

- Center Event: When the line passes over to the center of the cell

- Exit Event: When the line exists from the cell

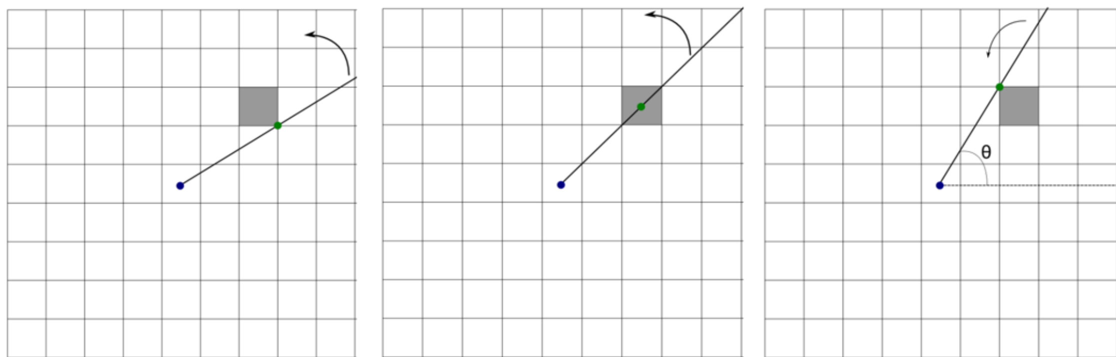



Figure 8: Event types (Enter, Center and Exit Events and Rotate Angle $\theta$)

It calculates the event list for each cell for three different events and sorts these events according to their rotate angle ($\theta$) from the lowest to the highest. Then it operates on the event list and keeps track of events with a balanced tree (agenda or active tree). As seen in Figure 9, the balanced tree sorts the cells according to their distances to the observer. The child to the right of the tree of the yellow cell is purple, which is more distant than

the yellow while the child to the left (red cell) is closer than the yellow cell. The same relationship applies to the blue and green cells.

Algorithm decides what to do with the event according to the event type while iterating over the event list.

If the event is a/an:

- Enter event: Calculate the slope of the cell (See 2.2) and insert it into the active tree
- Center event: Traverse active tree to find if there is a cell which is nearer than this cell with a higher slope angle. If there is, mark this cell as not visible and if not mark as visible
- Exit event: Remove the cell from the active tree

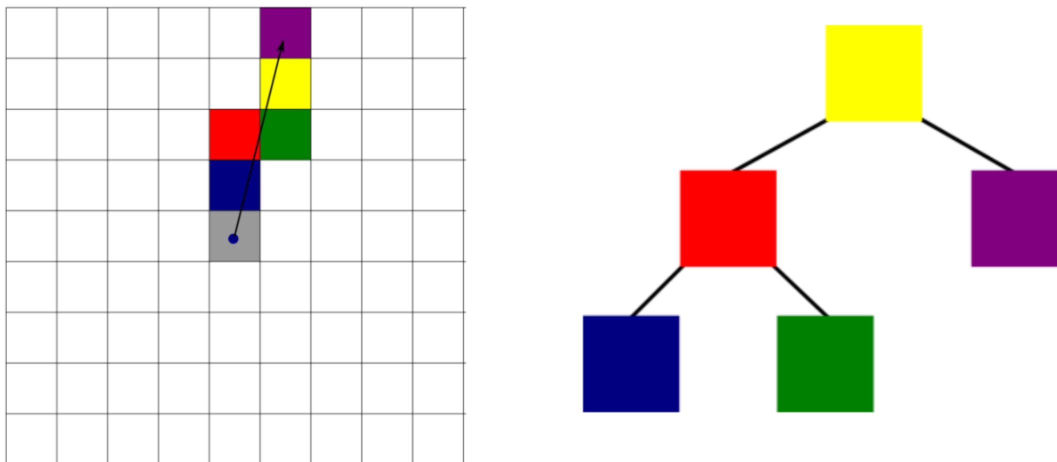


Figure 9: The active tree

To find the event (rotate) angle we need to calculate the enter and exit offsets of the cell. These offsets are calculated according to the slope of the line which connects the observer cell to the current cell. By looking at the slope, we can determine which quadrant contains the cell. We can make these calculations as follows:

$$\Delta x = c_i x - c_o x$$

$$\Delta y = c_i y - c_o y$$

Where:

- $c_i x$ : x coordinate of the current cell
- $c_o x$ : x coordinate of the observer cell
- $c_i y$ : y coordinate of the current cell
- $c_0 y$ : y coordinate of the observer cell

- If $\Delta x > 0$ and $\Delta y < 0$, it means that the cell is in the first quadrant:
    - $\text{offset}_{\text{enter}}(x, y) = (+0.5, +0.5)$
    - $\text{offset}_{\text{exit}}(x, y) = (-0.5, -0.5)$
- If $\Delta x < 0$ and $\Delta y < 0$, it means that the cell is in the second quadrant:
    - $\text{offset}_{\text{enter}}(x, y) = (+0.5, -0.5)$
    - $\text{offset}_{\text{exit}}(x, y) = (-0.5, +0.5)$
- If $\Delta x < 0$ and $\Delta y > 0$, it means that the cell is in the third quadrant:
    - $\text{offset}_{\text{enter}}(x, y) = (-0.5, -0.5)$
    - $\text{offset}_{\text{exit}}(x, y) = (+0.5, +0.5)$
- If $\Delta x > 0$ and $\Delta y > 0$, it means that the cell is in the fourth quadrant:
    - $\text{offset}_{\text{enter}}(x, y) = (-0.5, +0.5)$
    - $\text{offset}_{\text{exit}}(x, y) = (+0.5, -0.5)$
- If $\Delta x = 0$ and $\Delta y < 0$, it means that the cell is on the Y- axis:
    - $\text{offset}_{\text{enter}}(x, y) = (+0.5, +0.5)$
    - $\text{offset}_{\text{exit}}(x, y) = (-0.5, +0.5)$
- If $\Delta x < 0$ and $\Delta y = 0$, it means that the cell is on the X- axis:
    - $\text{offset}_{\text{enter}}(x, y) = (+0.5, -0.5)$
    - $\text{offset}_{\text{exit}}(x, y) = (+0.5, +0.5)$

- If Δx = 0 and Δy > 0, it means that cell is on the Y+ axis:
    - $\text{offset}_{\text{enter}}(x, y) = (-0.5, -0.5)$
    - $\text{offset}_{\text{exit}}(x, y) = (+0.5, -0.5)$
- If Δx > 0 and Δy = 0, it means that cell is on the X+ axis:
    - $\text{offset}_{\text{enter}}(x, y) = (-0.5, +0.5)$
    - $\text{offset}_{\text{exit}}(x, y) = (-0.5, -0.5)$

In the above calculations, the top is considered as Y-, while the bottom is considered as Y+, the left is considered as X- and the right is considered as X+ axis. The width and height of the cells are equal to 1. All the center event offsets are equal to 0 (See Figure 10).

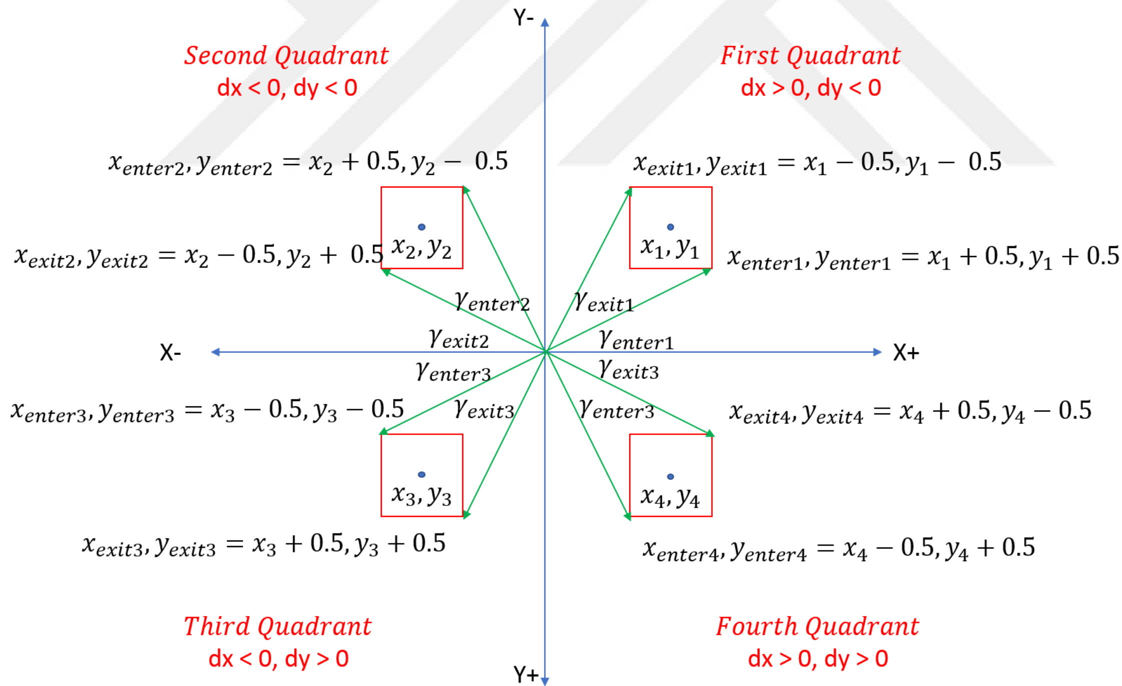



Figure 10: Event angles

After finding the correct offsets of the cells, we can calculate the event angle $\theta_{ei}$ as follows:

$$\theta_{ei} = \text{atan2}(c_o y - (c_i y + \text{offset}_{ei} y),\ c_o x - (c_i x + \text{offset}_{ei} x))$$

After the iteration over the event list is completed all the cells are marked as either visible or not visible (See **Algorithm 3**).

**Algorithm 3 : Van Kreveld's**

**Inputs**: $c_1, \dots, c_I$, $\boldsymbol{Elevation\ data\ cells\ of\ area\ of\ interest}$;
$c_O$, $\boldsymbol{Observer\ cell}$;
$h_O$, $\boldsymbol{Observer\ height\ above\ terrain}$;
**Output**: $v_1, \dots, v_I$, $\boldsymbol{Visibility\ flag\ of cells\ in\ area\ of\ interest}$
$\boldsymbol{for\ i = 1, \dots, I\ do}$
    $\boldsymbol{calculate\ three\ event\ offsets\ \ offset_{event}}$
    $\boldsymbol{calculate\ event\ angle}$:
    $\theta_{ei} = \boldsymbol{atan2}(c_o y - (c_i y + offset_{ei} y),\ c_o x - (c_i x + offset_{ei} x))$
    $\boldsymbol{add\ events\ to\ eventlist}$
**endfor**
$\boldsymbol{Order\ eventlist\ by\ event\ angle\ (lowest\ to\ highest}$
$\boldsymbol{for\ i = 1, \dots, size(evelist) do}$
    $\boldsymbol{if\ e_i\,.type\ equals\ enter}$
        $\boldsymbol{calculate\ slope}\ \ \beta_{e_i.c} = \frac{\mathbf{elev}(e_i.c) - (\mathbf{elev}(c_0) + h_o)}{\mathbf{dist}(c_0, e_i.c)}$
        $\boldsymbol{add\ \ e_i.c\ with\ slope\ \beta_{e_i.c}\ to\ the\ active\ tree}$
    $\boldsymbol{else\ if\ e_i\,.type\ equals\ center}$
        $\boldsymbol{traverse\ the\ active\ tree\ and\ find\ maximum\ elevation\ maxElev}$
        $\boldsymbol{if\ \beta_{e_i.c} >\ maxElev}$
            $\boldsymbol{mark\ cell\ as\ visible\ v_{e_i.c} = true}$
        $\boldsymbol{else}$
            $\boldsymbol{mark\ cell\ as\ not\ visible\ v_{e_i.c} = false}$
        **endif**
    $\boldsymbol{else\ if\ e_i\,.type\ equals\ exit}$
        $\boldsymbol{remove\ e_i.c\ from\ the\ active\ tree}$
    **end if**
**end for**

### 2.3.4 Parallel Algorithms for Viewshed Calculation

The popularity of parallel programing has increased over the years with the availability of frameworks such as CUDA and OpenCL. GIS applications also used these techniques to increase their performance. LOS and viewshed calculations are important parts of these applications, and several research studies have focused on implementing traditional sequential algorithms by means of parallel programming techniques.

