2013•2014
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: elektronica-ICT

Masterproef
Compoundly weighted Voronoi: a sequential and parallel implementation

Promotor :
ing. Dirk SMETS
dr. Kris AERTS

Promotor :
dr. R. WICHMAN

Vincent Boerjan
Masterproef voorgedragen tot het bekomen van de graad van master in de industriële wetenschappen: elektronica-ICT

2013•2014
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterproef
Compoundly weighted Voronoi: a sequential and parallel implementation

Promotor :
ing. Dirk SMETS
dr. Kris AERTS

Promotor :
dr. R. WICHMAN

Vincent Boerjan
Masterproef voorgedragen tot het bekomen van de graad van master in de industriële wetenschappen: elektronica-ICT

Acknowledgements

First and foremost it is a pleasure to thank the people who made it possible for me to write this thesis. I would like to show my deepest gratitude to Risto Wichman and Kris Aerts in particular. Without them I would never had the unique opportunity to work on my thesis in Aalto University. Their guidance made this thesis possible. And their knowledge was a source of inspiration. I would like to thank Dirk Smets as well for his feedback on the thesis.

It is a pleasure to thank my mother for always being there and helping whenever possible and my father who supported and advised me throughout my studies. My thanks also go out to Greet Raymaekers who helped me with all the paperwork and formalities for my Erasmus adventure.

A final word goes out to my friends and fellow students with whom I shared many great moments and who made my time in Diepenbeek and Helsinki memorable. It was a pleasure sharing my time as a student with them.

Table of contents

List of tables

List of figures

Abstract

SMARAD is the Smart and Novel Radios research unit at Aalto University in Helsinki. In the context of their smart radio research the area of influence of existing television transmitters is important data for the placement of experimental transmitters. Currently these areas are calculated with a regular Voronoi tessellation ignoring variation in transmitter characteristics. This thesis offers a more accurate generalised Voronoi implementation.

In this thesis the optimal construction method for a regional division with given characteristics is selected and implemented. This results in three different versions with a similar algorithm but on different platforms: Matlab, C and OpenCL. The Matlab version is the slowest but includes an additional API to map the result automatically to a topographic map of the target region using an internet source. The other implementations require an input image with the correct dimensions defined by the user to achieve the same results. The fastest implementation is the OpenCL version; however, cross-platform compatibility is limited. In this thesis the adaptive nature of the algorithm is also demonstrated by implementing different generalisations of the Voronoi tessellation to achieve a more accurate result for a given real world scenario.

The knowledge gathered from this thesis has applications for region division in signal processing as well as other scientific fields given the adaptive nature of the code. Examples of possible use include cellular and crystal growth, computer generated graphics and computational geometry.

List of abbreviations and symbols

API	Application Programming Interface
BMP	Bitmap image
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GPGPU	General-purpose computing on graphics processing units
IDE	Integrated Development Environment
I/O	Input/output
JPEG	Joint Photographic Expert Group
PNG	Portable Network Graphics
SDK	Software Development Kit

Introduction

Context

The purpose of this project is to make a computer based implementation of a generalized Voronoi plot, an improvement to the currently used regular Voronoi plots, to achieve a higher accuracy by adding two additional factors to the normal Euclidian distance formula used in regular Voronoi Tessellations. A proposed usage of this enhanced accuracy is the determination of the optimal placement for new transmitters in an existing radio wave network.

Research question

Regular Voronoi plots assume generator points of uniform strength and influence on the tessellation. When we take a look at a real life transmitter we know this assumption is not accurate, since not every transmitter has the same transmission power and path loss on the signal. To model this accurately we need at least two factors, an additive and a multiplicative factor. A problem with this multiplicative factor is that one small change in the dataset or an extra generator point can require a complete recalculation of the Voronoi plot rendering divide and conquer techniques useless and requiring a brute force approach. We need to find an implementation that performs better in such a situation.

Research goals

We can summarise all our goals in a short list:

- Implementation of a sequential method in Matlab
- Implementation of a parallel method
- Calling the parallel code from the Matlab implementation
- Comparing and selecting the optimal implementation

Method

To build the Voronoi diagram we must first study known algorithms and their limitations. The multiplicative factor in the problem greatly limits the possible algorithms and even standard approaches to numerical computing problems might prove unfeasible.

Since Matlab uses only one processor core we are restrained to single threaded sequential computing. This limits the possible performance of the application. Therefore we will examine different approaches of parallelising the application. First we take a look at the different concepts for parallelism to determine the most efficient way to parallelise our program. The next step is deciding the platform, the major candidates being CPU, GPU and FPGA.

After having chosen the platform we have to decide how we are going to address that platform. There are many tools available for creating a parallel application. The list of languages and API's we are going to evaluate is highly dependent on the choice of the platform. In order to decide which combination of language and API should be used we will evaluate them to certain criteria like speed, compatibility, I/O and difficulty of implementation.

The programming part of this thesis is started by making a Matlab implementation of the problem and check the speed of that implementation. This can optimised from a straight forward brute force method by adding sections of code for certain combinations of points. For example for two points with only multiplicative weights the edge between the two regions is always circular in shape. For our problem circles can be calculated faster than a the polyline generated by two generators with different multiplicative and additive weights.

Finally we can make the parallel implementation. Depending on the choices we made we may need to reconsider our approach from the sequential Matlab version. Every combination has different strengths and weaknesses that should be explored to achieve the best result possible. Next we compare it to the sequential Matlab implementation both in time and in complexity. In the expected case of the superiority of the parallel application we will make a final change to the Matlab code to call the parallel program.

1 Studying Voronoi Tessellation

The Voronoi Tessellation has been studied extensively and many algorithms have been developed to compute them in a time-efficient way. In this chapter we will evaluate their usefulness to our specific problem. First we take a look at what the compoundly weighted Voronoi tessellation exactly is.

1.1 Principles of Voronoi tessellations

Voronoi tessellations come in a lot of varieties but the basic principle is always the same. We divide a plane in regions by means of a list of generators and a distance formula. In an ordinary Voronoi diagram, the most common Voronoi diagram, this is the Euclidian distance formula and the regions contain all points closest to their respective generator. Points which are equidistant to two or more generators form the edges between regions.

$$D = \sqrt{(x_1 - x_{i1})^2 + (x_2 - x_{i2})^2}$$

Other variations utilise the Manhattan distance, farthest-point regions, power regions and for this thesis the most important variation: the weighted distance Voronoi tessellation.

1.1.1 Regular Voronoi

The most commonly used Voronoi tessellations are ordinary Voronoi tessellations.

Definition of a regular Voronoi diagram.

> "Given a finite set of two or more distinct points in the Euclidean plane we can allocate all locations in the plane to the point to which its Euclidian distance is smaller than to any other point in the plane. The resulting tessellation of the plane is called the planar ordinary Voronoi Diagram" [1].

We can write this definition mathematically [1]:

$P = \{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^2 | \, 2 \leq n < \infty$

With $P_i(x_{i1}, x_{i2}) \neq Pj(x_{j1}, x_{j2}) \; for \; i \neq j, i, j \in \; I_n$

We call $\mathrm{V(P_i)}$ a region where:$V(P_i) = \{x \mid \left||x - x_i|\right| \leq \left||x - x_j|\right| \; for \; i \neq j, i, j \in \; I_n$

$With\ ||x - x_i||$ the euclidian distance between the Cartesian points x and x_i
so $||x - x_i|| = \sqrt{(x_1 - x_{i1})^2 + (x_2 - x_{i2})^2}$

The combined regions finally form the Voronoi plot $\mathcal{V}$

$\mathcal{V} = \{V(p_1), V(p_2), \dots, V(p_n)\}$ (1)

In Figure 1 and Figure 2 we see an example of a Voronoi plot generated by our Matlab program. The added red lines demonstrate the fact that the edges are defined such that the distance to the two respective generators is equal. Note that this Voronoi plot is bounded by a region S where the definition above defines an unbounded region. So we alter formula (1) to

$\mathcal{V} = \{V(p_1) \cap S, V(p_2) \cap S, \dots, V(p_n) \cap S\}$ (1)

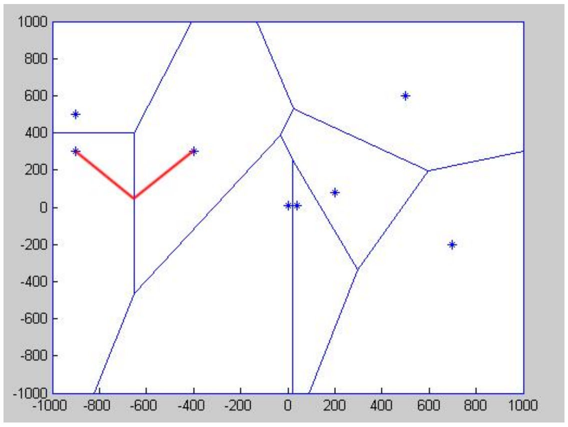


Figure 1 Ordinary Voronoi plot in Matlab

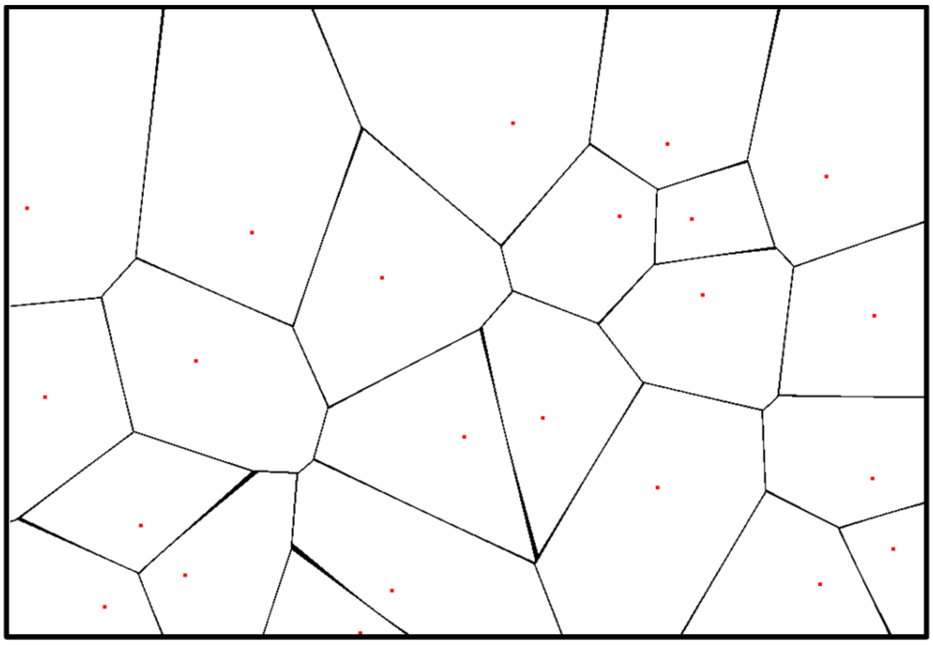


Figure 2 Ordinary Voronoi plot in OpenCL

1.1.2 Weighted Voronoi tessellations

Weighted Voronoi tessellations are a form of generalised Voronoi diagrams. In this thesis we discern three types of weighted Voronoi tessellations, additively weighted, multiplicatively weighted and compoundly weighted. The formulas for these generalised diagrams are displayed in Table 1.

Table 1 Overview of distance formulas for weighted Voronoi

Ordinary Voronoi	$D = \sqrt{(x_1 - x_{i1})^2 + (y_1 - y_{i1})^2}$
Additively weighted Voronoi	$D = \sqrt{(x_1 - x_{i1})^2 + (y_1 - y_{i1})^2} - Wa$
Multiplicatively weighted Voronoi	$D = \frac{\sqrt{(x_1 - x_{i1})^2 + (y_1 - y_{i1})^2}}{Wm}$
Compoundly weighted Voronoi	$D = \frac{\sqrt{(x_1 - x_{i1})^2 + (y_1 - y_{i1})^2}}{Wm} - Wa$

Additively weighted

The first of the weighted Voronoi Tessellations is the additively weighted Voronoi plot where a positive weight is subtracted from the regular distance function. Therefore an empty region can be generated when: $\min\{\left\|x - x_i\right\| - w_j, j \in I_n\} < -w_i$ [1]
Edges of additively weighted Voronoi tessellations are usually hyperbolic arcs or straight line segments if the distance between two generators is smaller than their combined weights.

Multiplicatively weighted

The second weighted Voronoi Tessellation is the multiplicatively weighted one. Here the distance formula is altered by multiplication with a positive weight greater than zero. The edges between two regions can be circular in shape or straight lines. Edges are only a straight line if they have equal weights. Areas can be completely contained in other areas and areas can grow around obstacles like other areas. We can see both effects in Figure 3 Multiplicative weighted Voronoi. The grey areas contain no generator but are part of the area under control by generator one even if the regions are disconnected. Generator 2, 3 and 4 each have the same weight, the borders between their areas are straight lines and region 5 is contained in region 6.

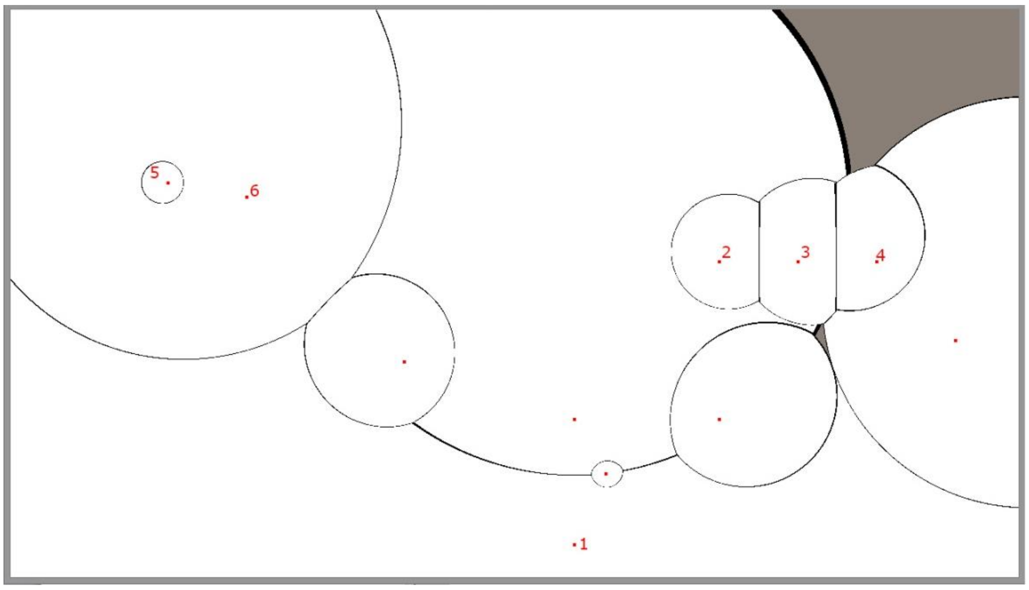


Figure 3 Multiplicative weighted Voronoi, eleven points

The multiplicatively weighted Voronoi diagram poses some serious problems to our program. Unlike the ordinary Voronoi diagram or the additively weighted Voronoi diagram point insertion is impossible here due to the big impact of the multiplicative weight. Where in an ordinary or additively weighted Voronoi plot the area of influence with point insertion is limited by the areas adjacent to the new generator this is not the case with multiplicatively weighted Voronoi diagrams as proven in Figure 4 and Figure 5. This is the situation if we enter one more point P1 in the diagram we created in Figure 3. We see that areas that are not connected to the region where the point was inserted also are affected. This implies that normal computational approaches like divide and conquer don’t work here. The tessellation has to be redrawn completely every time. [2]

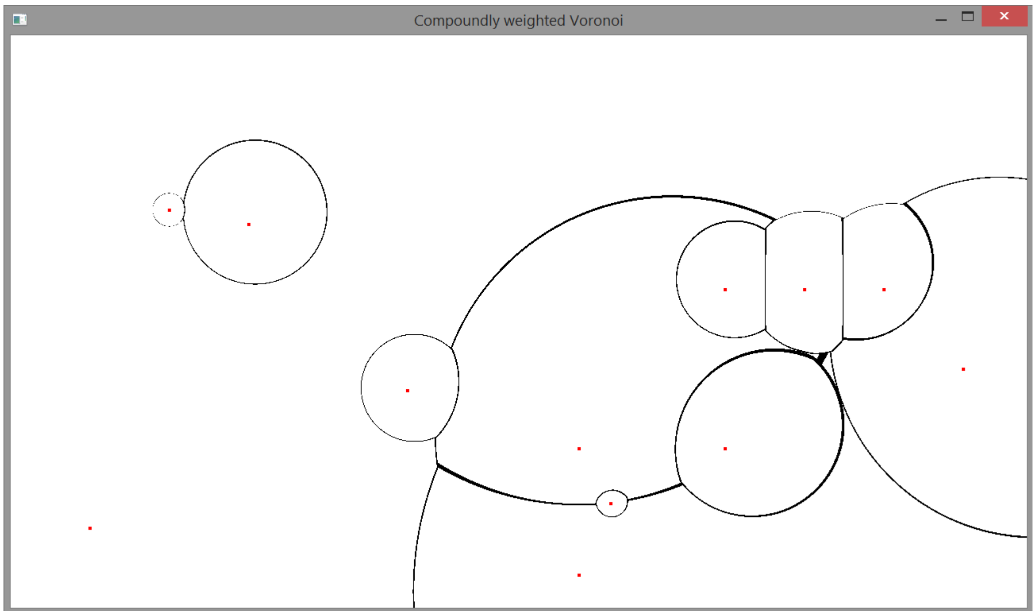


Figure 4 Multiplicatively weighted Voronoi, twelve points

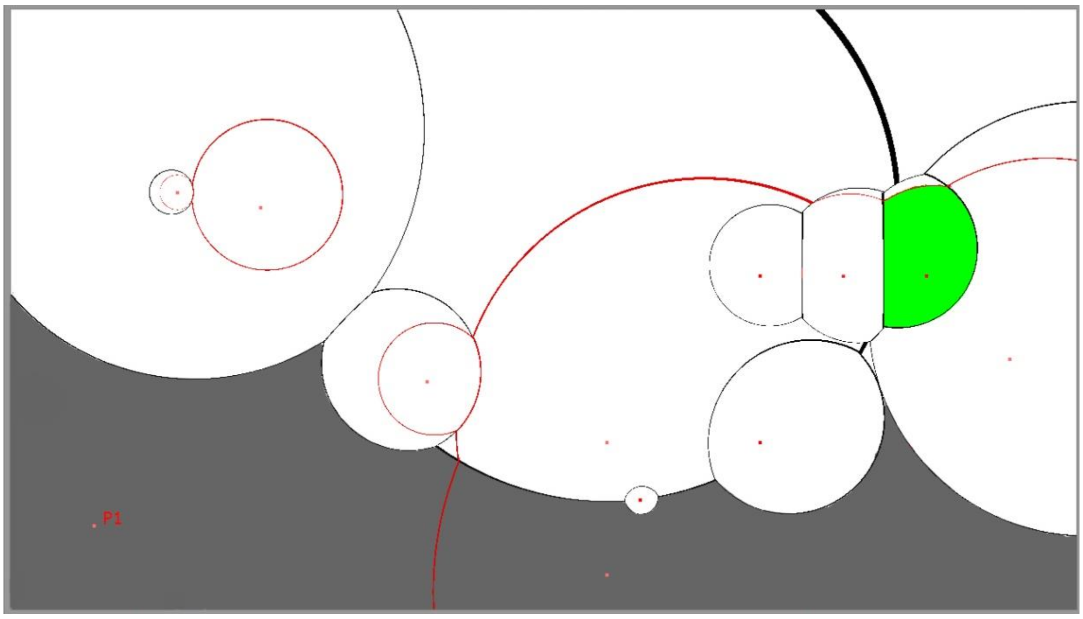


Figure 5 Multiplicatively weighted Voronoi, point insertion problem

Compoundly weighted

If we combine both weights in the distance formula we get the compoundly weighted Voronoi diagram. Now the edges are generally fourth order polynomial curves. The compoundly weighted distance becomes the additively weighted distance in case the multiplicative weight is one. When the additive weight is zero, this becomes the multiplicative weighted distance. Since this kind of Voronoi tessellation is a combination of additively weighted and multiplicatively weighted it also combines their characteristics. This means a brute force method is once again necessary.

