GPU Accelerated Parameter Estimation by Global Optimization using Interval Analysis

Master Thesis

Group: 12gr977

Mads B. Eriksen & Søren Rasmussen
<berre, sorenr>@es.aau.dk

Signal Processing & Computing

June 5, 2013

Supervisors:
Torben Larsen & Tobias L. Jensen

Aalborg University
School of ICT
Fredrik Bajers Vej 7B
DK-9220 Aalborg

School of ICT
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
http://www.sict.aau.dk/

AALBORG UNIVERSITY

STUDENT REPORT

Title:
GPU Accelerated Parameter Estimation by Global Optimization using Interval Analysis

Theme:
Master Thesis

Project Period:
September 2012 - June 2013

Project Group:
12gr977

Participant(s):
Søren Rasmussen
Mads Berre Eriksen

Supervisor(s):
Torben Larsen
Tobias L. Jensen

Copies: 5

Page Numbers: 90

Attachments: CD containing source code and a digital copy of this thesis.

Date of Completion:
June 5, 2013

Abstract:

This master thesis treats the topic of non-linear parameter estimation using global optimization methods based on interval analysis (IA), accelerated by parallel implementation on a Graphics Processing Unit (GPU). Global optimization using IA is a mathematically rigorous Branch & Bound-type method, capable of reliably solving global optimization problems with continuously differentiable objective functions, even in the presence of rounding errors.
The structure of the problems and methods considered is parallel by nature and fit the parallel architecture of modern GPUs well. Methods for efficiently exploiting this parallelism are presented, based on which a parallel GPU accelerated global optimization algorithm is implemented. A set of algorithmic variations of the parallel GPU accelerated algorithm are benchmarked and compared to corresponding sequential CPU based implementations. Results show speedups ranging from 1.43 to 60.4 times for the test problems and problem sizes used. Analysis shows that the GPU accelerated implementations do not utilize the GPU hardware fully. It is assessed that further utilization and speedup can be obtained by introducing an additional level parallelism. It is concluded that for problems with large numbers of measurements, use of the method presented has the potential of yielding significant speedups.

School of ICT
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
http://www.sict.aau.dk/

AALBORG UNIVERSITET

STUDENTERRAPPORT

Titel:
GPU Accelereret Parameterestimering ved brug af Global Optimering med Intervalanalyse

Tema:
Kandidatspeciale

Projektperiode:
September 2012 - Juni 2013

Projektgruppe:
12gr977

Deltager(e):
Søren Rasmussen
Mads Berre Eriksen

Vejleder(e):
Torben Larsen
Tobias L. Jensen

Oplagstal: 5

Sidetal: 90

Bilag: CD med kildekode og en digital kopi af denne afhandling.

Afleveringsdato:
5. juni 2013

Abstract:

Dette kandidatspeciale omhandler emnet non-linear parameterestimering ved brug af metoder til global optimering baseret på intervalanalyse (IA), accellereret vha. parallel implementation på en grafikprocessor (GPU). Global optimering med IA er en matematisk garanteret metode af Branch & Bound typen, i stand til på pålidelig vis at løse globale optimeringsproblemer med kontinuært differentiabele objektivfunktioner, selv når afrundingsfejl finder sted. Strukturen af disse problemer og metoder er parallel i sin natur, og passer godt til moderne GPU-arkitektur. Metoder til effektivt at udnytte denne parallelisme præsenteres og baseret på disse implementeres en parallel GPU-accelereret algoritme til global optimering. En samling af algoritmiske variationer af den parallelle GPU-accelererede algoritme benchmarkes og sammenlignes med tilsvarende sekventielle CPU-baserede implementationer. Resultaterne viser hastighedsforøgelser imellem 1,43 og 60,4 gange for de anvendte testproblemer og problemstørrelser. En analyse viser at de GPU-accelerede implementationer ikke anvender GPU-hardwaren til fulde. Det vurderes der ved at introducere et yderligere lag af parallelisme kan opnås højere anvendelsesgrad og hastighedsforøgelse. Det konkluderes at den presenterede metode giver potentiale for signifikante hastighedsforøgelser for problemer med højt antal målinger.

Contents

Preface

This is the master thesis of Mads B. Eriksen and Søren Rasmussen, Signal Processing and Computing group 12gr977 at Aalborg University, written in the period from September 2012 to June 2013, partly in Atlanta, Georgia, USA and partly in Aalborg, Denmark.

Throughout the report equations are referenced using paranthesis, e.g. "(4.2)" refers to equation 2 in chapter 4. Sections, chapters, theorems, corollaries, figures, tables and algorithms are referenced by type and then number, e.g. figure 2 in chapter 4 is referenced "Figure 4.2". Citations are in IEEE style, e.g. [3, ch. 9] refers to chapter 9 in source number 3 in the reference list.

The software developed is implemented in Python, C++ and CUDA. The user does not have to be familiar with C++ or CUDA but should know Python to use the software. The model provided by the user most however be implemented in C. All software developed along with an electronic copy of the thesis can be found on the attached CD.

The authors would like to thank AccelerEyes and its employees and Assoc.Prof. Patricio A. Vela (Georgia Institute of Technology) for their kind hospitality and their inputs to the project. Further a special thanks to Prof. Torben Larsen and Postdoc Tobias L. Jensen (Aalborg University) for their supervision of the project and all their inputs.

Aalborg University, June 5, 2013

Søren Rasmussen
sorenr@es.aau.dk

Mads Berre Eriksen
berre@es.aau.dk

Notation and Abbreviations

Sets

$\mathbb{R}$	The real numbers.
$\mathbb{R}_*$	The extended real numbers.
$\mathbb{IR}$	The finite interval numbers.
$\mathbb{IR}_*$	The extended interval numbers.
$\mathcal{S}$	A set or sequence, depending on the context.

Real numbers

Real numbers are denoted using lower-case letters.

x	Real scalar.
$\mathbf{x}$	Real matrix or vector.
$\mathbf{x}_{i,j}$	The i^{th} row and j^{th} column of the matrix $\mathbf{x}$.
$(\mathbf{x}_{\text{m}})_{i,j}$	The i^{th} row and j^{th} column of the matrix $\mathbf{x}_{\text{m}}$ (used when several subscripts are needed).
$\mathbf{x}_i$	The i^{th} element of the vector $\mathbf{x}$.
$(\mathbf{x}_{\text{m}})_i$	The i^{th} element of the vector $\mathbf{x}_{\text{m}}$ (used when several subscripts are needed).

Intervals

Intervals are denoted using capital letters. See Section 2.1 for more.

$\underline{X}$	Real interval.
$\overline{X}$	Upper bound of X.
$\underline{X}$	Lower bound of X.
$[a, b]$	Interval with lower bound a and upper bound b.
$\mathbf{X}$	Interval matrix/vector.
$\text{int}(X)$	Interior of X.
$\text{mid}(X)$	Midpoint of X.
$\text{wid}(X)$	Width of X.
$\text{vol}(\mathbf{X})$	Volume of $\mathbf{X}$.
$\text{hull}(\mathcal{S})$	Hull of the set $\mathcal{S}$.
$\text{cset}(f, \mathcal{S})$	Containment set of the function f over the set $\mathcal{S}$.

Symbols

Ψ	Number of parameters in the model function.
Φ	Number of measurements.
Γ	Dimension of the measurement points.

Abbreviations

ALU	Arithmetic Logic Unit
AMP	Asynchronous Multiple Pool
ASP	Asynchronous Single Pool
BB	Branch & Bound
FLOPS	Floating Point Operations Per Second
FMA	Fused Multiply Add
FPU	Floating Point Unit
GMU	Grid Management Unit
GPU	Graphical Processing Unit
GPGPU	General Purpose Graphical Processing Unit
GS	Gauss Seidel
IGS	Interval Gauss Seidel
IGO	Interval Global Optimization
IN	Interval Newton
IN/GB	Interval Newton/Generalized Bisection
IOPS	Interval Operations Per Second
MIMD	Multiple Instructions Multiple Data
NVVP	Nvidia Visual Profiler
OpenCL	Open Computing Language
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
SMP	Synchronous Multiple Pool
SSP	Synchronous Single Pool
ULP	Unit in the Last Place

Chapter 1

Introduction

This chapter serves as an introduction to the thesis. The objective of the thesis is to research the possibility of accelerating the non-linear least-squares parameter estimation problem using a Graphical Processing Unit (GPU).

In Section 1.1 the non-linear least-squares parameter estimation is introduced, followed by an overview of global optimization methods for solving the problem in Section 1.2. The global optimization method selected for use in this work is a Branch & Bound-type algorithm that uses Interval Analysis. Finally Section 1.3 provides an overview of the state of the art within methods for interval global optimization (IGO) and parallelization thereof.

1.1 Parameter Estimation

The task of parameter estimation often arises in the engineering sciences (see e.g. [1, 2, 3, 4]). Given a set of measurements on a system and a mathematical model of this system with unknown parameters, the task is to somehow find a set of parameters that makes the model predictions fit the system behavior as closely as possible. In terms of least squares, the optimal set of parameters is obtained by solving the following optimization problem[1]:

$$\mathbf{p}_{\text{opt}} = \underset{\mathbf{p}}{\operatorname{argmin}} f(\mathbf{p}, \mathbf{x}, \mathbf{y}) \tag{1.1}$$

$$f(\mathbf{p}, \mathbf{x}, \mathbf{y}) = \sum_{i=0}^{\Phi-1} (y_i - m(\mathbf{p}, \mathbf{x}_i))^2 \tag{1.2}$$

where Ψ is the number of parameters to be estimated and $\mathbf{p} \in \mathbb{R}^{\Psi}$ is a vector of the parameters for the model function $m(\mathbf{p}, \mathbf{x}_i) \in \mathbb{R}$. The measurements, the number of which is denoted Φ, are contained in the vector $\mathbf{y} \in \mathbb{R}^{\Phi}$. The corresponding measurement points are contained in the matrix $\mathbf{x} \in \mathbb{R}^{\Phi \times \Gamma}$ where $\mathbf{x}_i \in \mathbb{R}^{\Gamma}$ denotes the i^{th} row containing the i^{th} measurement point.