Zhao et al. (2013) proposed a parallel implementation of R3 algorithm using GPU. They implemented R3 algorithm with a different spatial indexing mechanism, and compared the algorithms with each other and also with the CPU version of the algorithm. Osterman (2012) adapted R2 algorithm to GPU and compared the results with the CPU version. Chao et al. (2011) proposed a different technique to render 3D terrains with invisible areas which are rendered as shadows. As opposed to using an LOS based calculation, they

made use of the programmable graphic pipeline to achieve this goal. They passed elevation data to vertex and pixel shaders and calculated the visibility of each pixel. Cauchi-Saunders & Lewis (2015) used another non-LOS based parallelization technique which is using a wave front algorithm, XDraw (Franklin et al., 1994).

Axell & Fridén, (2015) implemented both R2 and R3 with GPU, and compared the results for GPU and CPU. They mainly focused on optimization for R2, and used R3 as a comparison algorithm.

Ferreira et al. (2013) implemented a parallel version of Van Kreveld's Algorithm on CPU. They used a version of algorithm adapted from Fishman, Haverkort, & Toma (2009). They basically divided the area of interest into eight sectors, and calculated the viewshed of each sector with a separate thread on CPU.

For rendering, programmable graphic pipeline can be used, for example while generating a shadow map, shadows are calculated just like viewshed. However, with GPGPU the output viewshed data can be used for further analyses as the output is written directly to the global memory. In addition, the size of the analysis area can be much bigger than the shader-based approaches because of the limitations of the render buffer size (Khronos Group, 2017a). With a GPGPU, based approach, a graphics pipeline is not needed; this eliminates the need for using graphics APIs such as OpenGL and specific data types such as vertices and pixels.

## 2.4 GPGPU

GPU is a special hardware for processing graphics related data. Due to the nature of graphics operations, calculations are suitable to be done in parallel, thus graphics hardware architectures are inherently developed to handle massively parallel instructions. OpenCL and CUDA are the two main frameworks to make use of the capabilities of GPU hardware other things than rendering operations. This is widely known as the general purpose programming on graphical processing units (GPGPU).

OpenCL is the open standard for parallel programming. It is supported by several vendors including NVIDIA, AMD and Intel. A code written in OpenCL can also be run on CPU (Khronos Group, 2017b). CUDA is a framework developed by NVIDIA and it is a proprietary framework specific to NVIDIA GPUs (NVIDIA Corporation, 2017b).

These two frameworks share similar programming paradigms and essentially provide similar functionalities. There is performance comparison research which reports that CUDA has better performance for calculations but data copy operation performance has no significant difference in a Monte Carlo simulation variation called Adiabatic Quantum Algorithm (AQUA) (Karimi, Dickson, & Hamze, 2010). Benchmarks on image and video processing also revealed that while CUDA provides better performance for these specific applications, data copy operations could become the determining factor in performance by causing a bottleneck regardless of the choice of the framework (Temizel et al., 2011).

On the other hand, the authors note the evolving nature of these frameworks and suggest up-to-date comparisons of the latest versions. Another research indicates that there is no significant difference between OpenCL and CUDA (Fang, Varbanescu, & Sips, 2011). They used the benchmark suites RODINIA (Che et al., 2009) and SHOC (Danalis et al., 2010). Performance is not the only reason to choose a framework. OpenCL could be chosen for portability reasons, since CUDA only works for NVDIA GPUs. CUDA has a more easy development and shipment process than OpenCL with the available tools and sources (Klöckner, 2017).

In this thesis, we used CUDA as the GPGPU framework. Thus, the details of the CUDA framework is explained in the following section. Reasons underlying the choice of CUDA rather than OpenCL are maturity of available tools, and mobility is not an aim of this research. We also used a library called Thrust which is built on CUDA in this thesis.

### 2.4.1 CUDA

GPUs have a number of processing units called Streaming Multiprocessor (SM), which contains several processing cores. CUDA is a single program multiple data programming framework designed utilize this massively parallel architecture (NVIDIA Corporation, 2009). A thread is a set of execution instructions which can be run on CUDA Core. Each SM has its own L1 cache and Shared Memory space and registers. Each SM can access global memory and L2 cache. Figure 11 displays a brief overview of CUDA Architecture.

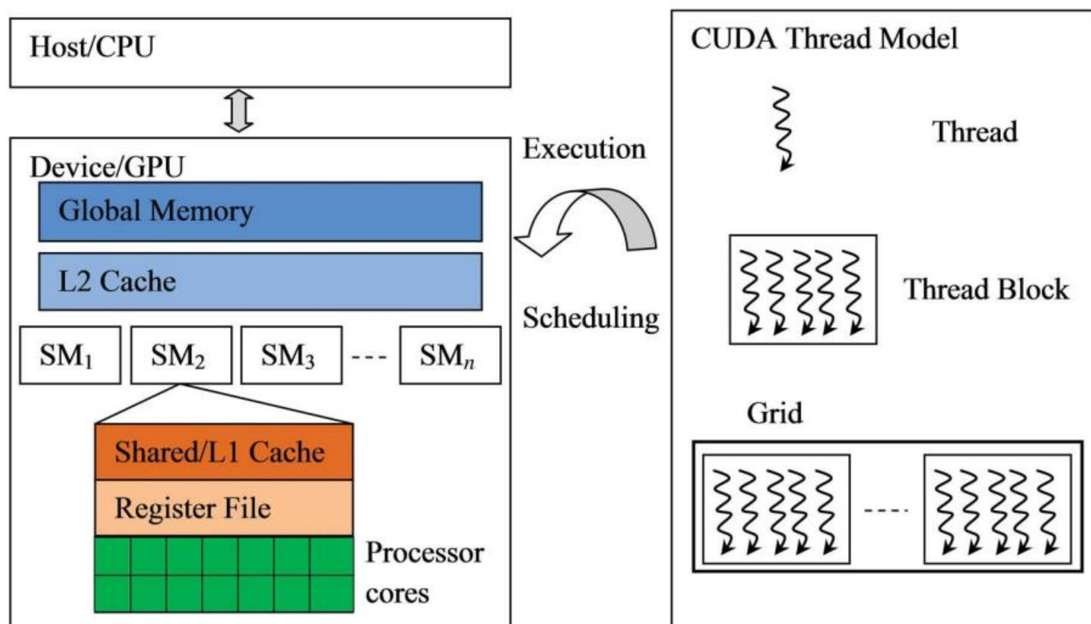



Figure 11: CUDA architecture (NVIDIA Corporation, 2017a)

L1 and L2 caches are hardware managed caches, L1 can be disabled with the API calls. They cache data when global memory is accessed, which means all the data accessed from global memory are cached to L2 and L1 (if enabled) then used by CUDA core. Registers are memory spaces for local variables of a thread. Each variable declared in the thread resides on registers. The global memory is the largest memory space that can be accessed from any thread in any SM, on the other hand, it is the slowest one (NVIDIA Corporation, 2009).

Parallel tasks are run on GPU in the form of code segments called "kernels". Many threads execute each kernel on different data in parallel. Threads are grouped into blocks and grids (See Figure 12). Grids and blocks are programming abstraction for thread grouping. Indeed, threads are assigned to SMs as blocks, and blocks are executed as thread groups called "warps". The size of the warp depends on the target architecture and the compute capability of the hardware (NVIDIA Corporation, 2017b).

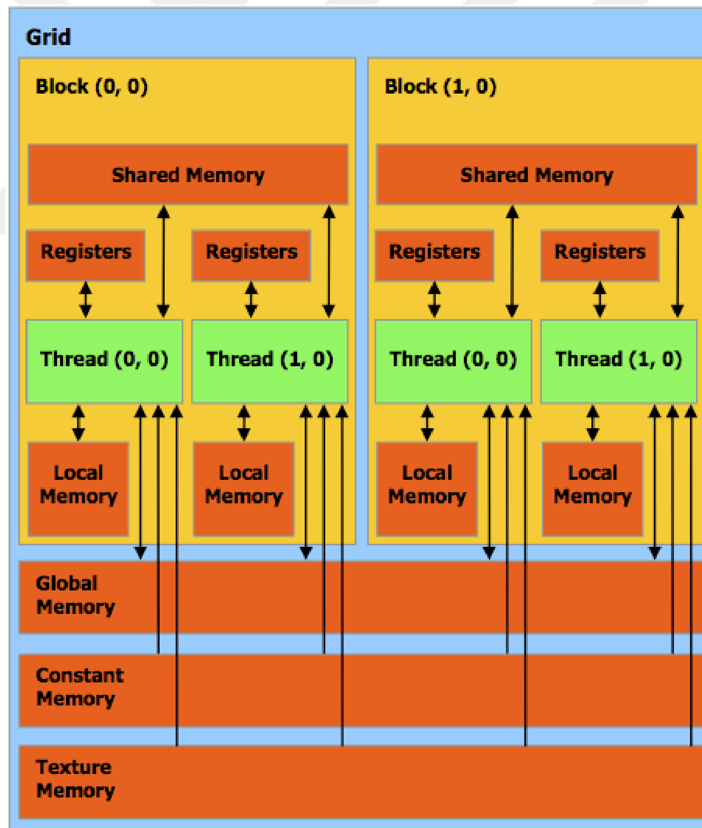



Figure 12: Thread grouping (NVIDIA Corporation, 2017a)

There are three types of memory which can be accessed from any thread (See Figure 12). Global memory can be used for reading and writing data and is slower than Constant and Texture Memory. Constant memory is a special place for constants, and it is read-only. Texture memory is like a constant memory, but provides hardware accelerated texture

fetching. If data are a set of values that need to be accessed with float values (like series of color), then texture memory can be used (NVIDIA Corporation, 2007).

Each thread has a unique id, which is calculated via block and thread indices, and has its own register space (Kirk, 2007). Indices could be in one, two or three dimensions to fit the needs of the problem. If two threads want to share data, they can use the shared memory as cache if they are on the same block, or could use the global memory, but sharing data with the global memory is slower than sharing the data on shared memory.

The general flow a kernel call is explained below:

- Copy data from the host (CPU) to the device (GPU)
- Call the kernel with the desired parameters
- After the kernel execution is completed, copy the data from the device to the host
- Use the data in the main program

To get high performance via GPGPU, threads that execute kernel should occupy the device as much as they can. To achieve this, programmers should carefully adjust their algorithms to operate on massive data with light-weight calculations in each thread and threads should execute without blocking each other. More independent threads result in higher occupancy and higher performance (NVIDIA Corporation, 2007). If the copy operation is time consuming, CPU implementation may yield higher performance than the GPU one even when the calculation is faster on GPU. Hence, data transfers need to be handled effectively and they need to be optimized and run asynchronously to the calculation whenever possible.

### 2.4.2 Thrust

Thrust is a C++ library for CUDA based on the Standard Template Library(STL) (NVIDIA Corporation, 2011). It uses STL like structures which are natively used in C++ programs, and provides powerful abstraction. Programmers could use Thrust, with a minimal programming effort, to benefit from CUDA in common problems, such as sorting, transformation, reductions, prefix-sums, reordering. It hides CUDA calls and operations, and can be used just like a normal C++ library. With this perspective any programmer that has no specific CUDA knowledge can benefit from GPGPU. Thrust is a highly-optimized library and generally yields a better performance than custom kernel calls. Thrust fill operation is reported to be 34 times faster than an un-optimized fill operation written in CUDA (Bell & Hoberock, 2012).

Thrust provides device abstraction, but it can also be used with other kernel calls, as input and output with device/host pointer transformations.