1.2 Strategies of Voronoi construction

Since the multiplicative factor prohibits the use of standard algorithms we have no other choice then a brute force algorithm. This is why we need brute force methods or approximation methods.

1.2.1 Approximation by a digital image

Input

This program takes the following input:

- X & Y, dimensions of the digital image D(X*Y)
- [x_n, y_n]set of n generators in boundaries of the digital image

Output

The program returns the following output:

- digital image D approximating the Voronoi diagram for the given generator set
  - value of a pixel if added to an area = number of the generator
  - value of a pixel if undeterminable = -1

Procedure

We place the generators on the digital image and grow the region outwards by evaluating each pixel in the plot according to these rules[1]:

- if a pixel has no neighbours that are not zero it stays zero
- if all the positive valued neighbours of a pixel have the same value it gets the same value
- if the neighbouring pixels have two or more different positive values it gets the value -1

We continue evaluating until all pixels have been assigned a nonzero value. This can take multiple iterations.

Result

An x*y array with values -1 or 1 to n. All the pixels with the same value except -1 form a group. Some groups are separated by a value of -1 but this is not always the case. To plot the borders we would need to run one last iteration that marks every pixel with a non-equal neighbour. If we plot these lines we get 2 pixel wide lines when there was no -1 separating the two regions and three otherwise. We could also assign a colour to each value and plot the images with those values. This version of the algorithm is only valid for the ordinary Voronoi but we can adapt it easily by not only marking the generator in the first step but also marking the area around it in a circular area with the radius equal to the additive weight. We should do this incrementally so that if there is an overlap we don't overwrite it. The multiplicative factor can be the grow rate of the areas but this might prove difficult to implement. If an obstacle is encountered the growth in that direction is stopped and so we lose the property of multiplicative Voronoi on disjoint regions.

1.2.2 Intersecting polygons

Input

This program takes the following input:

- List of generator coordinates and their additive and multiplicative weights
- Bounding box containing all the generators.

Output

The program returns the following output:

- List of edges or alternatively list of areas

Procedure

If we compare two distance functions for two generators we get the points that determine the border between those two generators.

$$\frac{\sqrt{(x_1 - x)^2 + (y_1 - y)^2}}{Wm_1} - Wa_1 = \frac{\sqrt{(x_2 - x)^2 + (y_2 - y)^2}}{Wm_2} - Wa_2$$

Knowing this we can shrink the bounding box by determining the intersections of the border and the bounding box, if any. The simplified procedure would be:

1. Input area = bounding box
2. Calculate the border first generator pair g1-g2
3. Find intersections g1-g2
    - No intersections → the border creates a closed area, find the area that contains g1. This is or the enclosed area or everything except the enclosed area
    - Intersections → find the area that contains g1
4. Go back to step 2 and change g2 with the next generator if there are generators left and change input area to the output of step 3
5. Go back to step 2. Set g2 back to the first generator of the list and g1 to the next generator if there are generators left in the list and change input area to the output of step 3

This method is not complete because it is not always trivial to determine the area containing the main generator e.g. with some additive weights there are more than two intersections with the input area. There are some possible optimisations as now every calculation is done multiple times and some calculations can be done faster e.g. the intersection between the polygon and a line segment. While the intersecting polygon is strictly convex we only need to find two intersections while with a concave polygon there might be more intersections, and so more computing time. To maximise performance we need to guarantee that the polygon is strictly convex as long as possible by planning the order in which we evaluate the generators.

Result

The result is a list of edges ranging from straight lines to fourth order polynomials or a list of areas. These can easily be plotted.

1.2.3 Distance lists

Input

This program takes the following input:

- x and y dimensions of the plane
- list of n generators and their respective weights

Output

The program returns the following output:

- digital image D approximating the Voronoi diagram for the given generator set
    - value of a pixel if added to an area = 0
    - value of a pixel if it is on an edge = 1

Procedure

For all points P(x,y) of the plane we calculate the distance from this point to all generators in the list according to the distance formula. This gives us an x*y*n matrix.

Next we look for the two minimum distances for each point P(x,y). If these distances are the same with a certain margin, this point, or pixel, is part of an edge between the two generators. If the distance is bigger than the margin the point does not belong to an edge but only to the area with the smallest distance.

Result

The result of this method is an array that can easily be printed to a binary image. Data on edges and vertices is not directly available. This method is a good candidate for parallelisation due to the many relatively easy independent calculations that have to be done.

2 Comparison of different computation approaches

2.1 Sequential computing

In traditional computing, software is constructed for sequential computation. This means the programmer analyses the problem and breaks it down into a discrete series of instructions. These instructions are queued for execution on a single processing unit on a single computer system. The instructions are executed one after another in the order they were queued in the instruction pipeline: an instruction has to be completed before the next one can be processed. Figure 6 displays the basic idea of sequential computing.

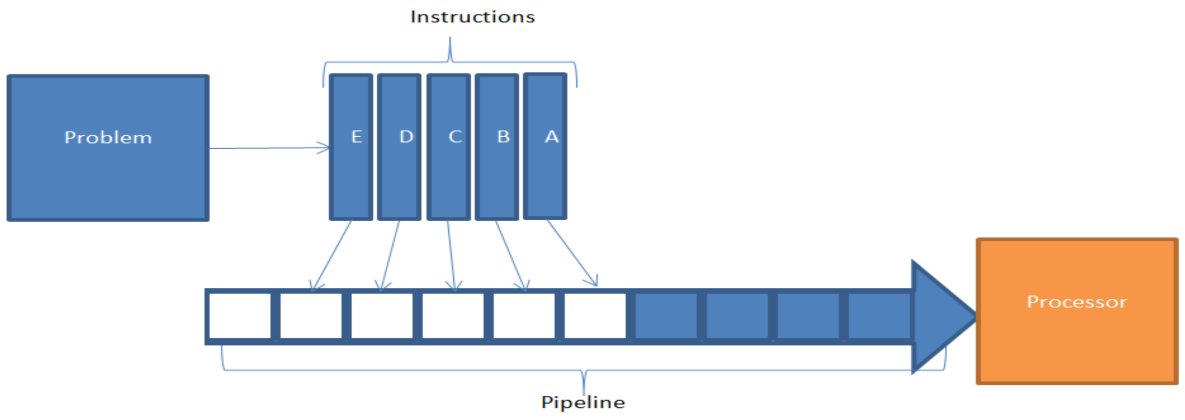


Figure 6 Principle of sequential computing

2.2 Parallel computing

The idea of parallel computing is nearly as old as the computer itself. One of the earliest examples is the Hollerith Tabulator , a mechanical computer created by Herman Hollerith in 1890.

It has also been clear for a long time that there are many different approaches to achieving parallel computation as indicated by this extract from a 1958 computer magazine.

> "So far the subject of parallel programming has been introduced only in situations where there are two or more separate parts of a machine, each dealing with its own branch of the program. However, very similar logical problems of programming can occur in a situation which does not involve more than one control unit to divide its time between two different activities" [3].

We will examine the three main philosophies of parallel computing and some of the modern methods of parallel programming.

2.2.1 Instruction level parallelism

The first way of parallelising computing is instruction level parallelism or ILP. ILP is the concurrently execution of multiple machine instructions. Consider this simple mathematical problem:

C = (x+y)*(z-w)

1. A = add x y
2. B = sub z w
3. C = mult A B

Instruction 3 is based on the results of instruction one and two and the results of those have to be known before this instruction can be executed. One and two are mutually independent. If our processing unit has two or more functional units capable of executing these instructions we can calculate both in the same time span where a sequential system would only calculate one of the instructions.

2.2.2 Data level parallelism

Data level parallelism is a special case of instruction level parallelism. In instruction level parallelism we execute different instructions over different processing units at the same time. In data level parallelism we execute the same instructions on multiple processing units at the same time but with different data. This is also called Single Instruction Multiple Data, SIMD. Alternatively multiple processing units can work more independently and execute multiple instructions or even complete programs independently. This is called Multiple Instruction Multiple Data.

A good example to demonstrate the possibilities of data level parallelism is the for loop. In Table 2 we give a simple for loop and its decomposition. We see that this can be parallelised in an easy manner because the actions that have to be done are always the same, only the (input) data changes. We can divide the task over all processing units with the required functionalities.

Table 2 Data level parallelism example

For loop	Decomposed for loop

For(int i = 1; i < n; i ++)
{
    DataOut[i] = DataIn[i]*2;
}
	DataOut[0] =DataIn[0]*2;
	DataOut[1] = DataIn[1]*2;
	…
	DataOut[n] = DataIn[n]*2;

2.2.3 Task level parallelism

Task or thread level parallel structured programs are designed in such a way that multiple processing units each get a subsequent part of a sequential program to execute. To design a program for task level parallelism you must divide the big problem in a number of smaller problems. These smaller problems get partitioned over the available hardware and solved simultaneously. This approach to parallelism only works when only a small amount of data dependencies exists. The fewer dependencies the more tasks can be executed in parallel. When there are dependencies, tasks can be grouped in task groups. Task level parallelism is scetched by Figure 7.

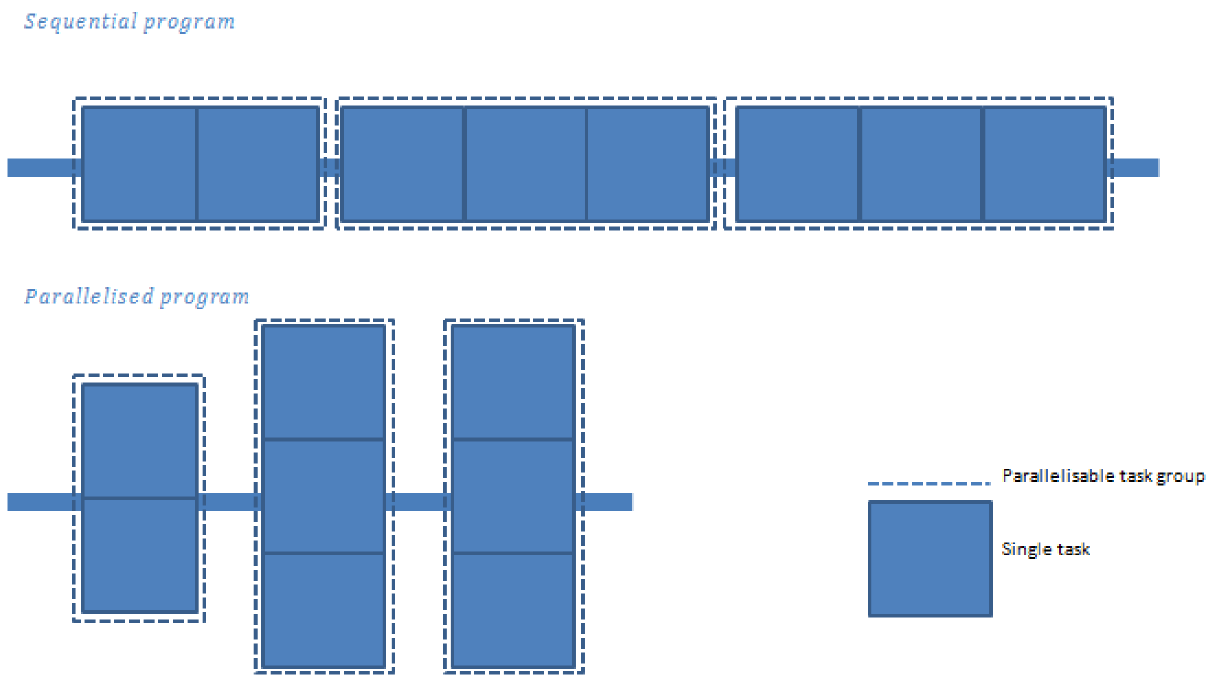


Figure 7 Task level parallelism

2.3 Approaches to parallelised computing

Before we can choose a programming language or API we need to research which platform would be best suited for our situation.

2.3.1 Parallel capable hardware

Central Processing Units

Around 2001 the first attempt at parallel computing on a single Central Processing Unit, or CPU, was performed by Intel with their hyper threading technology. Here a second execution core is added to an existing processor design sharing cache with the first core [4]. In 2005 the first real multicore CPU's were introduced with two or more cores on the same die, opening the path to continuously increasing core-counts in CPU's with current day processors ranging from 2 to 10 real cores and up to 20 hyper threading cores.

The multiple cores don't have to be all on the same device. Some motherboards are designed to support multiple processors and clustered computers can also combine their computational possibilities to get an even higher core-count. But while the hardware is capable of handling simultaneous execution of multiple threads most software is still programmed for only one processor core. Since the cores of a typical CPU are designed for heavyweight tasks the most common multithreading approach on CPU's is task parallelism.

Graphical processing units

While multicore CPU architecture is focussed on the simultaneous processing of a few heavyweight threads and a high performance per thread, GPU architectures take a different approach and focus on simultaneous execution of many lightweight threads and features many small cores. Single core capabilities of a GPU are generally poor and not all GPU's support double precision computation or performance is decreased significantly when using the double precision floating point format, see Table 3. In Figure 8 we see the G80 architecture of a Nvidia GPU. This architecture was first used in 2006 and was the first one to feature unified shader models.

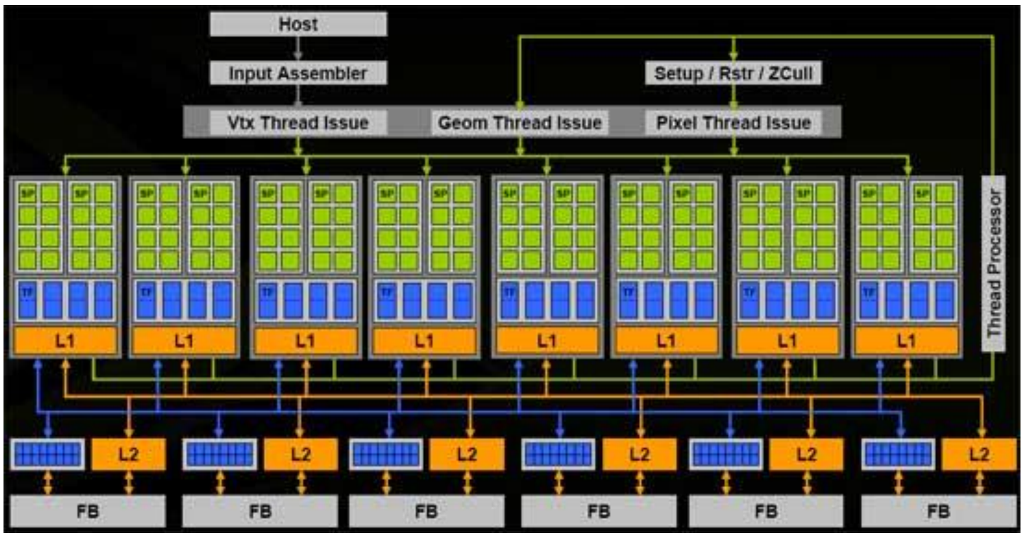


Figure 8 Architecture of a Nvidia 8800 GTX GPU [5]

The unified shader model implies that every core in the GPU can do the same task with consistent performance. This makes GPU programming easier and opens the door to General Purpose computation on the GPU[6] or GPGPU in short. The G80 architecture features 128 computation cores (green blocks) that share L1 cache per 16 cores. The cores are grouped in 16 workgroups [5]. These workgroups indicate the maximum amount of different tasks a GPU can execute simultaneously. On a GPU data parallelism is the preferred approach to multithreading because of this limitation but task parallelism is also possible.

When we do a one on one comparison between CPU's and GPU's in Table 3 we notice that high-end professional CPU's offer less options for data-parallelisation compared to mid-end consumer GPU's. It is worth noting that the professional GPU lacks a display output and is created purely for GPGPU applications.

Be aware that not all of the data in Table 3 is supplied by the manufacturer. For the CPU's we've done a measurement on the test system to get the FLOPS and for the other two CPU's we used the theoretical formula: 4*cores*clock*threads/core [7].

FPGA

FPGA designers have two choices when making parallel applications: programming an application in C on the soft processor or in a hardware description language on the hardware. When programming hardware parallel applications on FPGA the designer can create the hardware to suit his specific needs. The parallel cores can be created to the exact requirements of the application making denser logic possible compared to the general purpose CPU or GPU cores. FPGA's typically offer higher performance per watt. The limiting factor to the use of FPGA's in parallel computing is the I/O. In parallelised computing FPGA's are usually used as a co-processor to a regular x86 computer system. The communication between those devices is the weak point of the parallel system. The application should be designed with minimal bandwidth overhead between the host and the co-processor in mind and the fastest possible interconnect between the devices. Usually PCI is chosen because it is connected to the main processor bus. FPGA parallel computation usually is limited to low precision fixed point calculations.

Typical problems that need to be parallelised can be hard to write in VHDL or Verilog, the standard hardware description languages used in FPGA design. Instead C programmes are often written for a soft processor. The area performance however is about 13x worse and the speed performance about 17x with optimised vector code compared to a hardware design [8].

Conclusion

While multithreaded CPU's are available in almost all modern computers they are not really suited for this problem. CPU's are the best choice for relatively few heavyweight threads opposed to many lightweight threads needed for our problem. FPGA's can do many lightweight threads but the communication with the FPGA is harder and can potentially bring a lot of overhead. Parallelisation on GPU's is the most promising for this thesis. They offer a big improvement over sequential computation for a small increase in difficulty and no increase in cost as GPU's are commonly available in modern computers. GPU's also offer perspective on a higher level of parallelisation in distributed computer systems.

Table 3 CPU/GPU comparison [9, 10 ,11, 12, 13, 14, 15]

	Consumer CPU	Professional CPU	Test system CPU	Consumer GPU*	Professional GPU	Best performance / price GPU	Test system GPU
Brand and name	Intel core i7 3960X	Intel XEON E7-2850	Intel core i7 3740QM (mobile high end chip)	Nvidia Titan	Nvidia Tesla K20X	AMD Radeon HD 7970	Nvidia NVS 5200M (mobile low-midrange chip)
Cores	6 (12 with hyper threading)	10 (20 with hyper threading)	4 (8 with hyper threading)	2688	2688	2048	96
Peak GFLOPS Single / double precision	158,4 / 158,4 (theoretical)	192 / 192 (theoretical)	75 / 75 (benchmark)	4500 / 1500 (marketed speed)	3950 / 1310 (marketed speed)	3788,8 / 974,2 (marketed speed)	240 / 80 **
price	$999	$2558	$378	$999	$3199+	$280	Unknown < $200

* here we didn't take the GPU with the highest core count because it has a rather poor dual precision performance

** there is no official data on the NVS 5200m available, instead we took the data of the similar GeForce 620m which has the same architecture but different drivers.

