TECHNISCHE UNIVERSITÄT DRESDEN

DEPARTMENT OF COMPUTER SCIENCE
INSTITUTE OF SYSTEMS ARCHITECTURE
DATABASE TECHNOLOGY GROUP
PROF. DR.-ING WOLFGANG LEHNER

# Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

# Parallization of Parameter Estimators on GPUs

Mathias Rauh
(Born 25th June 1989 in Riesa)

Professor: Prof. Dr.-Ing Wolfgang Lehner
Tutor: Ulrike Fischer, Benjamin Schlegel

Hamburg, January 11, 2014

**Aufgabenstellung**

Ausgangssituation

Die Prognose von Zeitreihen spielt eine wichtige Rolle in vielen Bereichen, z.B. bei der Vorhersage erneuerbarer Energie zur Steuerung des Energiemarktes. Oft wird dabei die zukünftige Entwicklung der Zeitreihe in einem statistischen Prognosemodell abgebildet. Zur Berechnung von Prognosemodellen werden Parameterschätzer eingesetzt, die in mehreren Läufen über die Zeitreihe verschiedene Parameterkombinationen untersuchen und bewerten. Je nach Länge der Zeitreihe und Anzahl der zu schätzenden Parameter kann dies ein zeitaufwendiger Prozess sein.

Grafikprozessoren (GPUs) bieten Möglichkeiten zur Beschleunigung dieses Schätzvorganges, da diese oft ein Vielfaches der Rechenleistung von herkömmlichen Prozessoren besitzen. Problematisch ist jedoch, dass diese hohe Rechenleistung nur erreicht werden kann, wenn der zugrundeliegende Prozess ausreichend parallel ausgeführt wird. Ziel dieser Diplomarbeit ist es deshalb, parallele Parameterschätzer für GPUs zu entwickeln, implementieren und evaluieren.

Aufgabenstellung

Ziel der Arbeit ist es, Parallelisierungsmöglichkeiten von Parameternschätzern auf GPUs zu untersuchen und prototypisch zu implementieren.
Die Teilaufgaben der Arbeit sind dabei:
- Einarbeitung in die Zeitreihenprognosen und die Schätzung von Prognosemodellen.
- Einarbeitung in parallele Architekturen (speziell GPUs) und das OpenCL Framework.
- Klassifikation existierender Parameterschätzer/Prognosemodelle und Abschätzung von deren Parallelisierbarkeit.
- Entwicklung von parallelen Parameterschätzern, die für GPUs optimiert sind.
- Prototypische Implementierung der entwickelten Verfahren.
- Experimentelle Evaluierung der Implementierung mit synthetischen und realen Daten.

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Diplomarbeit zum Thema:

*Parallization of Parameter Estimators on GPUs*

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.
Hamburg, January 11, 2014

Mathias Rauh

**Abstract**

Nowadays whenever a complex, parameterised function needs to be optimized, a parameter estimator will be employed that varies the function's parameters according to some algorithm and evaluates the changes in the result and thereby eventually finds the optimal parameter set. One such estimation process would be the execution of the pattern search as first described by Hooke and Jeeves in 1961. This estimator or optimizer shall be the basis for a new parallel implementation suited for GPU computing using the OpenCL language that will be presented in this work. This solution will be described, discussed and evaluated using triple exponential smoothing as the function to optimize. The goal is to minimise the forecast error by setting the parameters accordingly. Several variants of the proposed algorithm are compared in experiment and for additional comparision they will run against a simple grid search.

# Contents

**1 Motivation** **3**

**2 General Parallelism Strategies** **5**
2.1 What to Parallelize? . . . . . 5
2.2 Challenges of Parellelizing . . . . . 5
2.2.1 Data Dependencies . . . . . 5
2.2.2 Multiple Instances . . . . . 6
2.2.3 Task Parallelism . . . . . 7

**3 Parallel Optimizers and Their Classification** **8**
3.1 Parallel Control Flow . . . . . 8
3.2 Parallel Data . . . . . 9
3.3 Hybrids . . . . . 10
3.4 Different Approaches . . . . . 10

**4 A Local Search Pattern Method - The Algorithm By Hooke And Jeeves** **11**
4.1 Desrciption Of The Classic Algorithm . . . . . 11
4.2 On Parallelizing Hooke Jeeves - Different Approaches . . . . . 12
4.3 A Parallel Hooke Jeeves . . . . . 13
4.3.1 On How to Calculate Neighbouring Points . . . . . 13
4.4 Comparision of the Classic and Parallel Hooke-Jeeves . . . . . 15

**5 A Parallel Implementations of Jooke-Heeves using OpenCL** **16**
5.1 OpenCL . . . . . 16
5.1.1 Logical Representation of Hardware . . . . . 16
5.1.2 On the Status of OpenCL - Ready for Productive Development? . . . . . 17
5.2 Special Adaption To Graphic Card Computing . . . . . 19
5.3 Limitations of the Implementation . . . . . 20

**6 Experiments** **21**
6.1 Setup . . . . . 21

6.2 Data Sources . . . . . 21
6.3 Difference in Execution Time between CPUs and GPUs . . . . . 23
6.4 Influence of the Input Size . . . . . 23
6.5 Comparision of Different Modes of the Parallel Hooke-Jeeves . . . . . 25
6.6 Influence of OpenCL Parameters . . . . . 26

**7 Conclusion** **29**

**Bibliography** **30**

# 1 Motivation

The focus of this thesis will lie on parallel techniques for solving optimization problems. As should be known today's challenge when trying to improve our computer programmes is to adapt them to parallel hardware architectures. Recently parallel computing on graphic cards became a trend. The typically offer the benefit to gain a large speed-up by spending relatively few money on commodity hardware. However most programmers are not familiar with GPU programming techniques. The author did experience that by himself during the course of this work.

Eventually the effort will pay off, and did pay off for the purpose of this work as the execution time of a programme could be reduced by a factor of several hundred until several thousand using a relelatively old consumer graphics card. Originally experiments on some kind of a high performance computer were planned but it turned out to be absolutely not necessary, since it would have been very difficult to keep such a machine engaged for more than a couple of seconds.

Before we want to delve into details, I shall briefly explain several terms that will occur quite often in this work. When I talk of a parameter optimizer or sometimes a parameter estimator I am refering to a programme that takes as input a search space of arbitrary dimension and a parameterised target function that shall be evaluated. All the parameters of that funcation vary within the bounds of the given search space. The parameter optimizer's task is to find such a set of parameters such that the ouput of the target function will be optimised.

As target function the exponential smoothing method has been chosen. It is used to forecast time series based upon historical data. There are different variants of this method with varying numbers of parameters. Initially, for early code development and testing, the single exponential smoothing that uses only one paramter served as the target function. With some training data used by the smoothing algorithm the parameter optimizer could tweak this parameter until a minimum forecast error could be achieved. The same is valid for the triple exponential smoothing that was later used for further experiments, with the small difference that we now have three parameters to optimise.

Of course there was need for same measurement function to determine the forecast error. For this purpose the choice fell upon the sum of squared errors of prediction or SSE for short. Since we con-

centrate on performance and speed-up tests the small difference between this and other forecast error calculation methods does not play a relevant role.

There is one central question in this chapter that has remained: Why put any effort in parallelising forecasting? This is quickly answered when thinking of time sensitive fields of applications. For example imagine a power plant that needs to adapt its production quickly to the near-future demand. The only way to assess this demand is to run some statistical calculations that can return a forecast. Since this is time critical, this calculations better be fast and that is the more difficult to achieve the more data is available. But more data most of the time leads to better forecasts so one absolutely wants to use all this data. Hence the need for parallelization to process all that in time.