In the case where the model function $m(\mathbf{p}, \mathbf{x}_i)$ is non-linear in $\mathbf{p}$, (1.2) may result in a non-convex optimization problem i.e. (1.2) may be multimodal and thus contain several local minima [7]. It is important to notice that (1.2) is not nesessarily non-convex if the model function $m(\mathbf{p}, \mathbf{x}_i)$ is non-convex. The present work focuses on

[1] An alternative approach, where the measurements themselves are regarded as uncertain and therefore included as variables in the optimization problem is the Error In Variables (EIV) approach. For more on this method, see e.g. [5, 6]

these cases, where the model function is non-linear. To solve non-convex optimization problems global optimization techniques are required.

For these types of problems, as with most other computational problems, it is natural to strive for as low execution time as possible in order to make it feasible to solve progressively larger problems within a given time frame. During the recent years, the floating point processing performance of Graphical Processing Units (GPUs) has risen significantly beyond that of common CPUs, when performing parallel computations [8]. This has resulted in a trend within high performance computing to utilize GPUs to accelerate the execution of algorithms [9]. As noted in e.g. [10, 11, 12], global optimization problems are parallel by nature. These facts combined makes it natural to attempt to accelerate algorithms to solve the non-linear parameter estimation problem using GPUs.

1.2 Global Optimization Methods

A large number of different methods for global optimization exist, and thus a complete survey of these methods is out of scope. Instead, this section aims to provide a classification of the methods for global optimization based on their properties. Classification of global optimization methods is dependent on the properties considered, and is therefore not unique. One classification[2], given in [7], defines the following categories:

Incomplete methods: Methods that use search heuristics and provide no guarantees of finding a global optimum.

Asymptotically complete methods: Methods that, assuming infinite run time and use of exact arithmetic, find a global optimum with probability one, but does not know when a global optimum is found.

Complete methods: Methods that, assuming infinite run time and use of exact arithmetic, find a global optimum, and after a finite amount of time is able to provide an approximate global optimum within specified tolerances.

Rigorous methods: Methods that, within a fixed amount of time, find an approximate global optimum within specified tolerances, even without use of exact arithmetic.

The latter two, and especially the last, of these categories are obviously stronger than the former two, due to the fact that they provide guarantees regarding the quality of the result.

In this work the focus is on rigorous methods, specifically the methods based on Interval Analysis (IA) (see Chapter 2 and 3). Methods for interval global optimization (IGO) are based on the *Branch & Bound* (BB) method [13]. BB uses a divide-and-conquer principle for global optimization where the initial search region is successively divided into subregions. For each subregion the bounds of the objective function are evaluated and based on this the subregion is split (branched), discarded from the search or accepted as a possible solution [14]. An interpretation of BB is that of a search tree structure, as illustrated in Figure 1.1. Starting with the entire search region, represented to the left on the figure by the top node and to the right by the black box, the search region is split into progressively smaller subregions, represented as nodes on the tree to the left in the figure.

[2] For other classifications, see e.g. [10].

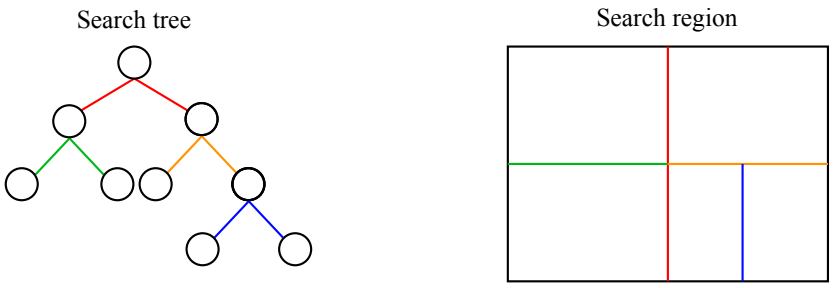


Figure 1.1: Illustration of a Branch & Bound search three. The search region, illustrated to the right, is split into smaller boxes. The subregions created by the splits are represented as nodes in the tree to the left. The colored lines in search region to the right represent the lines along which subregions are split. This is reflected by the colored lines connecting the nodes in the tree to the left.

In order to compute bounds of the objective function over a region, the method used must utilize some knowledge of the function. In the case of IGO, all methods require that the objective function f is continuous, and additionally most require that f is continuously differentiable and its gradient $\mathbf{g}_f$ and Hessian $\mathbf{h}_f$ must contain a finite number of isolated zeros [15, 16, 17, 18].

While most IGO algorithms are designed for unconstrained optimization (bounds-constrained in practice), some are able to work with constraints. The constrained case is not treated here. For more on constrained IGO, see [11, ch. 14-16].

A few practical examples of parameter estimation problems that can be solved using IGO include: vapor-liquid equilibrium problems in chemistry [19], sinusoidal modeling problems, used in speech analysis [20, 21] and the Error Vector Magnitude problem [22]. For more examples, see e.g. [23, 4]. An example of a category of problems that are not solvable by IGO methods is problems with discrete valued variables.

In practice, whether or not a parameter estimation problem can be solved within a reasonable time frame using IGO methods, is determined by the number of parameters to be determined. In [24] a collection of different parameter estimation problems, solved using the BARON solver, are collected. The maximum number of parameters solved are 7. The authors have not been able to find any examples of solvers based on interval analysis or other rigorous methods that solve parameter estimation problems with more than 7 parameters.

1.3 State of the Art

This section provides an brief overview of the state of the art within IGO methods and parallel implementations of IGO methods.

Much work has been done in IGO. The field of Interval Analysis was pioneered in the mid-sixties by Ramon E. Moore with publication of the book *Interval Analysis*[3]. In [25] a short overview of the early work Moores early work is given. In 1979 a method for global optimization using interval analysis was proposed for the one dimensional case by Hansen in [17]. The following year it was extended to the multidimensional case by Hansen in [18]. A comprehensive work on IGO including many of the developments until 2004 can be found in [11], in which advanced methods

[3]R. E. Moore - "Interval Analysis", Prentice-Hall, 1966. This book is unfortunately not available to the authors.

for both unconstrained and constrained IGO using heuristics to combine different methods are presented.

Most IGO algorithms make use the Interval Newton method – a method that can be use contract areas of the search space to enclose solutions or disqualify areas of the search space that do not contain solutions. As part of the Interval Newton method a linear system of equations must be solved. This is commonly done using the interval Gauss-Seidel method [26, 27, 28, 11]. In order to make the method more efficient, the system to be solved can be preconditioned. A standard approach is the inverse-midpoint preconditioner. This approach is, however, proven to be sub-optimal in some cases in [29]. Instead [29] presents an approach where an optimal preconditioner is computed using a linear programming approach. However, as this approach is computationally heavier than the inverse-midpoint approach, it does not always result in better overall efficiency. In [28] a pivoting preconditioning approach, based on ideas from [29] is presented and combined with the inverse-midpoint preconditioner in a hybrid preconditioning approach, which is shown to lead to substantial speedups for the problems tested.

Within parallel IGO, some notable works are [30, 31, 32, 33] and [34]. These all take the approach of processing nodes in the search tree in parallel. This is covered further in Section 5.1. To the best knowledge of the authors, no work on implementation of parallel IGO using GPUs has been published at the time of writing.

1.4 Summary

This project treats the subject on non-linear parameter estimation based on interval global optimization. These are problems that are parallel in nature and can be very time consuming to solve. Therefore the possibilities to parallelize the optimization algorithm using GPUs is investigated. Interval global optimization is an area where significant amounts of research has been done, however, to the best knowledge of the authors, no attempts to parallelize the method using GPUs have been published.

In Chapter 2 the central concepts of intervals and interval arithmetic are introduced. Chapter 3 describes how interval analysis can be to construct a method for global optimization, some approaches for accelerating the convergence of these methods, and considerations specific to the parameter estimation problem. In Chapter 4 the Nvidia CUDA GPU platform is introduced. The hardware architecture of CUDA GPUs, the CUDA programming model are described, and use of the CUDA platform for interval computations is treated. A scheme for parallelization and a GPU accelerated implementation of IGO based on this scheme is described in Chapter 5 along with a sequential CPU based reference implementation. The GPU accelerated implementation and the CPU reference implementation are benchmarked using a set of test problems and compared to measure the speedup in Chapter 6. Finally, Chapter 7 concludes on the present work and discusses possibilities for improvement of the implementation presented.

Chapter 2

Interval Analysis

This chapter introduces the central concepts of intervals and interval arithmetic. In Section 2.1, the basic concepts of intervals, finite interval arithmetic, functions of intervals, dependence and extended interval arithmetic are introduced, followed by a short introduction to computational interval arithmetic. This is followed by an introduction to systems of linear interval equations and how they are solved in Section 2.2.

This chapter is primarily included to give the reader a quick overview and understanding of some important concepts.

2.1 Interval Arithmetic

This section describes the basic concepts of finite and extended interval arithmetic. Throughout the section, some definitions and theorems are presented. For proofs and further details see [11, Ch. 2-4] and [13], on which this section primarily is based.

2.1.1 Finite Interval Arithmetic

In finite interval arithmetic, introduced in by R.E. Moore in 1966[1] [11, 13], the concept of a real *interval number* (or just *interval*) $X \in \mathbb{IR}$ describes a closed interval. X is defined by its real lower and upper endpoints $\underline{X}$, $\overline{X} \in \mathbb{R}$, such that [11, 13]

$$X = [x_\ell, x_u] = \left[\underline{X}, \overline{X}\right] = \{x \mid x_\ell \leq x \leq x_u\} \tag{2.1}$$

Here and in the following, capital letter variables represent intervals and lowercase letter variables represent real numbers. The lower- and upper bounds of interval variables are sometimes referenced using under- and overlined capital letters, respectively, as in (2.1). By this definition, a real number y corresponds to an interval of width zero (called a *degenerate* interval), where $Y = \left[\underline{Y}, \overline{Y}\right] = [y, y]$ [11, 13].

