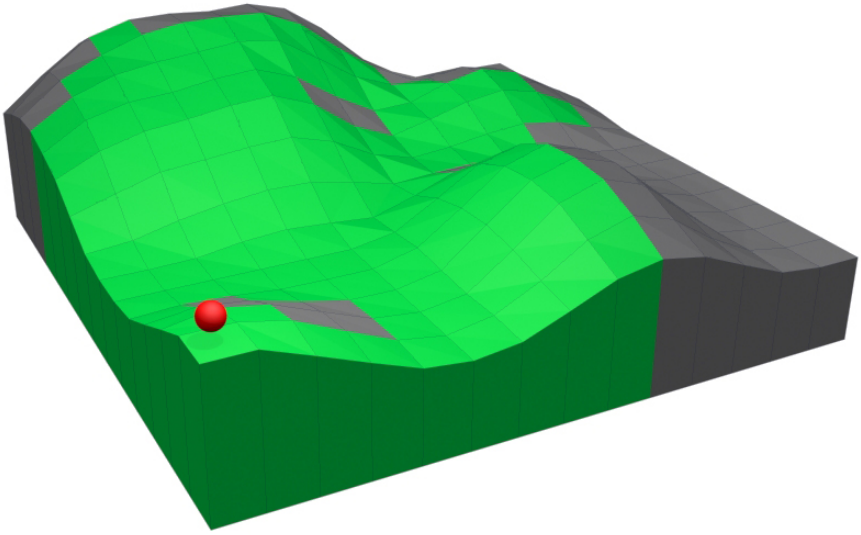


Comparison between GPU and parallel CPU optimizations in viewshed analysis

Master's thesis in Computer Science: Algorithms, Languages and Logic

TOBIAS AXELL
MATTIAS FRIDÉN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Comparison between GPU and parallel CPU optimizations in viewshed analysis

TOBIAS AXELL
MATTIAS FRIDÉN

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2015

Comparison between GPU and parallel CPU optimizations in viewshed analysis
TOBIAS AXELL
MATTIAS FRIDÉN

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Gothenburg, Sweden
Telephone +46 31 772 1000

Cover: A viewshed on a rolling landscape, the red ball marks the observer position.

Gothenburg, Sweden 2015

Comparison between GPU and parallel CPU optimizations in viewshed analysis
TOBIAS AXELL
MATTIAS FRIDÉN
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Parallel CPU implementations of a viewshed algorithm using both multithreading and SIMD vectorization and GPU implementations were implemented and compared in this study. The results show that parallelism is essential for achieving good performance on a CPU, and that data transfer can be partly overlapped by computations to hide some of the overheads in GPU implementations. The GPU implementation was the fastest with a performance approximately 3 times faster than the parallel CPU implementation for the hardware the tests were performed on.

Keywords: viewshed, line of sight, GPU, parallel CPU, SIMD, vectorization, R2.

Acknowledgements

We would like to thank Erik Sintorn for his supervision and for many great conversations, and Ulf Assarsson for being our examiner. We would also like to thank Anders and the others for help and support during the project.

Tobias Axell and Mattias Fridén, Gothenburg, June 2015

Contents

1
Introduction

A viewshed is an area that is visible from a given observation point. Every point that is within a given range from the observation point and is visible from the observation point is part of the viewshed. A viewshed can be used to find suitable points for optimizing path planning, find out if a camera has coverage of some area, or landscape planning.

As previous research indicates that the viewshed problem is suitable for parallel computation [1], it will be of interest to research and test different algorithms for both the CPU and GPU to find good algorithms for these. These algorithms will be implemented and optimized, and then the best implementation on each hardware will be compared to find out if the viewshed problem is better suited for the CPU or the GPU.

1.1 Problem

The aim of this thesis will be to research whether the CPU or GPU is the most suited computing platform for solving the viewshed problem in parallel, and to find out what it is that makes the implementation fast or not. This parallelization may not be trivial and tests needs to be performed to find out what makes implementations fast or not. To get a fair comparison between implementations, it is important that the hardware that the algorithms are tested on is fair to compare, or at least tested with the differences in mind. All implemented algorithms has to be tested to find time-consuming parts so that they can be optimized. It might be hard to estimate how an implementation will behave on other hardware than it has been tested on, thus the implementation needs to be tested on different hardware to be able to draw more general conclusions about the performance of the implementation on other hardware.

1.2 The Viewshed Problem

Computing a viewshed takes three parameters; a digital elevation model (DEM) which is a map with height information about the terrain, an observer position in three dimensions, and a range defining the area around the observer position for which to compute the viewshed. The task is then to compute which parts of the area inside the range that are visible from the observation point (see Figure 1.1).

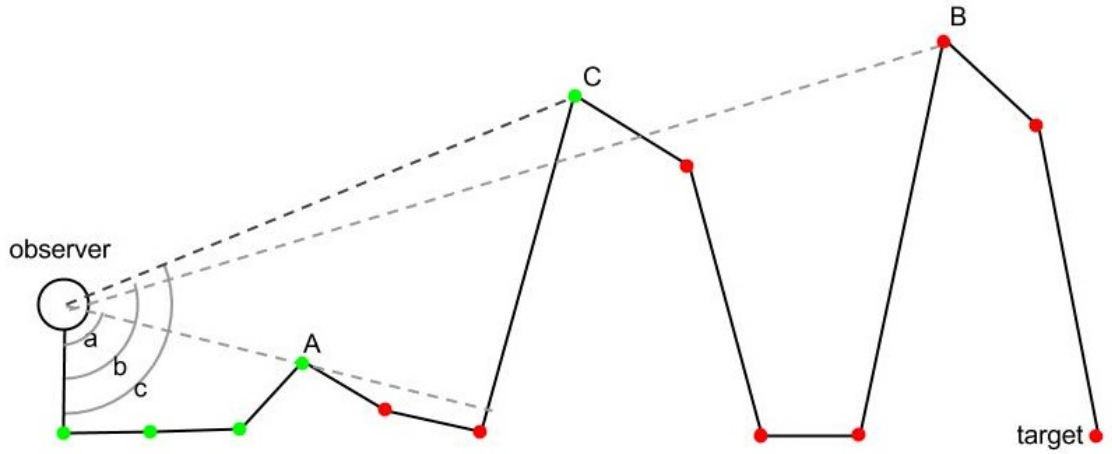


Figure 1.1: A cross section of a binary viewshed. Green dots indicates visible DEM points and red dots indicates obscured DEM points

1.2.1 Extended Problem

This paper will adopt the same solution format as Franklin et al. [2], computing the height above ground that is needed at each position in order to be visible from the observer position. For example, a height of 0 meter means that the ground on that position is visible and a height value of 1 meter means that an object on that position has to be at least 1 meter high (or elevated 1 meter above ground level) in order to be visible from the observation point (see Figure 1.2).

1.3 Previous Work

R3, R2, and Xdraw are three algorithms that has been studied by Franklin et al. [2] The R3 algorithm is non-approximate with an $\theta(n^3)$ complexity. The R2 algorithm has time complexity $\theta(n^2)$ and almost as accurate as R3. The Xdraw algorithm has the same algorithm complexity as R2, but with a better constant and with an even higher approximation error.

Wang et al. [3] has developed an algorithm that's similar to the Xdraw algorithm that uses reference planes to define local horizons based on previous results. This algorithm is also fast, but approximate.

Izraelevitz [4] has developed a modified version of the Xdraw that use a *back-tracking* method that sacrifices some performance in order to reduce approximation error.

Xia et al. [5] has implemented four versions of a line of sight based viewshed algorithm (it is unclear exactly which one as they do not describe it in detail) with different levels of parallelism: SMSR (Sequential matrix travarsial, Sequential

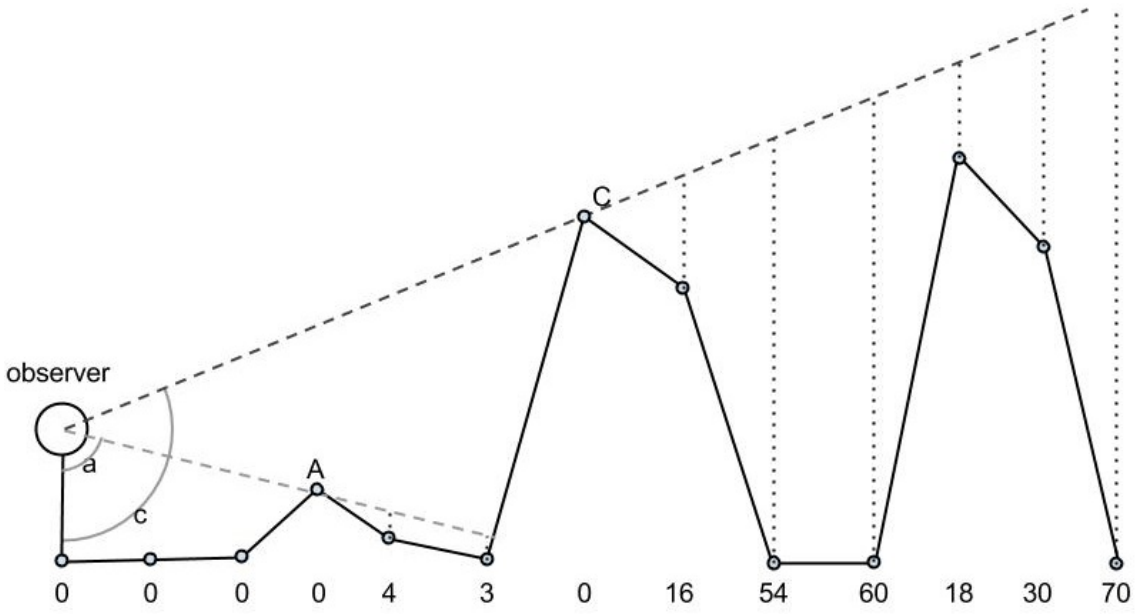


Figure 1.2: A cross section of an extended viewshed. The numbers indicate the required height above ground at each point in order to be visible from the observers position

ray traversial), PMSR (Parallel matrix travarsial, Sequential ray traversial), SMPR (Sequential matrix travarsial, Parallel ray traversial), and PMPR (Parallel matrix travarsial, Parallel ray traversial). The parallel parts were implemented on the GPU using CUDA and the sequential parts were implemented on the CPU. The version using PMSR were the one with the best results.

Stojanovic and Stojanovic [6] implemented the R3 algorithm on the GPU and compared the implementation to a sequential CPU solution. It differs from the algorithm implemented by Xia et al. as it computes a binary result and is using this fact to finish the computation of a sightline as soon as the target cell has been occluded, if it is occluded.

Blelloch, Sengupta et al. and Dotsenko et al. have developed algorithms for solving the prefix scan and prefix scan algorithms in parallel[7, 8, 9], and Blelloch has also shown that a parallel prefix scan algorithm can be used to solve the Line of Sight (LOS) problem in parallel, a problem closely related to the viewshed problem[7].

Lee et al. has argued that many claimed speedups of 10x to 1000x when using GPUs are exaggerated and shows that the average relation in execution speed (without data transfer overheads in their measurements) is a 2.5x advantage for GPUs for a set of 14 tested problems including Monte Carlo, Convolution image filtering and Fast Fourier Transform algorithms[10].

Gregg and Hazelwood argue for the importance of the context in which an algorithm is used, as data-transfer overheads from and to the GPU may have a large

impact on the runtime of a GPU implementation[11].

1.4 Purpose and Goals

The purpose of this thesis is to compare GPU and parallel CPU solutions of the viewshed problem in order to determine if either of these two classes of hardware is more suitable for solving this problem efficiently. For this thesis we have three goals:

1. Implement two algorithms for viewshed analysis (see 1.2.1): one for a CPU, and one for a GPU. The algorithms may be approximate but then with a justifiably small approximation error.

2. Benchmark the produced implementations to see whether a GPU implementation or a CPU implementation is better than the other in terms of performance.

3. Reach more general conclusions about what makes the algorithms/ implementations efficient (or inefficient) on GPUs and CPUs. Such conclusions could be used as basis for decision when choosing for which hardware to implement some solution to some other geospatial analysis problem.

1.5 Limitations

This thesis will only compare parallelization on CPUs and GPUs. We will not consider distributed computing, computer clusters or any other special hardware for parallel computation.

There are different kinds of DEMs, such as rasters/grids, triangulated irregular networks (TINs), and triangulated grids but this thesis will only consider algorithms working on raster DEMs and optimizations for them.

This study will not consider refraction in the atmosphere nor diffraction over and around terrain that may be of interest when computing viewsheds for, for example, radio signals.

1.6 Outline

Chapter 2 will present different algorithms for the viewshed problem as well as relevant theory for GPU programming and parallel optimizations for CPUs. Chapter 3 chapter will describe implementation choices, chosen algorithms, encountered problems, and suggested solutions. Chapter 4 will contain results where benchmarks of different implementations will be presented and compared. Chapter 4 will be followed by a discussion chapter, a conclusion chapter, and a chapter about future work.

2
Theory

This chapter will first present relevant algorithms for viewshed analysis and the closely related line of sight problem. Then some theory for parallel CPU optimization, and background theory for GPGPU programming is presented.

2.1 Line of Sight

The line of sight (LOS) problem is; given a DEM, an observer position and a direction; calculate which points on the ground along the sightline originating from the observer position with the given direction that are visible from the observer position.