# 2 General Parallelism Strategies

## 2.1 What to Parallelize?

As Marx showed in [Mar13] the largest fraction of execution time is generated by parameter estimation, while the actual function that is evaluated plays not really a significant role. This becomes evident if we think about how such a forecast system would work: the parameter optimizer calls the target function typically very often, each time with a different paramter set. It is now much more simpler to distribute all these calls on several computing cores (in GPU computing typically some thousands) than to parallelize the target function's code to achieve the same speed-up. Marx used the OpenCL language, that is also used in this work for GPU computing, and has found out, that the overhead is negligible.

Therefore the focus of parallelizing clearly lies on the parameter estimator. There are now different strategies on how to parallelize it. Partitioning the data and distribute these partitions on different computing units is probably the easiest one. Another way would be to start multiple instances of the algorithm, that are competing for the best results. This is especially useful for non-deterministic estimators that rely to a certain degree on random decisions. Also combination and variants of these strategies are imaginable. Later in this chapter, several such approaches for parallelizing that were published shall be explored. But before we shall discuss several challenges that occur when trying to parallelize algorithms.

Besides many target functions, for example the exponential smoothing, that we want to use here, are not really suited for parallel execution anyway. Hence the focus on parallelizing the parameter estimator.

## 2.2 Challenges of Parellelizing

### 2.2.1 Data Dependencies

In practise there are almost always some parts of an algorithm that have to be executed sequentially since there are data dependencies. A simple example of such a case would be a function that has to choose the best value out of a set of intermediate results. The best value cannot be found with

absolute surety before all results are available. However this function can start to compare the first results as soon as at least two of them become available. To further complicate the algorithmic design one could think of parallel comparators working at the same time on different subsets of data, each of them finds an optimal results and sends it to the next level of comparators until a final comparator selects the end result. Such a strategy might be known from the implementation of database operators belonging to the class of aggregators, like COUNT(), SUM(), etc.

This means the challenge of data dependency issues can be solved by finding smaller subsets of data that are independent from the rest subsets of data. Each subset can be distributed to a different computing unit and all these units then can work independent and in parallel. Within each unit, the data can be processed sequentially. The question is can you form enough subsets in order to fully utilize all your computing units? This is especially relevant when bearing in mind that graphic cards typically have a lot of processors that can executed a lot of operations in parallel, resulting in typically several hundred up to several thousand operations per cycle.

Another possiblility to overcome data dependencies is to employ some kind of synchronising mechanism between different computing units in order to solve issues. Whether this is efficient or not primarily depends on the hardware. As one can easily guess synchronizing over a large local area network or even the Internet would introduce significantly high delays that would result in poor performance. On the over hand modern CPU cores belonging to the same CPU can efficiently communicate over fast channels. Synchronizing often requires more coding effort. Since OpenCL cannot be called a convenient to use language the synchronizing approach was not pursued in this work (and it turned out not to be necessary anyway.)

### 2.2.2 Multiple Instances

If it is not possible to form enough sensible large partitions of your data, then one might think to launch additionally multiple instances of the algorithm that are may using different paramters. (Now these parameters are not the ones that we try to optimize, but parameters that determine how our estimator behaves). Or if the estimator employs random functionality multiple intances might also be a benefit in order to test for more variants than with just one. This concept of combining partitioning of data with starting of multiple instances will later be mentioned and evaluated further when presenting the author's implementation of a parallel Hooke-Jeeves-Optimizer that uses exactly that strategy.

### 2.2.3 Task Parallelism

Task parallelism, which parallelizes operations' executions, typically involves a careful algorithmic design that tries to exploit the underlying architecture as well as possible. In turn one has to consider the fact that such designs tend to be highly dependend on the architecture to run efficiently as briefly stated in [Kon04]. The use of task parallelism is very limited nowadays. It might be possible to achieve a speed-up of two to eight or ten, as has been shown. Adapting this technique for GPU computing would simply mean at least hundreads of cores are not utilised. Hence this approach shall not be discussed further as it is presenting absolutely no benefit for this work.

# 3 Parallel Optimizers and Their Classification

This chapter presents a survey of existing parallel optimizers that were proposed in other works and that shall be classificated and grouped. Some of these systems resemble others while differing only in some aspects. However that could still cause them to be classificated in different categories. One of these parameter estimators that appears in nearly every category and sub-categories is simulated annealing.

## 3.1 Parallel Control Flow

Some proposed systems try to parallelize the data others try to parallelize the control flow, which one might call thread-level parallelism. And some systems are hybrids doing both. A first classification criteria is therefore the level (data or thread) of parallelism.

Other possible criteria in order to group all the different approaches are whether the systems are work application specific or model specific. One example for an application specific system is presented in [KR86]. This article deals with parallel simulated annealing for cell placement on circuit boards. The primary goal is to minimize the total length of electrical conducters. Now the cells that need to be placed can be rotated and the wiring changes accordingly. The authors created a variant of the simulated annealing method that was specifically adapted to the use case. This approach can further be classificated as a global optimization technique what shall be clear when the process of simulated annealing is explained in the next paragraph.

Briefly explained this method lets you traverse a given search space, whereby an initial solution is selected and then another solution is derived. This is called a move. How moves are calculated is determined by a cooling schedule. This essentially describes how far the euclidean distance between the solutions can be. Over time this maximum distance will shrink, the process ”cools down” or the temperature drops. The idea behind this is that the temperature is high enough not to become trapped in local minima, but also not to miss a global minima.

Wether a solution is suitable is evaluated by a fitness function. This function can be quite application

specific for the system in the cited article. The approach to parallelize simulated annealing by the authors here was to execute multiple moves in parallel meaning they implemented some kind of thread level parellism. As a result intermediate results later needed to be synchronized in order to get the final result.

Examples of model specific approach are the works [AdBHvL86], using simulated annealing with a parallel cooling schedule and optimizing globally, [JT88], presenting a local parallel nelder-mead optimizer and [HKT01]. The latter describes a local parallel optimizer that is based on Hooke-Jeeves pattern search method. This method was also chosen for the optimizer developed by the author and is later described in detail. For the moment it is enough to say, that the Hooke-Jeeves method follows a search path as long as intermediate results are improving and backs up once they deteriorate. In the end the method will halt when reaching a local minimum.

## 3.2 Parallel Data

In [CFW98] a system is proposed that employs different starting points that serve as a basis for further calculations. In essence one could say the simply try different alternatives at the same time. Their optimizer, that also implements the already described simulated annealing method, can be characterized as an incrementally improving system, that starts with several initial points in the search space. These are then used to further traverse the search space. That means the space has not been devided into different subspaces (what would be considered partitioning) but rather we are regarding the full space and just parallise by beginning at several starting points and see how subsequent solutions evolve. That is all calculations are done on the same basic input data, but use different status data or intermediate results.

Another approach is the Hill-Climbing method that is typically classificated as a local optimizer. But when one starts to execute it starting from multiple initial solutions in the search space this classification is not clear anymore. One could argue that it has become a global optimizer. Two examples can be found in [OMGS04] and [GTG05].

We cannot only use different starting points, but also partition our data. Each partition is then distributed to another computing unit and hence a comparably easy to implement way of parallelizing results form that. In [CRSV87] and [VNR$^+$06] one can read on such approaches. The first one is again one system that is based on simulated annealing , while the second one is an implementation, called SCEM-UA. They partition the data, derive a solutions based upon previous solutions, then evaluate the result and if it is not satisfying they finally shuffle within a partition and start all over.