The basic arithmetic operations on intervals are extensions of their real counterparts. For intervals $X = [x_\ell, x_u]$ and $Y = [y_\ell, y_u]$ they are defined as in [11, 13]:

$$X \text{ op } Y = \{x \text{ op } y \mid x \in X, y \in Y\} \quad \text{for} \quad \text{op} \in \{+, -, \cdot, /\} \tag{2.2}$$

[1] In the publication R. E. Moore - "Interval Analysis", Prentice-Hall, 1966. This publication is unfortunately not available to the authors.

This leads to the following expressions for the basic operations [11, 13]:

$$X + Y = [x_\ell + y_\ell,\, x_u + y_u] \tag{2.3}$$
$$X - Y = [x_\ell - y_u,\, x_u - y_\ell] \tag{2.4}$$
$$X \cdot Y = [\min(x_\ell y_\ell, x_\ell y_u, x_u y_\ell, x_u y_u),\, \max(x_\ell y_\ell, x_\ell y_u, x_u y_\ell, x_u y_u)] \tag{2.5}$$
$$X/Y = [x_\ell,\, x_u] \cdot [1/y_u,\, 1/y_\ell]\,, \quad 0 \notin Y \tag{2.6}$$

Additionally exponentiation is defined for $n \in \mathbb{N}_0$,

$$X^n = \begin{cases} [1,\, 1] & \text{for } n = 0 \\ [x_\ell^n,\, x_u^n] & \text{for } x_\ell \geq 0 \text{ or } n \text{ odd} \\ [x_u^n,\, x_\ell^n] & \text{for } x_u \leq 0 \text{ and } n \text{ even} \\ [0,\, \max(x_\ell^n,\, x_u^n)] & \text{for } x_\ell \leq 0 \leq x_u \text{ and } n \text{ even} \end{cases} \tag{2.7}$$

Note that (2.6) is not defined when the interval in the denominator contains zero, as this would cause a division by zero in real arithmetic in (2.2), which is undefined.

As for real numbers, vectors and matrices of intervals can be used. Interval vectors are sometimes called *boxes*, as they describe an n-dimensional box. Some definitions for intervals X and interval vectors $\mathbf{X} \in \mathbb{IR}^n$ are shown in Table 2.1.

Concept:	**Interval:**	**Interval vector:**
Midpoint	$\mathrm{mid}(X) = \frac{1}{2}(\underline{X} + \overline{X})$	$\mathrm{mid}(\mathbf{X}) = [\mathrm{mid}(X_0), \mathrm{mid}(X_1), \ldots, \mathrm{mid}(X_{n-1})]^{\mathrm{T}}$
Width/diameter	$\mathrm{wid}(X) = \overline{X} - \underline{X}$	$\mathrm{wid}(\mathbf{X}) = \max_{i=0,\ldots,n-1} \mathrm{wid}(X_i)$
Abs. value/norm	$\lvert X\rvert = \max\{\lvert\underline{X}\rvert, \lvert\overline{X}\rvert\}$	$\lvert\mathbf{X}\rvert = \max_{i=0,\ldots,n-1} \lvert X_i\rvert$
Box volume		$\mathrm{vol}(\mathbf{X}) = \prod_{i=0}^{n-1} \mathrm{wid}(X_i)$

Table 2.1: Real functions on intervals and interval vectors [11, 13].

2.1.2 Functions of Intervals

There are two classes of functions on intervals; real functions of intervals, and interval functions. Real functions of intervals are real-valued functions that take interval arguments, such as the functions in Table 2.1. Interval functions are interval-valued functions that take interval arguments. The concept of interval functions is central to interval analysis [11]. An important condition that must hold for interval functions is the containment constraint [11]:

$$F(\mathbf{X}) \supseteq \{f(\mathbf{x}) \mid \mathbf{x} \in \mathbf{X}\} \tag{2.8}$$

A couple of important definitions lead to the so called fundamental theorem of finite interval analysis.

Definition 2.1 (Interval extension) [11, 13] An interval function $F(X)$ is called an *interval extension* of a real function $f(x)$ if $F(\mathbf{x}_0) = f(\mathbf{x}_0)$ for all $\mathbf{x}_0 \in \mathcal{D}_f$, where $\mathcal{D}_f$ is the domain of f.

⋆

Definition 2.2 (Inclusion isotonicity) [11, 13] An interval extension F is *inclusion isotonic* if $F(\mathbf{X}') \subseteq F(\mathbf{X})$ for all $\mathbf{X}' \subseteq \mathbf{X} \subseteq \mathcal{D}_f$.

⋆

Theorem 2.1 (The fundamental theorem of finite interval arithmetic) [11, 13] Let $F(\mathbf{X})$ be an inclusion isotonic interval extension of a real function $f(\mathbf{x})$. Then $F(\mathbf{X})$ contains the values of $f(\mathbf{x})$ for all $\mathbf{x} \in \mathbf{X}$ (i.e. (2.8) holds). ♢

This means that interval extensions can be used to bound the values of real functions over an interval. This is a very useful property that is used extensively in Chapter 3.

2.1.3 Dependence

Each time an interval variable occurs in an interval expression, it is treated as a separate interval. As an example, consider the operation of squaring an interval. For real numbers, squaring is the same as multiplying the number with itself, but for intervals, using the definitions of the operations in Section 2.1.1 on page 5:

$$X = [-3,\, 6] \tag{2.9}$$
$$Y_1 = X^2 = [0,\, 36] \tag{2.10}$$
$$Y_2 = X \cdot X = [-18,\, 36] \tag{2.11}$$

Observe that $\{y \,|\, y = x^2\ ,\ x \in X\} = Y_1 \subseteq Y_2$, that is; both results include the true result, but only Y_1 is a *tight* or *sharp* result. The extra width of Y_2 is caused by the effect called dependence. The implication of dependence is that the algebraic form of interval expressions has an effect on the quality of the result, contrary to real expressions [11].

Another example of dependence is subtraction. Consider a situation where $Y = X_1 + X_2$ where X_2 is known, and X_1 is unknown. To obtain the value of X_1, one would normally perform the manipulation

$$X_1 = Y - X_2 \tag{2.12}$$

However, when $X_1 + X_2$ is substituted for Y and the definition of subtraction in (2.4), the right-hand side of (2.12) becomes:

$$X_1 + X_2 - X_2 = X_1 + \left[\underline{X_2} - \overline{X_2},\, \overline{X_2} - \underline{X_2}\right] \neq X_1 \text{ when } \underline{X_2} \neq \overline{X_2} \tag{2.13}$$

Instead, to obtain X_1, one must use a dependent form of subtraction, which can be interpreted as the inverse of (2.3):

$$X \ominus Y = \left[\underline{X} - \underline{Y},\, \overline{X} - \overline{Y}\right] \tag{2.14}$$

Applying this to (2.13) yields:

$$X_1 + X_2 \ominus X_2 = X_1 + \left[\underline{X_2} - \underline{X_2},\, \overline{X_2} - \overline{X_2}\right] = X_1 \tag{2.15}$$

Each of the basic operations (2.3)-(2.6) has a corresponding dependent version [11].

2.1.4 Extended Interval Arithmetic

The interval system introduced in Section 2.1.1 has a large drawback in that not all operator-operand combinations are defined. An example of this is division by intervals containing zero (see (2.6)), which is quite a large restriction when constructing algorithms.

To eliminate this problem, an alternative interval system, based on the set of extended real numbers $\mathbb{R}_* = \mathbb{R} \cup \{-\infty, \infty\} = [-\infty, \infty]$ was introduced independently in 1968 by Kahan and Hanson[2] [11] and later developed into a mathematically consistent system presented in [11]. The set of intervals over the extended reals is denoted $\mathbb{IR}_*$.
A central concept when working with arithmetic over the extended reals is that of *containment sets*.

Definition 2.3 (Containment set) [11] The containment set of an expression evaluated on all points in the set $\mathcal{S}$, denoted $\text{cset}(f, \mathcal{S})$, is the smallest possible set of values taken on by all possible algebraic transformations of the expression evaluated on all points in $\mathcal{S}$.

$\star$

Contrary to the real numbers, the containment sets of the basic arithmetic operations on the extended reals are not always single values, but may be intervals or sets. The containment sets listed in Appendix D in Tables D.1-D.4.
The fact that containment sets may be quite complicated to represent and use, motivates the introduction of interval boxes that enclose the containment sets, named *containment set enclosures*:

Definition 2.4 (Containment set enclosure) [11] $F(\mathbf{X})$ is a *containment set enclosure* of f if $F(\mathbf{X}) \supseteq \text{cset}(f, \mathbf{X})$ for all $\mathbf{X} \in \mathbb{IR}_*^n$.

$\star$

This leads to the fundamental theorem of extended interval arithmetic:

Theorem 2.2 (The fundamental theorem of extended interval arithmetic) [11] Given the real expression $f(\mathbf{x}) = g(h(\mathbf{x}), \mathbf{x})$ and the containment set enclosure of h, $H(\mathbf{X})$, then $G(H(\mathbf{X}), \mathbf{X})$ is a containment set enclosure of $\text{cset}(f, \mathbf{X})$ for all $\mathbf{X} \in \mathbb{IR}_*^n$.

$\diamondsuit$

The containment sets of some operations have shapes that can be exploited in algorithms. Consider for example the expression $f(X, Y) = \frac{X}{Y}$, evaluated at $(X_0, Y_0) = ([1, 2], [-3, 5])$. The containment set enclosure F of f is $[-\infty, \infty]$ because Y_0 contains zero (see Appendix D Table D.4), but the containment set is:

$$\text{cset}(f, (X_0, Y_0)) = \left\{ \left[-\infty, -1/3\right], \left[1/5, \infty\right] \right\} \tag{2.16}$$

The containment set in (2.16) consists of the union of two distinct semi-infinite intervals, separated by a gap. Specifically, when $0 \notin X$ and $\underline{Y} < 0 < \overline{Y}$, the containment set of $f(X, Y) = \frac{X}{Y}$ consists of two semi-infinite intervals [11]. An algorithm where knowledge of this gap is described in Section 2.2.1.

[2]The publications W.M. Kahan - "A more complete interval arithmetic" - 1968 and R.J. Hanson - "Interval arithmetic as a closed arithmetic system on a computer" - 1968 are unfortunately not available to the authors.