3 Study of different programming languages and API's

3.1 Introduction

To evaluate available languages and API's we will look at their respective advantages and disadvantages as well as the key requirements summarised in Table 4:

Table 4 Comparison criteria

Available platforms	For a high speedup we would want to run the program on a computers GPU, this assures the highest possible level of multithreading without specialised hardware
Available languages	In case of an API we are interested in which languages can be used with the API.
I/O	The way the language or API can read and write parallel data
Compatibility	Some API's or languages work on any normal x86 computer with standard compilers, others require special drivers, compilers or even hardware.
Relative speedup factor	Speed gained compared to the same program executed single-threaded on the same hardware. This comparison is not straightforward as it usually strongly depends on factors like the total number of available processing units and uniform capabilities of those units

Since evaluating all existing parallel languages and API's goes beyond the scope of this thesis we decided to evaluate only a selected group of languages and API's which offer the best perspective and are well documented.

3.2 Matlab Parallel Computing toolbox

The most obvious way to start the parallel implementation is with Matlab. Matlab offers a parallel computing toolbox to enable running code on multiple computing units. The toolbox isn't limited to local resources but can also be used to run the application on big computer clusters in a distributed computing environment. This can be done with the same code as for a single desktop system.

The Matlab parallel computing toolbox supports up to 12 local workers on the CPU and can use the complete capacity of a Nvidia CUDA enabled GPU. On GPU there are only a few Matlab functions available. However the toolbox also provides the possibility of CUDA kernel inclusion in only one line of code. Table 5 provides an overview.

Table 5 Matlab Parallel Computing toolbox [16]

Matlab Parallel Computing toolbox conclusion	
Advantages	• No translation necessary from sequential Matlab code • Computer cluster and grid support • Some of the highly optimised mathematical functions of Matlab available
Disadvantages	• GPU functions are limited unless we include CUDA kernels • Bound to certain instructions • CPU: maximum 12 workers on single system
Available platforms	CPU, GPU (Nvidia)
Available languages	Matlab
I/O	Same as Matlab
Compatibility	System requirements are the same as Matlab
Relative speedup factor	Matlab's Parallel computing toolbox isn't really powerful on a normal desktop computer but only comes into its own in use on cluster computers ⇨ Medium (desktop multicore CPU) ⇨ High (desktop Nvidia GPU) ⇨ Extremely High (computer clusters)

3.3 OpenMP

When switching from programming for a system with a single processing unit to a system with multiple independent processing units to program applications in parallel the problem of memory sharing arises. If two independent processing units work on the same dataset a problem can occur if there are data dependencies. To understand the utility of Open Multi Processing, or in short OpenMP, we'll take a quick look at the given problem. In Figure 9 and Table 6 we see a possible problem when unified memory is used.

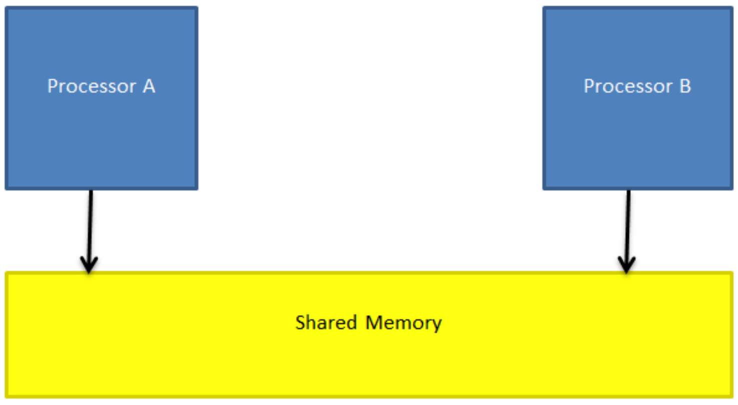


Figure 9 Unified shared memory model

Table 6 Using two processors to solve a problem

	Processor A	Processor B
1	Read x	z = read mem
2	Read y	z = mult z 2
3	z = add x y	print z
4	Store z mem	NOP

Processor B uses the output of Processor A but because both processors work independently and asynchronously processor B does not know if the value it reads from the memory is the right value. To be able to have a reliable result OpenMP was designed to address these kind of problems.

OpenMP is an API consisting of compiler directives, runtime libraries and environment variables, enabling programmers to efficiently use those shared memory systems. However OpenMP does not support GPGPU applications. Thus the maximum level of parallelism is limited to the number of available CPU cores. In typical workstations this translates to two to sixteen threads.

OpenMP is easy to use. The programmer only has to define which loops have to be parallelised and OpenMP takes care of thread creation, synchronisation and destruction. The amount of threads created is also determined by OpenMP [17]. Table 7 provides an overview.

Table 7 OpenMP [17,18]

OpenMP conclusion	
Advantages	• Explicit instead of automated parallelism. No extra intelligence in the compiler required to find the parallel parts • Scalable to systems with higher processing unit count like clusters • Translation from sequential code to OpenMP is relatively easy • Incremental parallelism • Obfuscates hardware o No need to understand underlying CPU/GPU architecture o Can also be a downside for optimising
Disadvantages	• Hard to parallelize loops of variable size • Overhead when using recursive code • Inefficient on non-shared memory systems • High overhead for loading the executable and load balancing compared to single threaded programs. Only interesting for longer programs
Available platforms	Only on CPU
Available languages	C, C++ and Fortran
I/O	Not specified in the standard. Major issues can arise if the threads write back to the same file. Complete responsibility of the programmer
Compatibility	Supported by many compilers ⇨ GCC, VS C++, Intel Oracle and IBM C/C++/Fortran All hardware can theoretically run OpenMP if the compilers support the hardware
Relative speedup factor	Highly depending on number of cores, can only use CPU cores ⇨ Medium

3.4 OpenACC

Another standard for parallel computing is OpenACC. Just like OpenMP it is an API designed for use with C, C++ and Fortran. Converting a program in any of those languages to be OpenACC enabled is very easy and can be achieved by just adding a few lines of code with compiler directives. These directives are ignored by incompatible compilers but compatible compilers know these sections of code have to be executed on an accelerator. Unlike OpenMP these accelerators can be CPU's as well as GPU's, providing a potential performance gain.

However just adding these compiler directives to an existing piece of code can be inefficient. To gain as much performance as possible the directives should be added around loops and the loops should be designed with parallelism in mind. When designing a program the programmer should try to keep in mind the envisioned execution model of the OpenACC group. Some key features of the execution model include[20]:

- Most of the user application is executed on the host device (e.g. The CPU)
- An accelerator takes care of the most tasking sections of code
    - These are typically coded in loops
- Even in regions tasked to the accelerator the host device can interfere with the execution on the accelerator e.g. by allocating memory and passing arguments and results back and forth
- The host program initiates the master thread
- The accelerators operation can be synchronous or asynchronous, the latter requiring some form of command queue

Being a high-level API memory management isn't a key point in OpenACC, it can be done implicit and is managed by the compiler. Table 8 provides an overview.

Table 8 OpenACC [20]

OpenACC conclusion	
Advantages	• Easy to implement • High level ○ No need to understand underlying CPU/GPU architecture ○ Can also be a downside for optimising • Portable
Disadvantages	• Performance is highly dependent on loop design (more than other languages) • Relatively new: not much documentation or support
Available platforms	CPU and GPU (AMD, Nvidia and Intel)
Available languages	C, C++ and Fortran
I/O	Not applicable, same as the chosen language
Compatibility	Only supported by a few compilers, other compilers are planning support ⇨ CAPS Enterprise HMPP Workbench, Cray CCE (only on Cray systems) and PGI Accelerator (C and Fortran only) ⇨ GCC (in a future version) OpenACC supports most Intel, Nvidia and AMD multicore hardware, details not provided
Relative speedup factor	Depending on the choice of accelerator, GPU capable ⇨ Medium-High

3.5 OpenGL

OpenGL, again an API, is entirely different from the other API's and languages in this list in that its main purpose is rendering graphics. Until recently it was very hard to do GPGPU computing with OpenGL: the programmer had to think in triangles and pixels and remap his problem to these objects. In version 4.3 (August 2012) of the OpenGL standard compute shaders were added. These structures are used for computations that don't necessarily lead up to rendering. Normal shaders work in well-defined boundaries and accept a range of user input and generate output whereas compute shaders are more abstract in nature and operate more autonomously, forcing the user to make explicit reads and writes from within the shader[21]. A simplified scheme is provided in Figure 10. The second stage actually consists of an explicit read, the required calculations and a write back.

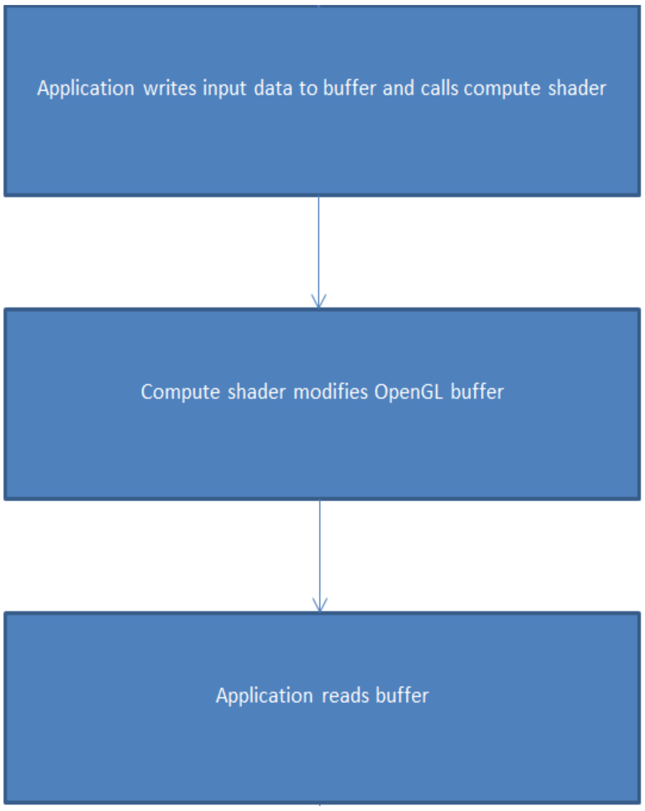

Application writes input data to buffer and calls compute shader
Compute shader modifies OpenGL buffer
Application reads buffer

Figure 10 OpenGL compute shaders principle

The origin of OpenGL makes it an interesting choice to render the results of the application if we write it in a compatible language that lacks having its own Window agent. OpenGL has no such functionality out of the box either but OpenGL libraries that do are widely available. One of the most widespread libraries is GLUT. GLUT offers window definition and control, basic drawing functionality for vectors and vertices as well as keyboard and mouse support[23]. GLUT is no longer in development but an open source alternative, freeGLUT, offers the same functionality and is still in development [24]. Table 9 provides an overview.

Table 9 OpenGL [22,23]

OpenGL conclusion	
Advantages	• No extra packages/API's required * • Highly scalable • Code is compatible between different GPU brands
Disadvantages	• Hard to implement (prior to 4.3) • Multiple distinct libraries • Memory management is done automatically. No way to assign certain blocks of memory to certain threads thus minimising possible performance gain for memory usage
Available platforms	GPU (AMD, Nvidia)
Available languages	OpenGL offers capability and language bindings with over 15 languages amongst others: C, C++, C#, Java, Fortran, Haskell, Visual Basic, Python. This choice can shrink depending on the chosen OpenGL library (GLUT, SDL, GLFW, CPW…)
I/O	Not applicable, same as the chosen language
Compatibility	No OpenGL specific compiler required. Other compatibility features depend on the library of choice. (free)GLUT offers compatibility and portability between the most common OpenGL implementations and platforms. On hardware level all GPU's from AMD and Nvidia support at least version 1.2 of OpenGL
Relative speedup factor	Can use the full parallel power of a GPU ⇨ High

* if we take into consideration that we will need OpenGL anyway to display the Voronoi plot.

3.6 OpenCL

OpenCL is the result of the increasing request for cross-platform GPGPU capable applications. OpenMP was limited to the CPU, OpenGL prior to version 4.3 was hard to code and isn't really suitable for GPGPU computing but rather rendering. The OpenCL API was specifically designed for computation so it has no rendering capability on its own. The OpenCL standard specifies that OpenCL code can be executed on CPU, GPU, accelerator hardware or any combination of those devices. In reality however this depends on the implementation you choose. OpenCL has three major releases: Nvidia, AMD and Intel. The former two are the biggest players on the GPU market. These two versions are not interchangeable and have some inconsistencies. For example the AMD API supports CPU and GPU while the Nvidia implementation supports only GPU. Table 10 provides an overview.

Table 10 OpenCL [25, 26, 27, 28, 29]

OpenCL conclusion	
Advantages	• Added support for accelerator (Intel API) • Easy definition of global and local work items ○ Global: partitioning workload between different devices (Crossfire / SLI setup or CPU+GPU) ○ Local: partitioning workload between different processing units of the same device • Possible to combine CPU and GPU effort for even greater capabilities • Programmer can plan memory management to maximise memory performance • Memory abstraction, this increases portability • Most recent release (November 2013) supports ARM hardware • Easy translation from CUDA to OpenCL
Disadvantages	• Relatively complex to set up • Bad compatibility between different versions of the API (see compatibility)
Available platforms	GPU (Nvidia), GPU & CPU (AMD), GPU & CPU & Accelerator (Intel)
Available languages	Amongst others: C / C++ / C# Fortran / Java / Python
I/O	OpenCL buffers can be read by the master program thus I/O depends on the language of the master program.
Compatibility	OpenCL has many different implementations which are not fully compatible. The main ones are Nvidia, AMD, Intel and IBM. Compiler-wise all compilers that can handle C/C++/Fortran can be used. OpenCL is compatible* with the following hardware: • AMD CPU's supporting at least SSE2 • AMD GPU's with unified shaders • Nvidia GPU's with unified shaders • Intel CPU's starting at the Core 2 family except Celeron, Pentium and Atom processors • Intel integrated graphics from the 3th generation Intel Core CPU's onwards • Many more In practice this means all devices from AMD, Nvidia and Intel designed after 2006 support OpenCL with exception of low end Intel CPU's * This again depends on the chosen implementation and version of OpenCL.
Relative speedup factor	Can use the full parallel power of a GPU and CPU simultaneously ⇨ Very high

3.7 CUDA

Before the first OpenCL specifications were released Nvidia already offered its own proprietary GPGPU package under the name Compute Unified Device Architecture or CUDA in short. CUDA offers parallel programming capabilities on supported Nvidia GPU's. CPU support is completely absent in CUDA and hardware from other manufacturers is not supported. Nvidia's recent hardware architectures Fermi (2010-2011) and Kepler (2011-2012) are designed to natively support more coding languages as well as CUDA. The CUDA API is mapped completely to the use of CUDA cores, the smallest computational unit on a GPU die, for high efficiency computing. OpenCL has the disadvantage that it cannot directly access the CUDA cores but it needs an abstraction layer. This creates a computational overhead and performance loss compared to CUDA. This performance loss is in the 10% range most of the time but on smaller datasets this overhead gets bigger [30,31].

CUDA programs are backwards compatible until the GeForce 8 series from 2006 that introduced unified shaders. Earlier cards without the unified shader architecture do not support CUDA. Translation between CUDA and OpenCL is relatively easy. The major differences are in the terminology as demonstrated in Figure 11Table 11. Other differences include a new compiler, buffer definitions and pointers being no longer compulsory[32].

Table 12 provides an overview of the CUDA API.

Table 11 Terminology changes between OpenCL and Nvidia [32]

CUDA	OpenCL
Thread	Work Item
Thread block	Work group
Shared memory	Local memory
Local memory	Private memory

Table 12 CUDA [22, 25, 33]

CUDA conclusion	
Advantages	• Easy definition of blocks and threads, analogue to global and local work items ○ Block: partitioning workload between different devices (SLI setup) ○ Thread: partitioning workload between different processing units of the same device • Programmer can plan memory management to maximise memory performance • Maximum theoretical performance from Nvidia cards • Kernel integration in Matlab can be done with only one line of code

Disadvantages	• Not compatible with cards from other manufacturers
Available platforms	GPU (Nvidia only)
Available languages	Amongst others: C / C++ / C# Fortran / Java / Python with appropriate language bindings
I/O	Identical to OpenCL
Compatibility	On hardware level CUDA is compatible with all Nvidia cards with the unified shader model. CUDA has its own compiler (NVCC) opposed to OpenCL
Relative speedup factor	Can use the full parallel power of a GPU, no overhead on Nvidia GPU's ⇨ Very high

3.8 Conclusion on Voronoi theory

Based on this evaluation of the most interesting aspects of each language we reviewed we can make a decision on which one suits us most.

Matlab's parallel toolkit only has limited GPU capabilities. We would have to make a compromise between compatibility and functionality. If we choose to support only CPU's and Nvidia GPU's or if only basic instructions would suffice Matlab would be a good option. However we prefer to have more cross-platform compatibility. OpenMP only has CPU capabilities. While providing a significant boost in performance over single threaded applications the maximum amount of threads an application with OpenMP can spawn will always be significantly less than a program that does utilise a GPU. OpenGL can use the power of the GPU but even with the compute shaders it is still a cumbersome task to remap the problem that has to be solved to OpenGL standards.

Then we are left with three API's: OpenACC, OpenCL and CUDA. While these three languages offer similar performance we decided to go with OpenCL. CUDA is faster on a Nvidia GPU but cannot use a GPU from another brand, OpenACC can run on both platforms but is relatively new and undocumented compared to the others. OpenCL also features the possibility of combining hardware for even more performance gain, something the other API's don't offer. This extra speed and compatibility with non-Nvidia GPU's make up for the slightly lower performance on single Nvidia cards.

4 Matlab implementation

Our first attempt at solving the compoundly weighted Voronoi problem is a Matlab implementation. Matlab already offers a Voronoi function but this implementation has no support for the additional weights we want. Adjusting this version to take weights is not possible; this implementation is based on the relation between the Delaunay triangulation and the ordinary Voronoi plot. This relation is only valid for ordinary Voronoi tessellations.

Instead we had to make our own implementation. As discussed there are no available methods for incremental construction or other divide and conquer techniques. This makes a brute-force method necessary. We originally designed this method for dynamic Voronoi Tessellations with only one weight but as the thesis progressed the subject changed to compoundly weighted Voronoi. We changed the code to also accept a second, additive weight.

Algorithm description

In our approach we first calculate the equation of the locus of each pair of generators. To calculate the locus we equalise our distance formula between g1 and P and the generator g2 and P. This yields an implicit equation. P is not a single point but a set of points. When we solve the resulting equation 3 for x and y we have a valid equation for the locus.