This can also be expressed in terms of angles, or slopes; a point p on the ray is visible if and only if the vertical slope of the line from the observer position to p is higher than the slopes of the lines from the observer to all the points on the ray that is between the observer position and the point p (see Figure 1.1) [7].

Franklin et al. also considers the extended version of the problem where not only a binary result is wanted, but also the height required to be visible from the observer (see Figure 1.2). They use this output to compare different solutions in terms of difference in meters between results of two algorithms, instead of just comparing the binary result if a cell can be seen or not[2].

2.1.1 Computing a Line of Sight

Consider an observer O and a point p_t which is the target position that the observer is looking at. By stepping along the sightline created from O to p_t, one can compute if p_t is visible or not by looking up the height of the ground at each step s to describe the point p_s on the sightline on step s. The position of p_s is used to first compute the index of that point in the DEM, and this index is used together with the DEM to create the point p_g describing the point on the ground at step s. p_g only occludes p_t if the height of p_g is greater than the height of p_s.

Another way of computing the same result is by computing the vertical angle v_t between O and p_t before stepping along the sightline, and then computing the vertical angle v_g for each point p_g on the ground at each step s. v_g can then be compared against v_t at each step s instead of comparing their heights. p_g is occluding p_t if and only if v_g is greater than v_t. The most occluding horizon for a point p_t is simply the point p_g with the greatest vertical angle that was computed from all steps along the sightline from O to p_t.

To compute the height that is required to elevate a point p from the ground in order for it to be visible from the observer position, the most occluding horizon h is computed for p, and the lift can then be computed from O, p and h .

As seen in Figure 2.1, the most occluding point isn't necessarily the highest point, but instead the point that has the highest vertical angle from the observer. An algorithm cannot stop once it finds a horizon that occludes the point in question, but has to compute the vertical angle for all points on the ground along the sightline in order to find the highest one.

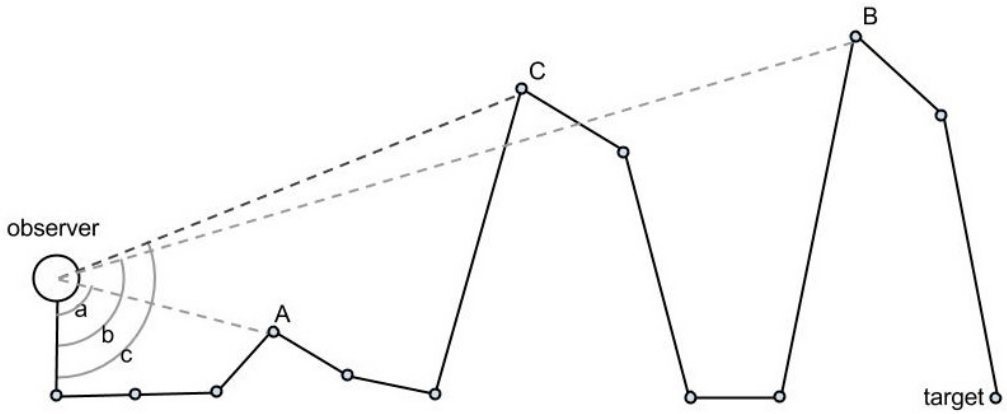


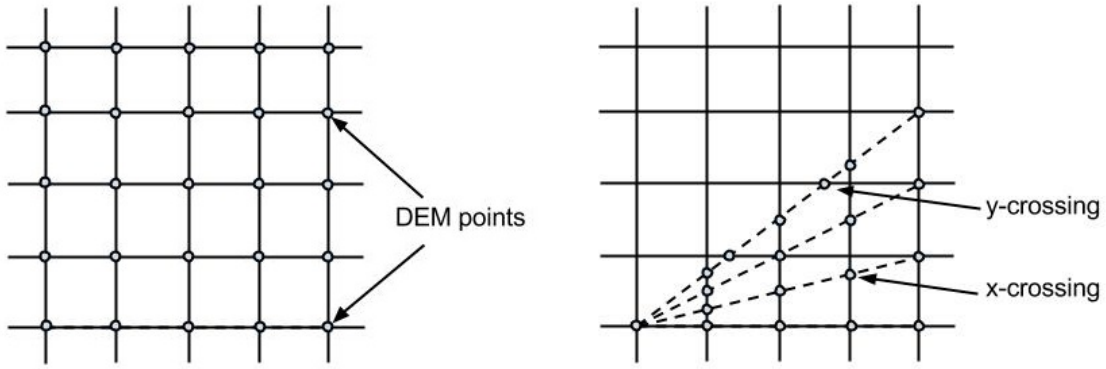
Figure 2.1: Visualization of horizons. The most occluding horizon is the point on the line between the observer and the target that has the highest vertical angle. In this image, its point C, as the angles fulfill a < b < c

2.1.2 DEM Interpretation and Interpolation

In order to know which points to consider when finding the most occluding horizon, a stepping method must be chosen, and the stepping method is dependent on how the data in the DEM is interpreted. The DEM represents an area of continuous terrain and the points in the DEM can be considered as samples as the DEM has a finite amount of points (see Figure 2.2a).

When using the DEM to reconstruct the height of an arbitrary point of the terrain, several methods for lookups can be used such as nearest neighbor, linear interpolation, and cubic interpolation. Nearest neighbor will sample from the closest point in the DEM and return the value without modification, while linear interpolation and cubic interpolation will make multiple samples and then return an interpolated result. Linear interpolation uses two samples and mixes them depending on the distance from the point that the result is wanted for and is considered a good choice of interpolation method to use for the viewshed problem according to Franklin et al. [2], as long as the positions to sample are on the edge segment between two points in the DEM. Linear interpolation of four values (arranged as corners of an

axis aligned rectangle) is performed by first performing linear interpolation of pairs along one dimention, and then interpolating the result along the other dimention. This is also called bilinear interpolation. The interpolation method used by samplers on the GPU is either nearest or bilinear interpolation [12], and interpolation has to be made manually on the CPU.



(a) A visualization of a raster DEM as a grid where the data points is located *at the intersections* of the grid lines

(b) Visualization of x and y-crossings in a raster DEM

Figure 2.2: Illustration showing where DEM points, x-crossings, and y-crossings are located

As linear interpolation is the recommended interpolation method, sample points should be made on line segments between two sample points and thus the stepping method will need to make a sample each time the sightline passes such a edge segment. Franklin et al. uses the terms x-crossing and y-crossings for for when a sightline passes between two points in the DEM where an x-crossing is when the sightline passes between two DEM points with the same x-position and an y-crossing is when the sightline passes between two DEM points with the same y-position. Figure 2.2b illustrates this.

According to Franklin et al. the earth's curvature must be accounted for when working with large scale problems. They propose a close approximation as

$$E_C = \frac{D_O^2}{2 * R_E} \tag{2.1}$$

where where D_O is the distance from the observer, R_E is the effective radius of the earth's, and E_C is the change in elevation due to earth curvature [2].

2.1.3 Parallel Prefix Scan

Blelloch has shown that the line of sight problem can be solved in parallel by using a parallel prefix scan algorithm [7]. The parallel prefix scan algorithm operates on an ordered set of n elements and uses a binary associative operator $\oplus$ to produce an ordered set of n elements, where output at index i is $input[i] \oplus output[i-1]$. The first element of the output is initially set to the value of the first element in the

input for an inclusive scan. In figure 2.3, the *max* operation was used to produce the result, and the operation can be interpreted as *the maximum number so far* in the input.

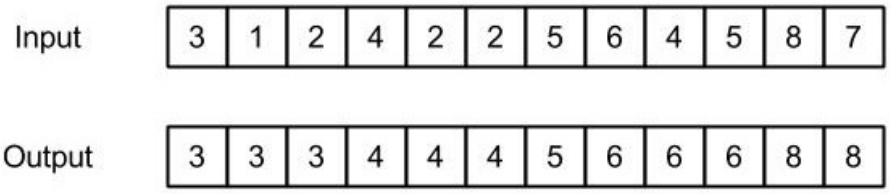


Figure 2.3: Visualization of the prefix scan algorithm using the max operator

Blelloch solves the line of sight problem in parallel by solving a sub problem of the prefix scan problem in parallel with $n/2$ processors by using two sweeps, one up sweep and one down sweep. The up sweep is performed by creating a tree with $log(n)$ levels. Figure 2.4 illustrates this tree with the *max* operation as $\oplus$. The levels are processed incrementally and level l uses $n/2^{l+1}$ processors, where each processor uses the $\oplus$ operator on two elements from level $l-1$ with the exception of the first level operates directly on the input set[7].

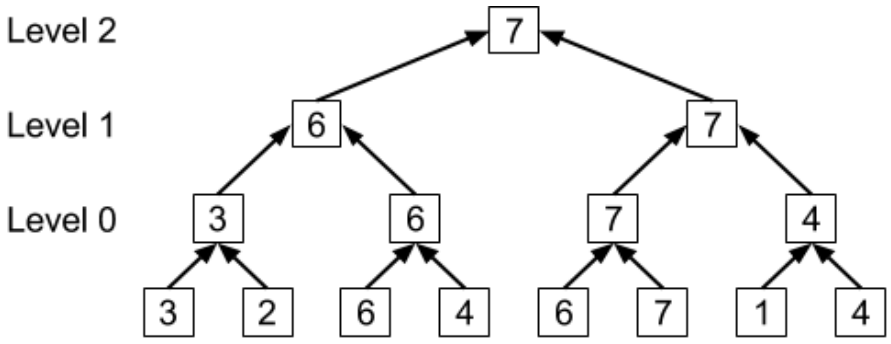


Figure 2.4: Up sweep in an example using the *max* operator. Level 0 uses 4 processors that each work on two elements on the previous level

The down sweep works on the same tree with the same number of processors at each level, but instead processes the tree levels in decreasing order. Initially the identity value is inserted at the root. The identity value for the max operation is negative infinity as $max(-\infty, x) = max(x, -\infty) = x$. Each processor p at level l writes to its right child the value from the $\oplus$ operator on the value at tree position p and the value of the left child. To the child to its left, the value in the tree at

position p is written. Figure 2.5 shows the tree after the execution of the down sweep.

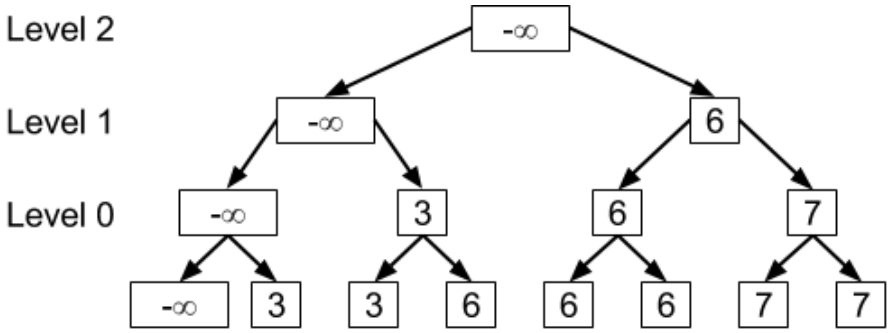


Figure 2.5: Down sweep by writing maximum of current value and value of left child to the right child, and writing the current value to left child

To get the solution to the inclusive prefix scan problem, a left shift of the elements in the array needs to be performed, followed by the final computation of the last index, $output[n-1]$, that will be set to $outpupt[n-2] \oplus input[n-1]$. Instead of doing this shift, the intermediate result can directly be used as a lookup table in the line of sight problem to look up any previous maximum angle.

2.1.4 Algorithm Criteria

Franklin et al. describe two criteria that they consider important for line of sight algorithms: the criterion of adequacy and the criterion of appropriateness.

The *criterion of adequacy* states that every point in a DEM that is on a sightline or a direct neighbor to the sightline (in the sense that there is no other point between the point and the sightline) shall influence the computation of the most occluding horizon for that sightline.

The *criterion of appropriateness* is the reverse of the criterion of adequacy, it states that any point in a DEM that is not on a sightline nor a direct neighbor of the sightline should not influence the computation of the most occluding horizon for that sightline [2].

2.2 Viewshed Algorithms

There are several existing algorithms for the viewshed problem described in literature. Franklin et al. [2] present the R3 and R2 algorithms that are based on solutions to the line of sight problem. They also present the Xdraw approximating algorithm that is not based on LOS, but is rather what they call a *wave front* algorithm [2]. Wang et al. present an algorithm based on reference planes that is very similar to the Xdraw algorithm [3]. Izraelevitz presents an algorithm that combines the wave

front and LOS approach in order to reduce the approximation error, but keep some of the good time complexity that the wave front approach gives [4].

2.2.1 R3

The R3 algorithm [2] is based on the line of sight problem. It works by computing the line of sight from the observation point to every cell in the raster DEM in order to determine the visibility of that cell. The R3 algorithm provides high correctness as it makes full use of the available elevation data [4] and satisfies both the criteria of adequacy and appropriateness, but it's time complexity is relatively high, $\theta(n^3)$, for an n by n raster DEM [2]. It steps on every x-crossing and y-crossong along a sightline and on every such crossing it computes the vertical angle between the observer, and uses the maximum of these angles to compute the height required to see the target point from the observer.

2.2.2 R2

The R2 algorithm [2], just like the R3 algorithm, works by sending out rays from the observer and computing the line of sight along that ray. What differentiates R2 from R3 is the way that the sightlines are computed. R3 computes a line of sight to every cell in the raster DEM and each line of sight computation only writes the visibility (or required height) value at the end of the sightline to the result raster. R2 only computes line of sight to the boundary cells of the area for which to compute the viewshed and lets these rays fill in the values of the cells they pass over from the observer out to the boundary. The boundary cells are defined as the set of cells that are inside the boundary and has at least one adjacent cell outside of the boundary [2].