2.1.5 Computational Interval Arithmetic

Directed rounding

When doing interval computations in practice in finite precision, rounding errors inevitably occur. Normally when using IEEE-754 floating point arithmetic, the round-to-nearest mode is used. Using this mode for interval computations, causes the finite precision result to sometimes become narrower than the infinite precision result, thus breaking inclusion isotonicity. In order to ensure that inclusion isotonicity holds, outward rounding of the interval bounds is used. That is, the lower bound is always rounded down to the nearest representable finite precision value and the upper bound is always rounded up [11, 35].

The level of support for directed rounding differs between hardware architectures and math libraries used. An example is the x86 architecture which supports directed rounding, but switching between rounding modes is often more time consuming than the actual operations [35].

Software for Interval Arithmetic

Many software libraries and a few compilers supporting interval arithmetic are available. This section mentions a few of these, but should not be considered a complete survey. Some of the libraries available are:

INTLAB: A MATLAB toolbox for reliable computing, which includes support for Interval Arithmetic [36].

Boost: A collection of C++ libraries, which includes an interval arithmetic library. The library has no direct support for transcendental functions, such as $\sin$, $\exp$ and $\log$ [37]. A CUDA port of a subset of the library was shipped with the CUDA 3.2 SDK Samples and is included but undocumented in the later versions (up to 5.0) [38].

filib++: A C++ extension of the ANSI-C library FI_LIB, which is short for *Fast Interval Library* [39].

RealLib: A C++ library for exact real arithmetic. The library makes use of the fact that the the rounding modes of the SSE-2 registers in Intel's x86 platform can be set independently of each other and of the x87 floating point unit (FPU). By using the SSE-2 registers for interval arithmetic and using the x87 FPU for other computations, switching of rounding modes becomes unnecessary. Speedups between 2.53 and 20.6 times speedup for the basic operations is reported comparing the Boost library in version 1.33 with RealLib3. The library does not support division by intervals containing zero [35].

The availability of compilers with support for interval arithmetic is more limited. One example is the Fortran 95 compiler by Oracle (formerly by Sun Microsystems) [40].

2.2 Systems of Linear Interval Equations

As with real numbers, systems of linear interval equations can be presented in the matrix-vector form

$$\mathbf{Ax} = \mathbf{B} \tag{2.17}$$

where $\mathbf{A} \in \mathbb{IR}_*^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{B} \in \mathbb{IR}_*^m$. The containment set $\mathcal{S}$ of the solution is:

$$\mathcal{S} = \{\mathbf{x} \mid \mathbf{ax} = \mathbf{b}, \mathbf{a} \in \mathbf{A}, \mathbf{b} \in \mathbf{B}\} \tag{2.18}$$

As the shape of $\mathcal{S}$ can be rather complicated, what is normally sought is the box that encloses the containment set, denoted the *hull*, of $\mathcal{S}$, $\mathbf{X} = \text{hull}\,(\mathcal{S})$ [11].

There are several different methods available for computing or enclosing this hull. A straightforward direct method is Gaussian elimination. As noted in [11] and [13], Gaussian elimination fails whenlk, $\mathbf{A}$ is not strongly diagonally dominant or when elimination requires division by an interval containing zero, and it is therefore only applicable problems where these conditions are not present.

Another direct method is the *hull method*. When $\mathbf{A}$ is non-singular, this method computes the exact hull of the solution. The hull method is not used in this project. For a description of the method, see [11].

An iterative method called the *Krawczyk method*, avoids the limitations of Gaussian elimination and the hull method, and is therefore more applicable [26, 11]. In [26] an interval version of the Gauss-Seidel method (sometimes called the *Hansen-Sengupta method*) was presented and shown to produce tighter bounds than the Krawczyk method at approximately the same computational cost. This method is broadly used and regarded as efficient ([26, 11, 13, 27, 41, 19]), and is therefore used in this project. The method is described in Section 2.2.1.

2.2.1 The Interval Gauss-Seidel Method

The Interval Gauss-Seidel (IGS) method presented in [26] is, as the name suggests, an interval version of the real Gauss-Seidel (GS) method (For a description of the GS method, see e.g. [42]).

Each row of the system in (2.17) describes an equation of the system. The i^{th} row describes the equation:

$$B_i = \sum_{j=0}^{n-1} A_{ij} x_j \tag{2.19}$$

which can be solved for x_i:

$$x_i = \frac{1}{A_{ii}} \left(B_i - \sum_{j=0}^{i-1} A_{ij} x_j - \sum_{k=i+1}^{n-1} A_{ik} x_k \right) \tag{2.20}$$

Given interval bounds $\mathbf{X}$ on each element in $\mathbf{x}$, the values of x_j and x_k in (2.20) are substituted with these bounds. This yields new bounds N_i on x_i:

$$N_i = \frac{1}{A_{ii}} \left(B_i - \sum_{j=0}^{i-1} A_{ij} X_j - \sum_{k=i+1}^{n-1} A_{ik} X_k \right) \tag{2.21}$$

The use of extended interval arithmetic guarantees that the solutions to (2.17) contained in $\mathbf{X}$ are also contained in $\mathbf{N} = [N_0, N_1, ..., N_n]^{\text{T}}$ and thereby also in the intersection of the two boxes $\mathbf{X}' = \mathbf{X} \cap \mathbf{N}$. This motivates the following iterative algorithm for contracting the interval bounds of the solutions:

Algorithm 2.1 Interval Gauss-Seidel step [26]

Input: $\mathbf{X} \in \mathbb{IR}_*^n,\ \mathbf{A} \in \mathbb{IR}_*^{n\times n},\ \mathbf{B} \in \mathbb{IR}_*^n$

1: **for** $i = 0 \to n-1$ **do**
2: $\quad N_i \leftarrow \frac{1}{A_{ii}} \left(B_i - \sum_{j=0}^{i-1} A_{ij} X'_j - \sum_{k=i+1}^{n-1} A_{ik} X_k \right)$
3: $\quad X'_i \leftarrow X_i \cap N_i$
4: **end for**
5: **return** $\mathbf{X}'$

If $X'_i \subset X_i$ for any $i = 0, \ldots, n-1$, then Algorithm 2.1 results in a contraction of $\mathbf{X}$ around the solutions of (2.17). If $X'_i = \emptyset$ for any $i = 0, \ldots, n-1$, there is no solutions within the original bounds $\mathbf{X}$, and the algorithm can be terminated.

Note that in some cases, when zero is contained in the denominator but not in the numerator, the containment set of the right-hand side of (2.21) consists of two semi-infinite intervals, as exemplified in Section 2.1.4. If N_i is computed as the containment set enclosure of these intervals, then $N_i = [-\infty, \infty]$ which leads to $X'_i = X_i$, i.e. no progression. In stead, labeling the semi-infinite intervals N_i^- and N_i^+ and modifying the intersection step in Algorithm 2.1 to

$$X'_i \leftarrow (X_i \cap N_i^-) \cup (X_i \cap N_i^+) \tag{2.22}$$

a tighter bound is produced if either $(X_i \cap N_i^-)$ or $(X_i \cap N_i^+)$ is empty. This improvement comes at the price of a very small computational overhead.

The new bound resulting from each loop of Algorithm 2.1 is used in the succeeding iterations, where the tightness of N_i is partially determined by the results of the preceding loops. Therefore it may be advantageous to run the iterations in a different ordering than shown in Algorithm 2.1 [11].

In order for the method to be efficient, i.e. produce reasonably tight bounds, some form of preconditioner can be applied. The matrix $\mathbf{A}$ and the vector $\mathbf{B}$ are replaced by preconditioned counterparts, $\hat{\mathbf{A}} = \mathbf{QA}$ and $\hat{\mathbf{B}} = \mathbf{QB}$ where the $\mathbf{Q} \in \mathbb{R}^{n\times n}$ is a real-valued preconditioning matrix. A simple choice of $\mathbf{Q}$ (e.g. used in [11]) is the *inverse-midpoint preconditioner*, presented in [43]. As the name hints, this preconditioner is computed as the inverse of the matrix generated by taking the midpoint of each element in $\mathbf{A}$, i.e. $\mathbf{Q}_{\text{inv.mid.}} = (\text{mid}(\mathbf{A}))^{-1}$.

In [29] the inverse-midpoint preconditioner was shown to be sub-optimal. An alternative choice preconditioner is described in Section 3.3.

2.3 Summary

In this chapter the fundamentals of finite and extended interval arithmetic were presented, along with some considerations on practical computational interval arithmetic and notes on the availablity of software libraries for interval arithmetic. Further, it was described how systems of linear interval equations can be solved using the interval Gauss-Seidel method.

Chapter 3

Global Optimization using Interval Analysis

This chapter describes the application of interval analysis for global optimization. The purpose is to provide the theoretical background required for understanding the algorithms used.

Section 3.1 describes the Interval Newton (IN) method, an interval version of the classical Newton method for finding zeros of a function. Further the Interval Newton/Generalized Bisection (IN/GB) method, an extension of the IN method, capable of enclosing all roots of a continuously differentiable function within arbitrarily narrow bounds [11, 16], is described.

Combined with methods for bounding the range of the objective function (see [15, 44]), the IN/GB method can be used to construct algorithms for global optimization [44, 17, 18, 11]. In Section 3.2 a method for global optimization is described, where the IN/GB method is combined with a set of tests designed to qualify or disqualify areas of the search space as containing possible solutions.

Several improvements (see e.g. [11, 28, 29]) can be made to accelerate the IN/GB method and the global optimization method of Section 3.2, a few of which are described in Section 3.3. In Section 3.4 the steps of the global optimization method of Section 3.2 are analyzed with focus on the parameter estimation problem, and finally in Section 3.5 a short summary of the chapter is given.

3.1 The Interval Newton Method

This section describes an interval version of the Newton method – a classical method for local optimization [45]. To show the concepts of how the method works, the univariate version of the Interval Newton (IN) method is described in detail in Section 3.1.1. The result of the extension of the method to its multivariate version is given in Section 3.1.2. Finally, some measures to accelerate the convergence of the method are described. This section is primarily based on [11].