1. $D1 = \frac{\sqrt{(g1.x-P.x)^2+(g1.y-P.y)^2}}{g1.M} - g1.A$
2. $D2 = \frac{\sqrt{(g2.x-P.x)^2+(g2.y-P.y)^2}}{g2.M} - g2.A$
3. $\frac{\sqrt{(g1.x-P.x)^2+(g1.y-P.y)^2}}{g1.M} - g1.A = \frac{\sqrt{(g2.x-P.x)^2+(g2.y-P.y)^2}}{g2.M} - g2.A$

Now we define a bounding box that contains all generators and we take the first generator in the list, g1, and all the equations of that generator and all other generators. We group these equations in four groups:

- Lines
- Circles
- Hyperbolas
- Others, generally fifth order polynomial

To create the Voronoi tessellation out of these lines and curves we are going to create polygons out of the intersection between these equations and the bounding box. In some particular cases the equation generates a locus that does not intersect with the bounding box. If there is no intersection we know the locus is a closed area and we can also make a polygon.

Lines are easier to process than the other types of borders. We know that a line intersects with the bounding box at exactly two points. After calculating these points we can split the bounding box in two polygons. We save the polygon that contains the generator g1. If this is the first polygon we calculated we temporarily save it and calculate the next one. If there´s already a polygon we use the Matlab function polybool to merge both polygons. Polybool takes an array of x-coordinates and an array of y-coordinates of two polygons and a flag indicating the way this polygon-polygon intersection should be calculated. We enter the x- and y-coordinates of the two polygons and the flag 'intersection'. This creates a new polygon matching the overlapping areas of the input polygons. We overwrite the temporary polygon with this new one and continue the cycle until the last generator pair is processed. The remaining region is the region for which all points have the smallest distance to input generator g1.

We could use a similar approach to the more complex borders. However they provide new problems. While lines always have exactly two intersections with a border this is not a case for these lines. A circle for example can have 0, 2, 4, 6 or 8 intersections with the bounding box. For this reason we implemented different functions for different equation types. A first step is to sample all line segments. This is possible with a solve function in a for loop but can be done much faster by vectorisation of the code as seen in Figure 11.

```
eqn = sqrt((G1(1)-x)^2+(G1(2)-y)^2)-G1(4) ...
       == sqrt((G2(1)-x)^2+(G2(2)-y)^2)-G2(4);
% solve for y, this gives two equations because eqn is implicit
Y = solve(eqn,y, 'MaxDegree', 4);
% determine the sample rate
beginpoint = -1100;
endpoint = 1100;
n = (endpoint-beginpoint)*0.5;
% make the vector
vect = linspace(beginpoint,endpoint,n);
% place the vector in the solution of the solve
Y = double(subs(Y,x,vect));
% adjust the result so that only meaningful data remains (no NaNs
% or imaginary numbers)
Y1 = Y(1,:);
Y1(imag(Y1)~=0) = nan;
```

Figure 11 Vectorisation of a piece of the curve sampling function

The second equation type is the circle. Circular borders occur when the additive weights of a generator pair equal zero and the multiplicative weights are not equal. When we encounter a circle we determine if the main generator is inside or outside of the circle. The generator is inside the circle only if the generator has a smaller weight then the other generator. After marking the region containing the generator we can again evaluate the region to the input polygon. If there is no input polygon present this region becomes the input polygon for the evaluation of the next generator.

The last two types of borders can be processed with the same function. Unlike circles these sections are not closed by definition. Polybool requires two closed regions as input. Instead we calculate the intersections of the polynomials with the bounding box. By combining these intersections with the sampled curve we now get a finite number of polygons. After this step we again evaluate each polygon and repeat the same procedure as with circles.

Optimisations

The process described in this chapter is very time consuming. To calculate intersection points of two borders we use the solve function or the more optimised fsolve. Both functions rely on the mupadmex file to calculate our intersections. On our system a set of only seven additively weighted generators takes 56 seconds. From those 56 seconds 53 are spent in the mupadmex file after a total of over 32.000 calls. After optimising we reduced the amount of calls to mupadmex for this problem size with 50% resulting in a time of 30 seconds for the complete program. This proved to be the minimum as the processing of all generator pairs was only done once.

To further increase the speed of the program there are two options:

1. Lowering the vector size for generating the sampled curve. This will cause a significant performance gain but lowers the accuracy.
2. Keep a list of already generated areas. For the first generator we would have to calculate the effect of all other generators, and for the subsequent generators we only calculate the next generators in the list. If this is done we temporarily save the output polygon and subtract all earlier generators from this polygon.

The only interesting option is the second one. It makes sure that the equation for each combination of generators is only calculated once. This was already the case with previous optimisations. When we analysed the timing of this function, we noticed the performance gain was minimal but another interesting effect occurred.

In Figure 12 we see the original output on the left. On the right we see the output with this optimisation. In the left image we filled the areas without generator with the Matlab patch function. These black areas are called remote areas and are separated from the main area of their generator by one or more areas. In the right image these areas are allocated to neighbouring areas. This makes this optimisation unsuitable for our implementation but in applications where remote areas are unwanted and all area should be occupied, e.g. crystal or cell growth, this can be interesting.

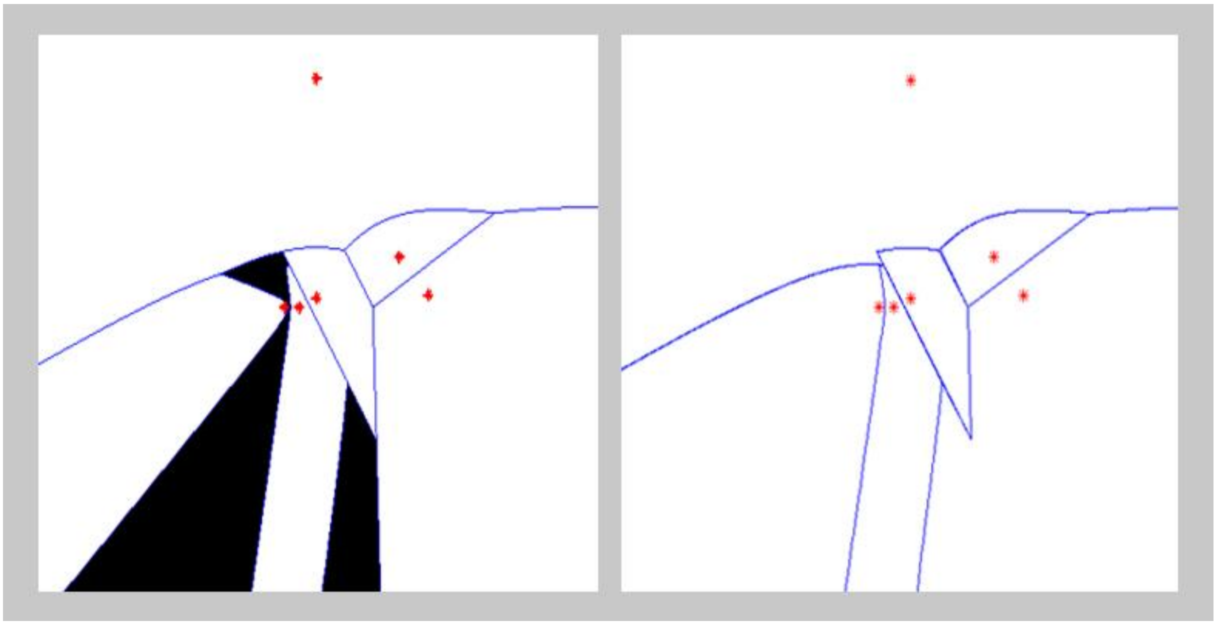


Figure 12 Marking holes in the tessellation

Conclusion

Given the O(n²) time complexity of the optimised program and the already long processing time for a small dataset we determined that this approach is unsuitable for our problem. Our 100 generator goal for the project would require significantly faster resources or a long processing time and make the program almost unworkable. Figure 13 demonstrates the polygon cutting principle while Figure 14 shows a timing breakdown.

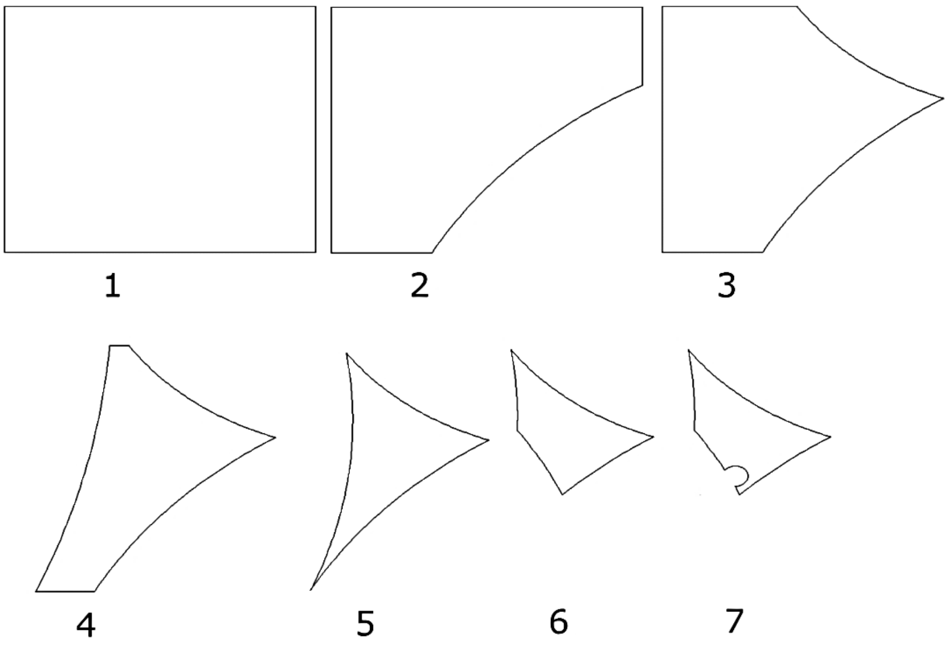
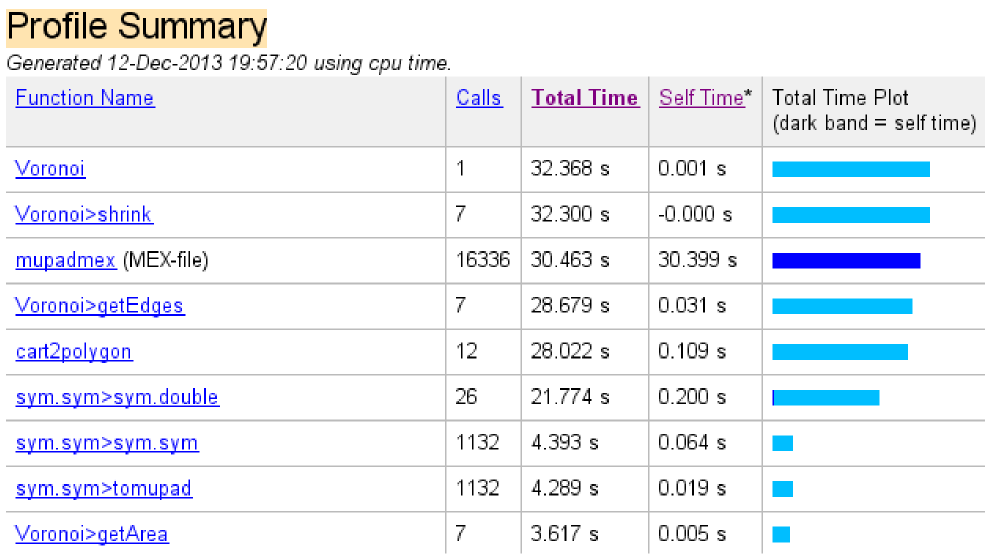


Figure 13 Cutting in a polygon.


Profile Summary

Generated 12-Dec-2013 19:57:20 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
Voronoi	1	32.368 s	0.001 s	
Voronoi>shrink	7	32.300 s	-0.000 s	
mupadmex (MEX-file)	16336	30.463 s	30.399 s	
Voronoi>getEdges	7	28.679 s	0.031 s	
cart2polygon	12	28.022 s	0.109 s	
sym.sym>sym.double	26	21.774 s	0.200 s	
sym.sym>sym.sym	1132	4.393 s	0.064 s	
sym.sym>tomupad	1132	4.289 s	0.019 s	
Voronoi>getArea	7	3.617 s	0.005 s	

Figure 14 Timing breakdown of the most optimised version with 7 points

Figure 14 shows the nine most time critical functions. It is clear that mupadmex causes the biggest problems by using up to 94% of the total execution time. Without the additive weights this method is three to thirty times faster for the same problem size.

5 OpenCL implementation

5.1 Environment

5.1.1 Introduction

The test system runs on Microsoft's Windows 8 Professional operating system. This makes the Microsoft Visual Studio IDE an obvious choice as development environment. Microsoft Visual Studio 2013 is not compatible with Nvidia's OpenCL SDK but Microsoft Visual Studio 2012 offers full compatibility. Visual Studio natively supports C, C++, C#, VB.NET and F#. So the choice for this project, C, is already supported and needs no further set-up. The professional version of Microsoft Visual Studio is available for students for free.

One downside to Microsoft Visual Studio is that the C11 standard is not supported and the C99 standard is only partially supported. Instead we are bounded to the first standard of C, C89 or ANSI-C. Figure 15 is the welcome screen of Microsoft Visual Studio.

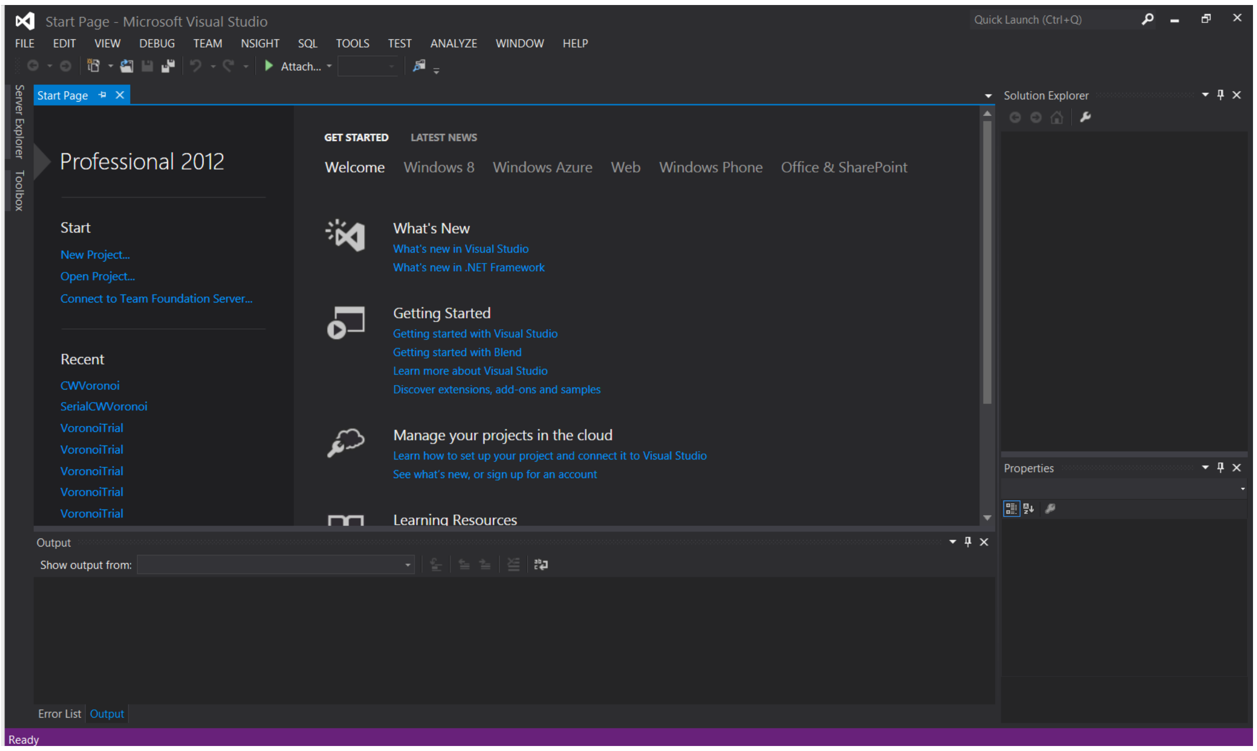
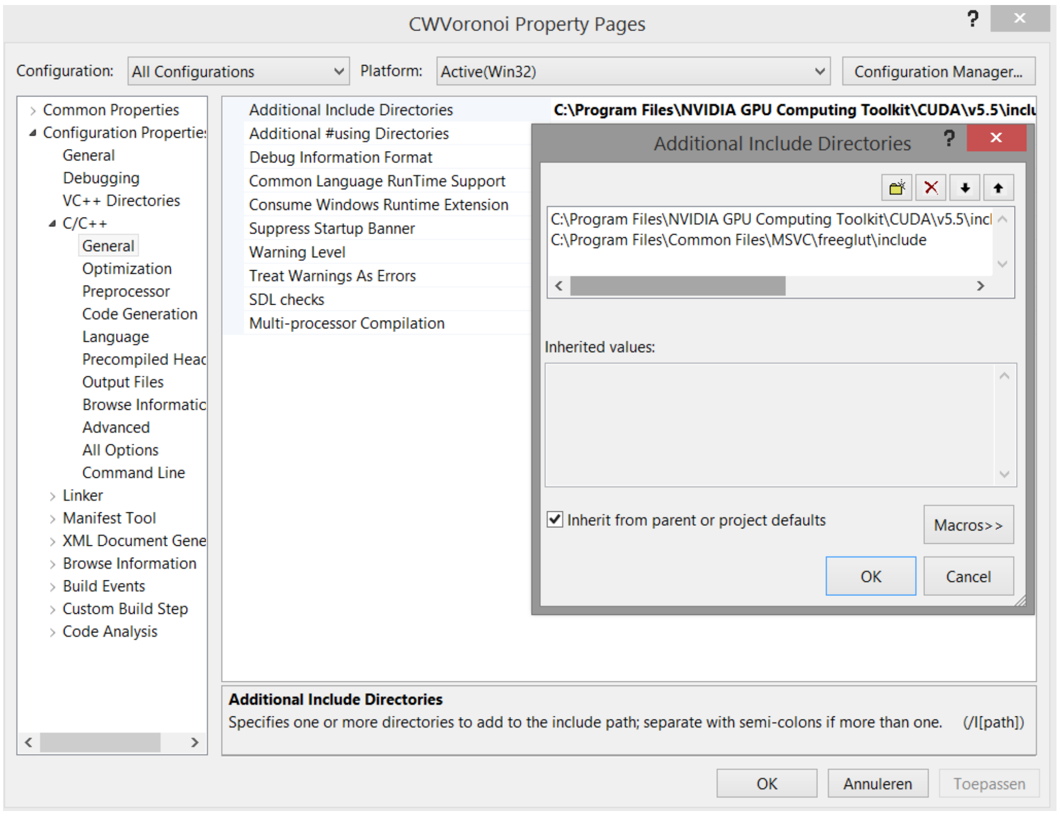

Start Page - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW DEBUG TEAM NSIGHT SQL TOOLS TEST ANALYZE WINDOW HELP
Attach...
Server Explorer Toolbox
Start Page
Solution Explorer
Professional 2012
GET STARTED LATEST NEWS
Welcome Windows 8 Windows Azure Web Windows Phone Office & SharePoint
Start
New Project...
Open Project...
Connect to Team Foundation Server...
Recent
CWVoronoi
SerialCWVoronoi
VoronoiTrial
VoronoiTrial
VoronoiTrial
VoronoiTrial
VoronoiTrial
What's New
What's new in Visual Studio
What's new in .NET Framework
Getting Started
Getting started with Visual Studio
Getting started with Blend
Learn more about Visual Studio
Discover extensions, add-ons and samples
Manage your projects in the cloud
Learn how to set up your project and connect it to Visual Studio
See what's new, or sign up for an account
Learning Resources
Properties
Output
Show output from:
Error List Output
Ready

Figure 15 Microsoft Visual Studio 2012 Professional

5.1.2 Set up

To set up Microsoft Visual Studio for OpenCL we need to install the SDK of our choice. For our system with an Nvidia GPU the Nvidia version is the most logical choice. The OpenCL package is part of the Nvidia GPU Computing Toolkit also used for CUDA GPGPU applications. Nvidia provides an installer that places all the necessary libraries, header files and other tools in the system and includes the Nsight menu in Visual Studio. The only thing the programmer has to do to get started with OpenCL is to set up Visual Studio. The setup menu is shown in Figure 16.