## CHAPTER 3

## GPU IMPLEMENTATIONS OF THE ALGORITHMS

In this chapter, GPU implementations of the algorithms are explained in detail. Thread organization, memory layout and copy operations are discussed.

### 3.1 GPU Implementation of R3

In R3 algorithm every LOS calculation can be done independently. For this reason, the algorithm is very suitable for parallel execution. Basically, each GPU thread is responsible for the LOS calculation of a single cell. This means we have to calculate the correct grid and block dimensions to have sufficient number of threads for each cell.

Elevation data are copied into the global memory of GPU as a one-dimensional array in row major order (See Figure 13).

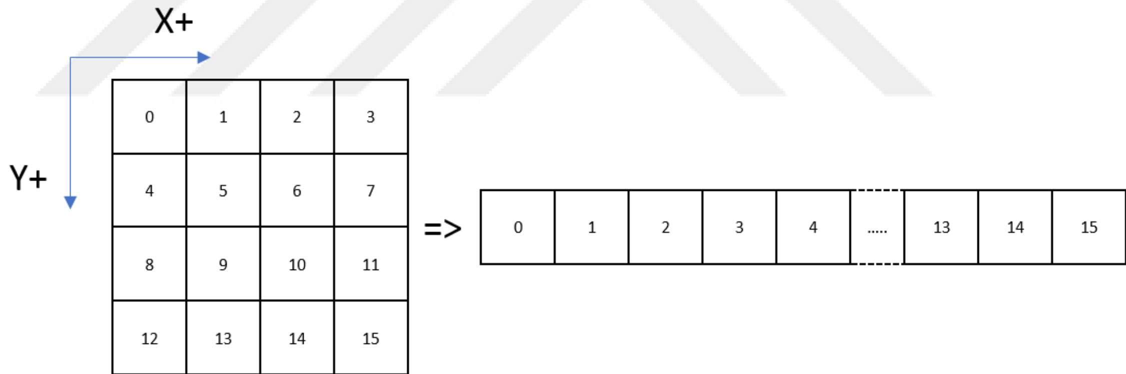



Figure 13: Elevation data on global memory

The observer index and the area of interest size can be changed; therefore, we may not use the whole elevation data for the problem (See Figure 14). The transformation of the area of interest (local) index to the global index (elevation data index) must be done for correct calculations. Below are the parameter definitions of a problem. The area of interests is a rectangular shape, width and height can be changed due to elevation data overflow. That is why initial width and height is called as *r*.

- $o_x$ , $o_y$: Global coordinates of the observer
- *r:* Radius of the area of interest
- $n_c$ , $n_r$: The size of elevation data (width/height)

With the given parameters, we can calculate the borders of the area of interests:

$$\min_x = \max(0, o_x - r)$$

$$\max_x = \max(n_c - 1, o_x + r)$$

$$\min_y = \min(0, o_y - r)$$

$$\max_y = \max(n_r - 1, o_y + r)$$

$$\text{width} = (\max_x - \min_x) + 1$$

$$\text{height} = (\max_y - \min_y) + 1$$

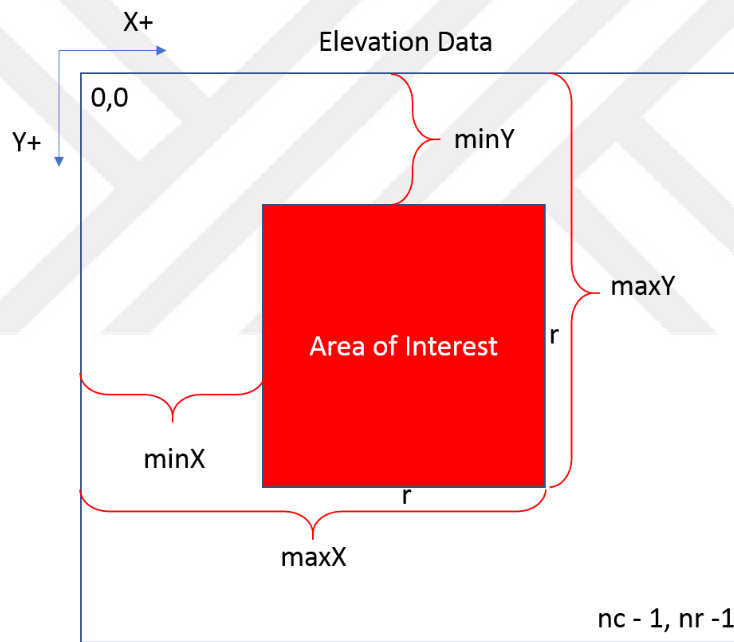



Figure 14: Area of interest in elevation data.

As mentioned above, in order to get the correct results, we need to calculate the grid and block dimensions. There are built-in parameters for thread indexing in CUDA:

- *threadIdx.x/y/z* : The local index of a thread in a block, can be 3D/2D/1D .
- *blockDim.x/y/z* : The size of a block, can be 3D/2D/1D
- *blockIdx.x/y/z* : The index of a block in a grid, can be 3D/2D/1D

Since the kernel will operate on 2D data, indexing will be done in 2D. Figure 15 illustrates cell to thread mapping. It should be noted that block dimension is fixed to 32x32.

Figure 15: R3 cell to thread mapping

Formulas to calculate indices are given below:

$$\text{gridDim}_x = \text{ceil}(\frac{\text{width}}{\text{blockDim.x}})$$

$$\text{gridDim}_y = \text{ceil}(\frac{\text{height}}{\text{blockDim.y}})$$

$$g_{Tx} = \text{blockIdx}_x \text{ x } \text{blockDim}_x + \text{threadIdx}_x$$

$$g_{Ty} = \text{blockIdx}_y \text{ x } \text{blockDim}_y + \text{threadIdx}_y$$

Where:

- $\text{gridDim}_x$ and $\text{gridDim}_y$ are the grid dimensions
- $g_{Tx}$ and $g_{Ty}$ are global thread indices

There could be more threads assigned to a kernel if the area of interest size is not divisible by 32. Each thread needs to compare its global indices with the width and height and if those values are higher than the area of interest size, they need to remain idle.

Since we know the global indices of threads, now we can calculate the global indices of cells, $g_x$ and $g_y$ and the elevation index $e_i$ :

$$g_x = g_{Tx} + min_x$$

$$g_y = g_{Ty} + min_y$$

$$e_i = g_x + g_y \text{ x } n_c$$

With the calculated values, threads can read their elevation data from global memory and make the LOS calculations for their respective cell. Output viewshed is also a one-dimensional array that has an equal size with elevation data and the same indexing mechanism. Threads can write output to viewshed array with the elevation index.

The slope angles of the cells could be calculated separately and these values could be inserted to a look up table. After this process, LOS slope angles of each cell could be gathered from that map. However, with this approach, extra global memory space for look up table needs to be allocated and global memory will be accessed for three time, for reading elevation data while calculating slopes, for writing calculated slope to the look up table, and reading slope while calculating LOS. For this reason, in each thread, slope angles are calculated after rasterization process. The calculated slope values are saved into registers. With this approach only one global access is required which is reading elevation data. In summary, following are the steps of the GPU implementation of R3 (See **Algorithm 1** for calculations):

- Allocate global memory on GPU for elevation data with the size of $n_c$ x $n_r$
- Copy the elevation data from CPU to GPU
- Allocate global memory on GPU for output viewshed with the size of $n_c$ x $n_r$
- Calculate the borders of the area of interest
- Call kernel with the parameters
- Calculate the global indices of cells and the elevation indices with each thread
- Make LOS calculations for each cell and write the output viewshed data with the calculated elevation index
- Copy the output viewshed data from GPU to CPU

## 3.2 GPU Implementation of R2

GPU implementation of R2 is very similar to R3 implementation. The only difference is that threads do not calculate LOS for each cell. Instead, they only calculate LOS for border cells of the area of interest. This leads different cell-to-thread mapping calculations for the algorithm.

The elevation data structure and the output viewshed data on GPU are the same with those in the R3 algorithm (See Figure 13). The calculation of borders and the width, height of the area of interest are also similar (See Figure 14). In the R3 algorithm, thread count needs to be equal to the total count of cells in the area of interest, however, now algorithm only needs threads for borders. To achieve this, threads need to be sorted along the borders of the area of interest, so one-dimensional grids are needed which entail one dimensional threads.

Since the maximum number of the threads that can be used in each block is 1024 in the target hardware, this value is chosen as the block dimension (NVIDIA Corporation, 2014). This maximum value can show variation for different hardware. Because of this, it should be queried and assigned as a block dimension. Figure 16 exhibits the block structure, local and global indices of threads.

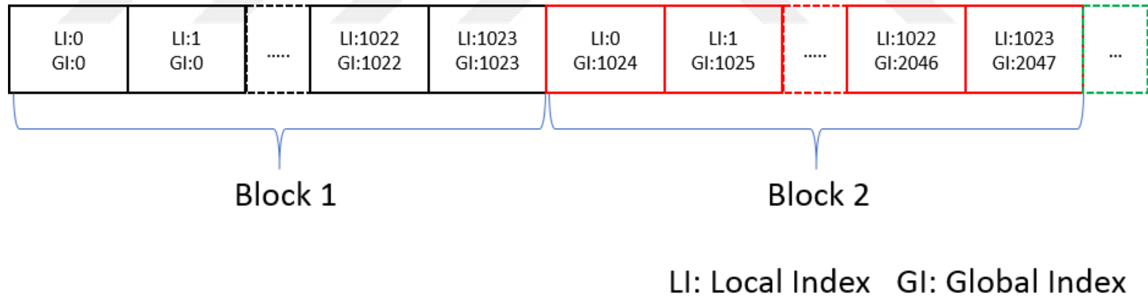



Figure 16: Local and global indices

Figure 17 displays the threads with global indices around the border of the example area of interest which has size of 7x5 cells.

To calculate the thread count we use the following formulas:

$$c_c = \left((\text{width} + \text{height}) \text{ x } 2\right) - 4$$

$$\text{gridDim} = \text{ceil}\left(\frac{c_c}{1024}\right)$$

Where:

- $c_c$ is the total cell count that we need to calculate LOS
- gridDim is the number of blocks we need, so we will call our kernel with a total thread of gridDim x 1024

If $c_c$ is not divisible by 1024 we will call our kernel with idle threads, those threads need to check their indices and they need to remain idle if it is higher than $c_c$.

Figure 17: R2 cell to thread mapping

Now we need to calculate the cell indices of each thread from their global indices. The global index of thread $g_{Ti}$ was calculated as follows (See Figure 16):

$$g_{Ti} = \text{blockIdx}_x \text{ x } \text{blockDim}_x \text{ x } \text{threadIdx}_x$$

By checking $g_{Ti}$ we can now determine the global index of cells, $g_x$ and $g_y$ (See Figure 17):

- If $g_{Ti} < \text{width}$ then this means thread is at the top side of border:

$$g_x = \text{min}_x + g_{Ti}$$

$$g_y = \text{min}_y$$

- If $g_{Ti} \geq \text{width}$ AND $g_{Ti} < \text{width} + \text{height} - 1$ then this means thread is at the right side of border:

$$g_x = \text{max}_x$$

$$g_y = \text{min}_y + (g_{Ti} + 1) - \text{width}$$

- If $g_{Ti} \geq \text{width} + \text{height} - 1$ AND $g_{Ti} < (2 \text{ x width}) + \text{height} - 2$ then this means thread is at the bottom side of border:

$$g_x = max_x - ((g_{Ti} + 1) - (\text{width} + \text{height} - 1))$$

$$gy = max_y$$

- If $g_{Ti} \geq (\text{width x } 2) + \text{height} - 2$ then this means thread is at the left side of border:

$$g_x = min_x$$

$$g_y = max_y - ((g_{Ti} + 1) - ((\text{width x } 2) + \text{height} - 2));$$

Since we know the global index of the cell in the elevation data we can read the elevation data of the cell with the elevation index $e_i$ and calculate LOS and write the output to the viewshed data:

$$e_i = g_x + g_x \text{ x } n_c$$

Each thread will calculate the LOS for the border cells separately with the calculated indices, and will write the viewshed results of all the cells intersected with their LOS line. The last value of the thread that writes the output will be used as the final result.