3.1.1 Univariate Version

Given a real continuous function $f(x) : \mathbb{R} \to \mathbb{R}$ and real points a and b, $a < b$ in the domain $\mathcal{D}_f$ of f, such that f is continuously differentiable in (a, b), the mean value

theorem states that there exists a point $c \in [a, b]$ such that [42, 16]:

$$f'(c) = \frac{f(b) - f(a)}{b - a} \tag{3.1}$$

where $f'(\gamma) = \frac{\mathrm{d}f(x)}{\mathrm{d}x}\Big|_{x=\gamma}$. When $f(b) = 0$, (3.1) can be reordered to

$$b = a - \frac{f(a)}{f'(c)} \tag{3.2}$$

Now consider the case where $f'(c)$ in (3.2) is replaced by its interval extension $F'(A)$ over an interval A where $a, b, c \in A$:

$$N = a - \frac{f(a)}{F'(A)} \tag{3.3}$$

Inclusion isotonicity (Definition 2.2) guarantees that $f'(c) \in F'(A)$. Now define the function

$$N_f(x, X) = x - \frac{f(x)}{F'(X)} \tag{3.4}$$

Theorem 3.1 and Corollary 3.1 follow:

Theorem 3.1 [11] The set of all $x \in X$ where $f(x) = 0$, denoted $\mathcal{S}_f(X)$, is contained in $N_f(x, X)$, i.e:

$$\begin{aligned} &\mathcal{S}_f(X) \subseteq X \quad \Rightarrow \quad \mathcal{S}_f(X) \subseteq N_f(x, X) \; \forall \; x \in X. \\ &\text{where} \quad \mathcal{S}_f(X) = \{x \in X \mid f(x) = 0\} \end{aligned}$$

♢

Proof 3.2 Follows from Theorem 2.2. □

Corollary 3.1 The set $\mathcal{S}_f(X)$ is contained in the intersection $X' = X \cap N_f(x, X)$ for all $x \in X$, i.e:

$$\mathcal{S}_f(X) \subseteq X \quad \Rightarrow \quad \mathcal{S}_f(X) \subseteq X \cap N_f(x, X) \;\; \forall \;\; x \in X.$$

♢

In its original formulation[1], the IN method is designed for use with finite interval arithmetic [16]. Therefore it is assumed that $0 \notin F'(X)$ as division by intervals containing zero is undefined. Because f is monotonous in X then X contains at most one zero of f. This method can be shown to converge to this zero under certain conditions, as stated in Theorem 3.2. The formulation of the method given in Algorithm 3.1 stems from [16].

Algorithm 3.1 Interval Newton Method (univariate) [16]

Input: $X_0 \in \mathbb{IR}_*, \;\; f, \;\; F'$

1: **for** $i = 0 \rightarrow n - 1$ **do**
2: | Pick a real point $x_i \in X_i$
3: | $X_{i+1} \leftarrow X_i \cap N_f(x_i, X_i)$
4: **end for**
5: **return** X_{n-1}

[1] Presented by R. E. Moore in "Interval Analysis", *Prentice-Hall*, 1966, which is unfortunately not available to the authors

Theorem 3.2 [16, 11] If $0 \notin F'(X_k)$ and x_i is chosen such that $x_i \in \text{int}(X_i)$ and $f(x_i) \neq 0$ for all iterations $i \geq k$ then Algorithm 3.1 converges to a zero of f if one exists.

♢

Proof 3.2[16, 11] If $0 \notin F'(X_i)$ then $0 \notin \text{int}\left(\frac{f(x_i)}{F'(X_i)}\right)$ and thus $x_i \notin \text{int}\left(N_f(x_i, X_i)\right)$. This means that $N_f(x_i, X_i)$ is always "on one side" of x_i and thereby $X_{i+1} = X_i \cap N_f(x_i, X_i)$ covers only the part of X_i on that side of x_i – see Figure 3.1. Thus, $\text{wid}(X_{i+1}) \leq \max(x_i - \underline{X}_i\,,\ \overline{X}_i - x_i) < \text{wid}(X_i)$ and X_i converges a zero if one exists for $i \to \infty$.

□

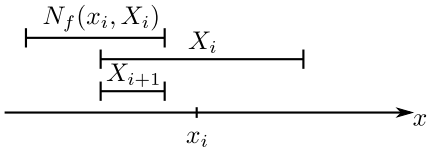


Figure 3.1: Illustration supporting the proof of Theorem 3.2.

In practice, $f(x)$ in (3.4) is evaluated using outward rounded interval arithmetic, i.e. by evaluating $F([x,\, x])$ and may therefore return a non-degenerate interval [17]. If $0 \in F([x_i,\, x_i])$ the assumption that $0 \notin F'(X_i) \Rightarrow 0 \notin \text{int}\left(\frac{f(x_i)}{F'(X_i)}\right)$ does not hold, but when $0 \notin F([x_i,\, x_i])$ the proof is still valid [11]. From the proof of Theorem 3.2 it can be deduced that $x_i = \text{mid}(X_i)$ is a sensible choice when $0 \notin F'(X_i)$, because then $\text{wid}(X_{i+1}) \leq \frac{1}{2}\text{wid}(X_i)$ [16, 11].

To develop an algorithm that converges even when $0 \in F'(X)$ it is useful to look at the possible outcomes of (3.4). The value of $N_f(x, X)$ can be categorized in three different cases (illustrated in Figure 3.2):

1. $X \cap N_f(x, X) = \emptyset$.
2. $N_f(x, X) \subseteq X$.
3. $X \cap N_f(x, X) \neq \emptyset$ and $N_f(x, X) \setminus X \neq \emptyset$.

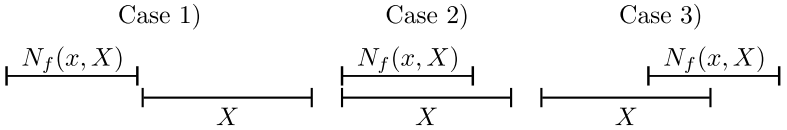


Figure 3.2: Categorization of values of $N_f(x, X)$ in (3.4). Case 1) $X \cap N_f(x, X) = \emptyset$. Case 2) $N_f(x, X) \subseteq X$. Case 3) $X \cap N_f(x, X) \neq \emptyset$ and $N_f(x, X) \setminus X \neq \emptyset$.

For each case, useful conclusions can be drawn regarding the existence of zeros of f in X. These conclusions are presented in Theorems 3.3-3.5.

Theorem 3.3 (Case 1) [11] If $X \cap N_f(x, X) = \emptyset$, then X contains no zeros of f, i.e:

$$X \cap N_f(x, X) = \emptyset \quad \Rightarrow \quad \mathcal{S}_f(X) = \emptyset$$

♢

Proof 3.3 By Corollary 3.1 the set $\mathcal{S}_f(X)$ of zeros of f in X is also contained in $X \cap N_f(x, X)$. Therefore, if $X \cap N_f(x, X)$ is empty, then $\mathcal{S}_f(X)$ is also empty.

□

In practice, when outward rounding is used, the proof still holds as $N_f(x, X)$ is only widened by the rounding [11].

Theorem 3.4 (Case 2) [11] If $N_f(x, X) \subseteq X$, then there exists a single simple zero (multiplicity 1) of f in X .

♢

For proof of Theorem 3.4, refer to [11, Theorem 9.6.9].

Theorem 3.5 (Case 3) [11] If $X \cap N_f(x, X) \neq \emptyset$ and $N_f(x, X) \backslash X \neq \emptyset$, X may contain an arbitrary number of zeros of f.

♢

Proof 3.5 (By example) If $0 \in F'(X)$ and $f(\hat{x}) \neq 0$ for some $\hat{x} \in X$, then $N_f(\hat{x}, X) = [-\infty, \infty]$, and case 3 applies. There may still be any number of points $x \in X, x \neq \hat{x}$ where $f(x) = 0$.

□

When $0 \in F'(X)$ and extended interval arithmetic is used, (3.4) yields infinite or semi-infinite intervals, and thus Case 3 applies. X may be completely or almost completely contained in $N_f(x, X)$, and so the iteration in Algorithm 3.1 does not contract the search region and gives no improvement. To resolve this problem, a method for use with extended interval arithmetic is used [16]. In this method, when there is no or little improvement, X is split into two or more separate intervals and the algorithm continues to work on each of these new intervals separately. This method is called the *Interval Newton/Generalized Bisection*[2] (IN/GB) method. Splitting of intervals is treated in more detail, for the more general multivariate case, in Section 3.1.2. Due to the nature of interval arithmetic, this algorithm does not yield exact solutions. Instead solutions are defined as intervals $\hat{X}$ that narrowly enclose a zero such that $0 \in F(\hat{X})$ and where $\text{wid}(\hat{X}) \leq \epsilon_w$ and $\text{wid}(F(\hat{X})) \leq \epsilon_f$ for predefined values of ϵ_w and ϵ_f.

The main loop of the algorithm consists of four primary steps:

1. **Function bound test:** A test of whether the function may contain a zero in the current interval.
2. **Solution qualification:** A test of whether the current interval qualifies as a solution.
3. **Interval Newton test:** A test of the outcome of the IN method.
4. **Split, discard or keep interval:** Based on the outcome of the Interval Newton test, a decision on whether to discard the interval, split it or keep it as is for further processing.

A work queue $\mathcal{W}$ used to store intervals to be processed, and solutions are stored in the list $\mathcal{L}$. The algorithm is listed in Algorithm 3.2.

[2] *Bisection* is the act of dividing into two parts. The term *Generalized Bisection* is used because the interval is not necessarily divided in just two.