Over time the best solution in each partition will be found.

## 3.3 Hybrids

Four works that describe hybrid approach, both involving control and data parallelism are [BSS88] (Quasi-Newton), [CBZ10] (Simulated Annealing), [FLZ06] (Nelder-Mead) and [CFW98] that combines simulated annealing with genetic algorithms. While this approach is certainly very interesting delving into this topic would be out of scope of this work since the system presented, or all of these hybrid systems, are naturally a little bit too complex to briefly explain here.

## 3.4 Different Approaches

A totally different field of optimizing parameters are machine learning techniques. One very prominent representantative of this branch are artificial neural networks. They were presented in all imaginable different variants in the literature. Since they are not relevant for this work this comment shall close with the statement that they exist and might be interessting to work with on GPUs too.

# 4 A Local Search Pattern Method - The Algorithm By Hooke And Jeeves

## 4.1 Desrciption Of The Classic Algorithm

As already pointed out the pattern search method developed by Hooke and Jeeves is a local optimization technique. In the following paragraphs the basic operation mode of this algorithm shall be characterized.

One starts from an initial guessed solution that consists of a set of values for all the parameters of the target function that has been selected for optimization. This set of values equals a vector of the seach space. Using this vector an initial function evaluation is executed in order to get a first result that shall serve as a reference for upcoming function evaluations. The Hooke-Jeeves-Optimizer then enters a loop in order to iteratively traverse the search space selecting new test vectors based on a certain pattern. These intermediate vectors (that I will also call neighbouring points later) are set as parameter input of the target function which is then evaluated. If a vector generates a better, a more optimal, return value the direction of traversing the search space will be further pursued, otherwise an alternate turn will be made.

At some point of time, that can be defined using constraints, we want the algorithm to stop. For example once the return value of the target function hits a certain threshold, that is considered to be a satisfiable result, the Hooke-Jeeves-Optimizer shall stop. Further constraints could involve the maximum number of iterations, or minimal distance between the test vectors of two subsequent rounds (which is useful considering that at some point the distance would be below the precision of relevant data types is reached).

The initially guessed solution marks a certain point in the search space. Further points can now be determined by adding or substracting a delta value, which desribes the distance between the two points. Typically this distance changes over the course of execution time whereby many alternative ways on how to change the distances are conceivable. The applying of deltas to the current solution produce new solutions which in turn are then evaluated. That is done as long as the evalution values are mov-

ing to the desired optimization target. That way the method approaches a local optimum and should the evaluation values start to increase again it would then mean the search overshot this optimum. A correction then needs to be made, which can be regarded as a backstep. The distance between the current and next solution will then be descreased what hopefully avoids overshooting a second time.

## 4.2 On Parallelizing Hooke Jeeves - Different Approaches

When parallelizing the Jooke Heeves method two different, promising approaches were pursued. As described earlier this algorithm iteratively calculates new neighbouring solutions and checks whether a benefit is thereby gained. If so that search direction will be further pursued. The first approach would be to calculate multiple neighbouring points in the search space each time and choose the best out of them for further investigation. Efficiency of this method depends on the distance of these points to eachother and how they are selected. Likewise influential is the sheer number of neighbouring points that are computed each round. This last parameter is also the one that lets one choose how many GPU-cores will be effectively used since each neighbour solution checking will involve a complete function evaluation that will run on the cores in parallel.

The second approach of parallelizing is to use several starting points instead of just one. Remember that Jooke-Heeves is a local optimization technique that starts with a initial guessed solution and tries to find better neighbouring solutions. When using several starting guesses these searches for better solutions can be executed in parallel with each instance investigating essentially a different subspace of the global search space. The more initial guesses one chooses to use the more this local optimizer actually resembles a global optimizer because the probability of missing the best solution will decrease significantly. Each instance will still converge towards a local minimum however one of the many local minimums found will probably be the global optimum.

These approaches can be combined to further increase the workload of GPU devices. Both are rather independent of each other and henceforth a combination can be considered all in all straightforward. This is exactly what has been done in the implementation phase.

## 4.3 A Parallel Hooke Jeeves

### 4.3.1 On How to Calculate Neighbouring Points

Especially when we calculate multiple neighbouring points in the search space, one very important question pops up. How do we select the points? Shall we just randomly select some points with varying (but limited) distance to a given base point? Or should we follow a strict pattern prescribing to determine points in a more regular fashion, for example points at a certain fixed distance to the given point in each dimension and each direction? The classic Hooke-Jeeves method uses the latter approach. For each dimension a unit vector is applied to the base point, whereas the vector is scaled by a value that could change every round. This is basically a step width. So there are n function evaluation in an n-dimensional search space, since we get n points by applying each of these unit vectors seperately.

For the purpose of this work a three-dimensional problem has been chosen, so when parallizing this approach in a naive way we could only achieve a speed-up of three with three function evaluations in parallel. We could increase that if we would choose more than one starting guess to start searches from. However we would have to select at least a hundred starting points to occupy even the simplest and oldest GPU platforms. A modern graphics card will be hopelessly under-utilised. Moreover if we would choose more and more starting guesses the search would increasingly resemble a grid search. A smarter algorithm for parallelly searching is necessary and one possible candidate shall be discussed now.

One drawback of the classic Hooke-Jeeves method is evident, because with engaging unit vectors we traverse the search space in a rectangular manner, while in reality going diagonally is often a considerable option for a shortcut. Or more generally speaking, adding any arbitrary non unit-vector to a base point can yield better results then going all the way around. So one can imagine that if we add a large amount of vectors pointing in different directions but also varying in length would yield a far better knowledge about the immediate surroundings of a base point. Then the most promising solutions are pursued for further evaluation. And once again many different vectors are added to this subset of points.

With parallel computing on GPUs we typically could evaluate hundreds, if not thousands of such points at the same time. That enables us to calculate a large variety of possible solutions at each step with the inherent risk that a lot of these could be useless for further steps of the algorithm. However due to employing massive parallel system we are in the luxury situation of not having to overly care for, if one would like to phrase it that way. However we do not want to escalate this behaviour and

aim for a quick return of result. So the next paragraphs shall outline and discuss how a more efficient version of this parallel algorithm will be like.

Finding a large set of arbitrary neighbouring points around a given base point, then selecting some of them and doing the same all over is obviously not the best search method. A much smarter way would be to take indeed a look around first but then focus on some more promising direction. Imagine you step into a large dark room, equipped with a torch, and determined to find a specific object. The width of the emitted light beam can be adjusted. Once you enter the room you probably would widen the light beam, yielding a wide light cone that covers a large area. Unfortunetly the torch does not shine brightly, since light emission's energy is constant and this energy is distributed over a large area. However that wide light cone helps you to quickly orientate in the room and maybe helps you to already find some objects that appear similiar to the one you are searching for. Once such objects arr located, you would probably step closer and simultaneously adjust the light cone of the torch so it becomes more focused and brighter to let you see clearer. That way you can inspect the object in question and quickly identify wether it is indeed the searched one.

Now transfer such a strategy to our mathematical problem. The dark room equals the search space. The searched object is a local optimum, the steps we walk is a search path, being a composite of a series of vectors. Our search technique is first to look around and then focus. When we apply a large amount of varying vectors to a base solution pointing in all directions and evaluate the results we should gain a good overview of our surroundings. That equals looking around with the wide cone of ligth emitted from the torch. Then we select say the best ten per cent of the found neighbouring points.