Following is the summary of the steps of the GPU implementation of R2 (See **Algorithm 2** for calculations):

- Allocate a global memory on GPU for the elevation data with the size of $n_c$ x $n_r$
- Copy the elevation data from CPU to GPU
- Allocate a global memory on GPU for the output viewshed with the size of $n_c$ x $n_r$
- Calculate the borders of the area of interest
- Call kernel with the parameters
- Calculate the global indices of the cells and the elevation indices with each thread
- Make LOS calculations for each border cell and write the output viewshed data with the calculated elevation index and write the output for each intersected cell as well
- Copy the output viewshed data from GPU to CPU

## 3.3 GPU Implementation of Van Kreveld's Algorithm

R3 and R2 algorithms are more suitable for parallel execution because they have independent tasks (LOS calculations for a cell), and their outputs do not need a merge operation. Each thread executes and writes without sharing data or blocking each other. On the other hand, Van Kreveld's Algorithm heavily depends on serial execution, but we can still gain benefit from parallel execution by dividing it into sub-steps.

Van Kreveld's Algorithm has three main steps (See Section 2.3.3 for details):

1. Calculating event angles for three event types for each cell
2. Sorting events according to their event angles from lower to higher
3. Iterating over the event list and calculating visibility for each cell

Step 1 has independent calculations for each cell. We need to calculate event angles for three event types, which means we can assign this task for one thread for one cell. Mapping of threads to cells can be done with the same method used in R3 algorithm (See Figure 15 for a visual representation and Section 3.1 for index calculations). Grid and block dimension calculations will be the same as well.

Step 1 does not need elevation data for event calculations, but it requires a memory block for the writing of the event calculations. We need the following fields and field types for each event in our event list:

- Event type (Enter, Exit or Center) (char – 1 Byte)
- Distance to the observer (float – 4 Byte)
- Event angle (float – 4 Byte)
- Index of cell, x y (unsigned short int – 2 Byte) (For x and y total 4 Byte)

This means that the event structure needs a total of 16 bytes of memory with padding. For each cell, we need three events. Therefore each cell needs a total 48 bytes of memory space. For an area of interest with size *width* x *height* the total event size $e_s$ is calculated as follows:

$$e_s = \text{width x height x 3}$$

This means that the algorithm needs approximately 1 GB of memory space for an area of interest with size 5000x5000 for Step 1 calculations, which is relatively high compared to other algorithms.

The event list is allocated as a one dimensional array on the global memory, similar to the elevation data (See Figure 13) Thread-to-cell mapping and global index calculations are the same with those in R3 algorithm. Threads can access their event data just like they reach their global elevation indices $e_i$ (See Section 3.1 for ei calculations), but since each cell has three event types and event list works with local cell index (index of cell in area of interest), the calculation of the event index $e_{vi}$ differs from the elevation index $e_i$

$$e_{vi} = (g_{Tx} + g_{Ty} \text{ x width}) \text{ x } 3$$

Figure 18 exhibits the event list on global memory, the grid shows the local cell indices $g_{Tx}$and $g_{Ty}$ , the bottom array shows event types, the cell and event indices.

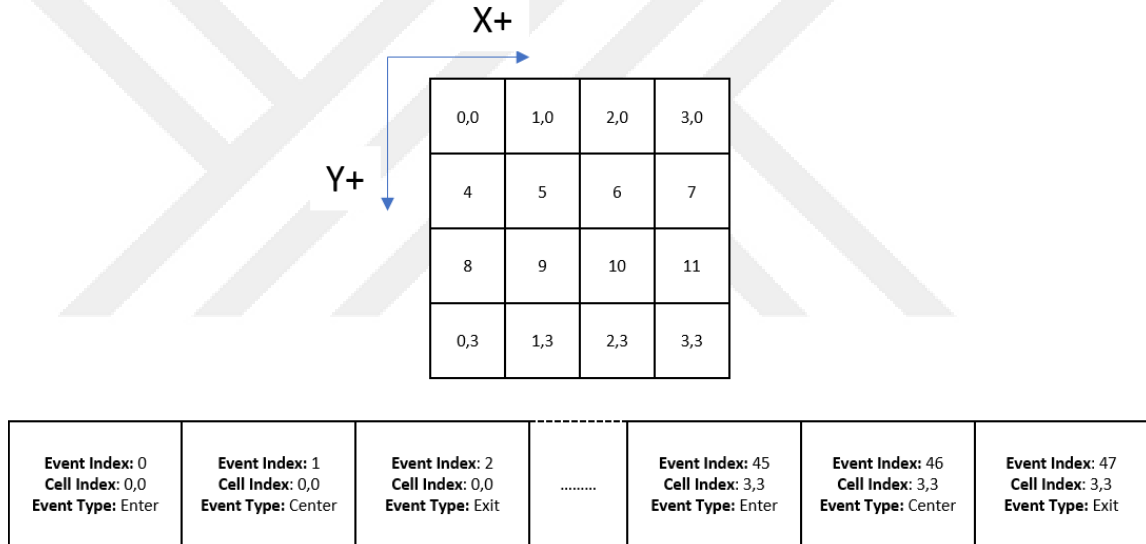



Figure 18: Event list in global memory.

Each thread needs to add an offset to write their respective event type into memory:

- $\text{EnterEvent } = e_{vi}$
- $\text{CenterEvent } = e_{vi} + 1$
- $\text{ExitEvent } = e_{vi} + 2$

All the parameters are then known for the calculation and writing events. Each thread calculates the needed values and writes them into the global memory (See 2.3.3 for event calculations)

After Step 1 we need to sort the events according to their event angles. Since the event data is not copied back to the CPU yet, Thrust can be used to sort events, with the usage of device pointer wrapper.

Step 3 is the sequential part of the algorithm, because this part iterates over the event list, and all the calculations depend on the calculations made previously. An alternative approach to parallelize this part would be to use the Thrust device vector to hold the active list. While iterating over the event list, events are inserted, searched and removed with Thrust calls. However, every step in iteration over the event list, makes this call highly time consuming and inefficient because at each iteration, the event list should be updated according to the event type, which needs a CPU-GPU memory transfer, and at each iteration only one event is searched, removed or added. The second approach entailed using methods like that used by Ferreira et al., 2013 and dividing the area into sectors. However, in that case, the event list needs to be calculated separately for each sector, and iterating over these event lists on GPU is not effective; indeed, it could be even slower, because of the need for a dynamic allocation while inserting events to the active tree. Moreover, calculations are too complex for lightweight GPU threads, because each thread will be iterated sequentially over its own event list. CPU parallelism is better for this algorithm, because iteration over eight divisions of a sector is more suitable for CPU threads; to gain advantage over CPU, dozens of GPU threads need to be executed at the same time. As a result, this part of algorithm remained the same with the original algorithm.

Below is a summary of the steps of the GPU implementation of Van Kreveld's Algorithm (See **Algorithm 3** for calculations):

- Allocate a global memory on GPU for the event list data with the size of $e_s$
- Calculate borders of the area of interest
- Call kernel with the parameters
- Sort the event list with Thrust
- Copy the event list data from GPU to CPU
- Iterate over the event list and calculate the viewshed on CPU

# CHAPTER 4

# RESULTS AND DISCUSSION

In this chapter, the results of the algorithms are presented and discussed, the algorithms are compared to each other, based on their execution time and memory usage.

## 4.1 Specifications of the Hardware and Software

**CPU:** Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz 8 Core

**GPU:** NVIDIA Tesla K40c Compute Capability 3.5

**RAM:** 32 GB

**OS:** Windows 7

**IDE:** Visual Studio 2013 / C++

**CUDA Version:** v7.5

**Architecture:** x64

***All CPU tests run with single-thread**

## 4.2 Results And Disccusion

Each algorithm was tested with CPU and GPU, and execution time results were calculated as average of 10 iterations. There are 5 different elevation data with size of 5000x5000, 10000x10000, 15000x15000, 20000x 20000, 25000 x 25000.

Because of the high memory usage of Van Kreveld's Algorithm, tests with large elevation data cannot be run; the upper limit of the algorithm is 10000x10000 elevation data with and area of interest radius of 3127.

The output viewshed is saved as a bitmap file that has an equal size with the elevation data. The visible pixels are painted green, not visible pixels are painted red. Outputs are compared with the comparison of bitmap files pixel by pixel.

All execution times were measured with the QueryPerformanceCounter function provided by Windows API. It is the highest precision timer of the Windows platform which has high resolution (<1 microsecond) (Microsoft, 2010). In timing tables, CPU and GPU

timings are given in milliseconds, and the Speed Up column shows the performance comparisons of CPU and GPU implementations of the respective algorithm.

Timing results were discussed according to usage of the algorithm in different applications. There are two kind of applications defined in the discussions:

- Real-time applications: Applications that use analysis results to provide on-the-fly information, such as games, simulations and moving map applications. The analysis time results should be as low as possible to achieve the desired frame rates, because the observer point is not fixed, and analysis results should be updated for every couple of frames. These update period and target frames rate depend on the needs of application.

- Offline applications: Applications that use analysis results to provide an information about a fixed-point observer. While it is desirable to get the results as fast as possible, the user of the application could wait for a couple of minutes to get the analysis results. GIS applications are examples of offline applications. The result of the algorithm could be used for a placement of an object.

### 4.2.1 Time Results

Table 1 presents the results of R3. As expected R3 gains significant benefit from GPU implementation; speed up reaches up to 168x. The speed up value increases when the output grows because GPU occupancy can still not reach its limits.

Table 1 : Results of R3 algorithm

| Area Size | CPU Execution Time (ms) | GPU Execution Time (ms) | Speed Up |
|---|---|---|---|
| 5000x5000 | 254128 | 2433 | 104 |
| 10000x10000 | 3034460 | 22111 | 137 |
| 15000x15000 | 10619550 | 75319 | 141 |
| 20000x20000 | 27337700 | 178960 | 153 |
| 25000x25000 | 59090500 | 350539 | 168 |

The timings show us that algorithm can be used in real time for area of 5000x5000; it lasts about 2.4 seconds, but the viewshed should not be updated for each frame. For offline calculations GPU implementation greatly reduces the calculation time. At an area of 25000x25000, the CPU version lasts for 16 hours, which is unacceptable for most applications. On the other hand, the GPU version lasts for only 6 minutes. Even if the

algorithm is not usable for real time usage, the offered algorithm can be used for offline calculations with workable time using the GPU version.

For an area size of 20000 x 20000, Zhao et al. (2013) gained a speed up value of 95 with GPU implementation of R3 compared to CPU implementation of R2. Their CPU version lasts 16917 seconds, while our implementation lasts 27337 seconds, and their GPU version lasts 203 seconds, while ours lasts 178 seconds. In addition to being a different implementation, the execution times also differ due to use of different test hardware. In addition, they use real geographical elevation data for their algorithms.

Table 2 : Results of R2 algorithm

| Area Size | CPU Execution Time (ms) | GPU Execution Time (ms) | Speed Up |
| --- | --- | --- | --- |
| 5000x5000 | 696 | 192 | 3.6 |
| 10000x10000 | 3429 | 398 | 8.5 |
| 15000x15000 | 8145 | 923 | 8.8 |
| 20000x20000 | 15832 | 1677 | 9.4 |
| 25000x25000 | 27333 | 2620 | 10.5 |

Table 2 presents the results of R2 which shows that the speed up reaches up to 10.5x. R2 speed up rates are significantly less than R3 because it does not make use of GPU threads as much as R3 does. On the other hand, its CPU and GPU execution times are significantly less than their R3 counterparts. Due to lower execution times, R2 algorithm can be used in real-time applications, and it can be updated per couple of frames without affecting application performance for areas 5000x5000 (which lasts 192 milliseconds) and 10000x10000 (which lasts 399 milliseconds). It is also reducing the analysis wait time for offline applications for area of 25000x25000; CPU version lasts about 27 seconds while GPU version lasts 2.6 seconds.