CWVoronoi Property Pages

Configuration: All Configurations
Platform: Active(Win32)

Additional Include Directories: C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.5\inclu

Additional Include Directories

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.5\incl
C:\Program Files\Common Files\MSVC\freeglut\include

Inherited values:

☑ Inherit from parent or project defaults

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

Figure 16 Setting up OpenCL

In the Properties of the project we need to change a few entries to include OpenCL. In Figure 16 one of these entries is visible. It is also noticeable that we have not only included Nvidia's toolkit but also freeGLUT. freeGLUT is used as window manager for rendering the output image. With mainCRTStartup "WinMain" is defined as the entry point for the application instead of "main". This is necessary because otherwise a reference error is encountered. Table 13 shows all the changes we need to make in the properties menu.

Table 13 OpenCL property menu entries

Location	Property
C/C++ > General > Additional include Directories	. \NVIDIA GPU Computing Toolkit\CUDA\v5.5\include .\freeglut\include
Linker > Gerneral > Additional Library Directories	. \NVIDIA GPU Computing Toolkit\CUDA\v5.5\lib\Win32 .\freeglut\lib
Linker > Input >Additional Dependencies	OpenCL.lib Opengl32.lib freeglut.lib
Linker > Advanced > Entry Point	mainCRTStartup

5.2 Choice of programming approach

We chose to use the distance list approach as discussed in chapter 1.2. We made this choice because its implementation can be done with one kernel file where the different kernel instances can be executed without data dependencies with the other instances. To calculate all distances to every generator for one pixel we only need a list of all generators with their coordinates and weights and the coordinates of the pixel. This data is static so the execution of one kernel instance has no effect on other instances. This means the kernels instances can be executed out-of-order. Approximation by a digital image is not as efficient to parallelise as we need to do multiple iterations with a changing image. The execution of one kernel instance would influence subsequent instances and necessitate in-order execution making the program more complex and partially sequential.

5.3 CWVoronoi.c

This is the main file of our program. It can be divided in three general parts: reading the input, processing the output of OpenCL and rendering the output image with OpenGL.

5.3.1 #Include and #define

In addition to the standard input-output library (stdio) and the standard library (stdlib) we include the math and time header files to give us access to mathematical functions as well as calculating the time spent in parts of the code. We also included OpenCL.h. This is the header to our OpenCL code, and the freeGLUT library for OpenGL rendering. We also defined a few macros. WIDTH and HEIGHT are the respective width and height of the plane in which all generators should be placed. We chose to make this the 720p HD resolution or 1280 vertical pixels by 720 horizontal pixels. We also defined the font we want to use with OpenGL and the maximum dataset size for the input. The last define is disable_output. If set to one the Voronoi tessellation will not be printed and no other output will be sent to the terminal. This is used for benchmarking the program.

5.3.2 Reading input file

Reading the input file is a trivial task in C. However it is notable that we use fopen_s instead of the ANSI-C fopen. The MSVC compiler used by visual studio marks a few commonly used I/O functions as deprecated and offers secure replacements. We could still use fopen and the other deprecated functions by including the macro `#define CRT_SECURE_NO_WARNINGS` if required for compatibility with other compilers.

Our read function detects signs of invalid data such as points outside of the defined plane, multiplicative weights with value zero or additive weights smaller than zero. In case one or more of these problems are encountered the application halts and prints the coordinates of the violating entry.

5.3.3 Processing the OpenCL data

The data returned by the OpenCL kernel is a 2D array with each element containing the number of the generator it belongs to. However this information is not directly printable. To be able to display the Voronoi plot we want to convert this data into a binary image. To achieve this we need to evaluate all elements in the array one final time. To determine if a pixel is a border we look at all its neighbours. If all these neighbours are assigned to the same area as the pixel we are currently evaluating this pixel is not a border pixel. If there is at least one pixel belonging to another generator this pixel is marked as a border pixel. After we processed all the items in the matrix we end up with a binary image. With this method two white pixels belonging to different regions are always separated by two black pixels. We are unable to process the edges of the plane in a similar way since we did not calculate all their neighbours. Instead we mark them automatically as being a border pixel. This has the added advantage of making all regions closed.

This whole process is demonstrated on small scale in Figure 17 where we examine what could happen at the border between the regions of generators '7' and '9'. In the top row we see how the 3x3 box starts at (1,1) and moves row by row to the position (N-1,N-1). If the 3x3 box only contains pixels owned by the same generator it will place a zero in the output matrix in the corresponding place. Otherwise a one will be placed. We can then map each element directly to a pixel to display it. This output does not include the positions of the generators. These are added by the GLUT draw function.

Because it is possible to parallelise this code it can be implemented as a kernel in a future version of this project. The current sequential version however is typically executed in 0.020 seconds or less. The time the program spends in this section of the code is independent from the complexity and size of the input and effectively constant. Due to these factors the potential benefits gained from implementing a secondary kernel are minimal and implementation has been deemed to be of low priority for this project.

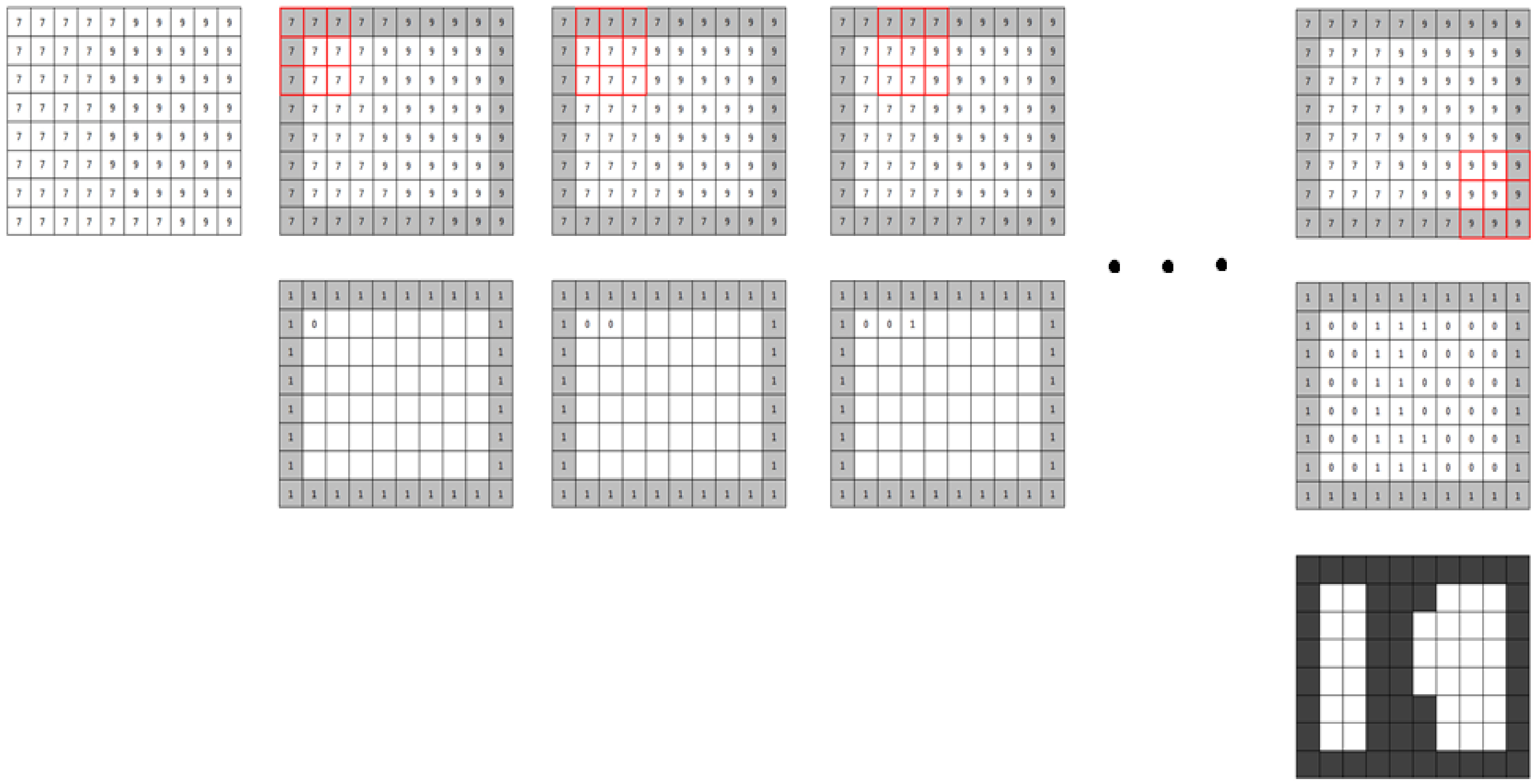


Figure 17 Border detection process

5.3.4 Rendering in OpenGL

Now we finally have a binary image of the Voronoi plot we still need to display it. The OpenGL part of the code for rendering the image consists of three functions: initGlut, mouseEvents, and drawPlot .

In initGlut we take necessary steps to prepare a window for displaying the results of the Voronoi plot. The initialisation code is shown in Figure 18.

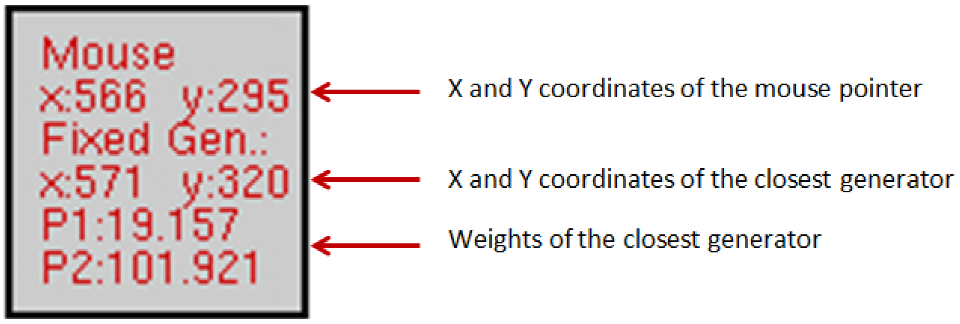
```
void initGlut(int argc, char **argv)
{
    // initialise GLUT
    glutInit ( &argc, argv );
    /* define window mode
     we choose the RGB color mode and a single buffer
     since we won't be displaying moving images this is less demanding
     for the system */
    glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
    /* define window, -1 means we let the window manager decide on x and y
coordinates */
    glutInitWindowPosition(-1, -1);
    /* define window size in pixels */
    glutInitWindowSize(WIDTH, HEIGHT);
    /* create the actual window with the required title */
    glutCreateWindow("Compoundly weighted Voronoi");
    /* set white background */
    glClearColor ( 1.0, 1.0, 1.0, 0.0 );
    /* set drawing range, we need to convert the size to GL datatypes */
    gluOrtho2D(0.0,(GLdouble)WIDTH,0.0,(GLdouble)HEIGHT);
    /* indicate the drawing function for GLUT
    cannot pass arguments to glutDisplayFun, this explains why global vars
are needed */
    glutDisplayFunc(drawPlot);
    /* track mouse */
    glutPassiveMotionFunc( mouseEvents);
    /* enable exiting the glutMainLoop (freeGLUT supports this, the
original GLUT doesn't) */
    glutSetOption(GLUT_ACTION_ON_WINDOW_CLOSE,
GLUT_ACTION_GLUTMAINLOOP_RETURNS);
}
```
Figure 18 GLUT initialisation

All functions are documented in the code. The last line of code is a particular interesting line. gluSetOption enables the programmer to set parameters for the OpenGL environment. This particular variable is the main reason why we chose for freeGLUT over the nearly identical GLUT. It enables us to give control back to the main program loop after exiting the window while the original GLUT would halt the application. This makes our application more suitable for further development and use in other programs. In the init function we also define the drawing function with glutDisplayFunc.

MouseEvents keeps track of the mouse position and uses that position to alter global variables, these variables are then used by the drawPlot function to give the user additional information about the plot. The coordinates of the mouse pointer are shown as well as the coordinates and weights of the nearest generator. In this function we also changed the standard ANSI-C function sprintf to the safer sprintf_s. The biggest advantage of this is that if the user enters values in the input file that overflow the allocated variables the program will now no longer crash as it did with the sprintf function. Whenever the mouse is moved glutPostRedisplay forces a refresh of the window.

The final OpenGL function, drawPlot, fills the canvas with the generated Voronoi tessellation. To draw the tessellation we will evaluate every item in the plot[] matrix. Each item in this matrix corresponds with a pixel in the window on the same coordinates. To draw this single pixel we need to delimit all the drawing actions with glBegin(GL_POINTS) and glEnd. This will make OpenGL treat all the vertices we define in this delimited group as single pixels. To define these vertices we have two choices depending on the required precision. Since our data is not sub-pixel precise glVertex2i suffices, otherwise glVertex2f could give us higher precision. In a similar manner we plot all the generators specified in the input file with larger, red pixels. A notable characteristic of the draw function is that no arguments can be passed to it directly. This means global variables are necessary.

In this function we also draw the overlay with additional information. This additional information is drawn over the current plot and again uses the same drawing principle we used to plot single points. Some of the information provided to the user is in text form. To print numbers and text we need to use the OpenGL function glutBitmapString. The data in this information box is seen in Figure 19.



```
Mouse
x:566  y:295
Fixed Gen.:
x:571  y:320
P1:19.157
P2:101.921
```

X and Y coordinates of the mouse pointer
X and Y coordinates of the closest generator
Weights of the closest generator

Figure 19 Information box

The information box is rendered in the top left corner of the Voronoi diagram. It features the mouse coordinates as well as coordinates of the nearest generator. The nearest generator is also rendered in green instead of the default red colour. This enables the user of the program to analyse the diagram in greater detail. The data is updated on every mouse movement.

5.3.5 Running the executable

This program is compiled to a 32 bit Windows executable. To pass arguments to the program we use the Windows command prompt. To load the program we use "CWVoronoi.exe data". The data parameter is a file. We prefer using .txt files due to their easy editing possibilities. In the file there should be four columns with from left to right the x coordinate, y coordinate, multiplicative weight and the additive weight all separated by spaces. The file length is not restricted but can be capped internally in the application. Currently the internal limit is 5000 points. All data after the 5000th line will be ignored. To run the program it is important that the freeglut.dll file is in the same folder as the executable.

If the user omits the filename parameter the program will use the coordinates.txt file if a file with that name is present in the current folder. If an invalid filename has been entered execution will halt.

5.4 OpenCL.c

Executing a program on the GPU of a computer with OpenCL requires a few mandatory and a few recommended steps. If we outline the general principle it comes down to some basic steps.

1. Query system information
2. Create access points to required hardware
3. Create and load memory buffers
4. Load and build kernel file or load binary
5. Set kernel arguments
6. Execute kernel
7. Retrieve results
8. Destroy objects and free resources

The order of step 2 and 3 and 4 can be changed if wanted all other steps have dependencies with previous steps. Table 14 shows all the OpenCL specific functions in the order we used them with a short description.

Table 14 Used OpenCL functions [34]

#	Function name	Explanation
1	clGetPlatformIDs	OpenCL works with the platform model, a platform is defined as a host connected to one or more OpenCL devices. On a regular computer there can be e.g. one platform consisting of one CPU and one GPU.

2	clGetDeviceIDs	Gets the list of devices available on the selected platform and chosen type (CPU, GPU, Accelerator).
3	clGetDeviceInfo	Query for device information such as name, number of cores and maximum array size. Not required but useful.
4	clCreateContext	A context is a container and manager for different objects (memory, program, kernel, command-queues) for the OpenCL runtime. A context can have multiple devices and all objects in a context can be shared by those devices. A context provides synchronisation points for all devices for e.g. memory updates.
5	clCreateBuffer	Create memory objects that can be accessed and used by both the host and the OpenCL device.
6a	clCreateProgramWithSource	Creates a program object from source code for the defined context.
6b	clCreateProgramWithBinary	Creates a program object from a binary for the defined context, mutually exclusive with clCreateProgramWithSource.
7	clBuildProgram	Builds executable from the program object created with clCreateProgramWithSource or clCreateProgramWithBinary.
8	clGetProgramBuildInfo	This command should only be executed when clBuildProgram fails to return CL_SUCCESS. It allows the programmer to view the problem with his code.
9	clCreateKernel	Creates a kernel object from the program object. A program object can contain multiple kernels. Each function in a program object with the __kernel qualifier can yield a kernel object.
10	clSetKernelArg	Set the arguments to a kernel object for execution. These are all arguments a function with the __kernel qualifier has.
11	clCreateCommandQueue	All commands on context objects such as memory are queued in the command queue. Queuing can be in-order as well as out-of-order .
12	clEnqueueWriteBuffer	Writes data from host memory to buffer accessible by both the host and the OpenCL device.
13	clEnqueueNDRangeKernel	Executes data parallel kernel on the selected OpenCL device.
14	clEnqueueReadBuffer	Copies a buffer object to the host memory.
15	clFlush	Forces all remaining commands in the command queue to the corresponding device.
16	clFinish	Holds the program until all commands issued to the corresponding device have been executed.
17	clReleaseKernel	Releases the defined kernel object.
18	clReleaseProgram	Releases the defined program object.
19	clReleaseMemObject	Releases the defined memory object.
20	clReleaseCommandQueue	Releases the defined command queue.
21	clReleaseContext	Releases the defined context.

Most consumer PCs can have one or two platforms installed: one platform for the CPU and another one for the GPU. In the current application we take the first platform that is found by the system. The next step is selecting the device from that platform. To do this we first query and select any devices of the "GPU" type. If that fails we fall back on a CPU type device. After the device selection we can create a context. Our context will only contain the selected device but selecting multiple devices for a higher level of parallelisation is also possible.

For our program we need two buffers: one input buffer and one output buffer. The input buffer contains four floats per generator point so the size of the required buffer is 4*lines*sizeof(float). The OpenCL kernel returns a 2D array with an integer for each pixel in the x-y plane. The size we need to reserve for this buffer is planeWidth*planeHeight*sizeof(int).