This method introduces a new problem; there will still be raster cells that are passed over by multiple rays, especially close to the observer. Since the rays in R2 writes results for all cells on the way out from the observer, several line of sight computations will write a result to the same cell. This could be unwanted because different sightline computations may have different approximation errors for a certain DEM point and the algorithm will then have an unnecessarily high approximation error if not the best result for each DEM point is used.

This can be solved by letting each sightline computation decide whether or not to write a cell to the result raster. Franklin et al. [2] suggests that when multiple sightlines pass over a raster cell only the one that passes closest to the cell's center should write its value to the result raster (see Figure 2.6) [2].

As described in 2.1.2, the points to sample are located on the x and y crossings in the DEM, but for the R2 algorithm a different stepping might be sought in order to make the algorithm faster. Franklin et al. propose to reduce the number of points to which complete line of sights are calculated by only considering the axis with the most crossings. Figure 2.7 shows the naming of the octants of a viewshed and as octants I, IV, V and VIII have more x-crossings than y-crossings, only x-crossings will be considered in these octants. In octant II, III, VI and VII, only y-crossings will be considered. This way of only stepping in one of the crossings does reduce

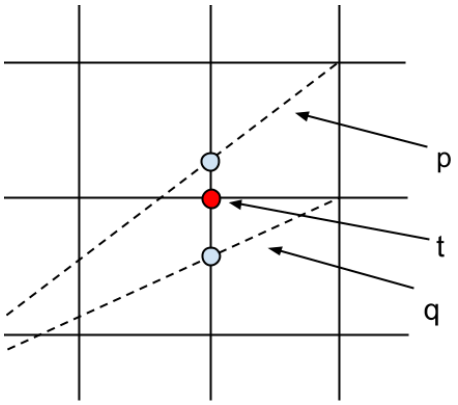


Figure 2.6: Visualization of two lines that both traverse a cell. The sightline p passes the point t slightly closer than the sightline q. Therefore the required height computed in p's x-crossing is chosen as the best approximate result for t

accuracy, but even with this modification, Franklin et al. still argue that the R2 algorithm fulfills the criterion of adequacy and the criterion of appropriateness in a broader sense.

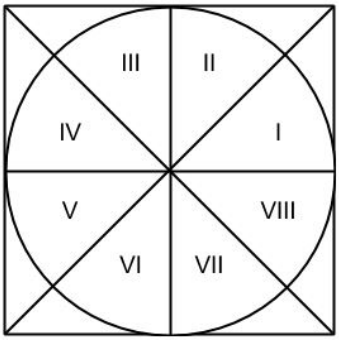


Figure 2.7: Visualization of octants

This makes the stepping method easier as the slope of the line which to step along, and a fixed step size, can be calculated before the actual stepping. With this stepping method, all instances of line of sight in the same octant will take an equal amount of steps on the sightline and the points of which to sample will all be on the same line in the same step of the algorithm (see Figure 2.8). This offers opportunities for caching, shared reads from the DEM, synchronization and loop divergency.

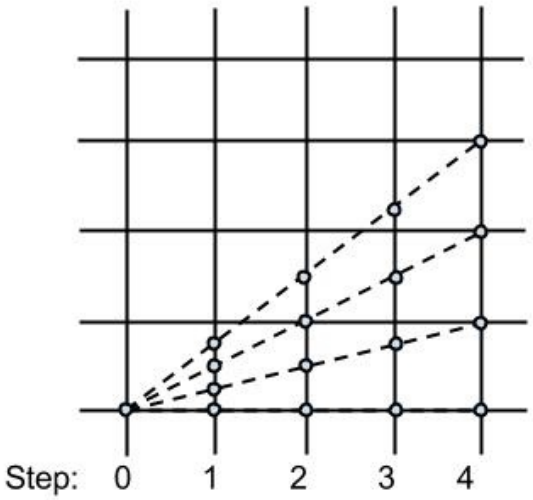


Figure 2.8: Visualization of stepping in octant I. The figure show how the steps are taken in the x-crossings

2.2.3 Wave Front Algorithms

Wave front algorithms are algorithms that are not based on a line of sight algorithm for computing the viewshed problem but rather computes visibility in layers outwards from the observer like a wave. Examples of wave front algorithms are the Xdraw algorithm presented by Franklin et al. [2] and the reference plane algorithm presented by Wang et al. [3].

The reference plane algorithm first assumes that the points on the map that are closest to the observer are always visible, and therefore has the required height 0, since there are no points between them and the observer that can conceal them. Then the DEM is divided into octants (see Figure 2.7) and the result for the horizontal, vertical and diagonal dividing lines are calculated with a LOS method as in R2. Then for every point inside the octants the required height for a point p is computed by finding the two reference points, r_{1p} and r_{2p}, that are the two points closest to p that a straight line from the observer to p would go between. The algorithm computes the results in a breadth first manner from the observer and outwards. This means that when computing the result for p all points in the octant that are closer to the observer than p (horizontally, if in octant I, IV, V or VIII, vertically otherwise), including r_{1p} and r_{2p}, has already been processed by the algorithm. The observer position, r_{1p}, and r_{2p} are then used to form a plane in three dimensions: the *reference plane*. The elevations used when forming the plane is the height of the observer and the required heights of r_{1p} and r_{2p}. If the point p is above or on the reference plane it is visible and the required height for p is 0, if p is below the plane, p is obscured and its required height is the distance from p to the plane in the height axis [3].

The Xdraw algorithm works in a similar fashion, but instead of creating a plane, Xdraw interpolates the point between the two reference points r_{1p} and r_{2p} that the straight line from the observer to p would pass through, and computes the required height for p as: if p is on or above the line the required height is 0, otherwise the required height is the distance from the point p to the line in the height axis [2].

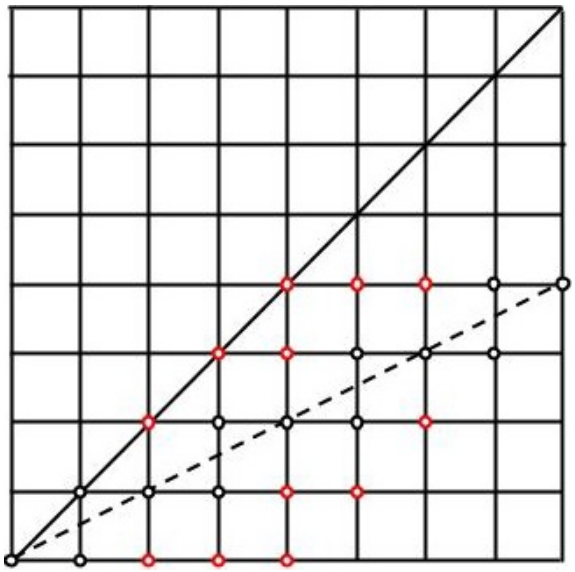
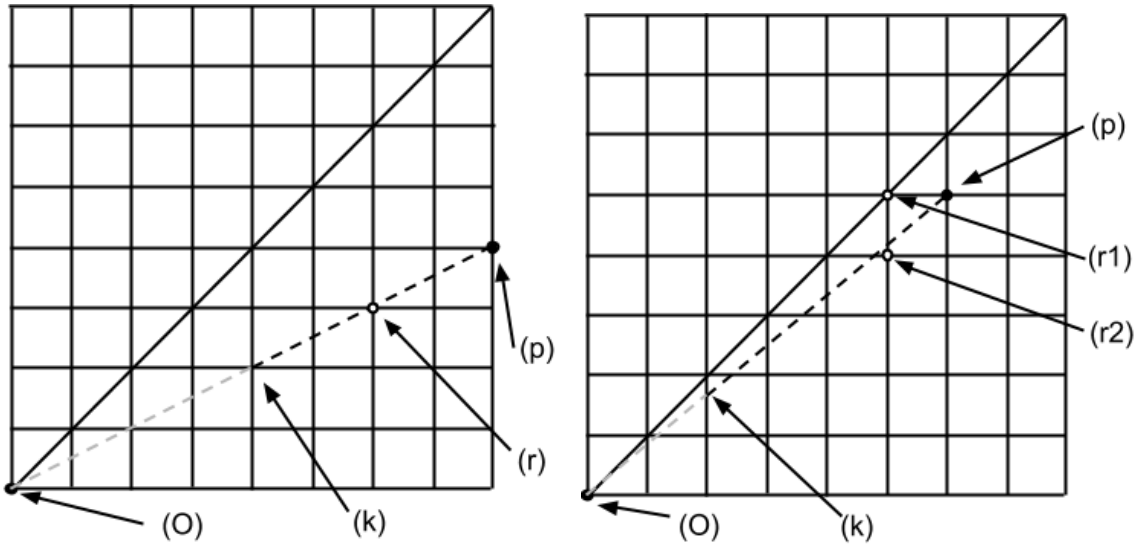


Figure 2.9: Illustration of why wave front algorithms does not fulfill the criterion of appropriateness. The point in the lower left corner is the observer position, the dashed line is an imagined sightline to a border cell. The circled intersections is the data points that computing the visibility for the end point of the sightline would depend upon. The red circles are inappropriate dependencies

Wave front algorithms such as Xdraw and reference plane fulfils the criterion of adequacy but breaks the criterion of appropriateness due to how the result for one point depends on two results for two other points. How the dependency for the result of one point in an octant propagates backwards is shown in Figure 2.9. Because of these inappropriate dependencies, the wave front method has a quite high approximation error [2].

A method for reducing the approximation error of the wave front method is introduced by Israelevitz [4]. Israelevitz's algorithm combines the wave front approach with the LOS approach by introducing a *backtrack order*. The algorithm computes the result from the observer and outwards like the Xdraw and reference plane algorintms so that when computing the visibility for a point p in the DEM, the points between the observer and p have already been processed. When computing the visibility for a point p in a DEM with Israelevitz's algorithm, the first thing done is to backtrack a number of steps defined by the backtrack order along a straight line from p towards the observer (see Figure 2.10). Steps are defined as y-crossings in



(a) A DEM point (r) was found during the backtracking, that will be used to compute the result for (p) with the LOS method

(b) The backtracking found no point to compute a LOS from, so the points $(r1)$ and $(r2)$ will be used to compute the result for (p) with the wave front method

Figure 2.10: Illustration of backtracking in Izraelevitz' algorithm in octant I with backtrack order 4. (O) is the observer position, (p) is the point to compute the result for, (k) is how far back the backtracking will go with backtrack order 4 and (r) or, $(r1)$ and $(r2)$ are the point or points that the algorithm will use to compute the visibility of the (p)

octants I, IV, V and VIII, and x-crossings in octants II, III, VI and VII (see Figures 2.7 and 2.2b). If the line passes directly over a DEM point, p_0, within those steps the visibility of p is computed as a LOS originating from p_0 with p_0's already computed required height above ground to p (see Figure 2.10a). If the line does not pass directly over any DEM point within the number of steps defined by the backtrack order, the visibility of p is computed like in the Xdraw algorithm using two reference points (see Figure 2.10b). This method reduces the approximation error that the inappropriate dependencies of the wave front method cause by computing some points with the LOS method and thereby stopping the propagation of inappropriate dependencies[4].

2.3 CPU Parallelism

There are two main ways of parallelizing programs on the CPU: one can distribute the workload of the program over several cores (assuming that the processor has more than one core) and one can use the processors SIMD vector units that executes operations on several data elements at a time, sometimes referred to as *vectorization* [13].

Single Program, Multiple Data (SPMD) is an execution model for parallel (or concurrent) programs. In the SPMD model there are several instances of the same

program running in parallel working on different data [13]. The model is therefore often used for data parallel applications [14].

2.3.1 Multithreading

Most modern CPUs have multiple cores and in order to make use of multiple cores in a program the program workload has to be distributed over multiple threads. There are a few different ways of doing this. One can either divide and distribute the work manually using threads, or one can use tools and frameworks such as OpenMP [13] or Cilk [15, 13]. Using multiple cores is more effective when the workloads for each thread are large and independent, as synchronization and communication between CPU cores is rather slow [15].

Hyper-threading is a term defined by Intel for running two threads on one CPU core by letting the threads share some of the hardware resources of the core. This can be beneficial in some cases, but not always. Performance gains of running two threads on one core is not doubled as the threads will compete for the same hardware resources [15]. Benchmarks of server applications have shown performance gains of up to 30% when using hyper-threading on Intel Xeon processors [16].

2.3.2 SIMD Instructions

Single Instruction, Multiple Data (SIMD) is a kind of instructions that performes some operation on several data elements at once [15]. Today's desktop processors usually have SIMD units and some instruction set extension such as SSE or AVX with SIMD instructions [15]. SIMD instructions operate on special vector registers of sizes typically between 64 and 256 bits on todays CPUs. SSE, for example, uses 128 bit registers and AVX uses 256 bit registers [15]. The notion of *SIMD lanes* is often used to denote the indices for different elements in a vector register that are being processed with SIMD instructions and often, but not always, one counts how many 32 bit items can be operated on at the same time. SSE, which operates on 128 bit vectors, can perform 4 32-bit operations with one instruction [13].