Figure 19 exhibits the speed up of GPU R2 compared to GPU R3. It gains a speed up to 153x.

For an area size 16001 x 16001 Axell & Fridén (2015) gained speed up value 29.4x with GPU implementation of their variant of R2 algorithm compared to CPU implementation of R2. They highly focused and optimized R2 algorithm, while our implementation only distributes LOS calculations to threads, which is why our speed up value is 10.5x for area size 25000x25000.

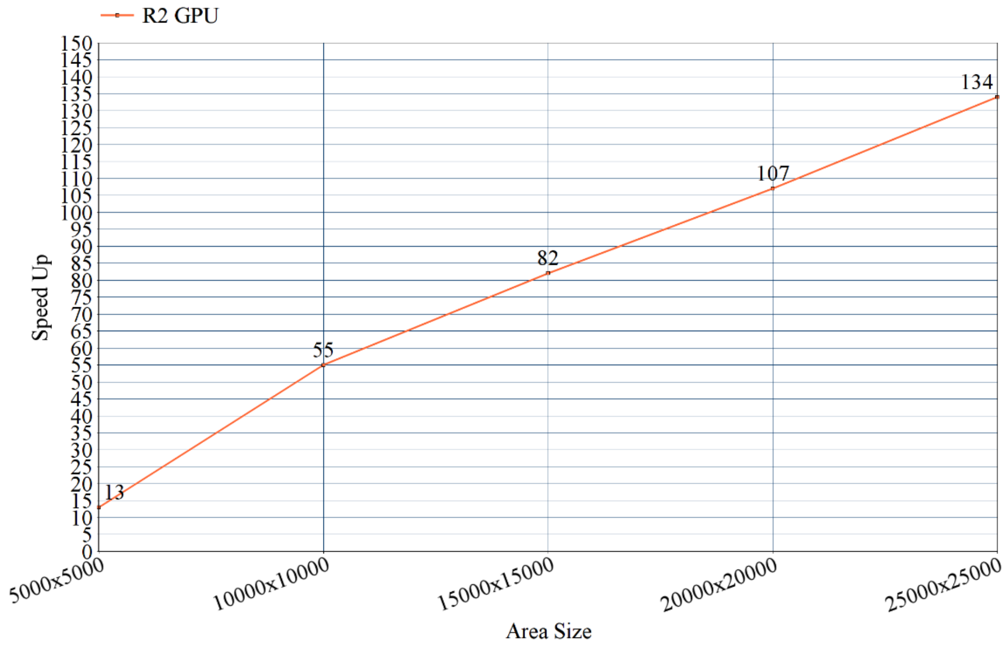



Figure 19: R2 GPU speed up compared to R3 GPU

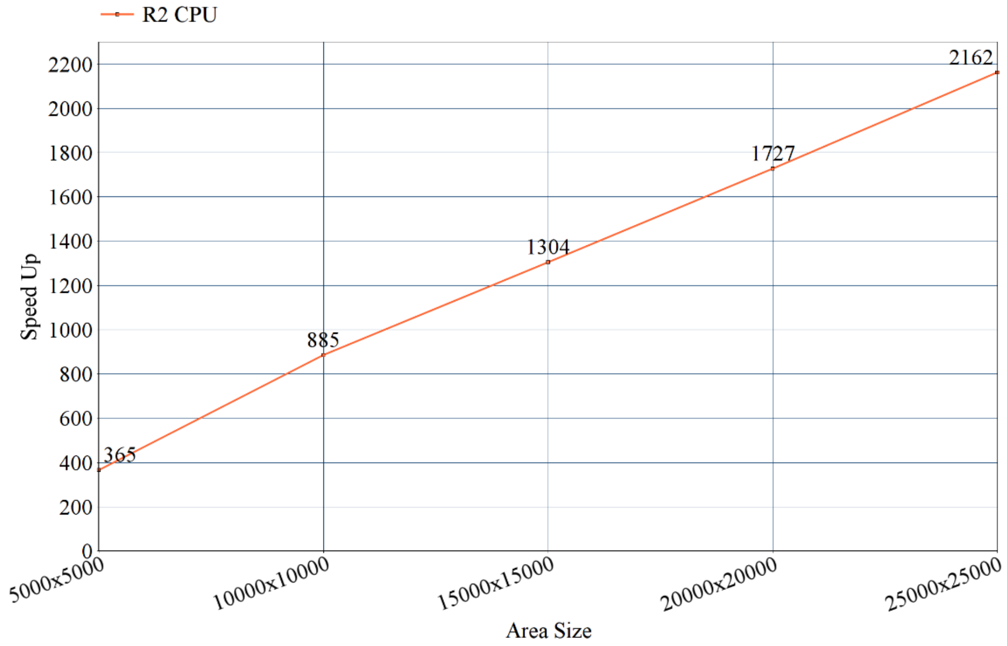



Figure 20: R2 CPU speed up compared to R3 CPU

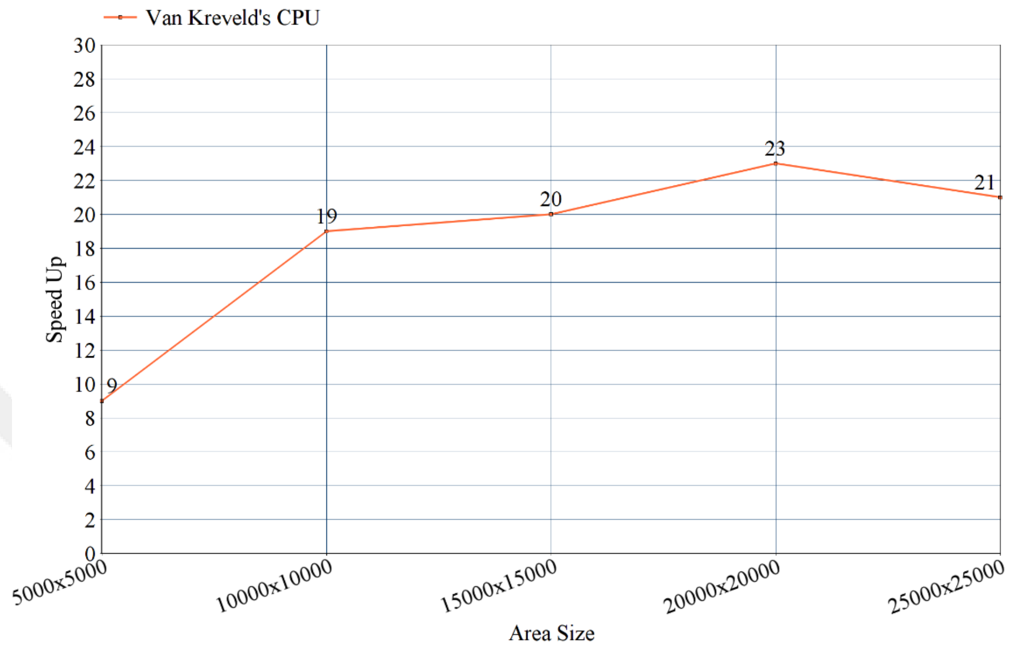



Figure 21: Van Kreveld's CPU speed up compared to R3 CPU

Table 3 : Results of Van Kreveld's algorithm

| Area Size | CPU Execution Time (ms) | GPU Execution Time (ms) | Speed Up |
|---|---|---|---|
| 5000x5000 | 29306 | 20208 | 1.5 |
| 6254x6254 | 59275 | 43600 | 1.4 |
| 10000x10000 | 160899 | N/A | N/A |
| 15000x15000 | 551290 | N/A | N/A |
| 20000x20000 | 1169660 | N/A | N/A |
| 25000x25000 | 2841270 | N/A | N/A |

Table 3 shows the results of Van Kreveld's Algorithm. As expected, the CPU version of the algorithm performance is between R3 and R2 algorithms (See Figure 20). However, the GPU version does not get benefit from GPU implementation very much and the speed up is only 1.5x. This proves that event calculations and sorting are not the time-consuming part of the algorithm but the event list iteration is.

Ferreira et al. (2013) gained 12x speed up with their parallel CPU Van Kreveld's algorithm compared to the original algorithm. This shows that Van Kreveld's Algorithm is much more suitable to CPU parallelism than GPU.

Appendix C shows the profiler results of algorithms R3 and R2. R2 threads have more distributed workload while R3 threads have more unorganized workload. This is expected because at R2 each thread calculates nearly same LOS, while at R3 threads that map to closer cells to observer have lower calculations. On the other hand, R3 has a lot more warps and threads to cover whole area that is shown while R3 uses more calculation power and its overall occupancy for each thread is lower. The speed up increase of R3 is higher than R2 because of thread count that R3 uses. Instruction per warp and instruction per clock shows R2 and R3 work distribution. R2: issued/executed ratio is high- meaning serialization, issue stall reasons are mostly "memory dependency" and "constant miss". Further optimizations of this algorithm could first focus on memory optimizations.R3: Issue stall is mostly due to "pipe busy". This algorithm is mostly arithmetically bound and further optimizations could first focus on instruction level parallelism and using higher throughput arithmetic operations.

### 4.2.2 Memory Usage

All the memory sizes in the tables are in megabytes. The CPU Memory column shows the total memory needed by the algorithm on CPU. The GPU Memory Column shows the total memory needed by the algorithm on GPU.

Table 4 : Memory usage of R3 algorithm

| Area Size | Elevation Data (MB) | Viewshed Data (MB) | CPU Memory Usage (MB) | GPU Memory Usage (MB) |
|---|---|---|---|---|
| 5000x5000 | 95 | 23 | 118 | 118 |
| 10000x10000 | 381 | 95 | 476 | 476 |
| 15000x15000 | 858 | 214 | 1072 | 1072 |
| 20000x20000 | 1525 | 381 | 1906 | 1906 |
| 25000x25000 | 2384 | 596 | 2980 | 2980 |

Table 4 shows the memory usage of the R3 algorithm; its memory usage depends on the size of the elevation data. It increases exponentially. Applications must be compiled as 64bit for bigger area sizes like 2000x2000 and 2500x2500 because the usage of memory exceeds 32bit limits (2 GB). Most of the modern computer setups has more than 4 GB RAM, and the CPU memory usage of the algorithm is enough for these setups. For 5000x5000 and 10000x10000 algorithm can be run on a normal CUDA enabled GPU but

for bigger area sizes it needs a separate CUDA enabled hardware that is not used for graphical output.

Table 5 : Memory usage of R2 algorithm.

| Area Size | Elevation Data (MB) | Viewshed Data (MB) | CPU Memory Usage (MB) | GPU Memory Usage (MB) |
|---|---|---|---|---|
| 5000x5000 | 95 | 23 | 118 | 118 |
| 10000x10000 | 381 | 95 | 476 | 476 |
| 15000x15000 | 858 | 214 | 1072 | 1072 |
| 20000x20000 | 1525 | 381 | 1906 | 1906 |
| 25000x25000 | 2384 | 596 | 2980 | 2980 |

Table 5 shows the memory usage of the R2 algorithm. It has a similar character with R3 algorithm. This is expected since two algorithms has different calculation methods for the same data. They do not need any different temporary memory space. They operate on elevation data and write the output to the allocated viewshed data.