Algorithm 3.2 Interval Newton/Generalized Bisection (IN/GB) [11, 16]

Input: $X_0 \in \mathbb{IR}_*,\ F,\ F'$

1: $\mathcal{L} \leftarrow \{\}$ *List of solutions*
2: $\mathcal{W} \leftarrow \{X_0\}$ *Work queue*
3: **while** $\mathcal{W} \neq \{\}$ **do**
4: Take an interval X from $\mathcal{W}$
5: **if** $0 \in F(X)$ **then**
6: **if** $\text{wid}(X) \leq \epsilon_w$ **and** $\text{wid}(F(X)) \leq \epsilon_f$ **then** *X is a solution; store it.*
7: Append X to $\mathcal{L}$
8: **else** *Perform Newton step*
9: $N \leftarrow N_f(x, X)$ for some $x \in \text{int}(X)$
10: $X' \leftarrow X \cap N$
11: **if** $X' = X$ **then** *No progression; split X*
12: $X_-,\ X_+ \leftarrow \text{split}(X)$
13: Append X_- and X_+ to $\mathcal{W}$.
14: **else if** $X' \subset X$ **then** *Progression; keep X′*
15: Append X' to $\mathcal{W}$.
16: **else if** $X' = \emptyset$ **then** *No solutions in X*
17: (X contains no zeros, discard it.)
18: **end if**
19: **end if**
20: **end if**
21: **end while**
22: **return** $\mathcal{L}$

Notes: The $\text{split}(X)$ splits the interval X at some point $x \in \text{int}(X)$ and returns the two parts. Lines 16-17 do nothing. They are included to clarify what happens when $X' = \emptyset$.

This type of algorithm can be proven to bound all discrete zeros in $\mathcal{S}_f$ to arbitrary accuracy (see e.g. [16] and [11, Theorem 9.6.3]).

3.1.2 Multivariate Version

In its multivariate version where $\mathbf{f}(\mathbf{x}) : \mathbb{R}^{n\times 1} \to \mathbb{R}^{n\times 1}$ the mean value theorem is expressed using the Jacobian $\mathbf{j} \in \mathbb{R}^{n\times n}$ of $\mathbf{f}$ at a point $\mathbf{c} \in \mathbb{R}^{n\times 1}$ on a straight line between $\mathbf{a} \in \mathbb{R}^{n\times 1}$ and $\mathbf{b} \in \mathbb{R}^{n\times 1}$ [42, 44]:

$$\mathbf{f}(\mathbf{b}) = \mathbf{f}(\mathbf{a}) + \mathbf{j}(\mathbf{c})(\mathbf{b} - \mathbf{a}) \tag{3.5}$$

As for the univariate version, let $\mathbf{f}(\mathbf{b}) = \mathbf{0}$ and replace $\mathbf{j}(\mathbf{c})$ with an interval expansion of the Jacobian $\mathbf{J}(\mathbf{A}) \in \mathbb{IR}_*^{n\times n}$ over the interval box $\mathbf{A} \in \mathbb{IR}_*^{n\times 1}$ such that $\mathbf{a}, \mathbf{b}, \mathbf{c}, \in \mathbf{A}$. $\mathbf{b}$ is replaced with an interval box $\mathbf{N} \in \mathbb{IR}_*^{n\times 1}$ that encloses the values of $\mathbf{b}$ where $\mathbf{f}(\mathbf{b}) = 0$:

$$\mathbf{J}(\mathbf{A})(\mathbf{N} - \mathbf{a}) = -\mathbf{f}(\mathbf{a}) \tag{3.6}$$

The system in (3.6) can be solved for $\mathbf{N}$ using the methods discussed in Section 2.2. The solution for a given $\mathbf{f}$, $\mathbf{x}$ and $\mathbf{X}$ is denoted $\mathbf{N_f}(\mathbf{x}, \mathbf{X})$. Just as with (3.4), the outcome of solving (3.6) falls into one of three categories, as illustrated in Figure 3.3.

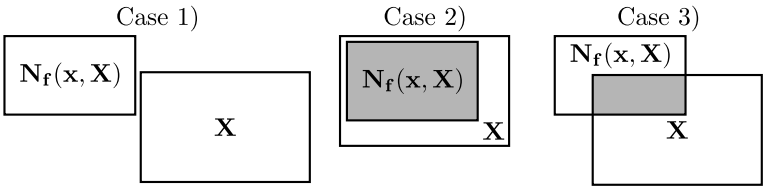


Figure 3.3: Categorization of values of $\mathbf{N_f}(x, X)$ in (3.6), 2-dimensional example. Case 1) $\mathbf{X} \cap \mathbf{N_f}(\mathbf{x}, \mathbf{X}) = \emptyset$. Case 2) $\mathbf{N_f}(\mathbf{x}, \mathbf{X}) \subseteq \mathbf{X}$. Case 3) $\mathbf{X} \cap \mathbf{N_f}(\mathbf{x}, \mathbf{X}) \neq \emptyset$ and $\mathbf{N_f}(\mathbf{x}, X) \setminus \mathbf{X} \neq \emptyset$. The grayed areas show the intersection $\mathbf{X} \cap \mathbf{N_f}(\mathbf{x}, \mathbf{X})$.

Algorithm 3.2 is trivially extendable to the multivariate case. Some considerations and details that apply both to the uni- and multivariate case are:

1. Splitting boxes
    - In Algorithm 3.2 the interval/box is only split when there is no progression in an iteration. The algorithm can be made more aggressive, in the sense that it splits boxes more often, by splitting whenever the progression is below some threshold. The amount of progression can for example be defined as the relative change on width or the volume of the box (the latter is only applicable in the multivariate version) [11, 46].
    - A box can be split along any number of any of its axes. The number of splits and choice of axes affects the convergence of the algorithm. On one hand the IN method is more efficient for smaller boxes, mandating splits into many subboxes [11]. On the other hand too many splits result in a very large work queue containing many split boxes, some of which may have been discarded using fewer iterations had they not been split.
    - A simple approach is to equally split along the axis where the box is widest. Due to variable scaling, this is not always the most efficient approach. One way of weighting the dependence of $\mathbf{f}$ on each dimension of a box $\mathbf{X}$ is using the weights $\delta_j = \text{wid}(X_j) \sum_{i=0}^{n-1} |J_{i,j}(\mathbf{X})|$ [11, eq. 11.8.1]. $\mathbf{X}$ is then split along the axis j with highest δ_j.
    - A straight forward choice is to split the box in half. Sometimes the Interval Newton process produces a gap in the box along an axis. When present, boxes can be split along these gaps to produce narrower subboxes. However, if the gap is narrow and close to the edge of the box, it may be more efficient to ignore it [11].
    - For more details and discussion on this topic, see e.g. [11, Sec. 11.8] and [34].
2. Empty interval boxes
    - A box is considered empty if it is empty in any of its dimensions.
    - An empty box can be considered having a volume of zero.
    - A box of volume zero is not empty, but represents a $(n-k)$-dimensional plane where k is the number of elements in the box of width zero. As a consequence of the above, a box of width zero represents a point.
3. Stopping criteria
    - When implemented using finite precision, it may be impossible to fulfill the stopping criteria for some choices of F, ϵ_w and ϵ_f. This must be handled by the developer.

- For a more thorough discussion on stopping criteria, see e.g. [16] and [11, Sec. 11.5].

3.2 Global Optimization using the Interval Newton Method

In this project it has been chosen to only work with model functions f that are twice continuously differentiable. The bounds on the objective function must include the global optima, i.e. the global optima must not be located on the edge of the search region. This is chosen to simplify the implementation and are by the authors considered reasonable limitations.

Further the method described is an unconstrained method. The method is however in practice constrained by the bounds on the parameters.

Due to this, a couple of observations can be done. Let $\hat{\mathbf{x}}$ denote any global minimum in the interior of the search space $\mathbf{X}_0$, then the following conditions apply [18, 11]:

1. Any global optimum is located at a stationary points of f, i.e. the gradient of f satisfies $\mathbf{g}_f(\hat{\mathbf{x}}) = \mathbf{0}$.

2. In a region around any global optimum, f is locally convex, i.e. the Hessian of f, $\mathbf{h}_f(\hat{\mathbf{x}})$ is positive semidefinite.

Note that these are necessary but not sufficient conditions for a global optimum (a sufficient condition is that $f(\hat{\mathbf{x}}) \leq f(\mathbf{x}) \; \forall \; \mathbf{x} \in \mathbf{X}_0$).

As described in Section 3.1, the IN/GB method can be used to enclose the zeros of a continuously differentiable vector function. Because f is twice continuously differentiable, the gradient $\mathbf{g}_f$ is continuously differentiable, and thus the zeros of $\mathbf{g}_f$, can be narrowly enclosed using the IN/GB method. By evaluating the bounds of the objective function over these intervals and comparing them, the global optima are found [17, 18].

Not all of the stationary points are local minima and thereby potential global minima; some are saddle points or local maxima. To avoid spending unnecessary time on narrowly enclosing these points, the second condition listed above can be applied. If the interval extension of the Hessian over the box being processed does not contain any positive semidefinite matrices, the box cannot contain a locally convex region and thereby cannot contain a global minimum. Thus, the box can be discarded from the search.

Performing a complete verification that the Hessian is positive semidefinite, e.g. by checking that all its eigenvalues are non-negative, is computationally expensive ($\mathcal{O}(n^3)$). A computationally much cheaper approach ($\mathcal{O}(n)$) is to check for a necessary but not sufficient condition: If any of the diagonal elements of $(\mathbf{H}_f)_{i,i}(\mathbf{X}) < 0$ for $i = 0, 1, \ldots, n-1$ then $\mathbf{H}_f$ evaluated over $\mathbf{X}$ is not positive semidefinite and therefore cannot contain a local minimum [11].

By combining the considerations above, an algorithm in six main steps is constructed:

1. **Bounds test:** A test to check that f may take on a value in $\mathbf{X}$ that qualifies as a global optimum.

2. **Monotonicity test:** A test to check that f may have a stationary point in $\mathbf{X}$.

3. **Convexity test:** A test on the Hessian $(\mathbf{H}_f)_{i,i}(\mathbf{X})$ of f over the current box to check that f might be convex in $\mathbf{X}$.

4. **Solution test:** A test of whether the current interval qualifies as a solution.

5. **Interval Newton test:** A test of the outcome of the IN method.

6. **Split, discard or keep interval:** Based on the outcome of the Interval Newton test, a decision on whether to discard the interval, split it or keep it as is for further processing.