Let's say for further explanation we pick just one point B out of this set, while our base point is called A. The direction (A,B) is a vector that describes our walking direction. In the next step we want to focus on B and it's surrounding area. But now we do not take a full look around but persue mainly the direction of our vector (A,B). Imagine that we add this vector to B and thereby define the center of a search cone with a certain width. Within this cone we choose several other vectors. Since we do not take a full look around but rather concentrate in a specific direction, the chosen vectors will all be pointing in a similar direction. It is imaginable that we even consider the same vector several times, but each time it will have a different length. In the next round using these vectors, we may find another point C, that yields even better results. Therefore we define the new, even more focused cone by stepping to C and adding vector (B,C) being the longitudinal axis of the cone. As long as we find better solutions we increasingly reduce the width of the cone.

Once no better solutions can be found, we take again a full look around. Then we either a solution

lying in a different direction then previously focused on will appear and we can change the search direction accorindgly and continue or still no better solution is found meaning that we hit a local optimum or are very close. To get a very precise final solution we test a lot of neightbouring points very close to the current point and the algorithm returns the result.

Apart from the previously explained method (and in the following called"smart"-mode) other techniques to determine neighbouring solutions are regarded for comparision. However these are rather simple and briefly explained. First around each point several random points are selected. The distance is limited, but otherwise they can come from a postion anywhere in the sub-search space around the given base point.

And another technique is simply using neighbours with an equal distance to each other. Around a base point there will be a grid of neighbouring solutions, so one can say this is essentially a very local grid search within the more globally operating Hooke-Jeeves.

These three search techniques, called the smart, random and equi-mode are later compared in experiment.

## 4.4 Comparision of the Classic and Parallel Hooke-Jeeves

First of all while the classic Hooke-Jeeves-Method performs clearly a local optimization, this cannot be clearly stated for the proposed parallel variant. Especially if we imagine that the density of starting points is relatively high compared to the search space. With a high probability the complete search space can be then traversed completetly using a more or less fine-grained grid. So in essence such a setup would resemble the global grid-search optimizer a lot and therefore could also be considered a global optimizer, rather than a locally operating one.

This becomes especially true if we use several startings solutions as a basis for our search. The more computing units are available the more we can use and the more similiar to a grid search the system will become.

# 5 A Parallel Implementations of Jooke-Heeves using OpenCL

## 5.1 OpenCL

What could be better than a platform-independent, and therefore flexible, language which on top of that resembles some programming language you already know? OpenCL tries to fulfil exactly that. Being an enhanced version of C, with special language extensions added that are needed to communicate for example with graphic cards but also with many other devices and bringing in its own keywords to address all the different types of memory, OpenCL lets you easily programme so-called kernels that can be executed on any supported device.

### 5.1.1 Logical Representation of Hardware

Here I shall explain briefly some technical terms of OpenCL that occur in later sections and chapters of this work. The smallest hardware unit is called computing unit. It represents a single core. Many such cores form a device, for example a array of cores on a graphics card. The graphics cards is called platform.

Each computing unit can execute a kernel. A kernel is written in a C-like dialect and should typically be not that large (Meaning the source code of the kernel's programm shall not be to the large). Before a kernel can executed it needs to be compiled, what can both be done during host compilation time or run time. Then it can be loaded onto the device in order to execute it. Once all the initial work is done, OpenCL lets you do that relatively easily.

Several classes of memory exist. Memory that is globally accessable, that is by all computing units on the same platform. A special type of such memory is constant memory that is shared read-only by all units. Of course each kernel also has private memory, only visible within a single unit. And lastly there is local memory that is available to a subset of units that work together in a workgroup.

The size of a workgroup is called worksize. This parameter essentially tells how many units shall be used for computation and therefore determines the maximum possible speed-up in combination with

the number of workgroups. For this work only one workgroup was used, that typically included all computational units that were available on the device.

### 5.1.2 On the Status of OpenCL - Ready for Productive Development?

In this section we shall cover some issues that have arisen during the software development phase of this work. We shall find out whether OpenCL is suited for productive development, that is whether one can programm efficiently using OpenCL or not.

Before one could start programming an environment has to be set up. OpenCL requires special driver software for each device it should run on, including not only GPUs but also CPUs if these should be employed too. Fortunetely these drivers seem to be available to all platforms, such as Windows 7, openSUSE Linux and Xubuntu Linux, which were all tested for capability by the author.

For efficient development one needs to engage special debugging tools to better investigate OpenCL-specific functionality. Debugging tools of AMD were tested and deemed rather unuseful by the author. Some versions tested on a Xubuntu system crashed upon start. Other versions of the debugger, that luckily crashed not immediately, were not able to debug an OpenCL-programm which is even more ridiculous. Obviously the QA-department of AMD forgot to test the tools properly on Linux-platforms. Furthermore these tools were not integrated with any IDE and jugding from the to some degree un-intuitive and rather basic graphical interface no real effort was put into developing these tools. For all these reasons the software development tools engaged for this work were not chosen from AMD but from Nvidia. The Tools used belong to the NSight plugin for Microsoft's Visual Studio.

Nvidia's debugging tools neatly integrate in Microsoft's Visual Studio. Therefore Windows has been chosen as the final development platform and version 2008 of the Visual Studio as the IDE has been used. The OpenCL debugging tools which were used (the author did not test every functionality) then did run always in a stable manner and were easy to use. The graphical interface is in the author's opinion clearly superior to the GUI of AMD's tools.

After having covered drivers and software development tools we shall now examine whether the OpenCL library and documentation do help the developer to produce working code. There are different bindings available, so OpenCL can be used with C-like function calls, but also encapsulating classes in C++ style or even C# bindings are available. Initially a certain platform independence for this work's code should be maintained, since the programm while developed under Windows, should

later also be compilable under Linux systems. Therefore C++ was given preference over C# and the .NET framework. (this was later much regretted). The author is aware that the Mono-Framework tries to offer a .NET-compatibel framework for linux sytems. The author also experienced troubles using Mono in previous projects and hence avoided this solution. Therefore the Host program has been developed using C++. However for some strange unknown reasons, the windows-installer, that delivered the OpenCL Dll with C bindings did not install the C++ bindings. After having lost a lot of time already investigating AMD's tools a quick solution was obviously to programm the host-code in C++ but use the C-style functions to actually engage OpenCL library functionality.

The steps to load an OpenCL kernel and call a kernel-function are numerous. First you need to discover the platform. For example ID's of devices with OpenCL support have to be retrieved manually. Such ID retrievals look rather counter-intuitive. For example first you call

```
clGetPlatformIDs(cl_uint 0, cl_platform_id[] NULL,
cl_uint &numPlatforms)
```

to find out how many IDs are found. The type `cl_uint` is basically an unsigned integer, and `cl_platform_id[]` is an array of integers. The parameter `numPlatforms` returns the number of IDs. Then we have to allocate memory by creating an array of `cl_platform_id` of the size `numPlatforms`. In a third step you call the same function again with slighty altered parameters:

```
clGetPlatformIDs(cl_uint numPlatforms, cl_platform_id[] platids,
cl_uint NULL)
```

whereat numPlatforms is the same value we have retrieved in step one, and platids is the just created array that shall be populated with actual IDs by the function. It is not clear why the OpenCL developers do not offer a function that creates the array of correct size internally and returns its size, while the caller just passes a pointer the memory where the array starts. This would reduce this three-step IDs discovery to a one-step discovery. Especially if you take into consideration, that you have to repeat the same procedure for each platform to retrieve the device-IDs since under each platform several devices can be subsumed. In another step additional information will be retrieved for each device in most cases. That way one could filter by the device's type if only GPUs and not CPUs shall be engaged, for example.