Table 6 : Memory usage of Van Kreveld's algorithm.

| Area Size | Elevation Data (MB) | Viewshed Data (MB) | Event List Data (MB) | CPU Memory Usage (MB) | GPU Memory Usage (MB) |
|---|---|---|---|---|---|
| 5000x5000 | 95 | 23 | 1011 | 118 | 118 |
| 6254x6254 | 381 | 95 | 1791 | 2267 | 1791 |
| 10000x10000 | 381 | 95 | 4487 | 4963 | 4487 |
| 15000x15000 | 858 | 214 | 10164 | 11236 | 10164 |
| 20000x20000 | 1525 | 381 | 18129 | 20035 | 18129 |
| 25000x25000 | 2384 | 596 | 28384 | 31364 | 28384 |

Table 6 presents the memory usage of Van Kreveld's Algorithm. As previously mentioned, this algorithm cannot run with an area larger than 6254x6254 in GPU because of the memory needed by the event list data. Furthermore, Thrust sort algorithm uses merge sort for the non-primitive type, which means, that it needs a temporary storage for sorting (Bell & Hoberock, 2012). The implication of this is that the algorithm cannot run

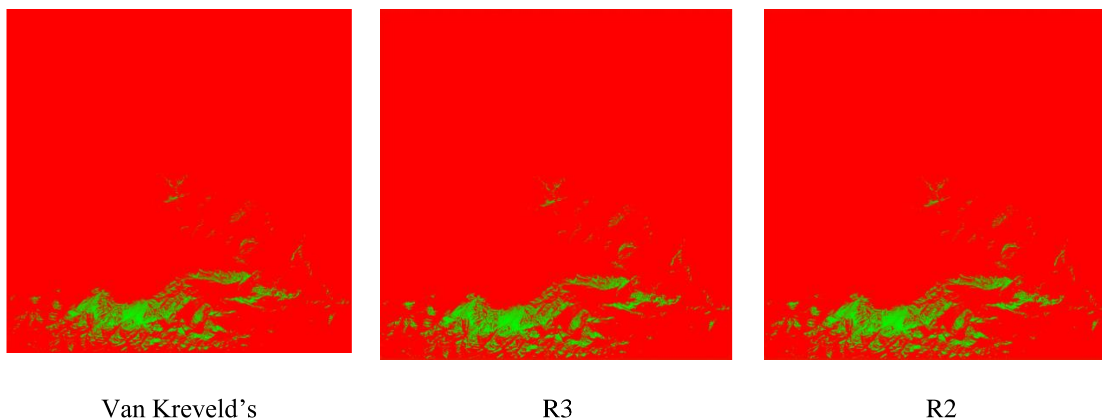
even if the allocated memory does not overflow; it cannot sort larger event list data. An area of 6254x6254 uses 10000x10000 elevation data, but since the implemented GPU version does not use elevation data on GPU, it is not a waste of memory on GPU.

Van Kreveld's Algorithm uses much more memory compared to the other algorithms, and only an area size up to 6254x6254 can be run on our target GPU. There is sufficient memory for greater area size, but as mentioned Thrust needs temporary memory for sorting. If we change the sorting method to some in-place sorting algorithm the overall speed up will be lower because Thrust uses a highly-optimized sorting technique. Even with this technique, the GPU benefit is not worth the effort. Changing the sorting mechanism for lower memory usage will make it worse. This algorithm needs a separate CUDA enabled hardware that is not used for graphical output for area sizes bigger than 5000x5000. The CPU memory usage is also very high; the target hardware needs to have more than 32GB RAM for area size 25000x25000. Hardware cost increases for this algorithm compared to the other algorithms, and performance gain is not satisfying enough to choose this algorithm.

#### 4.2.3 Viewshed Comparison

Figure 22 shows the outputs of three algorithms for area size 10000x10000. Visible cells are painted green and not visible cells are painted red.

Each GPU implementations produced the same viewshed output with their CPU counterpart. In Table 7 and Table 8 outputs of R2 and Van Kreveld's Algorithm are compared with that of R3. R3 is used as the base algorithm since R3 is the most accurate method within these three algorithms and we do not have ground truth about visibility results of these height maps. These are generated data, even if these were real elevation data of a terrain, it is a very time-consuming process to mark each DEM point visibility by checking the real terrain.



Van Kreveld's R3 R2

Figure 22: Sample viewshed outputs of different algorithms (10000x10000)

Table 7 : Different pixels of R2

| Area Size | Different Pixel Count | Different Pixel Ratio | False Negative Pixel Count | False Positive Pixel Count |
|---|---|---|---|---|
| 5000x5000 | 27760 | %0.11 | 15566 | 12194 |
| 10000x10000 | 221477 | %0.22 | 121924 | 99543 |

Table 8 : Different pixels of Van Kreveld's algorithm

| Area Size | Different Pixel Count | Different Pixel Ratio | False Negative Pixel Count | False Positive Pixel Count |
|---|---|---|---|---|
| 5000x5000 | 56559 | %0.22 | 54701 | 1858 |
| 10000x10000 | 343942 | %0.34 | 334571 | 9371 |

Table 7 and Table 8 show the different pixel count and ratio of R2 and Van Kreveld's Algorithm compared with R3 output. Results of R2 are more similar to R3 than Van Kreveld's Algorithm. The reason is that our R2 implementation uses the same mechanics with R3. On the other hand, Van Kreveld's Algorithm includes a tree, an event list mechanism. False negative means that algorithm marks cell as not visible while it is visible, and false positive means that algorithm marks cell as visible while it is not visible. R2 false negative-positive values are closer to each other while Van Kreveld's Algorithm is more inclined to mark cell as not visible. Both algorithms mark more cells as not visible than R3.

# CHAPTER 5

# CONCLUSIONS AND FUTURE WORK

## 5.1 Conclusions

We have parallelized and implemented three different algorithms on GPU: R3, R2 and Van Kreveld's Algorithm. R3 is the most time-consuming algorithm, while Van Kreveld's is the second and finally R2 is the least time-consuming, the fastest algorithm. The results show that R3 benefits from the GPU implementation more than the others. With the help of GPU parallelism, R3 can be used for very large terrains in offline applications; it gains up to 168x speed up with the GPU implementation.

R2 can be used for real-time applications for an area of 10000x10000. It only lasts about 400ms. It also has a very similar output and needs exactly the same memory as that of R3. If the performance is more important than accuracy, R2 should be chosen. R2 gains up to 10.5x speed up with our GPU implementation. On the other hand, there is no reason to choose Van Kreveld's Algorithm over R2 for GPU implementation because of high memory usage, worse performance and high error rate. The proposed method for Van Kreveld's Algorithm is not sufficient to gain benefit from GPU parallelism; it only gets 1.5x speed up.

## 5.2 Future Work

In this thesis, all algorithms use height map in Cartesian coordinate system. These algorithms can be compared with the geographical coordinate system and distances can be calculated with geodetic curves. R2 accuracy can be improved with the addition of a decision mechanism for which thread should write the final visibility of a cell.

R3 work distribution could be increased by combining closer cells' LOS calculation to one thread, but this requires different cell-to-thread mapping calculations, and calculation cost for each cell needs to be determined. R2 could be modified to use higher number of threads in order to benefit from the massively parallel architecture of the GPU.Our proposed Van Kreveld's Algorithm method can be combined with the method of Ferreira et al. (2013). In their method, the area is divided in sectors and each sector viewshed is calculated separately in CPU threads. Event lists and sorting of each sector can be calculated at GPU and then CPU threads can iterate over those separate lists. In order to make this method work with greater terrains, the memory usage of the algorithm should be decreased. This can be done with the usage of a different sorting technique that does not need temporary memory space.

# REFERENCES

Axell, T., & Fridén, M. (2015). *Comparison between GPU and parallel CPU optimizations in viewshed analysis*. Chalmers University of Techology.

Bell, N., & Hoberock, J. (2012). Thrust: A Productivity-Oriented Library for CUDA. *GPU Computing Gems Jade Edition*, 359–371. https://doi.org/10.1016/B978-0-12-385963-1.00026-5

Blelloch, G. E. (1990). Prefix Sums and Their Applications. *Computer*, 35–60. https://doi.org/10.1.1.47.6430

Bresenham, J. (1965). An incremental algorithm for digital plotting. *IBM Systems Journal*, 25–30.

Cauchi-Saunders, A. J., & Lewis, I. J. (2015). GPU enabled XDraw viewshed analysis. *Journal of Parallel and Distributed Computing*, *84*, 87–93. https://doi.org/10.1016/j.jpdc.2015.07.001

Chao, F., Chongjun, Y., Chen, Z., Yao, X., & Guo, H. (2011). Parallel algorithm for viewshed analysis on a modern GPU. *International Journal of Digital Earth*, 471–486.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009*, *2009*(c), 44–54. https://doi.org/10.1109/IISWC.2009.5306797

Cole, R., & Sharir, M. (1989). Visibility Problems for Polyhedral Terrains. *Journal of Symbolic Computation*, *7*(1), 11–30. https://doi.org/10.1016/S0747-7171(89)80003-3

Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., … Vetter, J. S. (2010). The Scalable HeterOgeneous Computing ( SHOC ) Benchmark Suite Categories and Subject Descriptors. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, (January), 63–74. https://doi.org/10.1145/1735688.1735702

De Floriani, L., & Magillo, P. (2003). Algorithms for visibility computation on terrains: A survey. *Environment and Planning B: Planning and Design*, *30*(5), 709–728. https://doi.org/10.1068/b12979

Fang, J., Varbanescu, A. L., & Sips, H. (2011). A comprehensive performance comparison of CUDA and OpenCL. In *In Parallel Processing (ICPP), 2011 International Conference* (pp. 216–225).

Ferreira, C. R., Andrade, M. V. A., Magalhes, S. V. G., Franklin, W. R., & Pena, G. C. (2013). A Parallel Sweep Line Algorithm for Visibility Computation. *Proceedings of XIV GEOINFO*, 85–96.

Fishman, J., Haverkort, H., & Toma, L. (2009). Improved visibility computation on massive grid terrains. *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '09*, 121. https://doi.org/10.1145/1653771.1653791

Franklin, W., Ray, C., & Mehta, S. (1994). Geometric algorithms for siting of air defense missile batteries. *A], Research Project for Battle*, (2756).

Gao, Y., Yu, H., Liu, Y., Liu, Y., Liu, M., & Zhao, Y. (2011). Optimization for viewshed analysis on GPU. In *Proceedings - 2011 19th International Conference on Geoinformatics, Geoinformatics 2011*. https://doi.org/10.1109/GeoInformatics.2011.5980830

González, C., Pérez, M., & Orduña, J. M. (2016). A Hybrid GPU Technique for Real-Time Terrain Visualization. *Proceedings of Computational and Mathematical Methods in Science and Engineering*.

Harris, M. (2002). About GPGPU. Retrieved from http://gpgpu.org/about

Izraelevitz, D. (2003). A Fast Algorithm for Approximate Viewshed Computation. *Photogrammetric Engineering & Remote Sensing*, *69*(7), 767–774. https://doi.org/10.14358/PERS.69.7.767

Karimi, K., Dickson, N. G., & Hamze, F. (2010). A Performance Comparison of CUDA and OpenCL. *ArXiv E-Prints*, *arXiv*(1), 1005.2581. https://doi.org/10.1109/ICPP.2011.45

Khronos Group. (2017a). OpenGL 4.5 Reference Pages. Retrieved May 20, 2017, from https://www.khronos.org/registry/OpenGL-Refpages/gl4/

Khronos Group. (2017b). The open standard for parallel programming of heterogeneous systems. Retrieved April 19, 2017, from https://www.khronos.org/opencl/

Kirk, D. (2007). NVIDIA cuda software and gpu parallel computing architecture. *ISMM*, 103–104. https://doi.org/10.1145/1296907.1296909

Klöckner, A. (2017). CUDA vs OpenCL: Which should I use? Retrieved April 19, 2017, from https://wiki.tiker.net/CudaVsOpenCL

Kreveld, M. Van. (1996). Variations on Sweep Algorithms : *In In Proc. 7th Int. Symp. on Spatial Data Handling*, 1–14.

Li, Z., Zhu, Q., & Gold, C. M. (2005). *Digital terrain modelling. Principles and methodology*. *New York*. https://doi.org/10.1201/9780203357132

Microsoft. (2010). QueryPerformanceCounter function. Retrieved April 9, 2017, from https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx

NVIDIA Corporation. (2007). CUDA C Best Practices Guide. Retrieved April 19, 2017, from http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

NVIDIA Corporation. (2009). NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Retrieved April 19, 2017, from http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

NVIDIA Corporation. (2011). Thrust. Retrieved April 19, 2017, from http://docs.nvidia.com/cuda/thrust

NVIDIA Corporation. (2014). NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM GK110/210. *Nvidia White Papers*.