These vector operations are suitable for computations on larger data sets where the same computation needs to be repeated many times for different data points, if the computations are independent and can be computed in parallel. Fields where SIMD instructions are often used are audio processing, and mathematical vector and matrix computations [15].

2.3.3 SPMD on SIMD and the ISPC Compiler

One way of implementing the SPMD model is by mapping program instances to SIMD lanes. In this way one can run multiple instances of one program in parallel in one thread. The *Intel SPMD Program Compiler* (ISPC) works in this way. It compiles a language (also called ISPC), that is based on C99. ISPC has also borrowed some features from C++ and introduced some features of its own for parallelism such as special parallel foreach loops and vector data types.

The foreach statement is the primary parallel feature of ISPC. It is similar to a regular for-loop in the sense that it performs the same block of code several times

with some index variable that is different each time. The difference between ISPC's foreach statement and a regular for-loop is that the semantics of the foreach loop does not specify any order in which iterations are executed. The compiler then distributes the loops workload over program instances so that multiple iterations are being executed in parallel.

Using SIMD instructions to execute several program instances in parallel is not a trivial task, and there are a few things that can cause problems, primarely two things; branching and memory operations (loads and stores). ISPC solves these problems, but they may cause performance issues in ISPC programs when these issues are not considered or are unavoidable.

Branching statements, such as if statements, switches, and loops can be one source of performance loss is ISPC programs. This is due to the fact that if program instances take different branches in a branching statement, the different branches that potentially (and most likely) do different things and cannot be executed in parallel with SIMD instructions as the very definition of SIMD is that the instructions do *the same* operation with several different operands. ISPC solves this by executing one branch at a time using an *execution mask*: a flag for each program instance that says if the program instance is active or not, in order to activate and deactivate program instances in the branches that specific program instance take and do not take.

Loads and stores (reads from, and writes to RAM) performed by program instances using SIMD vectors are called *gathers* and *scatters*. A gather or scatter usually access the same address or coherent addresses in memory, called coherent gathers or scatters, e.g. when each program instance read a constant variable or when the program instances read from or write to consecutive indices in an array. SSE and AVX provide vector load and store instructions, but only for data that is contiguous in memory. Gathers and scatters of data that the compiler cannot guarantee will be contiguous in memory have to be compiled to sequences of load or store instructions that load or store one vector element at a time[13].

2.4 GPU

The GPU is a processing unit mainly for computing graphics. Graphics computations are often parallel, and thus the GPU's architecture was been developed to reach as good parallel computational ability as possible. To achieve this, the hardware is built in hierarchies to split up workloads many times to enable parallelism. OpenCL is a a framework that can be used to target the graphics processing unit of a computer to use its capabilities for other things than just rendering. This can be referred to as general-purpose programming on graphics processing units (GPGPU). Code written for OpenCL can also be run on CPUs, and OpenCL is supported on multiple vendors, including Nvidia, AMD and Intel [17]. The following subsections will use OpenCL terminology. CUDA is another similar framework that supports GPU computing, which only targets Nvidia GPUs [18].

2.4.1 Kernel Execution

The smallest executing unit on the GPU is called a *work-item*. Each work-item executes a *kernel*, which is a function that will be executed on the GPU. Each work-item is a part of a *work-group* executing on a *compute unit*, and each work-item has a *global ID* and a *local ID* to distinguish them from each other. The global ID is derived from the number of global work-items that were specified when executing the kernel, and the local ID specifies a work-item ID within a work-group. Both global and local IDs are N-dimesional and each index begin at 0. [12].

An OpenCL *device* has one or more compute units, where each work-group is executed on the same compute unit. Each compute unit contains one ore more *processing elements*, and may also include dedicated *texture filter units*. A processing element is a virtual scalar processor, and each work-item may be processed on one or more processing elements. The work-items in a work-group will be executed in terms of *wave fronts* in AMD terminology [19] or *warps* in NVIDIA terminology [20]. The size of a wave front varies with hardware, but usually has the size 32 or 64. The size of a warp also varies with hardware, but is often 32 on modern hardware. The GPU executes other warps when one warp is paused or stalled, and according to NVIDIA, this is only way to hide latencies and keep the hardware busy [20].

A *command-queue* is used to queue commands that will be executed on a specific device. Several command-queues can be used to achieve concurrency and in some cases parallelism by overlapping transfers and computations [21].

2.4.2 Memory

Five different memory types are present on the GPU which can be used for different things to increase performance. The memory type with the most space is the *global memory*, which is used to pass data from host to device. This memory is accessible by all work-items executing in a context.

The constant memory, which is a region of the global memory, can also be used for this. The data transferred to constant memory must stay constant, and can only be read. The constant memory is a cached memory, and reads from constant-memory will be performed faster for work-items in the same work-group than global memory.

Texture memory is also a region of global memory, but has caches that is tweaked for two dimensional spatial locality. Images and textures is well suited for this memory. *Samplers* is used to describe how to sample from an image within kernels.

Local memory is only accessible by work-items in the same work-group and barriers can be used to perform synchronization between work-items in a work-group to ensure that writes or reads from local memory is executed in a synchronized matter.

Private memory is a region of memory private to a work-item, and can is not visible to other work-items. Variables placed in private memory will use registers if available, and is the fastest memory level. The number of accessible registers depend on the number of work-items that was specified when executing the kernel.

2.4.3 Programming Models

OpenCL supports two programming models, a data parallel and a task parallel programming model. The data parallel programming model defines sequences of instructions that is applied to multiple elements [20]. The task parallel programming model instead launches individual instances of kernels and parallelism is achieved by using vector data types or queueing multiple tasks.

3
Approach

The R2 algorithm was chosen for both the GPU and CPU implementations for three reasons:

- The R2 algorithm has the complexity $\theta(n^2)$ which according to Franklin et al. [2] is the lowest possible complexity order for a viewshed algorithm working on raster DEMs, since the raster DEM has a number of data points proportional to n^2 and some computation has to be performed for each data point.
- The R2 algorithm has a relatively small approximation error.
- The algorithm is easy to parallelize; as each LOS computation is independent from the other LOS computations, meaning they can be computed in parallel.

For measuring the approximation error of R2 an implementation of R3 was also implemented to act as ground truth.

To simplify the implementation of R2 and to deal with the fact that the raster type the implementations work with has to be rectangular we define the boundary of R2 as the square with the side $2r$, where r is the range of the viewshed to compute, and the observer position in its center (see Figure 3.1). No visibility computations are made for the raster cells that are out of range, but are inside the square boundary. Instead, a marker value indicating that the position is out of range is written in the result.

To parallelize R2 each LOS is given an index from which its target point in the square boundary can be computed. In this way the result of one LOS can be computed independently from all other LOS computations on either the CPU or the GPU.

The implementations of R2 uses the method suggested by Franklin et al. [2] for deciding which LOS computation that should write the result for a specific DEM point that is presented in 2.2.2. The implementations of R2 and R3 use Formula (2.1) for compensating for earth curvature.

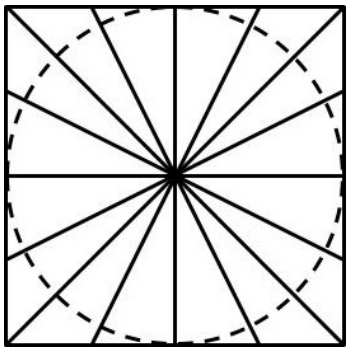


Figure 3.1: Illustration of how the sight lines are directed at the square border around an observer with some given range

3.1 GPU Implementations

The GPU implementations were made in OpenCL. We chose OpenCL for two reasons: compatibility with hardware from different manufacturers, and our previous experience and knowledge of OpenCL.

3.1.1 R3

This algorithm is primarily used as a ground truth for measuring the approximation error of R2. The R3 algorithm was implemented in OpenCL with one work-item per cell in the result raster and thus one line of sight computation per work-item. Each work-item steps in the way described in chapter 2.1.2, towards its target cell, and writes its result to the target cell once it has been reached. When stepping, each sample in the DEM is performed by linear interpolation to get a more accurate interpretation of the input data. This interpolation is performed in texture units on the GPU and is thus faster to perform on the GPU than on the CPU, which has to do the interpolation manually [22]. The interpolation method used by these texture units is bilinear interpolation, but the samples will always be located between two sample points as described in the algorithm earlier, so even if bilinear interpolation is used on the GPU, a linear interpolation on the CPU will have the same result.

3.1.2 R2

As the R2 algorithm sends sightlines to the border of the raster, and the GPU work-items can only differ with its indices [20], the kernels needs a way to figure out its index on the border. As we do not want to launch kernels with two dimensional indices, as only the indices along the border is needed, a one dimensional index is easier. This one dimensional index is then used to compute the two dimensional index in the border.

The height data is passed to the GPU as an image to make sure that the height data is stored into texture memory and that lookups from this image are cached. Reads from the raster from within the kernel is done with a sampler that is configured to use bilinear interpolation to get as accurate height estimations as possible for points on or between any points in the DEM. The raster space is cached which can be exploited to get a performance increase if the work-groups are arranged for spatial locality [20].

The size of work-groups for this algorithm are chosen to be 6 multiples of 32 to get 6 warps in each work-group that can be switched for hiding memory latencies. This also allows 192 work-items to access the same region of local memory that can be used for values that would otherwise have been frequently read from the global memory. The work-groups are grouped in the same order as they are indexed in the border to make sure that as many texture lookups as possible within a work-group are done close. Each work-group should share as much data as possible in local memory to minimize the number of reads from global memory, and as the viewshed problem has a lot of arguments, these will be copied from global memory to local memory by one work-item in each work-group before any work-item continues with computations.

3.1.3 Hiding API Call Overheads

Calls to the GPU can either be blocking or non blocking [12]. If the call is blocking, some overheads will be added for the command to reach the GPU, for the command to be fully executed, and for a response to be sent back to host. In order to hide these delays, or overheads, a command-queue was used to queue commands to the GPU, that does not block and does not wait for any result. Actions that get queues can have dependencies on other previously queued actions, to create a chain of dependencies that ensures that a result from a previous action is finished before another action uses that result. To actually get the result from a series of queued actions, a last action can be queued that blocks until it's processed, and that action can either be the read from GPU to CPU or even an "empty" action that only waits for the read to be finished. With the command-queue, the GPU can pull actions from a queue on the GPU instead of being idle between issued actions from the CPU.

3.1.4 Minimizing Allocations

If multiple calls to a viewshed algorithm is made with the same range and DEM resolution, the same amount of GPU memory will be allocated and released over and over again. To take advantage of this fact, the allocated GPU space can be kept instead of released, and reused if the algorithm is used with the same range and DEM resolution multiple times in a row.

3.1.5 Hiding Transfer Overheads

A GPU computation can be abstracted into three larger parts: data transfer to GPU, computation of the result, and transfer of the result from GPU to host memory.

The transfer from and to the GPU and kernel execution can be partially overlapped by splitting the computation into parts. By transferring only a part of the input data at a time, the GPU can begin execution once the first part of input data has been transferred, at the same time as the next part is being transferred. This can theoretically be implemented using any number of queues, but it's not practical to do it for more than four splits for the viewshed problem because of two major reasons: every sightline has to be located in only one of these parts that the input has been split in for it to get access to all necessary data, and the shape of the split parts has to be rectangular to be able to efficiently transfer a sub-image of the original input to the GPU. Figure 3.2 illustrates a scenario where the input were split into two parts, showing that 50% of the transfer overheads can be theoretically hidden by using two command-queues and queueing the writes, reads and computations for each half of the problem to different queues.

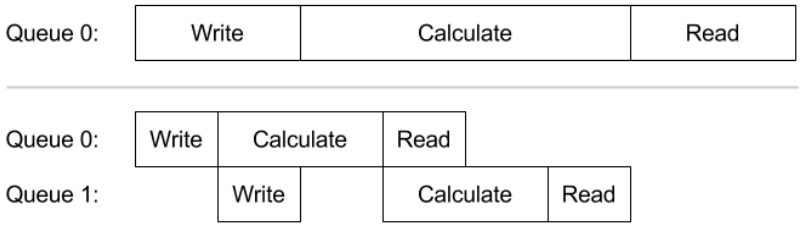


Figure 3.2: Visualization of possible speedup from queues by splitting transfers and kernel execution in half, and overlapping transfer overheads with kernel execution to achieve parallelism between the command-queues

3.1.6 Parallel Prefix Scan

The R2 implementation lets one work-item handle one sightline in the viewshed problem, but even the sightlines may need to be distributed over several threads if the GPU is still not saturated after such division of work. To do this, a parallel prefix scan algorithm can be used to compute the horizons and final results for each step along ~~the~~ a sightline in parallel.

The example by Blelloch that solves the line of sight problem in parallel cannot be used directly for the R2 algorithm, as the algorithm splits a sightline of length n into $n/2$ parts and uses one processor per part. In a viewshed problem on a raster containing 2000x2000 cells, the border will consist of 7996 cells which is also the number of sightlines that will be created. Each sightline will take 1000 steps, and with 500 processors per sightline, Blellochs approach would create about 4000000 threads to solve this instance. To use this approach on instances of viewshed instead of single instances of line of sights, Blellochs approach had to be modified. The proposed algorithm by Blelloch also works on a copy of the problem where the memory is modified each step, and in order to let multiple threads share the same area in memory on the GPU, the local memory needs to be used.