The resulting algorithm is listed in Algorithm 3.3.

Algorithm 3.3 Global optimization using (IN/GB) [18, 46]

Input: $\mathbf{X}_0 \in \mathbb{IR}^n_*,\ F,\ \mathbf{G}_f,\ \mathbf{H}_f,\ \rho_\tau \in (0,1),\ \epsilon_f \in \mathbb{R}_+,\ \epsilon_x \in \mathbb{R}_+$

$\mathcal{L} \leftarrow \{\}$ *List of solutions*
$\mathcal{W} \leftarrow \{\mathbf{X}_0\}$ *Work queue*
$b \leftarrow \infty$ *Best known upper bound*
while $\mathcal{W} \neq \{\}$ **do**
  Take an interval box $\mathbf{X}$ from $\mathcal{W}$
  $[f_\ell,\, f_u] \leftarrow F(\mathbf{X})$
  if $f_\ell \leq b$ **then** *Bounds test*
    $b \leftarrow \min(b, f_u)$ *Update best upper bound*
    if $\mathbf{0} \in \mathbf{G}_f(\mathbf{X})$ **and** $\mathbf{0} \leq \mathrm{diag}(\mathbf{H}_f(\mathbf{X}))$ **then** *Monotonicity+convexity test*
      if $\mathrm{wid}(\mathbf{X}) \leq \epsilon_x$ **and** $\mathrm{wid}(F(\mathbf{X})) \leq \epsilon_f$ **then** *Solution test*
        Append $\mathbf{X}$ to $\mathcal{L}$ *Store possible solution*
      else
        $\mathbf{x} \leftarrow \mathrm{mid}(\mathbf{X})$ *midpoint of* $\mathbf{X}$
        $\tilde{\mathbf{X}} \leftarrow \mathrm{Newton}(\mathbf{x}, \mathbf{X})$ *Interval Newton test*
        $\rho \leftarrow {}^{\mathrm{vol}(\tilde{\mathbf{X}})}/_{\mathrm{vol}(\mathbf{X})}$ *Improvement from Newton*
        if $0 < \rho \leq \rho_\tau$ **then** *Sufficient improvement*
          Append $\tilde{\mathbf{X}}$ to $\mathcal{W}$
        else if $\rho_\tau < \rho$ **then** *Insufficient improvement*
          $(\tilde{\mathbf{X}}_\mathrm{a}, \tilde{\mathbf{X}}_\mathrm{b}) \leftarrow \mathrm{split}(\tilde{\mathbf{X}})$
          Append $\tilde{\mathbf{X}}_\mathrm{a}$ and $\tilde{\mathbf{X}}_\mathrm{b}$ to $\mathcal{W}$
        end if
      end if
    end if
  end if
end while
$\tilde{\mathcal{L}} \leftarrow \{\hat{\mathbf{X}} \mid \hat{\mathbf{X}} \in \mathcal{L} \ \wedge\ \underline{F(\hat{\mathbf{X}})} \leq b\}$ *Filter list of solutions*
return $\tilde{\mathcal{L}}$

Notes: Note that the interval box is discarded when the path through the loop does not include a step where $\mathbf{X}$ (or a contracted or split version thereof) is appended to either $\mathcal{W}$ or $\mathcal{L}$.
The $\mathrm{Newton}(\mathbf{x}, \mathbf{X})$ function denotes a Interval Newton step, performed by solving the system $\mathbf{H}_f(\mathbf{X})(\mathbf{N}(\mathbf{x}, \mathbf{X}) - \mathbf{x}) = -\mathbf{G}_f(\mathbf{x})$ using the Interval Gauss-Seidel method, described in Section 2.2.1.

Most of the considerations regarding Algorithm 3.2 listed in Section 3.1 on page 18 also apply to Algorithm 3.3, with a few changes and additions:

1. Splitting boxes
   - The metric used to measure progression is the relative change in the volume of the box, $\rho = \frac{\text{vol}(\tilde{\mathbf{X}})}{\text{vol}(\mathbf{X})} \in [0, 1]$. A large progression results in a low value of ρ and small progression results in a value of ρ close to 1.
   - The dimension along which to split, are computed as $\delta \leftarrow \underset{i=0,1,\ldots,\Psi-1}{\text{argmax}} (\text{wid}(G_i) \cdot \text{wid}(P_i))$, from [11, eq. 12.13.1].
2. Selecting a box from $\mathcal{W}$ to process
   - The order in which boxes are processed affects the convergence of the algorithm, as good choices make the best known upper bound b decrease faster, thus making it possible to discard larger areas of the search space.
   - Many use a *best-first* approach, where the box in $\mathcal{W}$ where the lower bound of the objective function $\underline{F(\mathbf{X})}$ is lowest is chosen for processing [32, 34, 33]. Using this approach, processing of promising boxes is prioritized in the hope that the algorithm will converge faster.
   - Another approach, e.g. used in [46, 30], is a *depth-first* principle, where the latest interval added to the list is processed first. The result is that some boxes are quickly contracted and become small with correspondingly narrow bounds on the objective function. The hope is that these narrow bounds can quickly yield a low value of b. However, in cases where a branch of the search tree which does not lead to an improved value of b is chosen, this approach leads to unnecessary work being done. Therefore, the depth-first approach is only efficient when low value of b is known in advance, which is often not the case [32, 34].
   - A third approach, used in [18], is the *oldest-first* approach, where the box that has been in the work queue the longest is chosen for processing. This approach promotes search over the whole search tree, thereby making sure that no branches are left unexamined for longer periods.
   - The method used here, which empirically has shown good results, is a mixture of the depth-first and oldest-first approach where the interval to be processed are alternately chosen as the oldest and newest interval in the work queue in order to promote both global and local search.
   - For more thorough discussion on this topic, see e.g. [34] and [11].
3. The list of possible solutions
   - When the algorithm runs, boxes are appended to the list of solutions $\mathcal{L}$ when the conditions in lines 7 and 10 of Algorithm 3.3 are fulfilled. During the execution of the algorithm, the value of b changes and some boxes in $\mathcal{L}$ may thereby be invalidated. Therefore the list must be filtered as it is done in line 26 of Algorithm 3.3.
4. Other algorithms
   - Algorithm 3.3 is a fairly simple algorithm for global optimization using interval analysis, mainly inspired by the algorithm used in [46], but also using elements from [11]. More advanced algorithms can be constructed by using several contraction methods in stead of just the IN method and by applying heuristics to switch between them. An example of such an algorithm can be found in [11].

3.3 Accelerating the Convergence

This section describes a set of steps that can be taken to accelerate the convergence of Algorithm 3.3. Several other methods for accelerating the convergence exist that are not discussed here, as the scope of this project is limited (see e.g. [11, 18, 41, 47]). The methods described here are selected because they are fairly easy to implement and are reported to provide good speedups.

3.3.1 Preconditioning in the IN/GB Method

As mentioned in Section 2.2, the inverse-midpoint preconditioner has been proven to be sub-optimal in some cases. A linear programming (LP) approach for computing a preconditioner is presented in [29]. The preconditioner is concluded to be optimal in terms of the tightness of bounds, but in terms of CPU time it is not always more efficient than the inverse-midpoint preconditioner. In the concluding remarks in [29] it is suggested to investigate the use of a sparse preconditioner in order to simplify the LP problem. This suggestion is picked up in [28], where a hybrid preconditioning scheme that mixes a sparse or "pivoting" preconditioning approach with the inverse-midpoint approach. This method is described in this section.

The pivoting preconditioning approach, presented in [28], uses a sparse preconditioner $\mathbf{q}$. One element in each row is assigned a value of one and all other elements are assigned a value of zero. The non-zero element is called the pivot element and is labeled q_{ij} where i is the row number and j is the position of the pivot element in the row. For row i, the position of the pivot element j is selected such that, when the result of the interval Gauss-Seidel step (see Section 2.2.1 on page 10) is computed using the pivoting preconditioner, element i is either empty or has minimal width. The row q_i is then said to be discarding-optimal if N_i is empty, indicating that there is no solution in the current interval box, or contraction-optimal if N_i is non-empty and of smallest width possible for any position of the pivoting element in the q_i.

Using this scheme, $\mathbf{q}$ need not be formed explicitly, but is applied implicitly while at the same time performing an interval Gauss-Seidel step. As element i of the resulting box depends only on row i of $\mathbf{q}$, the rows of $\mathbf{q}$ can be constructed independently. The i^{th} row of $\mathbf{q}$ with the j^{th} element as the pivot element is labeled $(\mathbf{q}_i)_j$ and the result of the interval Gauss-Seidel step using this row is albeled $(N_i)_j$. The algorithm for performing an interval Gauss-Seidel step while computing and applying the pivoting preconditioner is shown in Algorithm 3.4.

Algorithm 3.4 Interval Gauss-Seidel step using pivoting preconditioning [28]

Input: $\mathbf{A} \in \mathbb{IR}_*^{n\times n}, \mathbf{X} \in \mathbb{IR}_*^{n\times 1}, \mathbf{B} \in \mathbb{IR}_*^{n\times 1}$

$w_{\text{opt}} = \infty$
for $i = 0, 1, ..., n-1$ **do**
  for $j \in 0, 1, ..., n-1$ **do**
    Precondition the j^{th} column of $\mathbf{A}$ with $(\mathbf{q}_i)_j$.
    Compute $(N_i)_j$ using the interval Gauss-Seidel method (Algorithm 2.1).
    $(X_i')_j \leftarrow (N_i)_j \cap X_i$.
    if $(X_i')_j = \emptyset$ **then**
      Return "no solutions"
    else if $\text{wid}((X_i')_j) < w_{\text{opt}}$ **then**
      $\hat{X}_i = (X_i')_j$
    end if
  end for
end for
return $\hat{\mathbf{X}}$

Compared to the inverse midpoint preconditioning approach, the pivoting preconditioner provides a relatively computationally cheap way of either discarding or contracting the box being processed. In some cases, however, the inverse midpoint approach yields a narrower result. Therefore it has been proposed by Gau and Stadtherr to combine the results of both approaches [28]. First, the Gauss-Seidel step is performed using the pivoting preconditioner in the hope that the current box can be discarded. If not, then the Gauss-Seidel step is performed using the inverse midpoint preconditioner and the two results are intersected to produce a final result.