NVIDIA Corporation. (2017a). Cuda C Programming Guide. Retrieved April 19, 2017, from http://docs.nvidia.com/cuda/cuda-c-programming-guide

NVIDIA Corporation. (2017b). Parallel programming and computing platform | cuda | nvidia | nvidia. Retrieved April 19, 2017, from http://www.nvidia.com/object/cuda_home_new.html

Osterman, A. (2012). Implementation of the r.cuda.los module in the open source GRASS GIS by using parallel computation on the NVIDIA CUDA graphic cards. *Elektrotehniski Vestnik/Electrotechnical Review*, *79*(1–2), 19–24.

Porwal, S. (2013). Quad tree-based level-of-details representation of digital globe. *Defence Science Journal*, *63*(1), 89–92. https://doi.org/10.14429/dsj.63.3768

Shapira, A. (1990). *Visibility and terrain labeling*. *Master's thesis, Rensselaer Polytechnic Institute*. Master's thesis, Rensselaer Polytechnic Institute.

Stefanescu, E. R., Bursik, M., Cordoba, G., Dalbey, K., Jones, M. D., Patra, A. K., … Sheridan, M. F. (2012). Digital elevation model uncertainty and hazard analysis using a geophysical flow model. *Proceedings of the Royal Society a-Mathematical Physical and Engineering Sciences*, *468*(2142), 1543–1563. https://doi.org/DOI 10.1098/rspa.2011.0711

Temizel, A., Halıcı, T., Logoglu, B., Taşkaya, T. T., Omruuzun, F., & Karaman, E. (2011). Experiences on image and video processing with CUDA and OpenCL. *GPU*

*Computing Gems Emerald Edition,* 547–567.

Ten Eyck, E. (2011). Safety at Colby College: Location of Emergency Call Boxes Marked by Blue Lights and Vulnerable Locations at Colby College. *Atlas Of Maine*, 2011.

Wu, E. (2008). Emerging Technology about GPGPU. *Technology*, (October), 2008.

Ying, Y. (2016). Organization and Management of Massive Terrain Data Using Block Quadtree. *International Journal of Simulation--Systems, Science & Technology*, 17(3). https://doi.org/10.5013/IJSSST.a.17.03.18

Zhao, Y., Padmanabhan, A., & Wang, S. (2013). A parallel computing approach to viewshed analysis of large terrain data using graphics processing units. *International Journal of Geographical Information Science*, *27*(2), 363–384. https://doi.org/10.1080/13658816.2012.692372

# APPENDICES

## APPENDIX A

## ELEVATION FORMAT

Elevation data format used in this thesis is a basic height map. The file contains integer values for elevation of each cell, using binary format. Elevation matrix stored in the file in column-major order. The file does not contain the size of the area stored, so user needs to access data with the correct column and row count. Here is the example code for reading elevation data as 1D and 2D:

```
typedef int elev_t;
int column = 5000;
int row = 5000;
elev_t** elev;
//Read elevation 2D
FILE* f = fopen(elevPaths[i], "rb");
elev = new elev_t*[row];
for (int k = 0; k < row; k++) {
	elev[k] = new elev_t[column];
	fread(reinterpret_cast<char*>(elev[k]), sizeof(elev_t),column, f);
}
fclose(f);

elev_t* elev1D = new elev_t[nrowCounts[i] * nColumnCounts[i]];
//Read elevation 1D
f = fopen(elevPaths[i], "rb");
int readCount = 0;
for (int k = 0; k < row; k++) {
	fread(reinterpret_cast<char*>(elev1D + readCount), sizeof(elev_t), column, 
f);
	readCount += column;
}
fclose(f);
```

# APPENDIX B

## CODE STRUCTURE

You can download the source code used in this thesis from:

https://github.com/dcfu/gpuviewshed/

Link includes .sln file for Visual Studio 2013. CUDA v7.5 needed to compile the source code.

Simply open “CUDAParallelViewshed.sln” file with Visual Studio and compile. No linux support is provided.

Here is the list of the files and libraries:

**EasyBMP**: Includes BMP image format manipulation codes, downloaded from: http://easybmp.sourceforge.net

**helper_cuda.h , helper_string.h:** CUDA error checking, provided by NVDIA

**kernel.h:** Contains signatures of kernel call wrapper functions

**kernel.cu:** Contains actual CUDA implementation of kernels

**thrustOperations.cu:** Contains methods that use Thrust

**rbbst.h, rbbst.cc:** Contains tree implementation used for active tree in Van Kreveld’s Algorithm, provided by Laura Toma, Bowdoin College - ltoma@bowdoin.edu, Yi Zhuang - yzhuang@bowdoin.edu

**main.cpp:** Contains actual test codes and methods

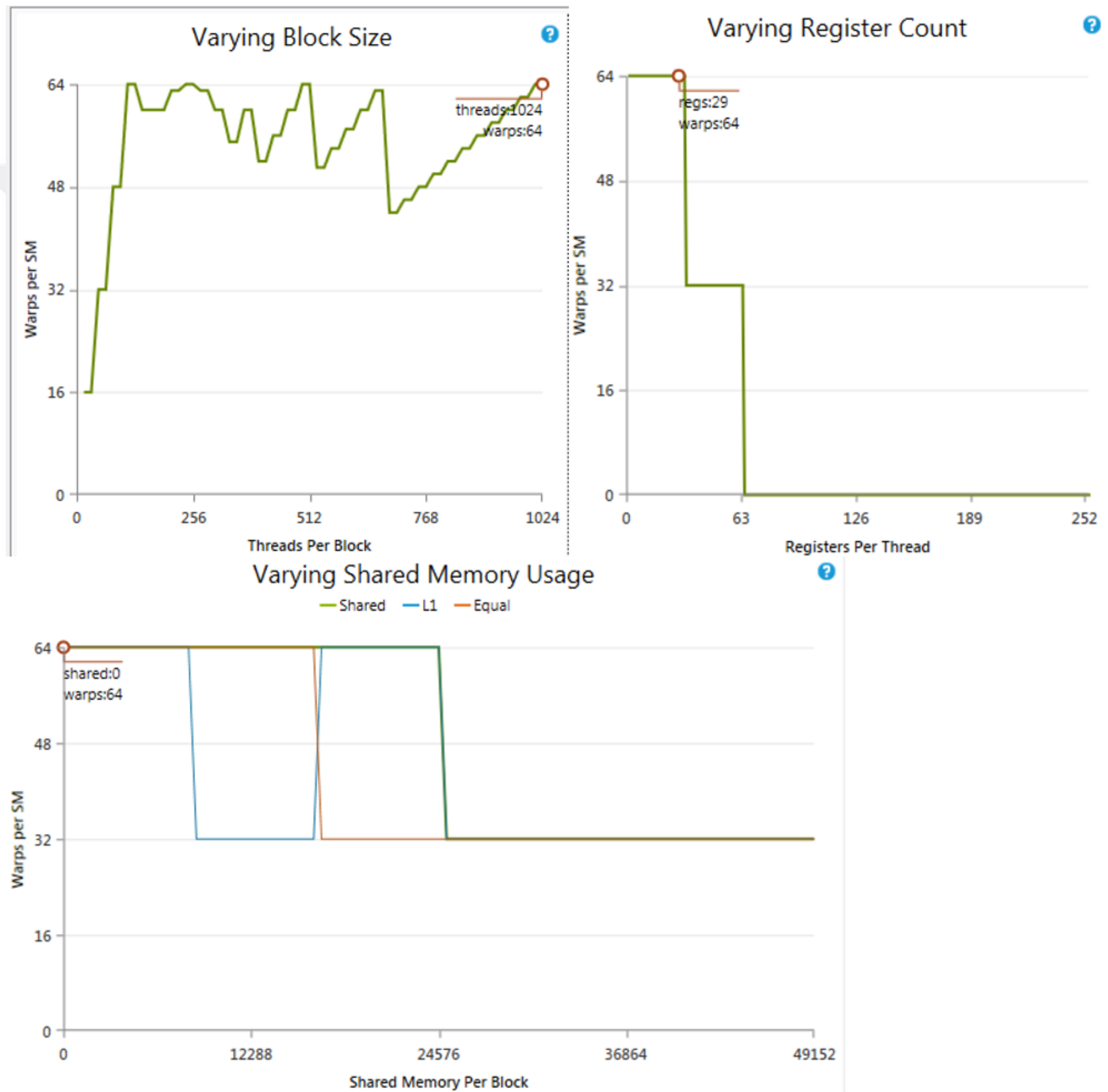
Note: Van Kreveld's Algorithm is based on code provided by :

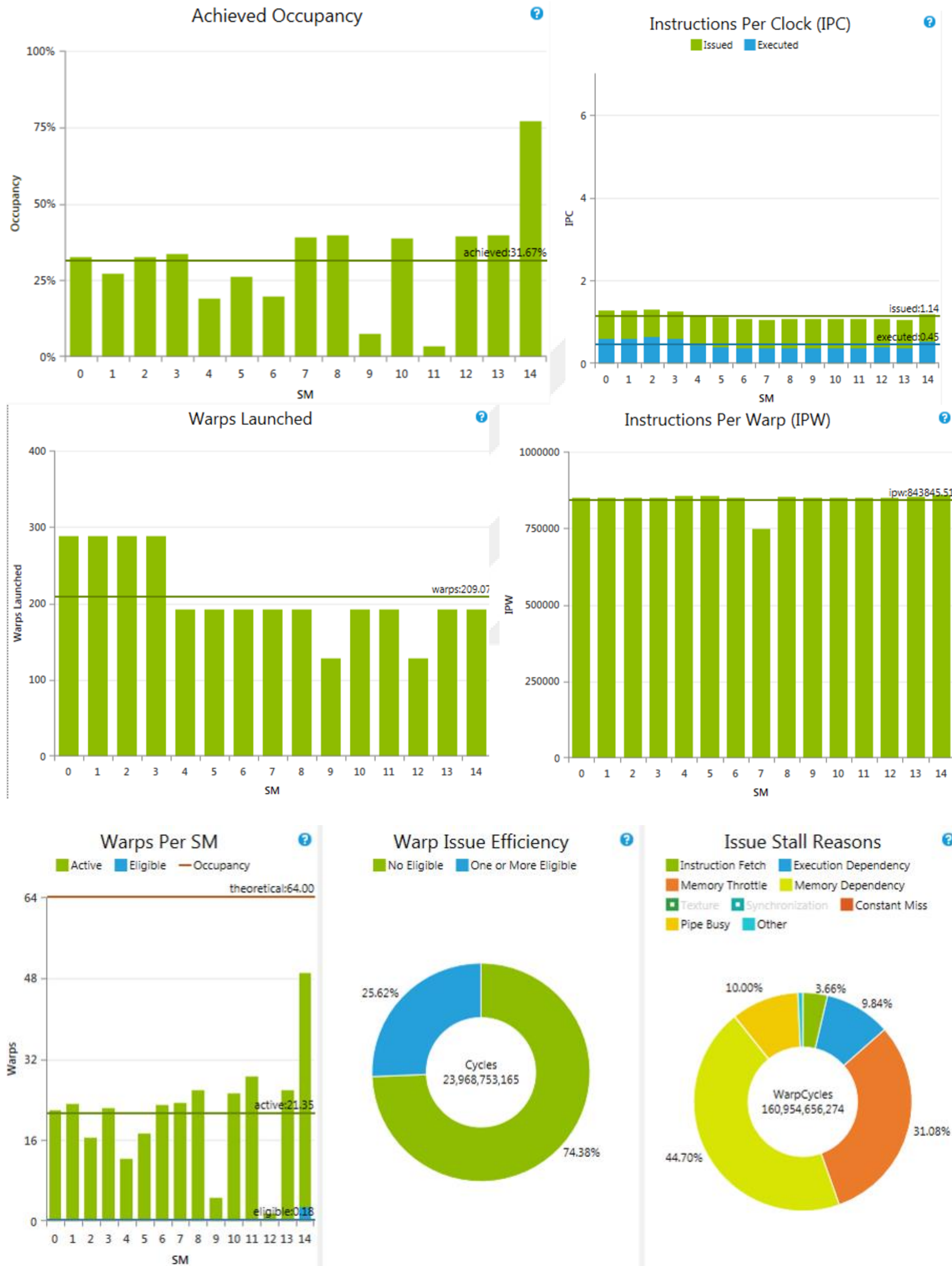
Ferreira, C.R., et al., 2013. A Parallel Sweep Line Algorithm for Visibility Computation. In: Proceedings of the XIV Brazilian Symposium on GeoInformatics (GeoInfo 2013), Campos do Jordão, SP Brazil: 85-96.