To allow fewer number of processors than $n/2$, each thread will instead treat a segment of a line of sight as a separate prefix scan problem, for which the last cell in the segment is of relevance to the next segment in the sightline. All segments needs to be in the same work-group to use the local memory and synchronize between each step to know that all writes were completed, and 4 segments with a workgroup-size of 4 would not even fill a warp, and would not allow multiple warps to hide memory latency. To get the worksizes up to 192, which fits 6 warps within a workgroup, 49 sightlines needs to be packed into the same workgroup. To store the all these sightlines in local memory, memory for 49000 floats would be needed to store all angles of all 49 sightlines, which requires a storage of $49kb$ to save the data. This already exceeds the maximum local memory size of GPUs with *compute capability 3.5*[23] and the space requirement will only increase when the input problem size increases.

To solve this, only the last step in each segment stores the computed prefix scan value to the local memory. The number of segments for each sightline is relatively small as they are chosen manually to increase the numbers of threads by a small multiple, and thus there will be even fewer steps in the up and down sweep as the number of steps is dependent on the number of segments. As these steps are so few, one of the threads in the sightline could instead handle the prefix scan of the elements stored in the local memory, and each segment only has to do one read from the local memory to get the relevant prefix-scan value from all previous segments to use for the final sweep in the segment. As the data for all steps in the sightline can't be saved in local memory, each thread handling a segment has to compute the angle again when doing the second sweep. See Figure 3.3 for an example of a line of sight solved by the parallel prefix scan implementation.

This implementation should have more impact on GPUs with more cores as it increases the number of work-items that may solve the problem in parallel. Smaller problems with this implementation may saturate the GPU earlier than the basic implementation and possibly execute faster, but only as long as all work-items that this algorithm produces can be run in parallel. This algorithm should not yield any performance increase once the GPU implementation without the parallel prefix scan algorithm i fully saturated.

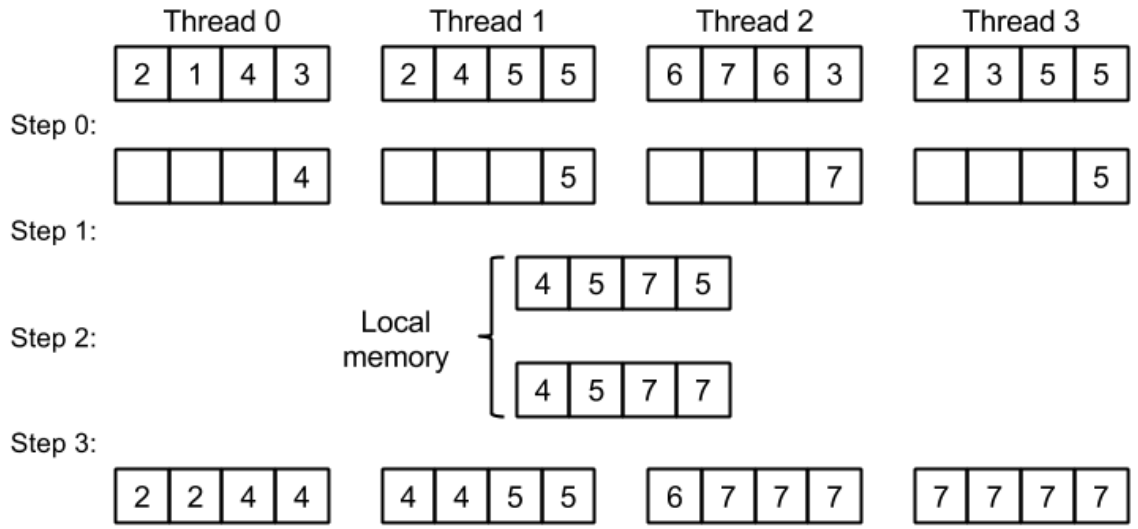


Figure 3.3: Image showing the parallel prefix scan algorithm using the *max* operator, computing the horizons for all steps of a sightline. Step 0: thread i computes local prefix scan for elements in segment i, but only saving the last horizon. Step 1: The last horizon is stored in index i of array in local memory. Step 2: thread i computes a prefix scan of elements in local memory. Step 3: thread i computes local prefix scan for segment i, but with value from local index $i-1$ as initial value. Segment 0 does not start with an initial value

3.2 CPU Implementations of R2

In order to test and evaluate the two different ways of parallelization on the CPU (see Section 2.3), four different versions using threads and SIMD instructions were developed: one with no parallelism, one that distributes the work over threads, one that uses SIMD instructions and one version that uses both threads and SIMD instructions.

The parallel versions of the R2 algorithm was implemented as SPMD programs that compute multiple lines of sight in parallel. The scalar (non-vectorized) implementations were written it C++ and the vectorized implementations were written in a combination of C++ and ISPC. The two dimensional indices of the border cells are computed from the one dimensional index of the each LOS computation in the same way as in the GPU implementation.

Multithreading was implemented in C++ using semaphores and threads and is used to distribute the LOS computations over multiple cores. We chose to do this without any previously mentioned frameworks (OpenMP and Clik) as only the most basic functionality was needed such as creating, pausing and resuming threads. Vectorization is done using ISPCs foreach loops and is used within the threads (or on its own) to compute several lines of sight in parallel with SIMD instructions on

each core that is used.

Since the CPU does not have texture units like the GPU, linear interpolated reading from DEM had to be implemented in software for the CPU versions of R2. When the R2 algorithm performs a lookup in the DEM, two DEM points will be read and interpolated between.

3.2.1 Memory Issues With R2

In RAM, a raster DEM is stored as a 1 dimensional array where the rows of the DEM are laid out after each other, so index 0 of a such an array contains the first height value of the first, and index w, where w is the length of a row, contains the first height value of the second row.

The way that the R2 algorithm traverses a DEM is in *sightlines* from the center and out towards the border. When several lines of sight are computed in parallel the sightlines in octants I, IV, V and VIII will spread out over different rows in the DEM. Since elements in the columns of the DEM are not contiguous in memory, ISPC is not able to compile the parallel reads of DEM points and writes of result values to coherent gathers and scatters. And since the code reading from the DEM is the same code no matter where in the DEM the read is done, and vice versa for the writes, all gathers from the DEM and scatters to the result in the vectorized implementation will be incoherent.

4
Result

The results in this chapter were produced on a computer with 8GB ram, an Intel i7 3610QM @ 2.3 GHz processor with the AVX instruction set extension and a NVIDIA GTX 660m graphics card. All examples were run on a DEM with 10 meter resolution over the area around Lake Tahoe, California/Nevada, USA. The observer is assumed to stand in the middle of a cell in the DEM. Each measurement using range can be converted to raster resolution by doubling the range to get a diameter, dividing by 10 meter as a DEM with 10 meter resolution were used, and then adding one since the observer is in the middle of a cell and the range is extended in opposite directions. With a range of 10000 meter, a raster of 2001 x 2001 pixels would be used for the viewshed computation. The max range used in a few examples is 80000 meter which would result in a 16001x16001 raster.

The performance tests were performed by starting with zero range, going to 80000 meter with steps of 1000 meter. For each step, 30 viewshed computations were made and the time was measured and averaged to get consistent results. The error measurements were made by performing 100 viewshed computations with 10000 meter range, evenly spaced over 51x51 km.

The implementation of the R3 algorithm made the driver for the graphics card crash due to being locked for longer that two seconds when reaching a range of 15000 meter. An estimation of R3:s performance above 15000 m range was done with polynomial regression (of degree 3 since R3 has the complexity $\theta(n^3)$). The performance comparisons between the R2 and R3 are not meant to affect the CPU vs GPU suitability discussion for the viewshed problem, but only to point out the importance of the choice of algorithm.

To get a better perspective of each optimization that was made, the CPU and GPU optimizations are first presented separately and then compared at the end, together with some error measurements.

The GPU version mentioned in this chapter as the basic GPU implementation is an implementation without the optimizations mentioned in 4.2.

All presented performance gains in the text of this chapter was measured at the range of 80000 meter which corresponds to a 16001x16001 raster on a DEM with 10 meter resolution.

4.1 CPU Implementations of R2

In this section the performance results for the different CPU implementations of the R2 algorithm produced. Comparisons between different levels of parallelization will

be presented as well as the impact of using linear interpolation.

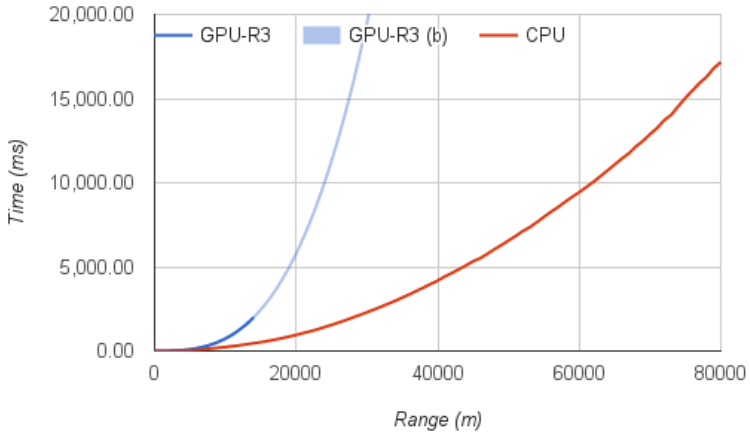


Figure 4.1: Performance comparison between the R3 algorithm (*GPU-R3*) implemented for the GPU, and the sequential implementation of the R2 algorithm (*CPU*). *GPU-R3 (b)* is a polynomial regression of *GPU-R3* estimating how the curve for the R3 algorithm would continue

In figure 4.1 we can see the performance of the R2 algorithm (with $\theta(n^2)$ complexity) for the CPU implemented in C++ with no parallelism in comparison to the R3 algorithm (with $\theta(n^3)$ complexity) for the GPU implemented in OpenCL. The brighter blue curve is the polynomial regression of R3.

4.1.1 Multicore Parallelism

Figure 4.2 shows a performance comparison between the completely sequential implementation and the implementation using only threads for parallelism. Four instances of the multithreaded implementation is shown using 1, 2, 4 and 8 threads. As seen in the graph, the sequential implementation *(b)* has almost exactly the same performance as the multithreaded one when using only one thread *(a)*. Distributing the workload over two threads gives approximatly a 1.9 times performance increase, four threads gives approximatly a 2.8 time performance increase and eight threads (in order to use all cores and hyper-threading) gives approximatly a 3.0 times performance increase compared to the sequential implementation.

4.1.2 Vectorization

The implementation that is parallelized using vectorization is shown in figure 4.3 compared to the sequential implementation. Using SIMD parallelism gives approx-

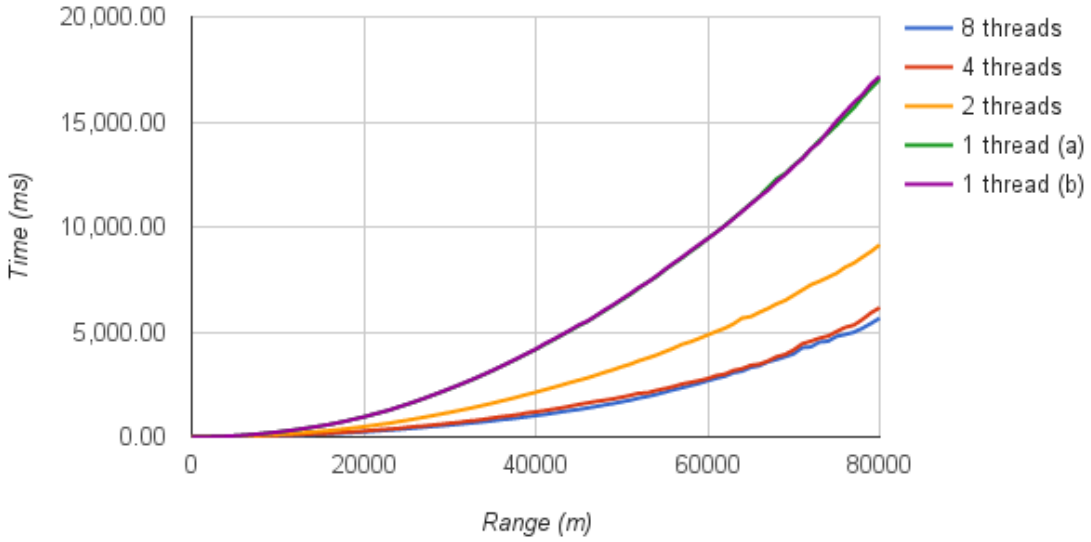


Figure 4.2: Impact on performance of multicore parallelism. Here we see the performance of the sequential implementation *(b)* compared to four instances of the threaded implementation with different numbers of threads running

imatly a 2.6 times increase in performance compared to the scalar implementation.

4.1.3 CPU Summary

Figure 4.4 shows a compilation of how multicore and SIMD parallelism as well as the combination of the two affect the performance of the R2 algorithm running on the CPU. Using both multicore and SIMD parallelism gave the CPU implementation a 9.4 times increase in performance in comparison to the implementation using no form of parallelism.

Another aspect of the implementation that affects performance is the choice of interpolation method. Because the linear interpolation used by the CPU implementations is implemented in software and is used many times by the algorithm it has a certain performance cost. See Figure 4.5 for a comparison of a CPU implementation with linear and nearest neighbor interpolation. The graph shows a 14% increase in performance when changing from linear to nearest neighbor interpolation. Changing interpolation method will however affect the result which may be undesirable, therefore will all performance comparisons in the report but this one consider CPU implementations that use linear interpolation.