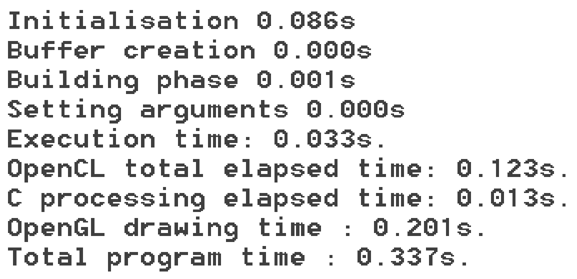
After reading and building the kernel the last step is to create a command queue before we can start with the actual execution of the kernel. This command queue is bound to a single device in a context and the runtime API. The runtime API now can be used to define and modify all memory objects. We first load the input buffer with the float array from CWVoronoi.c, modify it with a data parallel kernel and finally we copy the result to the output buffer. These three actions are all executed in the OpenCL runtime API.

Note that compilation of the OpenCL kernel files is not done at the same time as the rest of the program. Instead the OpenCL Toolkit is responsible for building the kernels. OpenCL supports two different compilation strategies: online and offline compilation. With online compilation the kernel file will be compiled at runtime. With offline compilation a binary file is loaded instead of a readable .cl file. The advantages and disadvantages can be found in Table 15.

Table 15 Online vs. Offline compilation

	Online	Offline
Advantages	• Higher portability, can be compiled on all OpenCL conformant systems	• Faster, less overhead in program execution as compiling the kernel is one of the most time consuming steps in the OpenCL initialisation
Disadvantages	• Source code is readable, might be unwanted in certain applications • More time consuming	• Not portable between different systems

In this implementation we made the choice to work with online compilation. Although it has a speed disadvantage it offers greater portability between different systems. The speed disadvantage also proves to be minimal as the total building time is around one microsecond as shown in Figure 20. Changing between online and offline compilation is trivial. The code to read the source code .cl file is identical to the one reading the binary .clbin file. clCreateProgramWithSource and clCreateProgramWithBinary should be swapped and finally clBuildProgram is used only with online compilation.



```
Initialisation 0.086s
Buffer creation 0.000s
Building phase 0.001s
Setting arguments 0.000s
Execution time: 0.033s.
OpenCL total elapsed time: 0.123s.
C processing elapsed time: 0.013s.
OpenGL drawing time : 0.201s.
Total program time : 0.337s.
```

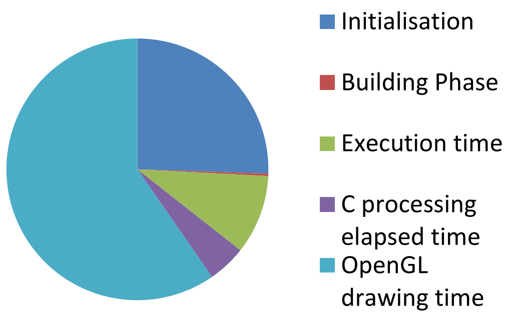


Figure 20 OpenCL timing breakdown

5.5 CWVoronoi.cl

The last part of the OpenCL implementation is the actual OpenCL kernel. The kernel is written in OpenCL C, a variation on C99 that enables parallel programming. The kernel itself is a single function marked by the __kernel qualifier. OpenCL .cl files can contain multiple kernels.

A kernel is executed for each work item defined by the OpenCL runtime. Work items are the smallest division of the problem that can be parallelised. A work-item is defined by its global ID, the coordinates in the index space. Work-items are grouped in work-groups. These work-groups are often the upper limit for simultaneously executable work. This limit is imposed by hardware limitations. Some implementations of the OpenCL API tolerate work-groups that exceed device specifications while guaranteeing correct execution. This feature should be avoided for portability of the code.

Our kernel function as seen in Figure 21 has five inputs, two of them are buffers: one input buffer with the generator coordinates and weights and a second buffer for the output. Both buffers are placed in the global address space. We have to place both buffers in the global address space because all instances of the kernel need access. The other three inputs are integers. The first two integers are the dimensions of the output array. This helps us to calculate the position of the kernel instance in the global scheme. The final integer is the amount of rows in the input array. Kernels always have to be of the void type, returning data is not part of the OpenCL standard.

```
__kernel void VoronoiOne(__global float *inputArray, __global int *result,
int width, int height, int points)
```
Figure 21 Kernel function header

For data parallel applications OpenCL uses get_global_id(0) to determine the position of a kernel instance in the global scale. However this global ID is only one dimensional so we need to do some calculations on it to get usable x and y coordinates.

To convert the 1D position to 2D coordinates we make the assumption we go through the plane in horizontal direction first and then in vertical direction. First we take the modulus of the global ID and the width. This gives us the x position in the current line. For the y position we divide the global ID by the width. Since both these numbers are integers this will give us the line number. We cast both the x and y position to the float type to be able to calculate the square root. Figure 22 shows the complete formulas.

```
//width and height of current pos(s)
xPos = (float)((globalID ) % width);
yPos = (float)(globalID/ width);
```

Figure 22 Determining the position of this kernel instance in the global scale

To calculate the closest generator to a certain point P(xPos, yPos) in the x-y plane we use an iterative process. We start with the maximum possible distance between two points. This is the distance between point P(0,0) and P(width, height). According to the Pythagoras theorem this is

$$D_{max} = \sqrt{(0 - width)^2 + (0 - height)^2}$$

Then we will calculate the distance from the point P(xPos, yPos) to each generator in the generator list updating D each time a smaller value is found. The distance formula here is the weighted distance formula so we need not only the x and y position of the generator but also the additive and multiplicative weights of that generator.

Each time the distance D is updated we also update an integer that holds the number of the generator linked to that distance. Once we processed all generators for a certain point we save this value in the output buffer on the same coordinates xPos and yPos. Since the output buffer is another 1D array we can use the global ID for this.

It is worth mentioning that in an OpenCL kernel double precision computation is not enabled by default. We initially decided to keep double precision disabled to provide higher speeds and compatibility as not all GPU's support double precision computation. If GPU's provide double precision calculations there is a typical theoretical performance loss of factor 3. To test if there was a major difference in accuracy of the output image or other benefits we created a double precision kernel. Our Nvidia Quadro NVS card is double precision enabled by default so the only changes needed were to the kernel. As demonstrated in Figure 23 and Figure 24 there is no noticeable difference between the single and double precision output.

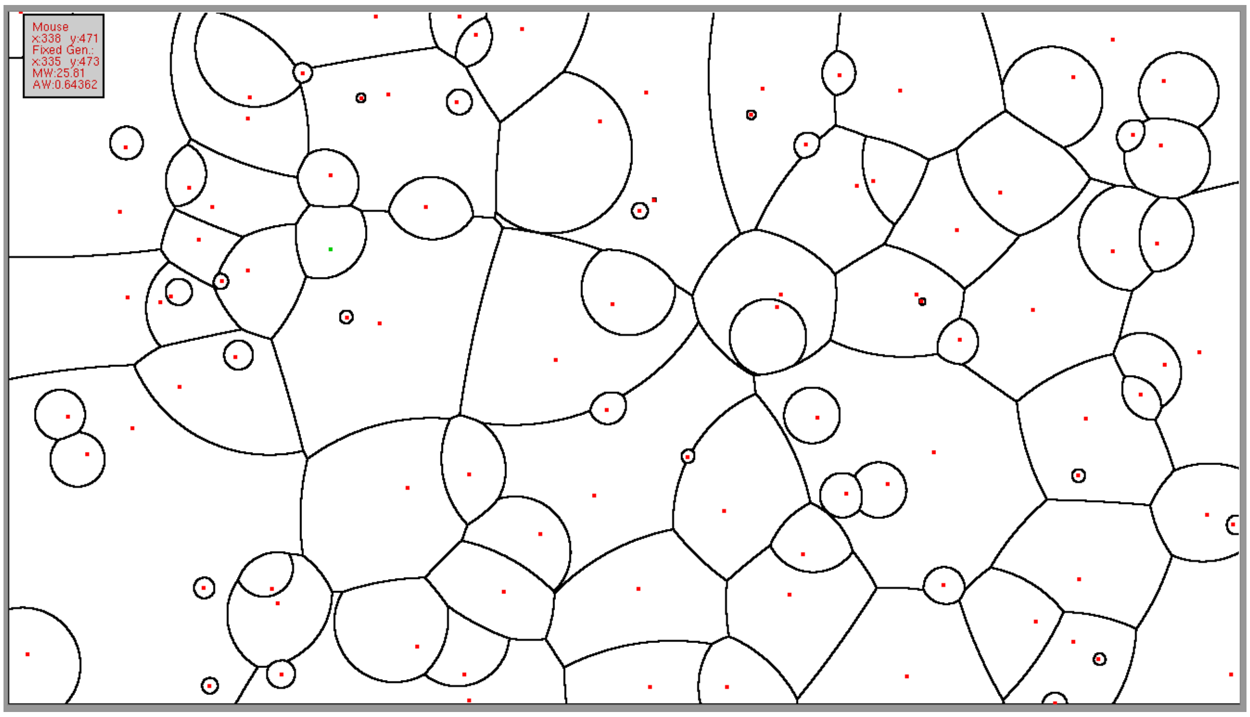


Figure 23 Single precision floating point Voronoi

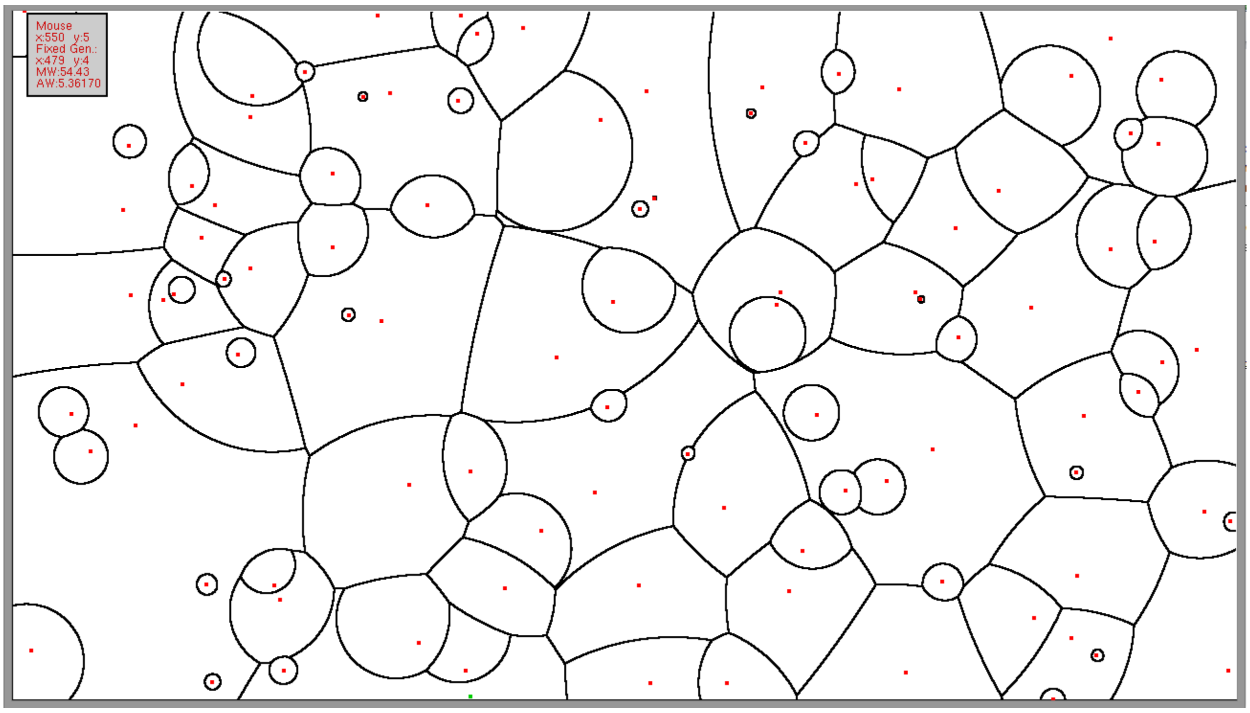


Figure 24 Double precision floating point Voronoi (same dataset)

Although there are no visible changes between the single and double precision floating point pictures we note that the total program time for this particular dataset almost doubled. The OpenCL execution time was ten times higher opposed to the predicted theoretical three times. For this relatively small dataset this is not an issue but larger datasets will become about ten times slower as the OpenCL execution time will become a more important factor with larger datasets while the other parts of the code are executed in constant time. Two of the measurements are printed in Figure 25 and Figure 26. Note that for this comparison the calculation of the tessellation is done multiple times. We took an average run for both cases.

```
OpenCL execution time: 0.038s.
OpenCL total elapsed time: 0.135s.
C processing elapsed time: 0.013s.
OpenGL drawing time : 0.190s.
Total program time : 0.338s.
```

Figure 25 Single precision floating point timing results

```
OpenCL execution time: 0.349s.
OpenCL total elapsed time: 0.415s.
C processing elapsed time: 0.013s.
OpenGL drawing time : 0.189s.
Total program time : 0.617s.
```

Figure 26 Double precision floating point timing results

From these observations we can conclude that our initial motivation to make a single precision kernel proved valid. In some applications the double precision accuracy might be more valuable than the execution speed but for our application the accuracy of the single precision floating point kernel is adequate.

5.6 Calling the OpenCL program from Matlab

One of our goals was being able to execute the program directly from the Matlab environment with Matlab variables. Because our program has its own printing function we decided to make Matlab write the data to a file and let the OpenCL program read that file. To achieve this we need only two Matlab commands: one to write the file and a second one to execute the program. Both functions are printed in Figure 27.

```
dlmwrite('coordinates.txt', A, 'newline', 'pc', 'delimiter', ' ');
!CWVoronoi.exe;
```

Figure 27 Linking matlab to OpenCL

Dlmwrite will create a file 'coordinates.txt' or overwrite an existing file with that name. In the file it will place the contents of matrix A. Subsequent numbers in the matrix are delimited by a space and rows are ended with a newline symbol.

The program will now execute as normal. A Voronoi Tessellation will be displayed if it can be generated from the provided data. If the user saves bad data to the coordinates.txt file the program will throw an error and halt execution. On error or successful execution output data will be printed to the Matlab console.

6 Comparison

6.1 Sequential implementation

To make a valid comparison we wrote a simple sequential version of this code in C and Matlab. The basic principle is the same. For the C version the opencl.c file has been replaced by the sequential.c file that contains similar code to the OpenCL CWVoronoi kernel.

The sequential implementation needs at least one extra for-loop to calculate all points but the distance functions and main principles are the same. We chose to use two for loops: one for the x coordinates and one for the y coordinates to make it easier to find the position in the grid. A similar approach as in the kernel could be used to determine the position. The sequential implementation yields exactly the same output as the OpenCL version. This is demonstrated in Figure 28.

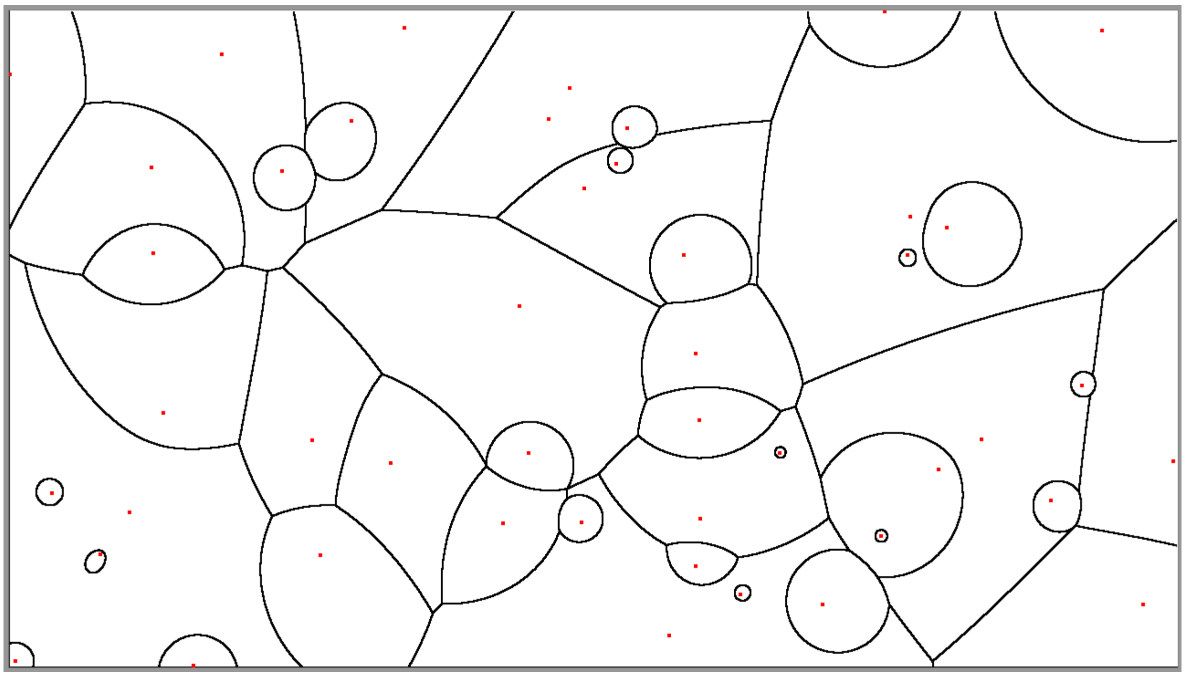


Figure 28 The output of the sequential code is identical to the OpenCL code

The reason we evaluate this sequential version is that this code does not need any of the OpenCL initialisation or buffer transfers. These parts are a big overhead for the OpenCL program so a sequential program could be faster in small datasets. Our 100 point goal is considered a small dataset for GPGPU computation.

6.2 Speed and input data sizes

To compare the sequential and the parallel OpenCL version we made some minor adjustments to the code. We disabled all the prints and the entire OpenGL part. A Matlab script was made to generate a new random input file and start the executable one hundred times. With the Matlab profiler we checked what amount of time was spent in the executable for different dataset sizes. This means we compare the complete programs including input file reading and OpenCL overhead. We tested for dataset sizes 10 to 10.000. Larger datasets work on the sequential versions but the OpenCL version ran out of resources. In theory it is possible to run larger datasets in the OpenCL implementation but it would require better hardware then the available hardware in the test system. The result of this comparison can be found in Table 16 and Figure 29. For this comparison we also translated the sequential code to a Matlab file and ran the double precision variant of the kernel.

Table 16 Average timing results for total program time

Dataset size	Matlab	Sequential C	OpenCL Single precision	OpenCL Double precision
10	0,837s	0,238s	0,170s	0,206s
100	5,069s	1,455s	0,191s	0,353s
1.000	46,928s	12,367s	0,500s	5,012s
10.000	465,911s	123,276s	3,016s	N/A
100.000	4670,870s*	1233,762s*	N/A	N/A

* For 100.000 points we only ran ten simulations instead of 100 because linear time was expected and the simulations could potentially take multiple days.