Such inconvenient calls and rather long preperation until a kernel can finally be called (a lot of required steps were not even mentioned yet) raise the initial effort to programm even the simplest OpenCL-solutions. Of course one would create code that encapsulates all these OpenCL calls, saving a lot of time for future projects. But then, why did the OpenCL developers not provide a more careful

designed library to begin with? The author assumes OpenCL libraries will become more convenient to use eventually, since it naturally takes some time until feedback is given and the OpenCL developers react upon.

A much more serious problem is the very basic, far to undetailed, documentation that is currently available for version 1.2. The author experienced several times that despite employing all available documentation, questions were not answered at all. Trying out different possibilities eventually lead to a solution and thereby solved the intial question. Of course such an approach costs a lot of time and is unacceptable for productive code development.

All in all the author deems OpenCL in the tested version (1.2) not ready for productive development. While the limitation to Nvidia's debugging tools in combination with the Visual Studio, if one wants to employ a reasonably efficient tool chain, might not pose difficulties for many developers, the lack of detailed documentation probably will be too large a stumbling block to tread the path of efficient OpenCL development under the pressure of the market. It might be a different case once you have sufficient time.

## 5.2 Special Adaption To Graphic Card Computing

To clarify what actually should be run on the computing units I can offer a short answer: only the target function. In our case a triple exponential smoothing forecast method as described by Holt-Winters. This function does not really has to be changed a lot since it has been decidied to let each kernel execute the target function on his own since it would be no benefit to parallelize the target function as already pointed out earlier. Hence our goal is to run as many such function evalutions with different arguments parallelly. First we store vectors of these arguments onto the graphics device. Then initiate the kernel execution on each computing unit and tell each kernel to fetch their arguments (an own set for each kernel). So in essence each computing unit executes the forecast independently from the others using its own parameters.

It is a little more complex to prepare this calls on the host and evaluate the results. Obviously it is unavoidable to do some sequential work in these two steps, which can limit the overall speed-up a little but. However that does not play a significant role.

At least until now the largest problem for graphics accelerated computing is the problem of how efficiently you can shovelling your data onto the device, meaning simply the memory bus is the bottle neck. There is actually a bright outlook as Intel recently announced plans for a new approach in

using shared memory allowing to skip the transporting of data from CPU-associated memory to GPU-associated memory. Unfortunetely this hardware has not been available at the time of composing this work. I therefore suppose it will be quite likely to further improve the performance of the developed software in the future when adopting it to this new kind of hardware what further motivated me to comment the source code a little more than usual.

Taking a deeper look on the memory bus problem you would certainly agree it is wise to minimize the amount of data copyied between the different memory chips. For example it would be less effective to first run several forecasts with different parameters on the input data, then pass the result back to the host and there choose the best forecast by running some error calculation algorithm on the results. A better way would be to calculate just the errors on the computing units, and let the host select it's preferred solution. This is why the prototype developed here is not suitable for calculating forecasts. It is only suitable to determine the best parameter set to run the forecast with. Of course all the functionality is basically there, the forecast results are just not passed to the host for optimization reasons. Also commercial usage of the software is thereby effectively impeded.

## 5.3 Limitations of the Implementation

The software that was developed is not suitable for productive usage since the aim was to generally demonstrate the feasability of a fast parallel variant of the Jooke-Heeves optimization method on graphic cards. Several issues were not concentrated on, for example the maximum length of the input is essentially limited by the size of of the graphic device's memory. For larger input data one needs to split the input data into smaller parts and operate on one part at a time. The overall computional effort would increase by the operations needed for additional copying of data from and to the device's memory. That would of course affect the execution time, however every implementation would suffer similar penalties when input data reaches the maximum memory size and that is why such cases were not tested.

The task was to develop a prototype and hence there was not put much effort into programming a user-friendly framework what would cost anyway way too much time, since C++ is not really convenient for such tasks. Just for a few parameter that can be used error checks are conducted, so the software could crash if unreasonable parameters are chosen (accidentally).

# 6 Experiments

## 6.1 Setup

The development platform used is a rather simple PC with a 3Ghz 2-core CPU by AMD, 4 GB RAM as main memory, a SATA hard disk and a GeForce 9500 GT graphic card from Nvidia. This graphic card has 32 stream processors, 550 Mhz core clock speed, 1400 MHz shader clock speed, 800 MHz memory clock speed. Installed are 256 MB memory. As is clearly the case this system is in no way special or expensive to purchase. Even the greater is the suprise how fast such a system can perform the parallel parameter estimations. In fact it provided enough speed to go without testing the produced software on a high performance cluster. This came in handy since the required additional effort to ensure the code will run on multiple platforms (keep in mind the development platform has been a Windows system) is perhaps to much for the scope of this work.

The development tools used were Mircosoft's Visual Studio 2008 Express and NSight, version 2.2.0.12313, which is Nvidias tools collection for GPU programming, that also provides support for OpenCL in the version 1.2. The author is aware that a newer version of OpenCL has been released but since the date was after the beginning of this work, switching to the new version has been strictly ruled out. Earlier described problems with OpenCL may not be present anymore.

## 6.2 Data Sources

Initially a large set of data series appeared to be perfectly suitable for the purpose of our experiments. As it turned out, that was not always the case in reality.

The first major problem is that many records for example on economic data or especially energy market data is only available for sale. Even the United Nations Statistics Divisions ([UNS]) only offers data in a suitable format for money what is comprehensible by the author. Such economic politics shall not in any way be supported and therefore data that can only be acquired commercially is totally out of question.

Another possible source for time series data was the M-competition ([MCO]). Unfortunetely the data

was too short. Most series were not longer than sixty entries, some contained roughly a hundred entries. Experiments using this data showed expected side effects of quite large errors but were also harder to measure since the overall execution time was only few miliseconds. In fact many economic statistics are only recorded on monthly or yearly bases and are as a result very short.

The timeSeries package ([TSR]) for GNU-R provides some example data that was indeed suitable and large enough (over three hundred items) to be processed, though data series with several thousand entries still would have been much better. Some of this data is displyed in the following diagramm.