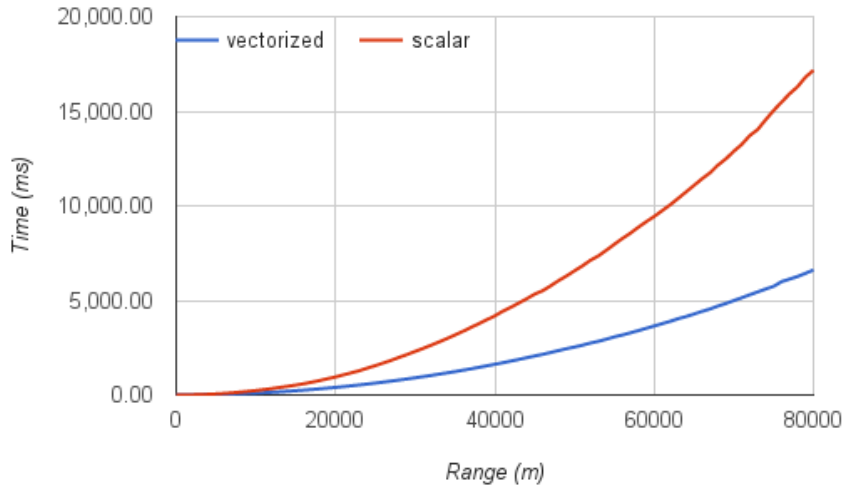


Figure 4.3: Impact on performance of vectorization. In this graph we see the sequential implementation compared to the vectorized version implemented partly in C++ and partly in ISPC targeting the AVX instruction set

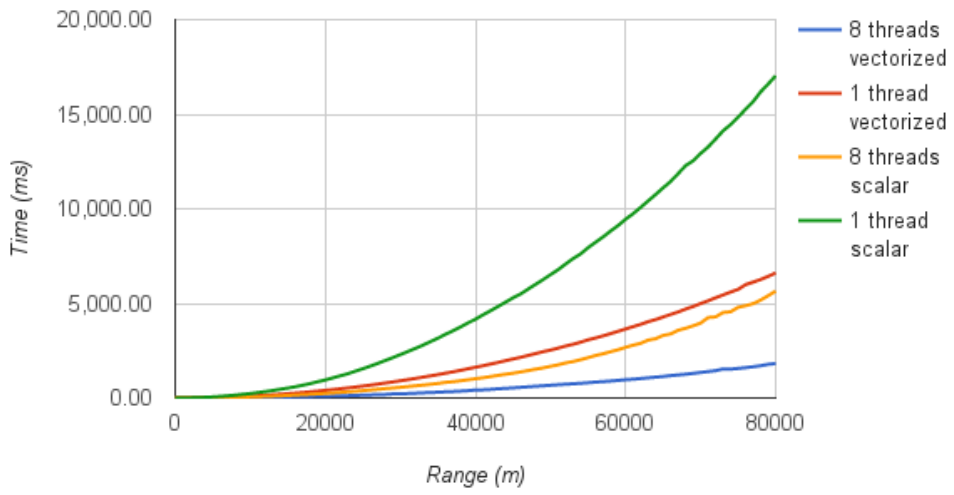


Figure 4.4: Impact on performance of multithreading and vectorization in the CPU implementations

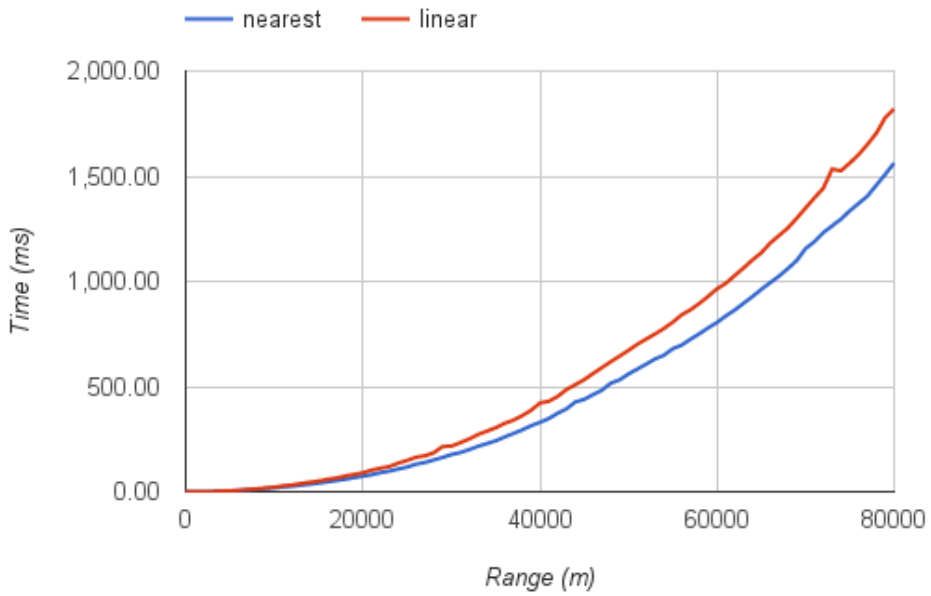


Figure 4.5: Impact on performance of using linear interpolation compared to nearest neighbor interpolation on the CPU

4.2 GPU Implementations of R2

Figure 4.6 show the performance of the R3 algorithm and the R2 algorithm, both implemented on the GPU. As the graph shows, R2 scales better with the input compared to R3. This follows our expectations as the number of sightlines traversed by R3 is of complexity $\theta(c * n^3)$ while R2 is $\theta(c * n^2)$, where c is affected by the parallelism.

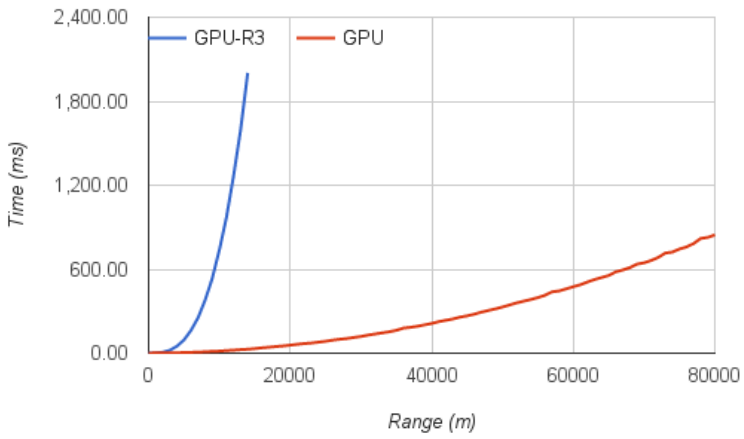


Figure 4.6: Comparison between GPU implementations of R3 and R2

To further improve the R2 implementation on the GPU, the implementation was split into 3 bigger parts that can be addressed in order to gain performance; Allocation, transfer and computation. The results of these optimizations will be discussed in the following subsections.

4.2.1 Minimizing Allocations by Reusing Memory

The rests for GPU implementation of R2 that reuses previously allocated GPU space to minimize the number of new allocations required is shown in figure 4.7 compared to an implementation that does not reuse memory. As the tests perform 30 runs of the algorithm at each range and then averages the runtime, the algorithm will only reallocate the space the first time at each range and then keep reusing it until the range changes and more space needs to be allocated.

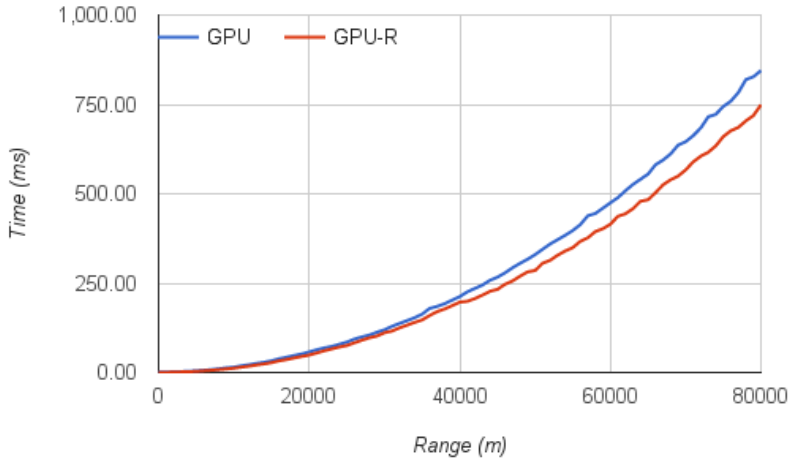


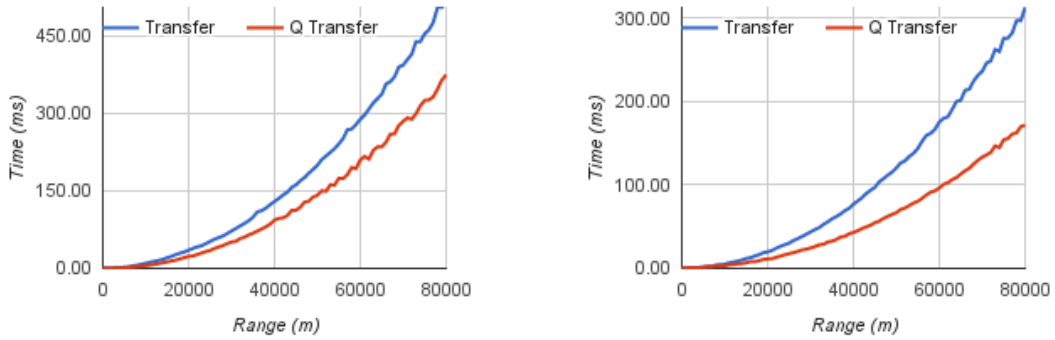
Figure 4.7: Impact on total run time of reusing previously allocated memory. *GPU-R* keeps and reuses memory when possible while *GPU* always allocates new GPU memory at each run and releases it afterwards

4.2.2 Hiding Transfer Overheads Using Multiple Command-queues

To deal with transfer overheads, multiple queues were used to be able to start the kernels before all of the data was completely transferred to the GPU, effectively hiding transfers behind computation.

To get a closer look what actually happens, figure 4.8a illustrates only the transfer time, with and without the multi queueing optimization.

As Figure 4.8a indicates, the results do not get a 50% reduction of transfer time which was the aim from the theory chapter, and Figure 4.9a shows why. The computation is not taking enough time to hide the reads from the GPU. If the computations had been more time consuming, or if less data had been transfered back from the GPU, more of the read could have been hidden by the computations. To try this out, another test was performed where the data type of the result was changed from a float raster to a short raster, i.e. 16 bits per raster cell instead of 32 bits per raster cell. The performance gain seen in Figure 4.8b is the result of half of the result reads being hidden behind the kernel computations, which is also confirmed by Figure 4.9b. The actual performance gain is about 45% of the measured transfer time, which is an improvement to previous 27% and quite close to the theoretical maximum of 50%. Even though this yeilds a better performance increase, 32 bit floats per raster cell will be used through the result chapter to maintain the a high accuracy for the algorithm.



(a) 16 bit inputs, 32 bit outputs **(b)** 16 bit inputs, 16 bit outputs

Figure 4.8: Impact on exposed transfer time by hiding parts of the transfer using double command-queues (*Q Transfer*) compared not hiding any transfer time (*Transfer*)

(a) 16 bit inputs, 32 bit outputs **(b)** 16 bit inputs, 16 bit outputs

Figure 4.9: Visualisation of how data transfer is hidden behind computation (see also figure 3.2). Green boxes are transfer and red boxes are computation. The images was created with NVIDIA NSight [24]

4.2.3 Dividing Computations Using Parallel Prefix Scan

To be able to saturate the GPU at shorter ranges, a parallel prefix scan algorithm was used to compute each sightline in parallel. Figure 4.10 shows that the number of segments does not seem to be affecting the performance, and compared to the implementation without the parallel prefix scan implementation Figure 4.11 shows that increased parallelism do not decrease the computation time even in smaller ranges. This is probably because the number of sightlines in the viewshed problems is relatively high even in smaller ranges, which saturates the GPU that it was tested on early. For further discussion on the performance of the parallel prefix scan argorithm, see Section 5.3.