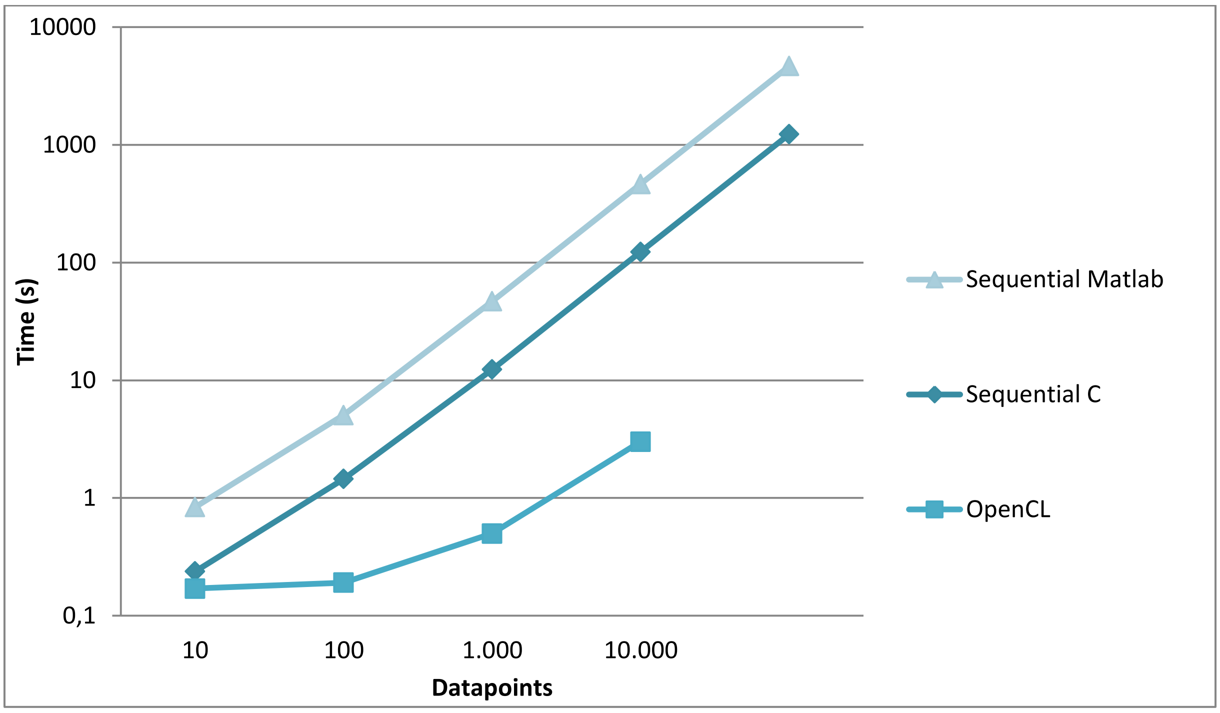
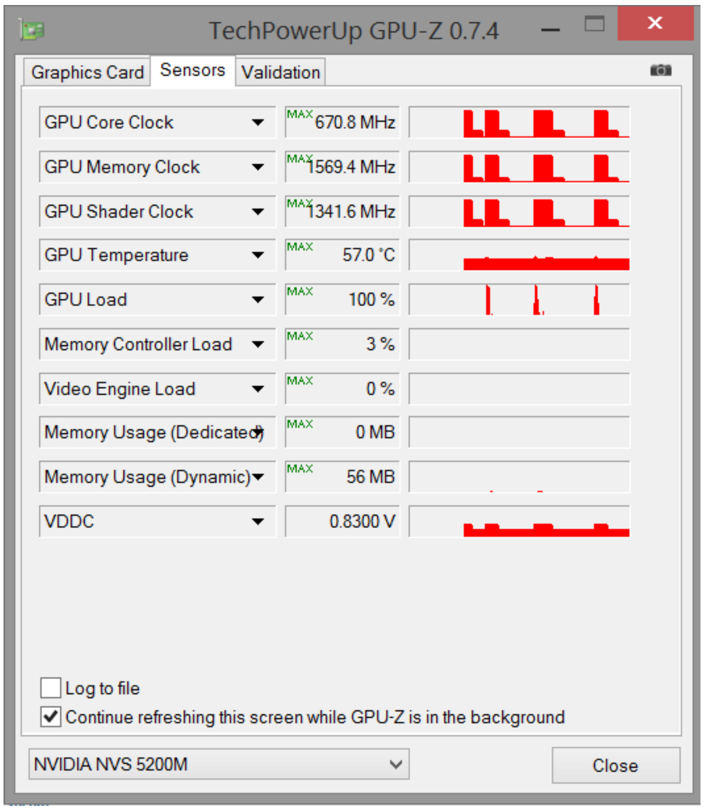


Figure 29 Timing results for sequential Matlab, sequential C and OpenCL (single precision)

We see that for small data sets the advantage of OpenCL over the sequential C program is almost negligible. Although 40% faster is significant, both programs still execute in under a second. At a dataset size of 1000 points the difference increases further. The OpenCL version is now almost 25 times faster and still executing in under a second. The speed of both sequential programs appears to have a linear relation to the dataset size. At 100.000 points the OpenCL version failed, crashed the driver and returned error code -5: CL_OUT_OF_RESOURCES. This can have two causes: we are trying to access too much memory or we try to execute the kernel with local workgroup sizes that exceed the hardware specifications. The double precision version already failed at problem sizes just over 1.000 generators with the same symptoms.

We ran an analysis tool on the GPU to locate the problem but the memory usage appeared to be well within boundaries at any time. Figure 30 is a screenshot of our troubleshooting attempt.



TechPowerUp GPU-Z 0.7.4

Graphics Card | Sensors | Validation

```
GPU Core Clock              MAX 670.8 MHz
GPU Memory Clock            MAX 1569.4 MHz
GPU Shader Clock            MAX 1341.6 MHz
GPU Temperature             MAX 57.0 °C
GPU Load                    MAX 100 %
Memory Controller Load      MAX 3 %
Video Engine Load           MAX 0 %
Memory Usage (Dedicated)    MAX 0 MB
Memory Usage (Dynamic)      MAX 56 MB
VDDC                        0.8300 V
```

☐ Log to file
☑ Continue refreshing this screen while GPU-Z is in the background

NVIDIA NVS 5200M

Close

Figure 30 GPU-Z, GPU monitoring software

We see three spikes where GPU activity is 100%. All these spikes are attempted computations of datasets of 100.000 generators. In the Memory Usage (Dynamic) field we see two corresponding small peaks were memory usage is maximum 56MB. On the third run less memory was used at the time of the driver crash and there is no visible peak. From this we can exclude excessive memory usage as possible cause of the crash. Instead the problem is located with the local workgroup sizes.

Common practice, when designing OpenCL programs, is to let the OpenCL API determine the local workgroup size. Most API's are designed by the hardware manufacturer and implement a way to determine the optimal work group size for faster execution. We tried to gain more control over the program by changing the local workgroup size to a size the GPU could definitely handle. This workgroup size might not be the optimal for speed purposes but is guaranteed to be within the bounds our GPU's hardware capabilities. First the maximum local workgroup size supported by the GPU is queried. This maximum is used when we queue the kernel. The code can be seen in Figure 31.

```
/* Query the maximum workgroup size supported by the hardware */
clGetDeviceInfo(deviceID, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(cl_uint),
&maxWorkGroupHW, NULL);

/* Use that value in the kernel execution */
clEnqueueNDRangeKernel(commandQueue, kernelOne, 1, NULL, &globalItemSize,
&maxWorkGroupHW, 0, NULL, &k_events[0]);
```

Figure 31 Code changes for maximum workgroup size

Executing this program takes longer than the original one and exceeds the five second mark when the Windows GPU Watchdog halts the kernel execution by rebooting the display driver. This is standard behaviour in Microsoft Windows and many other operating systems and is a safeguard for the operating system in normal circumstances to prevent the computer from locking up. This behaviour is possibly the reason why the API tried to create larger workgroups.

From this we can conclude that we will always have a problem with an input dataset of 10.000 or more generators on the hardware present in our test system. When we take local workgroup sizes that comply with the hardware specifications we exceed the five second mark and the watchdog kills the program. If we take bigger workgroups or let the OpenCL API decide the workgroup size will be too big for the hardware and the driver will crash. The only solution to this problem is to run the program on hardware with greater capabilities or to split up the problem in smaller groups.

6.3 Real world scenarios

The envisioned application of this project was to plot the areas of influence for different transmitters. Instead of dummy data we would load in real-world coordinates and characteristics of antennas placed in the Helsinki region. At this point we concluded that it was impossible to extract an additive weight from the data provided from the transmitters. To describe the signal strength at certain coordinates we replace the current distance formula Dist = M*D+A for the new formula Dist=P1*log(D)+P2 with $P1 < 0$, $P2 > 0$ and D the Euclidean distance to the transmitter. The normal Voronoi Tessellation calculates the shortest distance. Another adjustment is required to convert this to the highest signal strength. Due to the construction of our kernel file implementing these changes in the OpenCL version is a trivial task. The sequential C and sequential Matlab version are also easy to change although recompilation is necessary for the C version.

To make the resulting plots more readable we loaded in a background image of the region under observation. Rendering a background is the most convenient in the Matlab versions. We used a Matlab implementation of the Google Maps API by Zohar Bar-Yehuda[35] that is covered under the BSD license and free for redistribution and use in source and binary forms. This API renders a map to the currently opened plot. To determine which map to render the maxima and minima of the x- and y-axes are used as longitude and latitude respectively.

For the Matlab version of the code this meant we had to make some changes to the code. Since the transmitters are often positioned relatively close integer pixel coordinates aren't accurate enough. Transmitters located at a few kilometres apart will be mapped to the same coordinates. Our specific dataset is included in appendix A. We scale up this dataset so it's stretched between 0 and 1280 for x and 0 and 720 for y. We choose these values arbitrarily because 1280x720 is also the working resolution of our OpenCL implementation and the data could be used there as well. After upscaling the data we can calculate the Voronoi tessellation and downscale the results to the original size. Plotting this matrix instead of the upscaled version enables us to use the Google API without further adjustments. Figure 32 is our result for the Helsinki region.

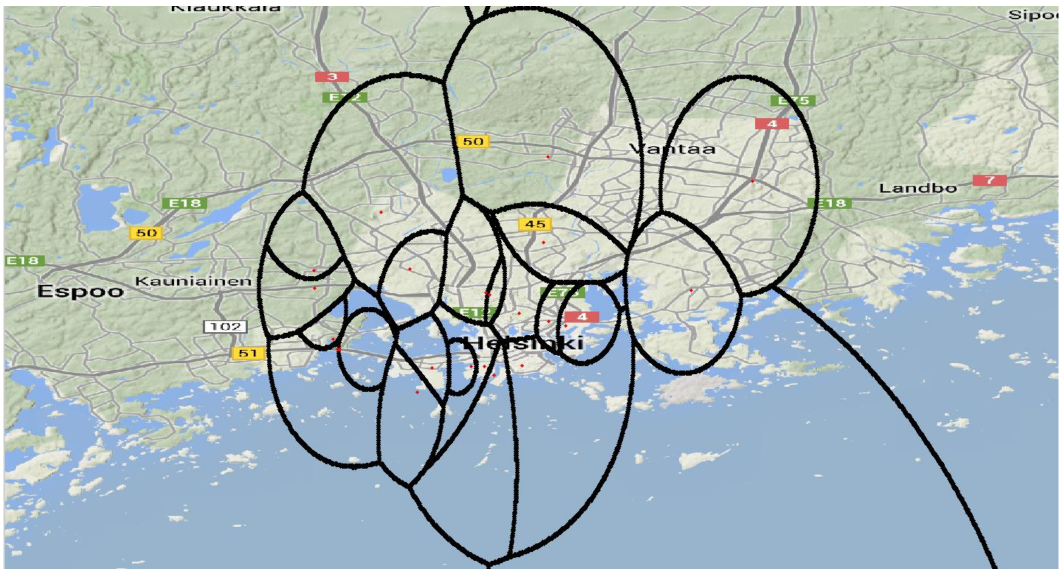


Figure 32 Sequential Matlab with Google Maps API background

For the sequential C and OpenCL versions there is no plug-and-play version of the API available. To be able to display a map we expanded the OpenGL code to draw a background image from a bitmap file. This background image has to be delivered by the user and the coordinates of the generator should be relative to the origin of this image. To draw the bitmap we use the glDrawPixels function. This function is not the fastest option to draw a predefined image but offers an easier implementation opposed to the default printing of images by loading textures and rendering them with quad structures.

We choose to limit the background capabilities of the program to bitmap images only. Bitmap images require no extra decompression steps opposed to jpeg, png or other compressed image formats. Figure 33 is a plot for a similar dataset for the Helsinki metropolitan area with the OpenCL code.

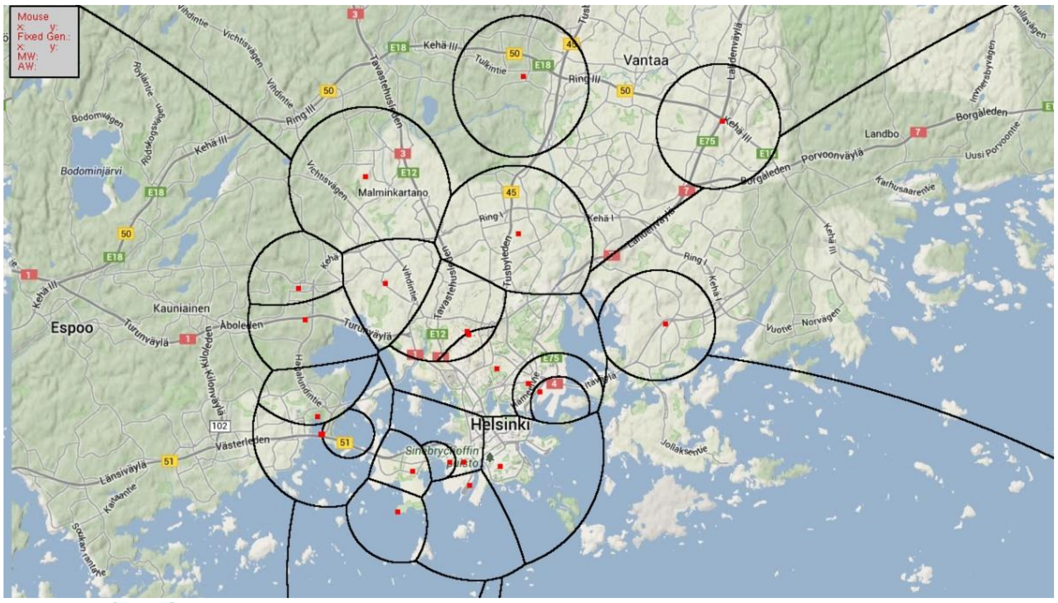


Figure 33 OpenCL Voronoi with background

To generate a Voronoi plot with map we need to add the image name as an additional parameter to the command “CWVoronoi.exe Helsinki.txt Helsinki.bmp”. For now this image has to be a 24 bit bitmap image with the same resolution as defined in CWVoronoi.c. This resolution is currently 1280 by 720 pixels but can be changed by altering two macros in the source code and recompiling the program.

7 Conclusion and future work

When we wrote our sequential C and sequential Matlab version for comparison purposes we expected them to be much slower then they proved to be. Our typical problem size is 100 points or less and at this size both of the sequential versions are competitive with the OpenCL version. While the Matlab version is about 26 times slower and the C version seven times slower for this problem size this can be considered acceptable. The double precision OpenCL implementation is only two times slower but due to compatibility issues with some graphics cards we already excluded this version.

The major tradeoff for the speed of our application is the lack of a scaling feature. When the user would zoom in on the plot the edges between different regions are not recalculated or rescaled and become visually bigger. This disadvantage is shared with both sequential applications but is absent in the much slower polygon intersection implementation. In Figure 34 we see why this lack of rescaling could be an issue.

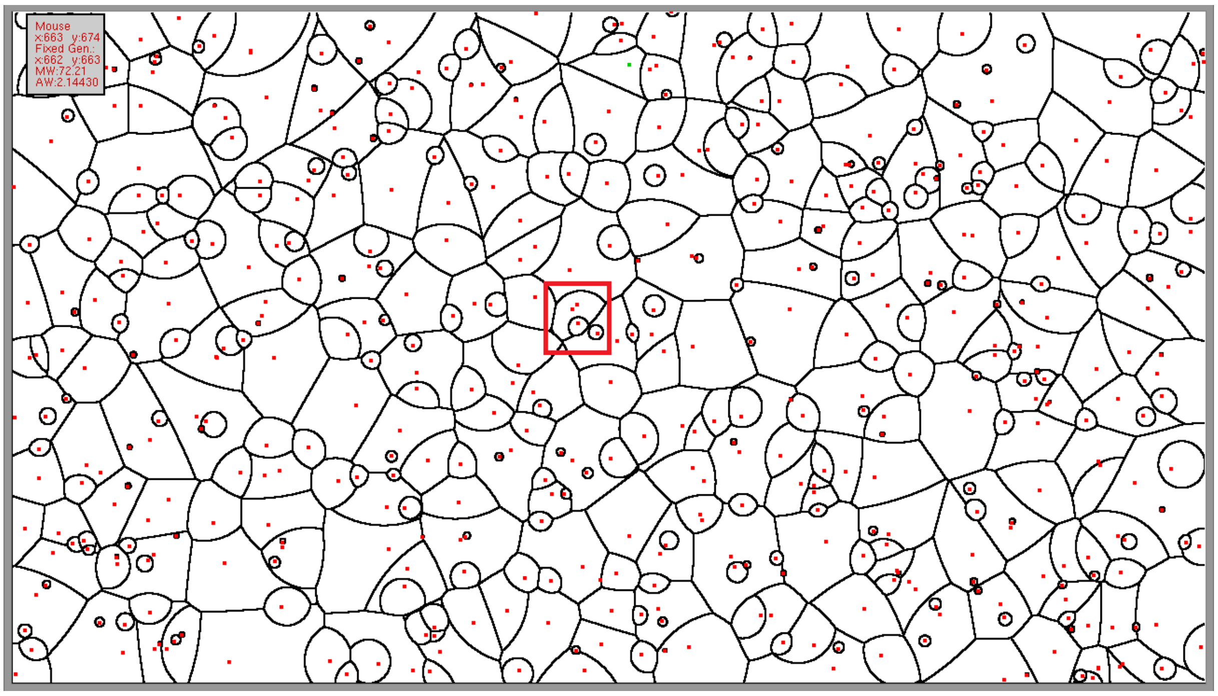


Figure 34 Voronoi Tessellation with 1.000 points.

When we plot some points close to each other some regions can become smaller than a single physical pixel and so it would seem one region contains two generators as seen in the red box. With another method of Voronoi construction such as the intersecting polygon method we could zoom in and examine this region. This is not possible with this implementation. The area would remain invisible and other borders would become wider. A possible improvement to the program would be to make such a zoom function. A suggested approach for this is:

1. Register mouse clicks. The first click is the first corner of the region of interest, the second click is the opposed corner. If required we could draw a rectangle between the first click and the mouse pointer position to give the user feedback on what he is doing. The right mouse button could be a possible "cancel" function.
2. Calculate horizontal and vertical scaling vectors from this input
    a. Scale_x = original_x/ input _x
    b. Scale_y = original_y/ input _y
3. Alter the generator list. Removing generators that are outside of the selection box should be avoided because this can have effects in the box. Instead we remap all coordinates. This requires more memory space so a maximum zoom factor should be established. Remap the generators by multiplication of the coordinates with the scaling vectors.
4. Recalculate the Voronoi tessellation with the new generator list, adjusted widths and heights.
5. Remap the output so that only the area in the bounding box gets plotted. This can be done with altering the parameters of the for loops in the draw function.