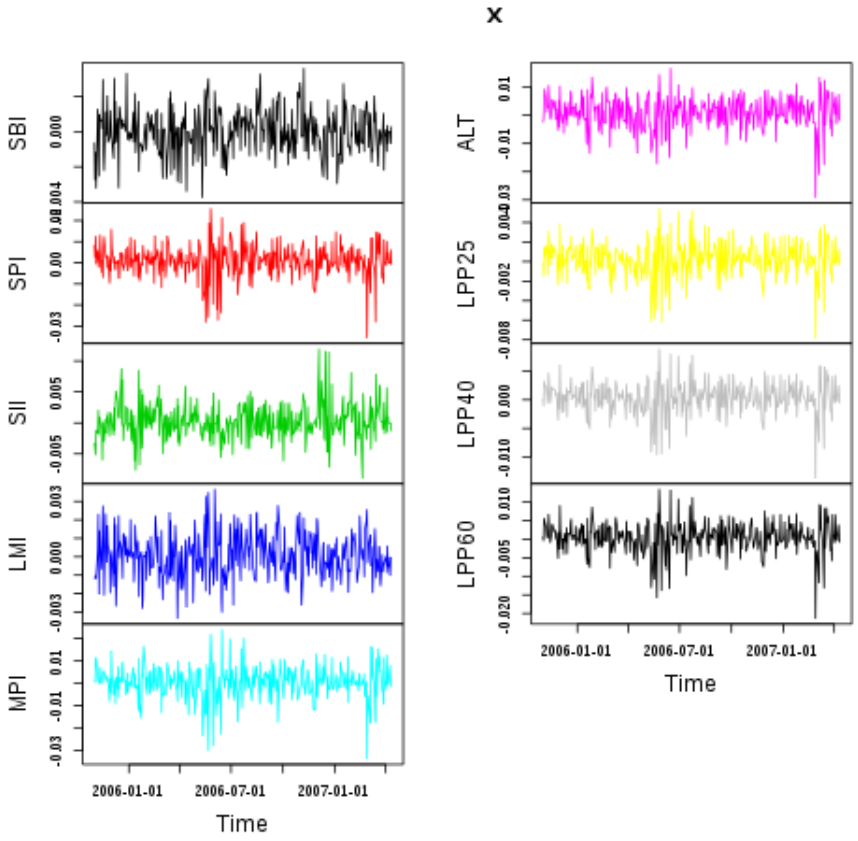


Figure 6.1: timeSeries package sample data

In addition to collected data from the real world, synthetic data was generated to test the algorithms. These series consisted of uniformly distributed random data. This turned out to be the only way to really test the performance of the developed algorithms, since for such tests series containing tens of thousands of items are necessary. The author is aware that this data does not contain seasonality

which is at first glance a problem for the triple exponential smoothing method here used as the target function. However this plays only a role when comparing the forecasts errors of different parameter optimizers (or comparing the different modes they run in). Hence synthetic data was not used in that case but it turned out to be suitable for all other performance test. An earlier completetion of the parameter optimzing process occasionally occurred with results showing for example that one parameter was not altered at all but on the other hand such instances were relatively seldom and therefore posed no significant contamination of the experiments.

## 6.3 Difference in Execution Time between CPUs and GPUs

For better understanding how dramatically the execution time can be reduced when using a commodity graphics card instead of a commodity CPU to calculate (especially when floating point number operations are involved), the following graph shall be provided.

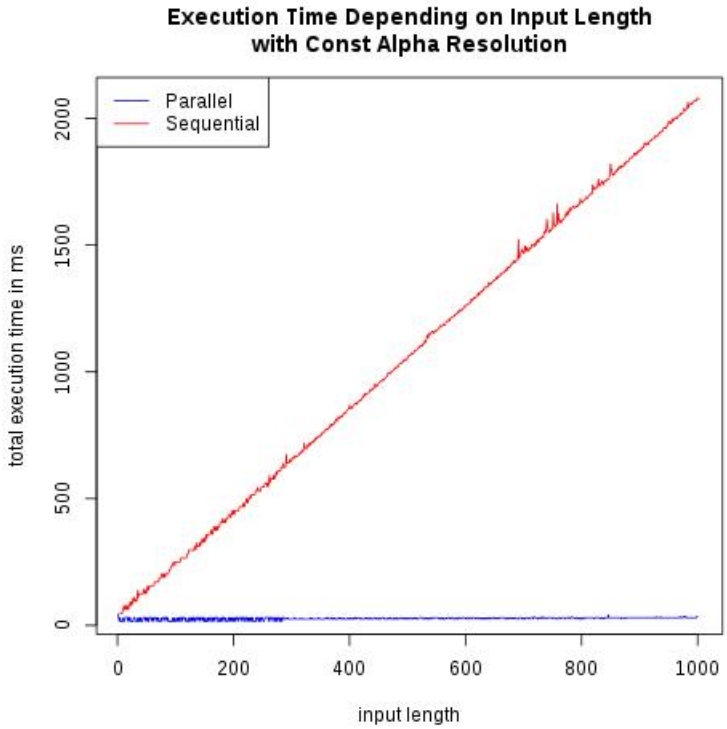


Figure 6.2: Difference in execution time for sequentially operating CPU core and parallel operating GPUs

Of course is it not a fair comparision. Clock frequencies of both devices differ, as does the width and

speed of the memory bus. Also the number of instructions per cycle and so on differ. However this graph can outline that transfering suitable tasks to a GPU can save a large amount of time. Please observe that the execution time almost appears to be constant in the paralllel case whereas it is rising clearly in the sequential case. In fact the parallel execution time line also rises steadily but in comparision with the other line the change is so small that it cannot be detected when both lines are present in the same graph.

For this experiment a simple grid search has been used to outline the difference. Roughly the same result can be achieved with any parameter estimator that uses at some point many iterations and therefore calls of the target function to determine the optimal parameters.

## 6.4 Influence of the Input Size

The next graphics shall be given to outline the influence of the input length on total execution time. It demonstrates this for one variant of the parallel Hooke-Jeeves.

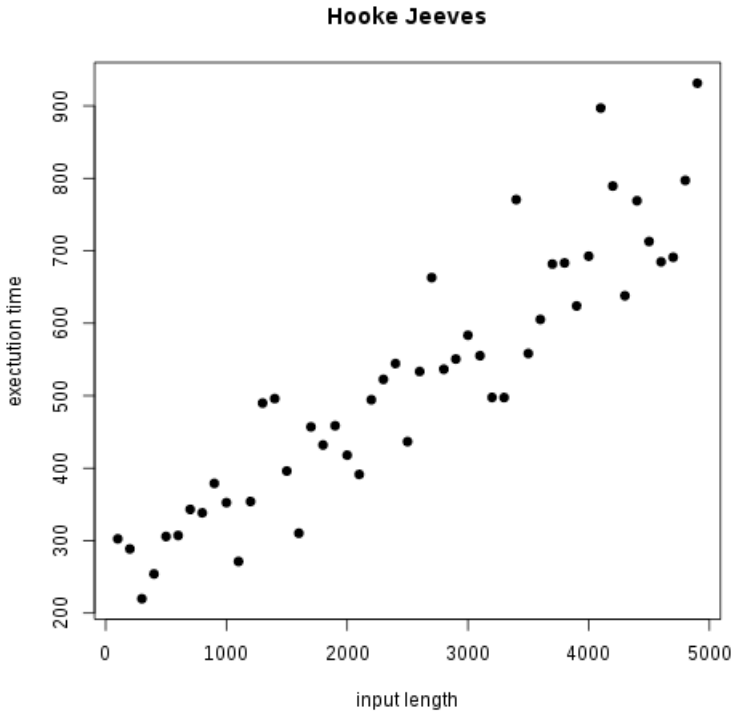


Figure 6.3: Hooke Jeeves, dependent on input length, execution time in ms

As can be seen the increase of execution times is roughly linear with a lot of outliners since the whole

process is randomized. The Hooke-Jeeves variant used is the so-called ”smart” variant, that tries to find random neighbouring points alongside, and gradually getting nearer, a given search vector. How the three implemented variants of Hooke Jeeves differ in theory has already been described earlier, how the differ in practise will be the topic in the next section.

While the size of the input grows by a factor of five, the total execution time only increases by a factor of two to three. Compare for example the execution times of 300-350 miliseconds for an input size of 1000, with 700-900 miliseconds for an input size of 5000.

For this experiments multiple forecasts were conducted for each input length and each time the average execution times were calculated.

## 6.5 Comparision of Different Modes of the Parallel Hooke-Jeeves

The next two diagramm present results from comparing all three implemented modes of the Hooke Jeeves. Each colour represents one mode, each dot stands for forecasting of one real-world data set. The first one involved three starting guesses, or instances, while the latter one involves ten.