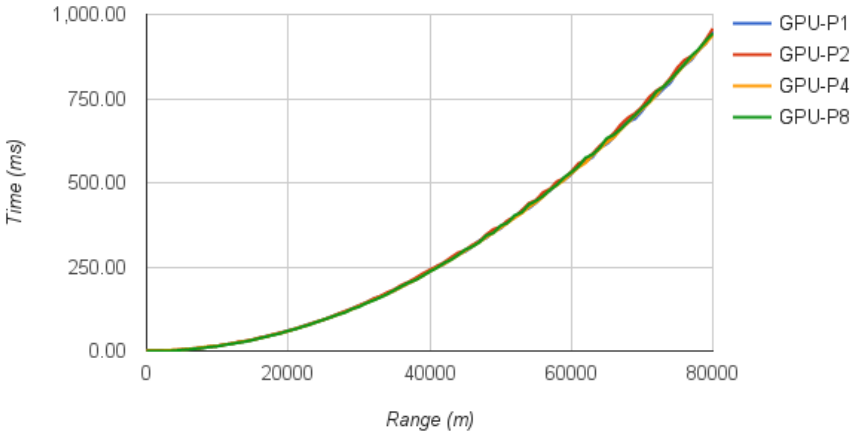


Figure 4.10: Different variations of the parallel prefix scan implementation where *GPU-Px* is the GPU implementation using parallel prefix scan with x segments

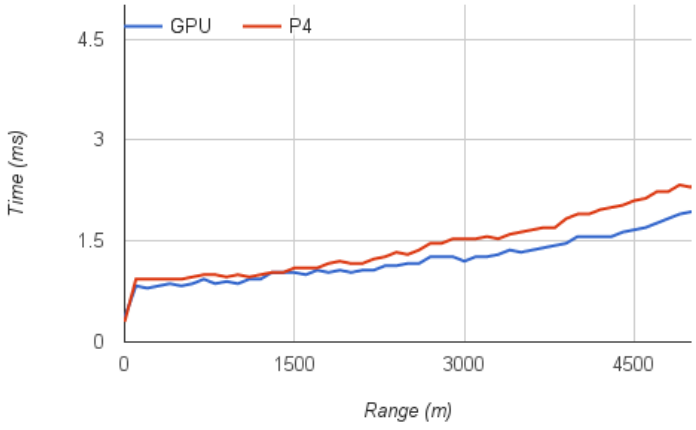


Figure 4.11: Impact on computation time of parallel prefix scan implementation where *P4* is the parallel prefix scan implementation with 4 segments for each sight-line and *GPU* is the basic GPU implementation

4.2.4 GPU Summary

Finally all GPU optimizations are shown together in Figure 4.12. The allocation and queue implementations show a 11.3% and 16.3% increase in performance over the basic implementation. The combination of these implementations increase the performance by 30% while the parallel prefix scan implementation slightly decreases the performance overall.

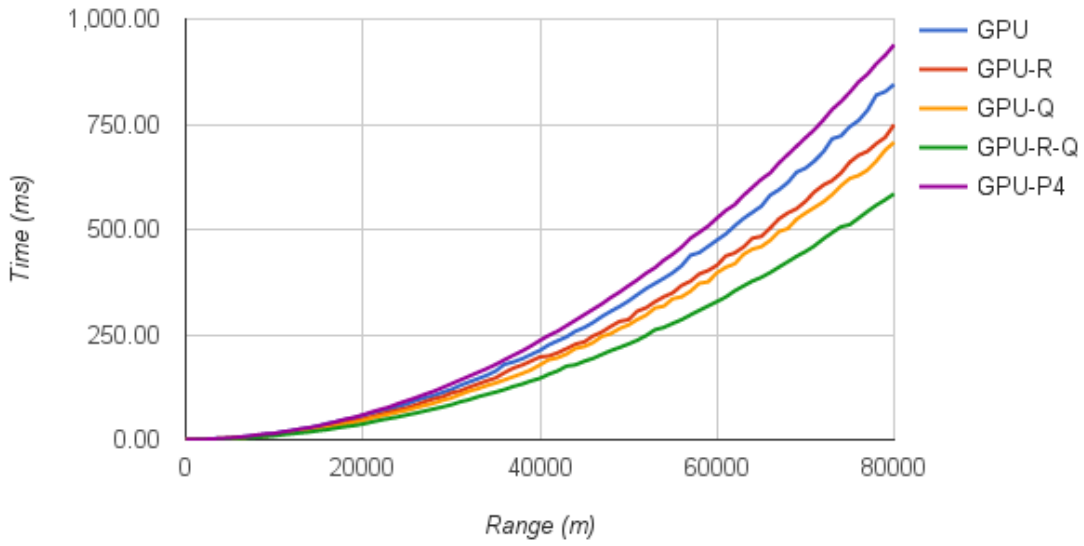


Figure 4.12: Comparison of different GPU optimizations where *GPU* is the basic implementation, *GPU-R* reuses memory, *GPU-Q* uses two command-queues, *GPU-R-Q* both reuses memory and uses two command-queues, and *GPU-P4* uses the parallel prefix scan implementation with 4 segments per sightline

4.3 Performance CPU vs GPU

The effect of parallelism can be see in Figure 4.13, which reveals a performance gain of 9.4x to 29.3x compared to the non-parallel CPU implementation.

The basic GPU implementation has a performance gain of 20.3x compared to the non-parallel CPU implementation and a 1.9x compared to the multithreaded, vectorized CPU implementation, and the GPU variant that both reuses the GPU memory and uses two queues reach a performance improvement of 29.4x compared to the non-parallel CPU implementation and 3.1x compared to the multithreaded, vectorized CPU implementation.

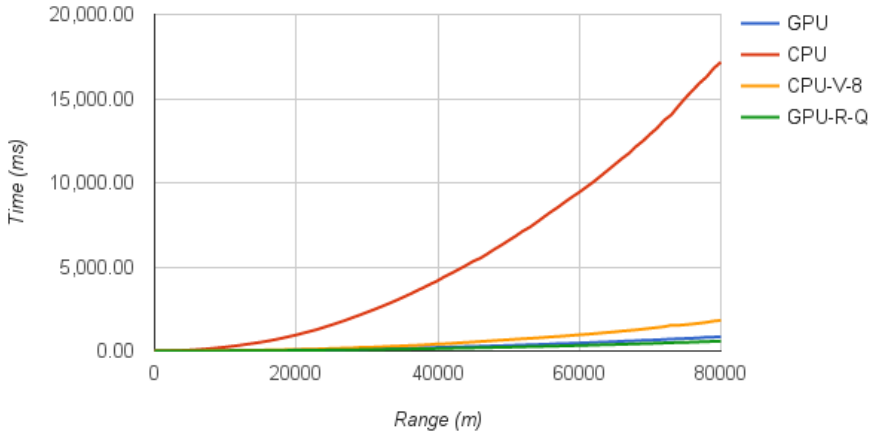


Figure 4.13: Effect of parallelism. *GPU* is the basic GPU implementation, *CPU* is the non-parallel CPU implementation, *CPU-V-8* is the both vectorized and multithreaded (8 threads) CPU implementation and *GPU-R-Q* is the GPU implementation that reuses GPU memory and uses two command-queues

The final performance comparison between the fastest of the CPU implementations and the fastest of the GPU implementation can be seen in Figure 4.14. The GPU implementation with memory re-usage and two command-queues compared to the CPU implementation that is both vectorized and multithreaded yields a 3.1x performance gain for the GPU implementation.

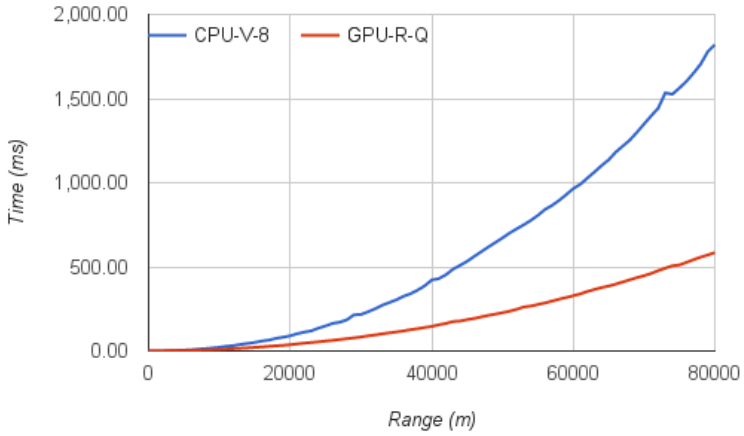
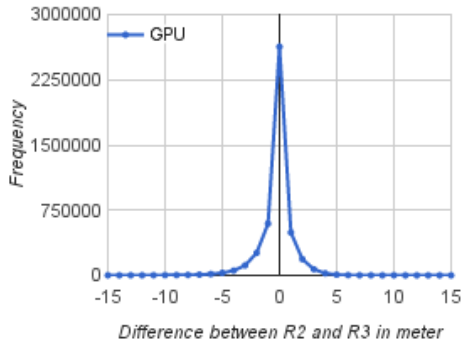


Figure 4.14: Comparison of between the vectorized multithreaded (8 threads) CPU implementation (*CPU-V-8*) and the GPU implementation that reuse memory and uses two command-queues (*GPU-R-Q*)

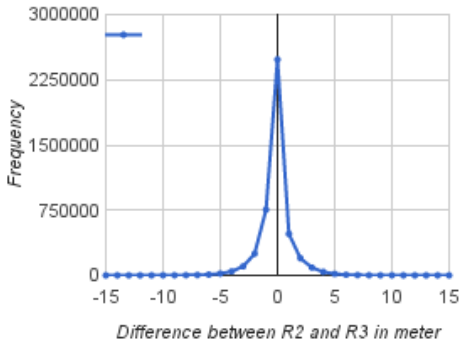
4.4 Approximation Error of R2

As the R2 algorithm is an approximation of the correct result, some measurements of how close to R3 it is would give some insight of how big this approximation error is. As both algorithms solve the extended viewshed problem, measuring the difference between the outputs of the algorithms will show how big the error is, and if there is some correlation of where the error happens. Figure 4.15 shows the frequencies of differences of different sizes between the R2 implementations and the R3 implementation. An optimal result would be that all differences were located at the center bin (meaning all differences are 0). Our results has the majority of the result in the center bin, and a small fraction in the bins closest to the one in the center, which shows that the error is small. The first and the last bin captures all differences that is lower than -15.5 meters and bigger than 15.5 meters. If our result is compared to Franklin et al., the results are very similar. The difference is that our error seems to be slightly more gathered around the center bin, which could be the result of different implementations of interpolation, ways of determining which sightline that is responsible for which cell, or the resolution of the DEMs used in the tests test.

Figure 4.16 shows the approximation error of the CPU implementation using nearest neighbour interpolation. Comparing it with Figure 4.15b one can see that the nearest neighbour interpolation gives a slightly lower peak in the histogram, and the histogram is somewhat skewed to towards more positive differences. That nearest neighbor interpolation gives a lower peak than linear interpolation is not



(a) Differences between the GPU implementation of R2 and R3



(b) Differences between the CPU implementation of R2 using linear interpolation and R3

Figure 4.15: Histograms showing the approximation error of the GPU (a) and CPU (b) implementations compared to R3.

very strange since the ground truth algorithm also uses linear interpolation.

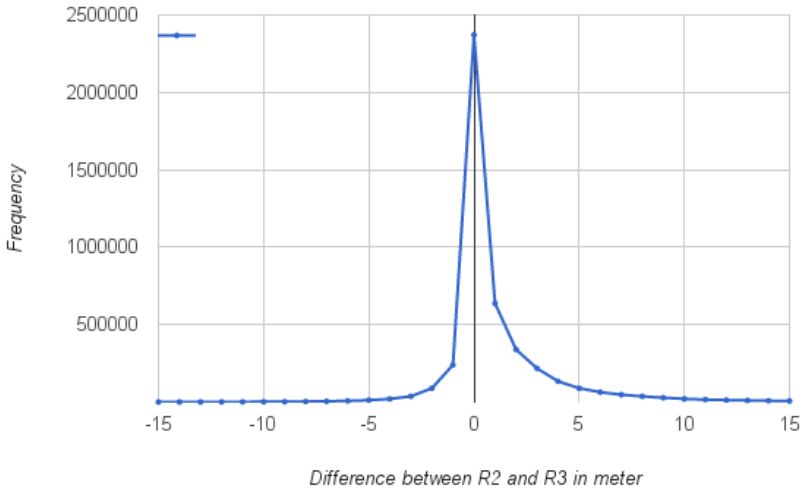
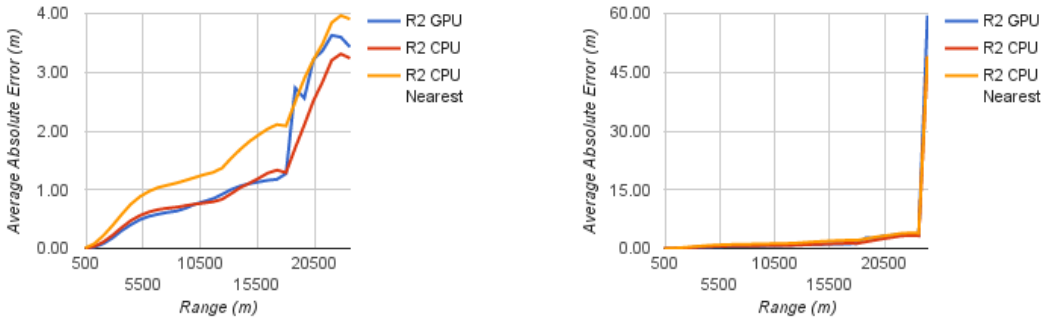


Figure 4.16: Differences between the CPU implementation of R2 using nearest neighbour interpolation and R3

Figure 4.17a shows that there is a correlation between the distance from the observer, and the average size of approximation errors. The values of the bins are

the averaged absolute values of the error at each distance.



(a) Error of CPU and GPU in different ranges compared to R3 with last two bins excluded

(b) Error of CPU and GPU in different ranges compared to R3 with last two bins included

Figure 4.17: Error of CPU and GPU in different ranges compared to R3

In Figure 4.17a, the values of the two rightmost bins were ignored, as the figure initially looked like in Figure 4.17b. This is probably because the two last bins is the borderline of where DEM points are close to the observers range. The R2 algorithm only guarantees that a cell gets written by the sightline that is closest to it's center, but the position of where it passes the cell can still be outside of the range while the target position is inside, see Figure 4.18 for an example. The value that was chosen to be written for cells that are out of range value was a very high value, which may explain the high average shown in the last two bins in Figure 4.17b.