Another way to raise the accuracy of the drawing itself is by replacing `glVertex2i` with `glVertex2f`. This enables sub-pixel precision.

A second weakness of the OpenCL and sequential C implementations is not related to the Voronoi tessellation itself but rather the lack of a convenient way to display a background image accurately. This can be resolved in a future version by implementing the Google Maps API like in the Matlab version or alternatively by sending the OpenCL output to a Matlab program that can use this API. However for this application the coarse mapping of the background is suitable. The bitmap we currently render can be considered a proof of concept.

In the opencl.c file some improvements are possible as well. Currently we let the system automatically select the platform that is detected first. On some systems there are multiple platforms available and this might not be the best choice. We could query the amount of devices first with `clGetPlatformIDs(0, NULL, &platformsFound)`. If we do not send an object containing a list of platforms and an integer containing its length this function will only return the number of platforms. We can then use this number to make a list of all platforms. With all platforms known we compare the names property. This returns the company name of the manufacturer of the API. We can then select the API we want by comparing those names. In our application we would prefer Nvidia: since Nvidia only supports GPU's we are sure the code will be executed on a GPU whereas Intel is on CPU and AMD can be both CPU and GPU.

The capabilities of the OpenCL code are somewhat limited to the available hardware. On our test system a maximum of 10.000 points could be calculated. This value can be higher on systems with more powerful hardware. The desired feature, solving problem sizes of about 100 generators, can be achieved easily. For this problem size an OpenCL implementation is not the only possible solution. Even the much compacter sequential C or sequential Matlab code can solve this problem size in acceptable time. The OpenCL code however remains capable of fast execution with large datasets where the speed of the other applications drops off dramatically. For small problem sizes the OpenCL implementation is also the fastest but with a smaller margin. Since the OpenCL kernel is automatically compiled at runtime it is more convenient to change the distance formula compared to the sequential C implementation since recompilation of the executable is not required. From these arguments we can conclude that the OpenCL version is the most appropriate solution for the given problem.

Appendices

Appendix A: Dataset Helsinki

X	Y	P1	P2
24.8073	60.2177	-18.7478	96.6675
24.8082	60.2067	-19.4634	101.8989
24.8200	60.1755	-19.1553	101.8080
24.8228	60.1692	-19.9216	107.7748
24.8235	60.1688	-18.8743	96.4893
24.8240	60.1692	-19.8026	106.6524
24.8507	60.2538	-19.8435	108.7242
24.8692	60.2188	-18.8685	96.2597
24.8742	60.1427	-18.8627	96.0203
24.8835	60.1577	-18.8743	96.4893
24.9087	60.1583	-19.0671	101.7647
24.9170	60.1583	-20.1127	112.2409
24.9180	60.2040	-19.5253	103.8193
24.9207	60.2028	-19.1566	101.9207
24.9233	60.1530	-20.0718	112.6212
24.9393	60.1913	-19.9720	112.4409
24.9415	60.1592	-19.8522	112.5329
24.9550	60.2353	-19.1085	103.0721
24.9582	60.2880	-19.8934	110.5542
24.9583	60.1862	-18.9518	100.4958
24.9693	60.1835	-18.3331	93.4968
25.0502	60.2057	-18.7321	97.2313
25.0897	60.2728	-19.7954	110.3761

Appendix B: OpenCL and sequential C background image

```c
int readBackground(char *background)
{
    /* vars */
    FILE *f;
    char *extension;
    /* tF is a global var, it is not possible to pass a local variable to a
     * draw function
     */

    extension = strrchr(background,'.');
    if(extension != NULL )
    {
        /* check extension of the input, currently only bitmaps can be
         * drawn
         */
        if(strcmp(extension,".bmp") == 0)
        {
            /* read background file */
            fopen_s(&f, background, "r");
            if(f != NULL)
            {
                tF = (GLubyte*)malloc(WIDTH*HEIGHT*3);
                fread(tF, WIDTH*HEIGHT*3, 1, f);
                fclose(f);
                return 1;
            }
        }
        else
        {
            printf("Image is not a bmp and will not be displayed.\n");
            return 0;
        }

    }
    else
    {
        printf("Image file not loaded.\n");
    }
    return 0;
}

void printBackground()
{
    if(tF != NULL)
    {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        /* reset raster position */
        glRasterPos2f(0,0);
        /* draw bitmap */
        glDrawPixels (WIDTH, HEIGHT, GL_BGR_EXT, GL_UNSIGNED_BYTE, tF);
        glClear(GL_DEPTH_BUFFER_BIT);
    }
}
```

Appendix C: Calculation of signal strength

The characteristics of the transmitters are estimated based on the radio wave propagation recommendation bundle by the International Telecommunications Union. The version of the recommendation that was used is ITU-R P.1546-5 “Methods for point-to-area predictions for terrestrial services in the frequency range 30 MHz to 3 000 MHz” [36].

This bundle offers graphs where the signal strength is given in relation to the Euclidean distance. Different graphs are included for different frequencies, areas and antenna heights. Figure 35 shows a similar graph.

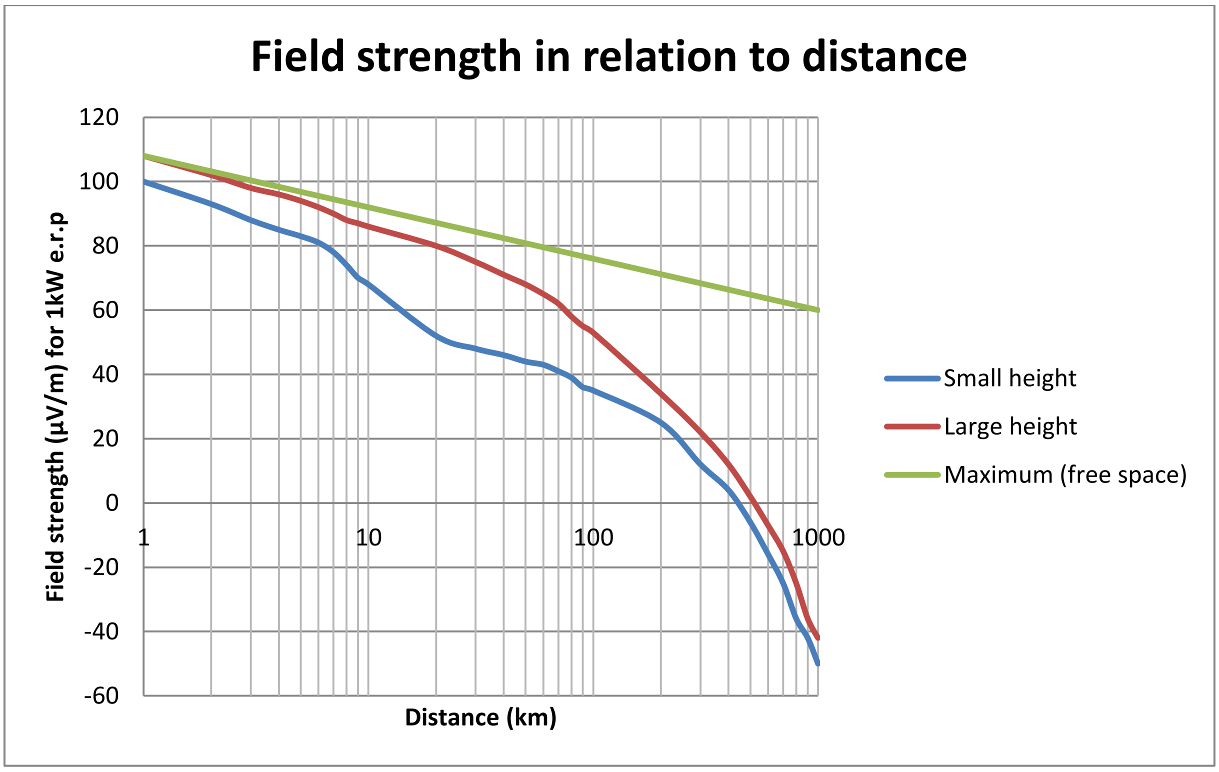


Figure 35 Field strength

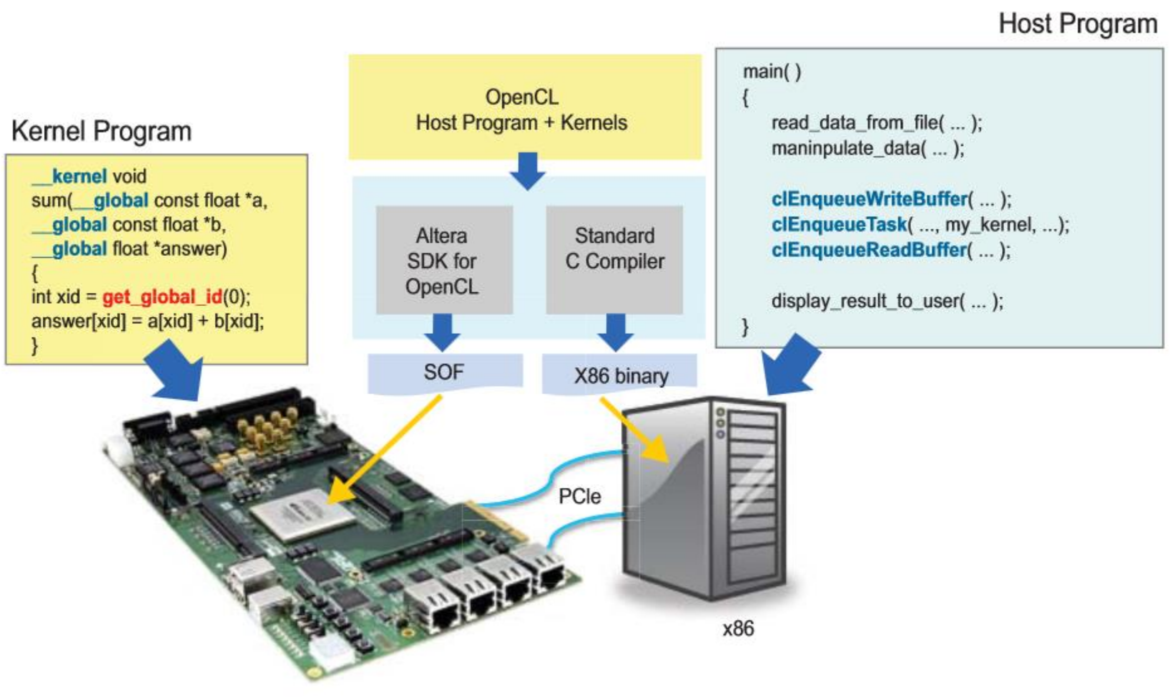
We approximate the curves in these graphs by fitting an equation with two variables P1 and P2 with the model P1*log(distance)+P2 so P1 < 0 and P2 > 0.

Appendix D: OpenCL on FPGA

OpenCL programs can also be run on FPGA's by specialised SDKs such as the Altera OpenCL SDK. The advantage of FPGA's over GPU's in massive multithreaded applications is the customisability of the device. The hardware can be fine-tuned for maximum performance for the envisioned application giving higher performance and potential lower power and lower costs. The OpenCL compiler enables the designers to model their problem with a high level language instead of a low level HDL languages. For optimal performance the FPGA can be connected via the PCI bus. Table 17 shows comparison of an OpenCL program on different devices. Figure 36 shows a possible setup.

Table 17 Monte Carlo Black-Scholes simulation results

Platform	**Power** Watts (W)	**Performance** Simulations per seconds (Bsims/s)	**Efficiency** Simulations per second per Watt (Msims/s/W)
W3690 Xeon Processor (CPU)	130	0.032	0.0025
Nvidia Kepler 20 (GPU)	212	10.1	48
BittWare S5-PCIe-HQ (FPGA)	45	12.0	266



```
__kernel void
sum(__global const float *a,
__global const float *b,
__global float *answer)
{
int xid = get_global_id(0);
answer[xid] = a[xid] + b[xid];
}
```

```
main( )
{
    read_data_from_file( ... );
    maninpulate_data( ... );

    clEnqueueWriteBuffer( ... );
    clEnqueueTask( ..., my_kernel, ...);
    clEnqueueReadBuffer( ... );

    display_result_to_user( ... );
}
```

Figure 36 FPGA as OpenCL host connected to x86 machine [37]

Bibliography

[1] A. Okabe et al., “Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. (2nd ed.)”, Wiley, 2000

[2] L. Ickjai and M. Gahegan, “Interactive analysis using Voronoi Diagrams: Algorithms to support Dynamic Update from a Generic Triangle-Based Data Structure.”, unpublished, 2000

[3] G. Stanley “Parallel programming,” The Computer Journal, Vol. 1, No. 1, April 1958, pp 2-10

[4] A. Valles, “Performance Insights to Intel ® Hyper-Threading Technology”, November 20 2009, [online]. Available: http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology. [Opened November 2013]

[5] S. Wasson, “Nvidia’s GeForce 8800 graphics processor”, November 8 2006, [online]. Available: http://techreport.com/review/11211/Nvidia-geforce-8800-graphics-processor. [Opened October 2013]

[6] J. Owens and UC Davis, “GPU Architecture Overview,” SIGGRAPH, San Diego, 2007

[7] Novatte, “How to calculate peak theoretical performance of a CPU-based HPC system”, date unknown, [online]. Available: http://www.novatte.com/our-blog/197-how-to-calculate-peak-theoretical-performance-of-a-cpu-based-hpc-system. [Opened November 2013]

[8] P. Yiannacouras, J. G. Steffan and J. Rose, “Data Parallel FPGA Workloads: Software versus Hardware”, 2009, [online]. Available: http://www.eecg.toronto.edu/~jayar/pubs/yiannacouras/yiannacourasfpl09.pdf. [Opened October 2013]

[9] Intel, “Intel ® Xeon ® Processor E7-2850”, 2011, [online]. Available: http://ark.intel.com/products/53573. [Opened November 2013]

[10] Intel, “Intel ® Core ™ i7-3960X processor extreme edition”, 2011, [online]. Available: http://ark.intel.com/products/63696. [Opened November 2013]

[11] Intel, “Intel ® Core ™ i7-3740QM processor”, 2012, [online]. Available: http://ark.intel.com/products/70847. [Opened November 2013]

[12] Nvidia, “Tesla K20X GPU accelerator”, November 2012, [online]. Available: http://www.Nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf.

[Opened November 2013]

[13] AMD, “AMD Radeon ™ HD 7970 Graphics”, 2011, [online]. Available: http://www.amd.com/us/products/desktop/graphics/7000/7970/Pages/radeon-7970.aspx#3. [Opened November 2013]

[14] Nvidia, “Tech Specs”, date unknown, [online]. Available: http://www.Nvidia.com/object/nvs_techspecs.html. [Opened November 2013]

[15] Nvidia, “GeFore GTX TITAN specs”, 2012, [online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications. [Opened November 2013]

[16] Matlab, “Parallel Computing Toolbox”, date unknown, [online]. Available: http://www.mathworks.se/products/datasheets/pdf/parallel-computing-toolbox.pdf. [Opened October 2013]

[17] R. Gerber, “Getting Started with OpenMP”, July 6 2012, [online]. Available: http://software.intel.com/en-us/articles/getting-started-with-openmp/. [Opened, October 2013]

[18] P. Kennedy “Intel® C++ and Fortran Compilers now support the OpenMP* 3.1 Specification”, September 2 2011, [online]. Available: http://software.intel.com/en-us/articles/intel-c-and-fortran-compilers-now-support-the-openmp-31-specification/. [Opened November 2013]

[19] B. Chapman, “Using OpenMP: portable shared memory parallel programming”, The MIT Press, 2008

[20] OpenACC group, “The OpenACC Application Programming Interface V2.0”, August 2013, [online]. Available: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf . [Opened October 2013]

[21] OpenGL group, “Compute Shader”, November 27 2013, [online]. Available: https://www.opengl.org/wiki/Compute_Shader. [Opened November 2013]

[22] R. Amorim et al., “Comparing CUDA and OpenGL implementations for a Jacobi iteration,” International conference on high Performance Computing &

[23] D. Göddeke, “GPGPU::Basic Math / FBO Tutorial”, 2005-2006, [online]. Available: http://www.seas.upenn.edu/~cis565/fbo.html . [Opened October 2013]

[24] OpenGLUT team, “OpenGLUT Project”, February 5 2005, [online]. Available: http://openglut.sourceforge.net/. [Opened November 2013]

[25] V. Hindriksen, “OpenCL vs. CUDA misconceptions”, 22-06-2011, [online]. Available: http://streamcomputing.eu/blog/2011-06-22/opencl-vs-cuda-misconceptions/ . [Opened November 2013]

[26] A. Munshi et al. “OpenCL Programming Guide”, Addison-Wesley Professional, 2011

[27] M. Shevtsov , “OpenCL advantages of heterogeneous approach”, 2013, [online]. Available: http://software.intel.com/sites/default/files/article/381646/opencl-advantages-of-heterogeneous-approach.pdf . [Opened October 2013]

[28] Khronos Group, “OpenCL conformant Products”, November 12 2013, [online]. Available: http://www.khronos.org/conformance/adopters/conformant-products#opencl. [Opened November 2013]

[29] Intel, “Intel® SDK for OpenCL* Applications XE 2013 R2 - User’s Guide”, 2013, [online]. Available: http://software.intel.com/sites/products/documentation/ioclsdk/2013XE/UG/index.htm . [Opened November 2013]

[30] K. Karimi, N. G. Dickson, F. Hamze, “A Performance Comparison of CUDA and OpenCL”, May 16 2011, [online]. Available: http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf. [Opened November 2013]

[31] J. Fang, A.L. Varbanescu, H. Sips, “A Comprehensive Performance Comparison of CUDA and OpenCL”, International Conference on Parallel Processing (ICPP), Taipei City, 2011

[32] J. Meredith, “From CUDA to OpenCL”, February 21 2012, [online]. Available: http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2012-02-20/08-opencl.pdf. [Opened November 2013]

[33] R. Farber, “CUDA Application Design and Development”, Morgan Kaufmann, 2011

[34] A. Munshi, “The OpenCL Specification”, June 10 2009, [online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.x-latest.pdf#page=29. [Opened December 2013]

[35] Z. Bar-Yehuda, “plot_google_map”, May 16 2010, [online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/27627-plotgooglemap. [Opened December 2013]

[36] ITU, “Methods for point-to-area predictions for terrestrial services in the frequency range 30 MHz to 3 000 MHz”, September 2013, [online]. Available: http://www.itu.int/rec/R-REC-P.1546/en. [Opened December 2013]

[37] Altera corporation, “Implementing FPGA Design with OpenCL Standard”, November 2013, [online]. Available: http://www.altera.com/literature/wp/wp-01173-opencl.pdf. [Opened November 2013]

Auteursrechtelijke overeenkomst