# APPENDIX C

# PROFILER RESULTS

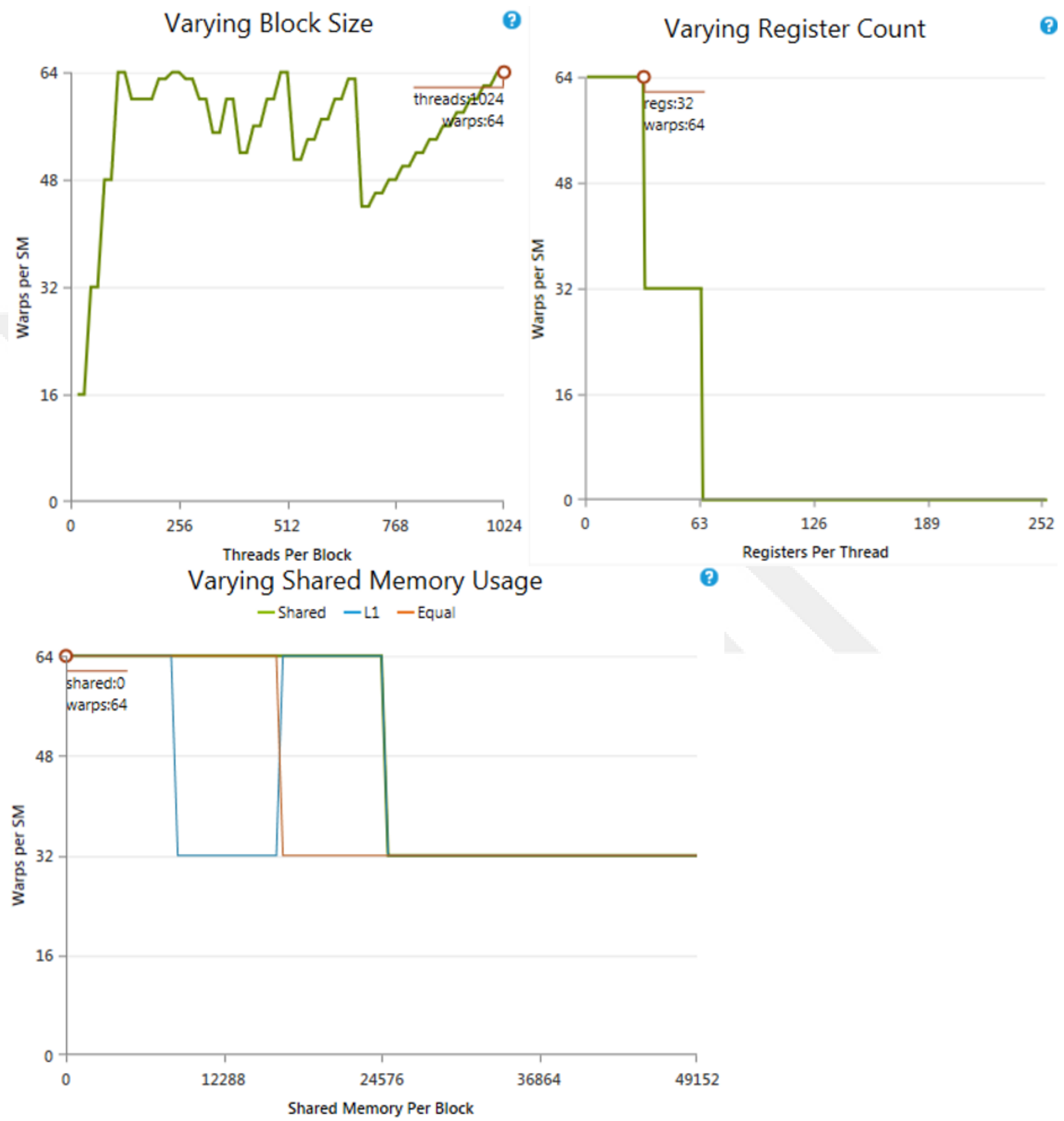
**R2**

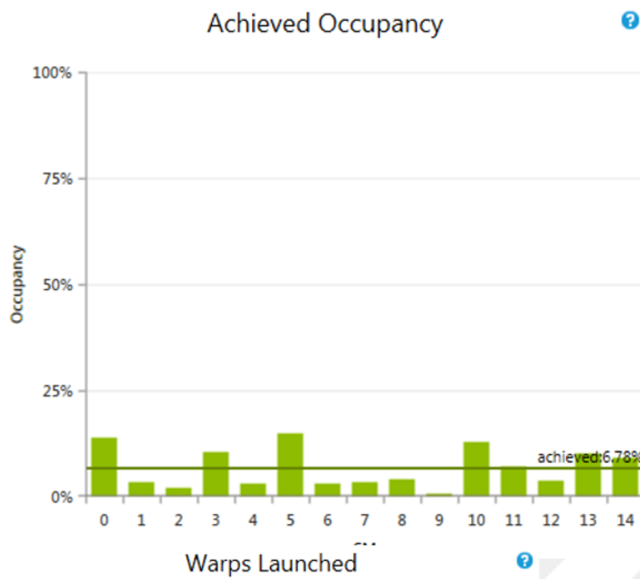


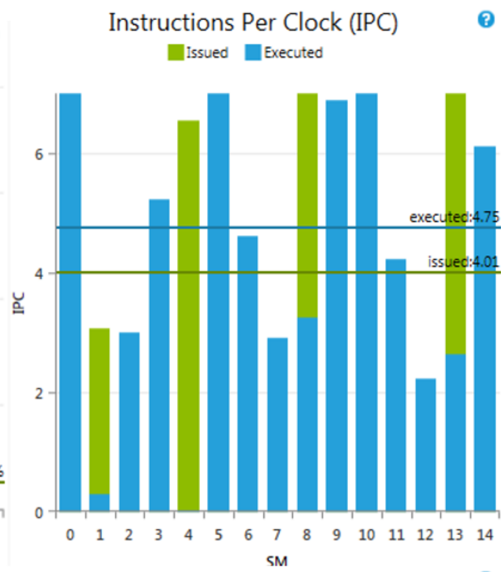


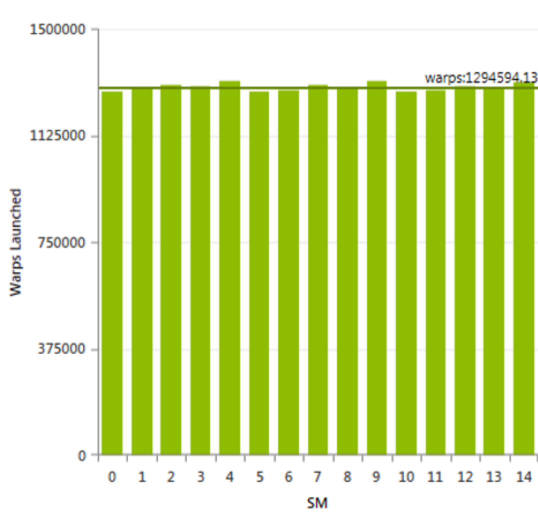
Achieved Occupancy
Occupancy
100%
75%
50%
25%
0%
achieved:31.67%
SM
Instructions Per Clock (IPC)
Issued
Executed
IPC
issued:1.14
executed:0.45
SM
Warps Launched
Warps Launched
warps:209.07
SM
Instructions Per Warp (IPW)
IPW
ipw:843845.51
SM
Warps Per SM
Active
Eligible
Occupancy
theoretical:64.00
Warps
active:21.35
eligible:0.18
SM
Warp Issue Efficiency
No Eligible
One or More Eligible
25.62%
Cycles
23,968,753,165
74.38%
Issue Stall Reasons
Instruction Fetch
Execution Dependency
Memory Throttle
Memory Dependency
Texture
Synchronization
Constant Miss
Pipe Busy
Other
10.00%
3.66%
9.84%
WarpCycles
160,954,656,274
31.08%
44.70%

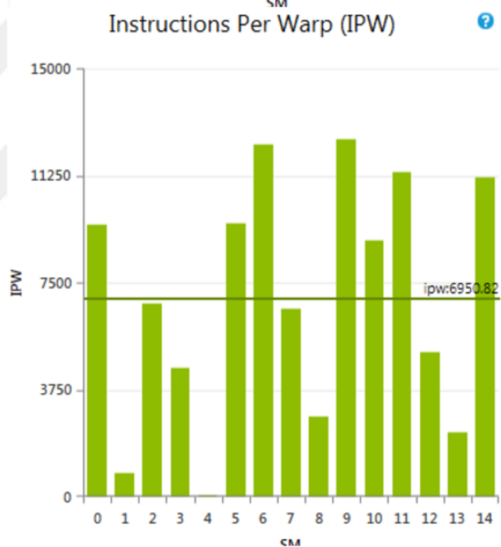
**R3**

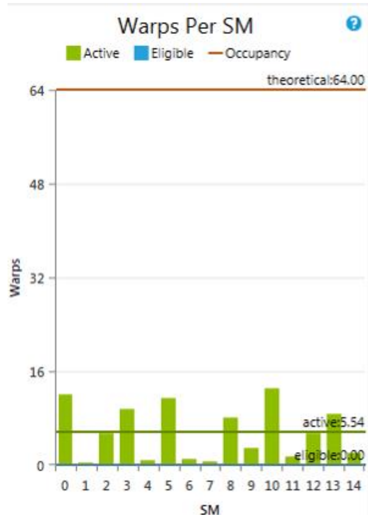


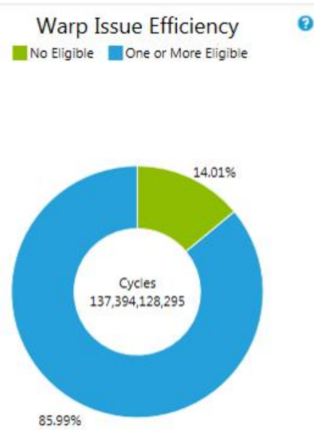

Achieved Occupancy
100%
75%
50%
25%
0%
Occupancy
achieved:6.78%
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

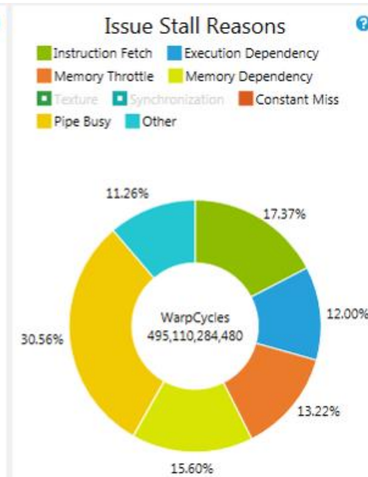

Instructions Per Clock (IPC)
Issued
Executed
IPC
executed:4.75
issued:4.01
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
SM


Warps Launched
1500000
1125000
750000
375000
0
Warps Launched
warps:1294594.13
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
SM


Instructions Per Warp (IPW)
15000
11250
7500
3750
0
IPW
ipw:6950.82
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14


Warps Per SM
Active
Eligible
Occupancy
theoretical:64.00
64
48
32
16
0
Warps
active:5.54
eligible:0.00
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
SM


Warp Issue Efficiency
No Eligible
One or More Eligible
14.01%
Cycles
137,394,128,295
85.99%


Issue Stall Reasons
Instruction Fetch
Execution Dependency
Memory Throttle
Memory Dependency
Texture
Synchronization
Constant Miss
Pipe Busy
Other
11.26%
17.37%
WarpCycles
495,110,284,480
12.00%
30.56%
13.22%
15.60%