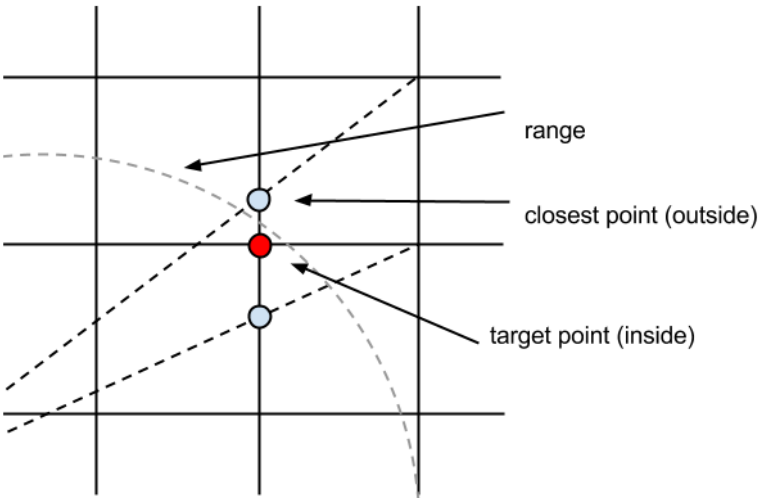


Figure 4.18: Illustration of a sightline from R2 that determines that a point on the sightline is further from the observer than the range, while the target point is inside the range

5

Discussion

5.1 Performance Comparisons

As the R3 algorithm is primarily used for measuring approximation errors of R2, it did not get as much attention as the R2 implementation in terms of optimizations to make it execute faster. The performance comparisons with R3 were made to point out the effect of algorithm complexity, and not to show any kind of exact performance increase.

Our result of 3.1x performance gain of the best GPU implementation over the best CPU implementation is close to the average performance gain of 2.5x for GPUs reported by Lee et al. [10].

5.2 Cache Efficiency in the R2 Implementations for the CPU

Caching may work a little differently for different sightline computations depending on octant and the direction of the sightlines in relation to the raster DEM, due to the way that the R2 implementation traverses the array containing the DEM. The sightline computations in octants II, III, VI and VII will step in y-crossings meaning they will read from different rows of the DEM in every step. This may cause caching to work less efficiently since elements on different rows are separated in memory. However the vectorized implementation that computes several sightlines in parallel will make its gathers from same row in each step which may be beneficial for caching, as the DEM points read are close to each other in memory. In octants I, IV, V and VIII the gathers performed in the sightline computations will read from columns in the DEM rather than rows which is probably not very cache efficient. However the sightline computations in those octants will step in x-crossings and will often run along the rows of the DEM which may be more cache efficient. This is only speculation and we have unfortunately not been able to measure how efficiently the implementations use caching and have not been able to verify that this really is how the implementation would use the cache.

5.3 Results of the Parallel Prefix Scan Implementation

The purpose of the parallel prefix scan implementation was to provide even more parallelism for the viewshed problem by letting multiple processors handle a single line of sight. If not enough resources exist to keep all work-items computing at the same time, the segments will be computed in sequence and the overheads for splitting the computation will only entail a performance decrease. As the parallel prefix scan implementation did not get any noticeable performance increase at any range, this extra parallelism seem to be unnecessary on the hardware it was tested on. The same conclusion was made by Xia et al.[25], even though they got greater performance decrease by using parallel sightline computation.

5.4 Power Consumption

It might be interesting to weight the results with power consumption, but the closest heuristic that we have for both devices is the thermal design power(TDP), which is a measure of heat that the device dissipates under relatively heavy workloads. Even though TDP does not directly reflect the amount of energy that the processors consume, they may give some sense of relativity between the CPU and GPU. Figure 5.1 shows the performance weighted by their TDP values which was 45 W for the CPU[26] and 75 W for the GPU[27].

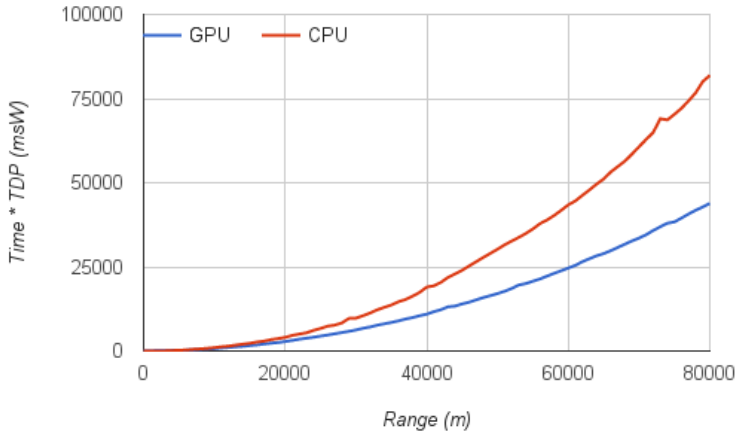


Figure 5.1: Performance measurements scaled with TPD

5.5 Conclusions about performance on other hardware

We were a bit limited in terms of hardware during this study, and the tests were not performed on other hardware than the one specified in the beginning of 4. As stated in 1.1, such tests are necessary to draw conclusions on other hardware, and thus we could not give more general performance estimations.

5.6 R2 and R3 out of Range Disagreement

The disagreement between R2 and R3 if a result from a point on a sightline is inside the range of the observer or not can be avoided. R2 could check if the cell currently being processed is within range or not, instead of checking for the point on the sightline that the algorithm actually stepped on. This was a behavior that was noticed after all benchmarks were performed and after the decision were made to not change the implementation further, but this change would probably increase the agreement between the R2 and R3 algorithms for cells that are close to the edge of the observer range.

6
Conclusions

Several implementations were made for both the CPU and the GPU to show the effect of some optimizations. The chosen algorithm was R2 which is an approximate algorithm. The approximation error of our implementations were measured by comparing the result of the R2 implementations with results from the R3 algorithm. The differences/errors are presented in the Figures 4.15 and 4.17a and we consider these approximation errors to be well within acceptable levels.

As the results show, the GPU implementation is faster than the parallel CPU implementation, with a speedup of about 3.1 times faster on a DEM of 16001x16001 data points. Both parallel CPU and the GPU implementations are preferable to the sequential CPU solution for the viewshed problem, where the GPU reach a 29.3x performance increase and the parallel CPU implementation reach a 9.4x performance increase.

On the GPU, which is massively parallel, a problem that can be split into many pieces that can run in parallel is crucial. The work-items that handle these pieces should cooperate when possible, making use of shared memory or caches. As the DEM in the viewshed problem can be seen as a texture, the texture memory on the GPU can offer hardware implemented interpolation methods when reading from the DEM and caches that are optimized for spatial locality. For the R2 algorithm, the input and output data is relatively small compared to computations, which is suitable for the GPU, and as parts of the viewshed problem can be computed in batches, parts of the transfers can be hidden by computations by using multiple command-queues. This problem is very parallel in general and fits the GPU very well.

As seen in the results, CPU implementations of the R2 algorithm benefit quite a lot from parallelism; vectorization, multithreading or a combination of the two. Due to how the R2 algorithm traverses the DEM, in sightlines from the observer position in the center and outwards, our vectorized implementation does have uncoherent gathers when reading from the DEM and uncoherent scatters when writing the result values. An algorithm/implementation that does not suffer from gathers and scatters would likely be faster, however, even if the implemetation has uncoherent gathers and scatters it still benefits from vectorization.

7
Future Work

7.1 Parallel Prefix Scan Using Global Memory

The intermediate result of the parallel prefix scan algorithm could be stored in global memory instead of being recalculated after the prefix scan has been made in parallel but due to the large amount of memory required for this operation, the choice was made to recalculate the angle instead to minimize the memory usage, and let the GPU do some extra computational work in order to increase the parallelism of the problem. An implementation that saves the result for use after the first scan in our implementation of the parallel prefix scan algorithm would be interesting to see in the future.

7.2 OpenCL on CPUs

Comparisons between viewshed algorithms implemented in OpenCL but run on a CPU, and handwritten vectorized and multithreaded implementations would be interesting to see.

7.3 Improving CPU Implementations

The CPU implementations have some memory related performance issues that we have not overcome. It would be of interest to look deeper into memory optimizations for parallel CPU implementations of R2 and possibly also other viewshed algorithms. The issue we know we have is incoherent gathers and scatters in the vectorized implementation due to how the R2 implementation traverses the DEM. At this moment we do not know if incoherent gathers and scatters can be be avoided in the R2 algorithm or if there is some other viewshed algorithm that can be parallelized with SIMD instructions without having incoherent gathers and scatters. This would be of interest to look further into as this most likely is one of the things that affect performance negatively in our implementation of R2.

It would also be of interest to further study and measure how to use caches effectively in a parallel implementation of R2.

7.4 Parallel Implementation of Izraelevitz' Algorithm

It would be interesting to see if Izraelevitz' viewshed algorithm could be implemented to run efficiently on parallel CPUs and how it would perform (considering both speed and approximation error) in comparison to R2.

7.5 Implementation Using Both CPU and GPU

It would also be interesting to see if a viewshed algorithm could be made that takes full advantage of both CPU and GPU for computations.

Bibliography

[1] Y. Gao, H. Yu, Y. Liu, Y. Liu, M. Liu, and Y. Zhao, "Optimization for viewshed analysis on gpu," in *Geoinformatics, 2011 19th International Conference on*. IEEE, 2011, pp. 1–5.

[2] W. R. Franklin, C. K. Ray, and S. Mehta, "Geometric algorithms for siting of air defense missile batteries," *A], Research Project for Battle*, no. 2756, 1994.

[3] J. Wang, G. J. Robinson, and K. White, "Generating viewsheds without using sightlines," *Photogrammetric engineering and remote sensing*, vol. 66, no. 1, pp. 87–90, 2000.

[4] D. Izraelevitz, "A fast algorithm for approximate viewshed computation," *Photogrammetric Engineering & Remote Sensing*, vol. 69, no. 7, pp. 767–774, 2003.

[5] Y.-j. Xia, L. Kuang, and X.-m. Li, "Accelerating geospatial analysis on gpus using cuda," *Journal of Zhejiang University SCIENCE C*, vol. 12, no. 12, pp. 990–999, 2011.

[6] N. Stojanovic and D. Stojanovic, "Performance improvement of viewshed analysis using gpu," in *Telecommunication in Modern Satellite, Cable and Broadcasting Services (TELSIKS), 2013 11th International Conference on*, vol. 2. IEEE, 2013, pp. 397–400.

[7] G. E. Blelloch, "Prefix Sums and Their Applications," *Computer*, pp. 35–60, 1990.

[8] S. Sengupta, A. E. Lefohn, and J. D. Owens, "A work-efficient step-efficient prefix sum algorithm," in *Workshop on edge computing using new commodity architectures*, 2006, pp. 26–27.

[9] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 205–213.

[10] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 451–460.

[11] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 134–144.

[12] Khronos Group. The opencl specification. Accessed: 2015-07-01. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[13] M. Pharr and W. R. Mark, "ispc: A spmd compiler for high-performance cpu programming," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–13.

[14] F. Darema, "The spmd model: Past, present and future," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2001, pp. 1–1.

[15] A. Fog, "Optimizing software in C++," pp. 1—-160, 2014. [Online]. Available: http://www.agner.org/optimize/optimizing_cpp.pdf

[16] S. D. Casey. How to determine the effectiveness of hyper-threading technology with an application. Intel Corporation. [Online]. Available: https://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application

[17] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. Accessed: 2015-06-18. [Online]. Available: https://www.khronos.org/opencl/

[18] NVIDIA Corporation. Parallel programming and computing platform | cuda | nvidia | nvidia. Accessed: 2015-06-18. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[19] Advanced Micro Devices, Inc. Amd accelerated parallel processing opencl programming guide. Accessed: 2015-06-18. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

[20] NVIDIA Corporation, "NVIDIA OpenCL Best Practices Guide," *Optimization*, vol. 181, no. 1.0, pp. 2175–2184, 2009. [Online]. Available: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[21] NVIDIA Corporation, "OpenCL Best Practices Guide," p. 54, 2010.

[22] D. Ruijters, B. M. ter Haar Romeny, and P. Suetens, "Efficient gpu-based texture interpolation using uniform b-splines," *Journal of Graphics, GPU, and Game Tools*, vol. 13, no. 4, pp. 61–69, 2008.

[23] NVIDIA Corporation. Programming guide :: Cuda toolkit documentation. Accessed: 2015-06-24. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

[24] NVIDIA Corporation. NVIDIA Nsight | NVIDIA. Accessed: 2015-06-23. [Online]. Available: http://www.nvidia.com/object/nsight.html

[25] Y. Xia, Y. Li, and X. Shi, "Parallel viewshed analysis on gpu using cuda," in *Computational Science and Optimization (CSO), 2010 Third International Joint Conference on*, vol. 1. IEEE, 2010, pp. 373–374.

[26] Intel Corporation. Ark | intel; core i7-3610qm processor (6m cache, up to 3.30 ghz). Accessed: 2015-05-28. [Online]. Available: http://ark.intel.com/products/64899/Intel-Core-i7-3610QM-Processor-6M-Cache-up-to-3_30-GHz

[27] Futuremark Corporation. Nvidia geforce gtx 660m review. Accessed: 2015-05-28. [Online]. Available: http://www.futuremark.com/hardware/gpu/NVIDIA+GeForce+GTX+660M/review