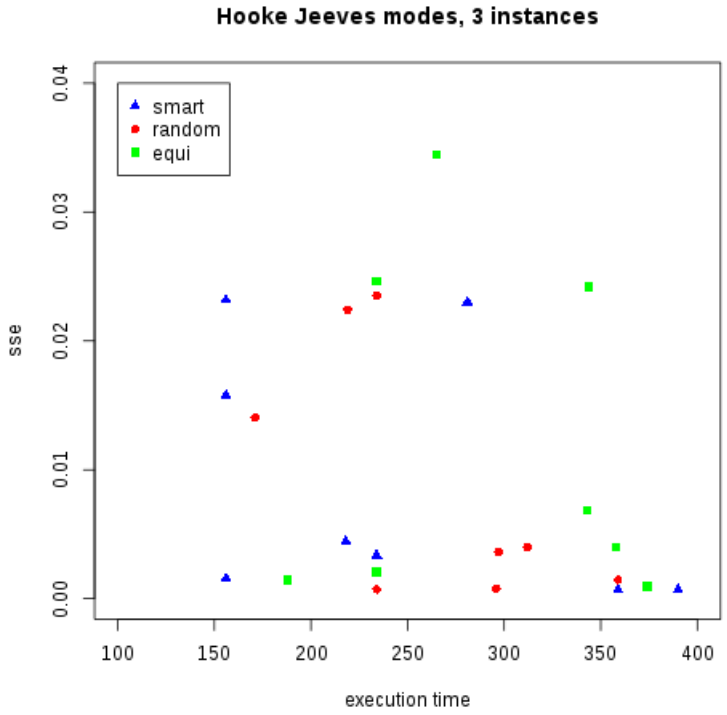


Figure 6.4: Hooke Jeeves, comparison of the three modes, with 3 instances.

First of all both the smart-variant and the random-variant show very similar behaviour. In fact one could argue that the additional effort for programming the smart-variant did not pay off. Both variants are heavily relying on randomized determination of parameters. To really compare them one would need a much larger amount of sample data series, that are each also of much larger size, meaning containing many values. As outline at the beginning of this chapter it was not possible to acquire large enough data to do that.

However the equi-variant does not rely much on random data, hence the results will not differ that much as is seen for the other two modes of the parameter estimator. Additionally the equi-mode is

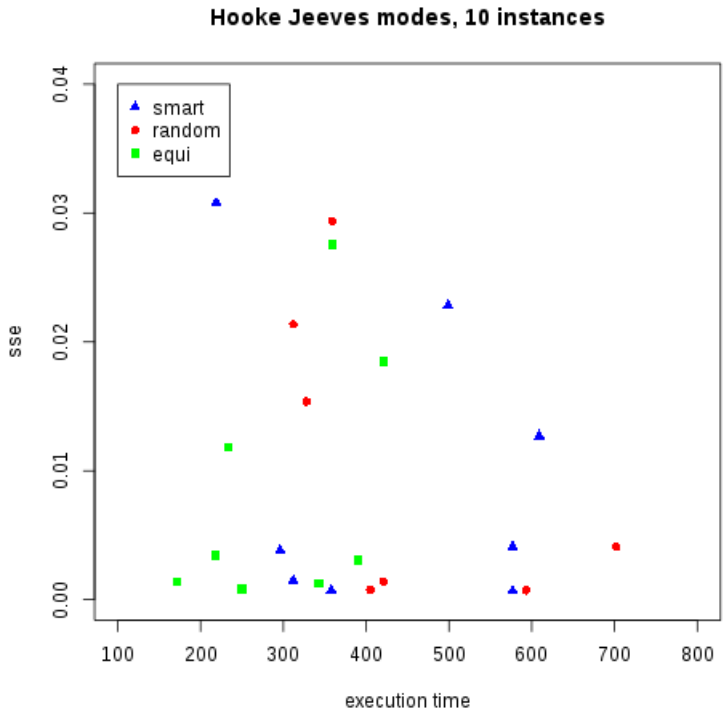


Figure 6.5: Hooke Jeeves, comparison of the three modes, with 10 instances.

even performing comparably well, if not better than the other ones, especially when looking at the second diagramme. This is not suprising since the more starting guesse or instances are regarded the more similar this mode resembles the grid search without the disadvantage of trying all possible values but with choosing only search paths that seem promising according to the pattern search method.

The equi-mode also might be provide the additional benefit that its runtime is more easily to predict than the runtimes of more randomized modes of the parameter optimizer. Especially if such algorithms shall be used for time critical forecasts this would be relevant in practise.

## 6.6 Influence of OpenCL Parameters

The first diagramme shows just a glimpse of the longer experiment that produced the second diagramme. Here we can clearly see how the speedup is capping for certain input lengths while at first it is relatively small. Each of the three curves represents a different value of the so-called worksize parameter. This is used by the OpenCL framework to determine on how many computing units the code shall be run in parallel and therefore essentially defines how much of the device is utilized.

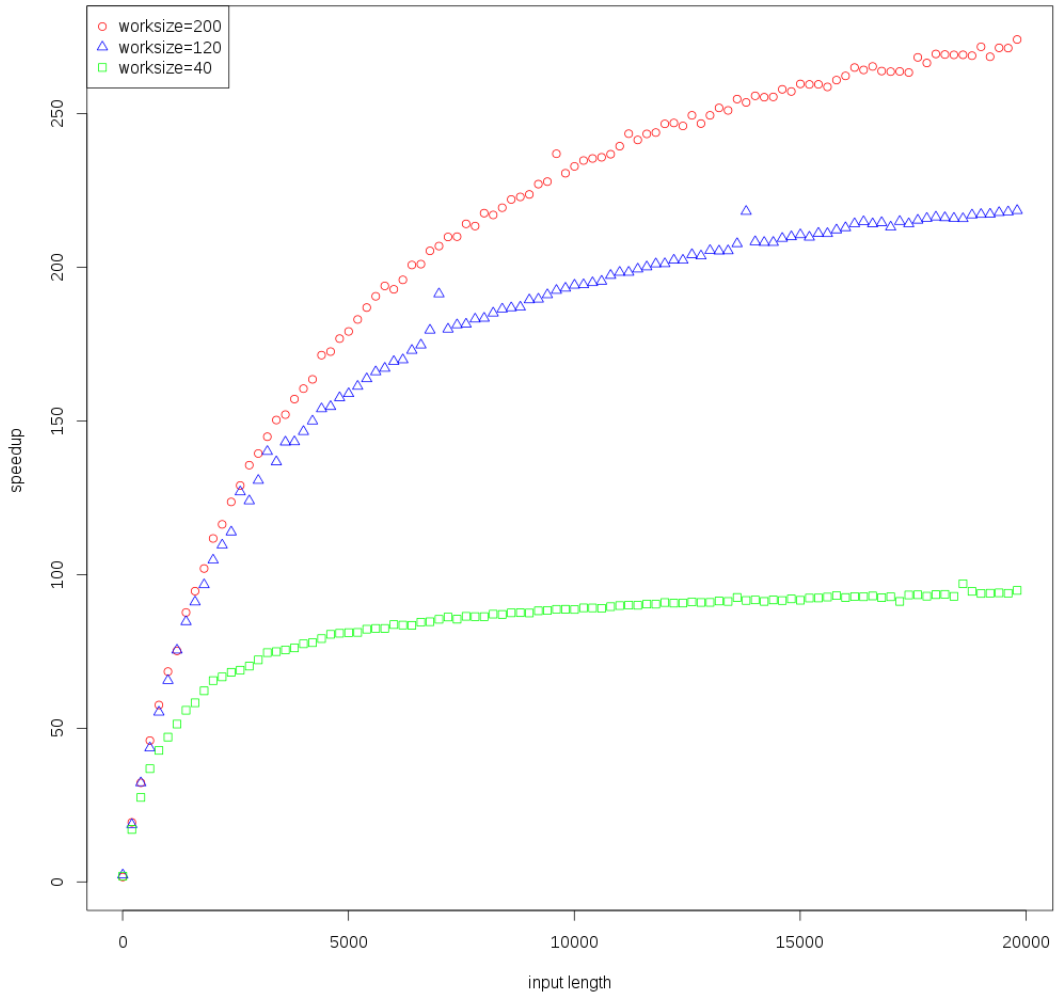


Figure 6.6: Speedup depending on worksize. Using grid search.

The second diagramme shows us a much longer run experiment, that demonstrates how the benefit

of increasing the worksize eventually vanishes as the hardware is fully utilized at a worksize value of around 200 hundred. From then on no further reducing of the overall execution time can be achieved. When remembering that the test system uses 32 stream processor, that can each perform up to 8 floating point operations at the same time, then you would end up with a maximum speed-up of 256. Therefore roughly twenty percent of the performance used up for the overhead and memory operations.

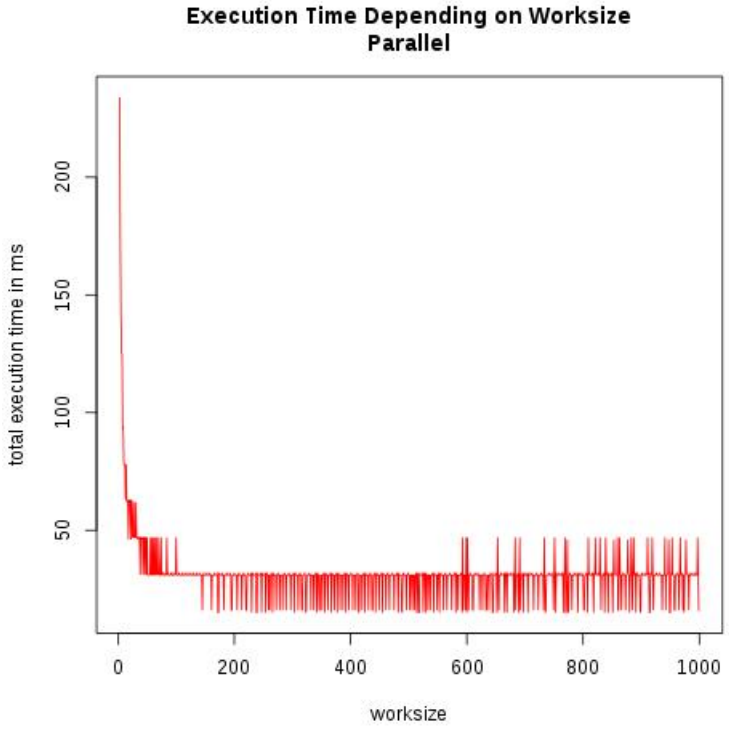


Figure 6.7: Longer experiment showing speedup depending on worksize. Using grid search.

Chronologically this experiment was conducted far earlier than the other ones and therefore the maximum number of the worksize that is reasonable for the experiments conducted afterwards had been set to 300 hundred, which is safely higher than the maximum in practise, since some side effects might result in higher speed-ups than 200 (but lower than 256) depending on the input data or operating system issues, hardware cooling and so on. In any way a higher set worksize does not harm the performance. The OpenCL framework manages this well enough.

# 7 Conclusion

First of all, there will always be new hardware improvements. For example graphics card vendors are working on special shared memory that enables simultaneous access of main memory data by the host and graphics cards. Such an improvement will render previously described concerns about memory transfer operations obsolete. Furthermore the new version of OpenCL has not been reviewed by the author so far, but it can be expected that a lot of issues were fixed, as the jump in version numbers from 1.2 to 2.0 suggests.

Regarding the parallel implementation of the Hooke-Jeeves I can summerise that the parameter estimator is clearly able to deliver forecast results within the brink of a second. Especially on newer graphics cards even further speed-up should be gained. The algorithm is able to adapt to a higher number of computing units without much effort, just by altering parameters, for example these that determine how many neighbours are calculated each round or how many starting points are used.

# Bibliography

[AdBHvL86] Emile Aarts, Frans de Bont, JHA Habers, and Peter JM van Laarhoven. *"A parallel statistical cooling algorithm"*. 1986.

[BSS88] Richard H Byrd, Robert B Schnabel, and Gerald A Shultz. "Parallel quasi-Newton methods for unconstrained optimization". *Mathematical Programming*, 42, 1988.

[CBZ10] Alexander Choong, Rami Beidas, and Jianwen Zhu. "Parallelizing Simulated Annealing-Based Placement using GPGPU". *2010 International Conference on Field Programmable Logic and Applications (FPL)*, 2010.

[CFW98] Hao Chen, Nicholas S Flann, and Daniel W Watson. "Parallel genetic simulated annealing: a massively parallel SIMD algorithm". *Parallel and Distributed Systems, IEEE Transactions on*, 9, 1998.

[CRSV87] Andrea Casotto, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. "A parallel simulated annealing algorithm for the placement of macro-cells". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6, 1987.

[FLZ06] Shu-Kai S. Fan, Yun-Chia Liang, and Erwie Zahara. "A genetic algorithm and a particle swarm optimizer hybridized with Nelder-Mead simplex search". *Computers and Industrial Engineering*, 50, 2006.

[GTG05] Brian Gerkey, Sebastian Thrun, and Geoff Gordon. "Parallel stochastic hill-climbing with small teams". *Multi-Robot Systems. From Swarms to Intelligent Automata*, 3, 2005.

[HKT01] Patricia D Hough, Tamara G Kolda, and Virginia J Torczon. "Asynchronous parallel pattern search for nonlinear optimization". *SIAM Journal on Scientific Computing*, 23, 2001.

[JT88] J. E. Dennis Jr and Virginia Torczon. "Parallel implementations of the nelder-mead simplex algorithm for unconstrained optimization.". *International Society for Optics and Photonics*, 1988.

[Kon04] Zdeněk Konfršt. “Parallel Genetic Algorithms: Advances, Computing Trends, Applications and Perspectives”. *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[KR86] Saul A. Kravitz and Rob A. Rutenbar. “Multiprocessor-Based Placement by Simulated Annealing”. *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, 1986.

[Mar13] Ronny Marx. “Parallelisierung Von Parameterschätzern Auf APUs”. 2013.

[MCO] “M-Competition”. `http://forecasters.org/resources/time-series-data/m-competition/`.

[OMGS04] Francisco Javier Ovalle-Martínez, Julio Solano González, and Ivan Stojmenović. “A parallel hill climbing algorithm for pushing dependent data in clients-providers-servers systems”. *Mobile Networks and Applications*, 9, 2004.

[TSR] “TimeSeries package for GNU-R”. `http://cran.r-project.org/web/packages/timeSeries/timeSeries.pdf`.

[UNS] “United Nations Statistics Division”. `http://unstats.un.org/`.

[VNR+06] Jasper A. Vrugt, Breannda´n Ó Nualláin, Bruce A. Robinsona, Willem Bouten, Stefan C. Dekker, and Peter M.A. Sloot. “Application of parallel computing to stochastic parameter estimation in environmental models”. *Computers and Geosciences*, 32, 2006.

# Copyright Information