3.3.2 Parallelization of the Interval Newton Step

Two methods for performing the Newton step in Algorithm 3.3 have been described, namely the interval Gauss-Seidel method using inverse midpoint preconditioning in Section 2.2 and the Gauss-Seidel method using pivoting preconditioning in Section 3.3.1. This section describes how the computations performed in each of these methods can be parallelized.

The interval Gauss-Seidel method using inverse midpoint preconditioner consists of two main steps; 1) a preconditioning step and 2) an interval Gauss-Seidel step. Here, the focus is on the second. The interval Gauss-Seidel step from Section 2.2 is repeated below in Algorithm 3.5 for convenience. In the form shown here it runs through the loop sequentially n times, each iteration dependent on the result from the last, and it can as such therefore not be parallelized on the outer iteration level. Inside the loop in the two summations, $n-1$ independent multiplications are performed and the results summed. If n is large, parallelizing these summations can yield a speedup, but when n is small it is assessed that gain is insignificant.

Algorithm 3.5 Interval Gauss-Seidel step [26]

Input: $\mathbf{X} \in \mathbb{IR}_*^n,\ \mathbf{A} \in \mathbb{IR}_*^{n\times n},\ \mathbf{B} \in \mathbb{IR}_*^n$

1: **for** $i = 0 \to n-1$ **do**
2: $\quad N_i \leftarrow \frac{1}{A_{ii}} \left(B_i - \sum_{j=0}^{i-1} A_{ij} X'_j - \sum_{k=i+1}^{n-1} A_{ik} X_k\right)$
3: $\quad X'_i \leftarrow X_i \cap N_i$
4: **end for**
5: **return** $\mathbf{X}'$

Another possible approach for parallelization is to alter the method in Algorithm 3.5 to make the iterations of the loop independent as shown in Algorithm 3.6. The iterations of the loop can then be run in parallel, reducing the computation time to $t_{\text{par}} = \frac{t_{\text{seq}}}{n}$ where t_{seq} is the execution time using the form with dependent iterations. The cost of letting the bounds be independent is that the results for previous iterations are not used to obtain better bounds. To remedy this, the parallel method can be run several times in sequence and possibly obtain a better result than Algorithm 3.5 using less or equal time but more computational resources. Due to time limitations, this experiment is left as a topic of further research.

Algorithm 3.6 Interval Gauss-Seidel step with independent iterations

Input: $\mathbf{X} \in \mathbb{IR}_*^n,\ \mathbf{A} \in \mathbb{IR}_*^{n\times n},\ \mathbf{B} \in \mathbb{IR}_*^n$

1: **for** $i = 0 \to n-1$ **do**
2: $\quad N_i \leftarrow \frac{1}{A_{ii}} \left(B_i - \sum_{j=0, j\neq i}^{n-1} A_{ij} X_j\right)$
3: $\quad X'_i \leftarrow X_i \cap N_i$
4: **end for**
5: **return** $\mathbf{X}'$

With the pivoting preconditioning approach in Algorithm 3.4, the preconditioning and the Gauss-Seidel step are performed simultaneously. For each element of the resulting vector $\mathbf{N} \in \mathbb{IR}_*^{n\times 1}$, the effect of using up to n different pivot positions is computed. Each of these n^2 computations involve a iteration of the loop in Algorithm 3.6, and can be performed in parallel using n^2 threads. For each element in the resulting vector, n of the results of these computations are then compared to select which is used. This can be done in parallel using n threads.

3.3.3 Narrowing the Gradient and the Hessian

In [48] a method for replacing some interval quantities with the midpoint of the interval (degenerate intervals in practice) in a special manner when computing the gradient and the Hessian is presented. Replacing interval quantities with scalars naturally results in narrower elements of the gradient and Hessian, in turn resulting in narrower bounds resulting from the Interval Newton method. For details on this method, see e.g. [48, 16, 11]. As an example, for a 3-dimensional problem, the gradient and the Hessian can be computed as follows:

$$\mathbf{G}(\mathbf{x}, \mathbf{X}) = [G_1(X_1, X_2, X_3), G_1(x_1, X_2, X_3), G_1(x_1, x_2, X_3)]^{\mathrm{T}} \tag{3.7}$$

$$\mathbf{H}(\mathbf{x}, \mathbf{X}) = \begin{bmatrix} H_{1,1}(X_1,\, x_2\,,\, x_3\,) & H_{1,2}(X_1, X_2,\, x_3\,) & H_{1,3}(X_1, X_2, X_3) \\ H_{2,1}(X_1, X_2, X_3) & H_{2,2}(\, x_1\,, X_2, X_3) & H_{2,3}(\, x_1\,,\, x_2\,, X_3) \\ H_{3,1}(X_1,\, x_2\,,\, x_3\,) & H_{3,2}(X_1, X_2,\, x_3\,) & H_{3,3}(X_1, X_2, X_3) \end{bmatrix} \tag{3.8}$$

Additionally, according to [11], any negative part of the diagonal elements of $\mathbf{H}$ can be deleted without compromising the inclusion property. By utilizing these methods, the elements of the gradient and the Hessian can be significantly narrowed without use of additional computations, resulting narrower bounds, thereby speeding up the algorithm.

3.4 IN/GB for Parameter Estimation

In this section, this special case is analyzed in terms of parallelism and reusable computations when evaluating the objective function, gradient and Hessian in Algorithm 3.3.

Note that for the parameter estimation problem, the free variables (parameters) are denoted $\mathbf{p} \in \mathbb{R}^{\Psi\times 1}$, the matrix containing the measurement points as columns is denoted $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\Phi]^{\mathrm{T}} \in \mathbb{R}^{\Phi\times\Gamma}$ where $\mathbf{x}_i \in \mathbb{R}^{\Gamma}$ is the i^{th} row containing the i^{th} measurement point and the vector containing the measurements is denoted $\mathbf{y} \in \mathbb{R}^{\Phi}$, respectively. The interval versions of $\mathbf{p}$ and $\mathbf{x}$ are denoted $\mathbf{P} \in \mathbb{IR}_*^{\Psi\times 1}$ and $\mathbf{X} \in \mathbb{IR}_*^{\Gamma\times\Phi}$ respectively. This means that $\mathbf{p}$ and $\mathbf{P}$ take the places of $\mathbf{x}$ and $\mathbf{X}$ in Algorithm 3.3.

3.4.1 Evaluation of the Objective Function, Gradient and Hessian

In order to make clear the parallel nature of the computations required to evaluate the objective function, the gradient and the Hessian, they are derived here in matrix-vector form. The equations are derived in real form, but are trivially extendable to interval form by replacing the relevant scalars with intervals.

The objective function f is:

$$f(\mathbf{p},\mathbf{x},\mathbf{y}) = \sum_{i=0}^{\Phi-1} \left(y_i - m(\mathbf{p},\mathbf{x}_i)\right)^2 = \underline{\underline{\mathbf{e}(\mathbf{p},\mathbf{x},\mathbf{y})^{\mathrm{T}}\mathbf{e}(\mathbf{p},\mathbf{x},\mathbf{y})}} \tag{3.9}$$

where $m(\mathbf{p},\mathbf{x}_i) \in \mathbb{R}$ is the model function, $\mathbf{e}(\mathbf{p},\mathbf{x},\mathbf{y}) = \mathbf{y} - \mathbf{m}(\mathbf{p},\mathbf{x})$, $\mathbf{y} = [y_1, ..., y_\Phi]^{\mathrm{T}}$ and $\mathbf{m}(\mathbf{p},\mathbf{x}) = [m(\mathbf{p},\mathbf{x}_1), m(\mathbf{p},\mathbf{x}_2), ..., m(\mathbf{p},\mathbf{x}_\Phi)]^{\mathrm{T}}$.

The gradient of f is given as:

$$\mathbf{g}(\mathbf{p},\mathbf{x},\mathbf{y}) = [g_1(\mathbf{p},\mathbf{x},\mathbf{y}), g_2(\mathbf{p},\mathbf{x},\mathbf{y}), \ldots, g_n(\mathbf{p},\mathbf{x},\mathbf{y})]^{\mathrm{T}} \tag{3.10}$$

$$\begin{aligned} g_j(\mathbf{p},\mathbf{x},\mathbf{y}) &= \frac{\partial f}{\partial p_j}(\mathbf{p},\mathbf{x},\mathbf{y}) \\ &= \frac{\partial(\mathbf{y}^{\mathrm{T}}\mathbf{y} + \mathbf{m}(\mathbf{p},\mathbf{x})^{\mathrm{T}}\mathbf{m}(\mathbf{p},\mathbf{x}) - 2\mathbf{m}(\mathbf{p},\mathbf{x})^{\mathrm{T}}\mathbf{y})}{\partial p_j}(\mathbf{p},\mathbf{x},\mathbf{y}) \\ &= 2\left(\frac{\partial \mathbf{m}}{\partial p_j}(\mathbf{p},\mathbf{x})\right)^{\mathrm{T}}(\mathbf{m}(\mathbf{p},\mathbf{x}) - \mathbf{y}) = \underline{\underline{-2\mathbf{a}_j(\mathbf{p},\mathbf{x})^{\mathrm{T}}\mathbf{e}(\mathbf{p},\mathbf{x},\mathbf{y})}} \end{aligned} \tag{3.11}$$

where $\mathbf{a}_j(\mathbf{p},\mathbf{x}) = \left[\frac{\partial m}{\partial p_j}(\mathbf{p},\mathbf{x}_1), \frac{\partial m}{\partial p_j}(\mathbf{p},\mathbf{x}_2), ..., \frac{\partial m}{\partial p_j}(\mathbf{p},\mathbf{x}_\Phi)\right]^{\mathrm{T}}$ is a vector containing the j^{th} element of the gradient of $m(\mathbf{p},\mathbf{x})$, with respect to $\mathbf{p}$ evaluated at the points in $\mathbf{x}$.

The Hessian of f is given as:

$$\mathbf{h}(\mathbf{p},\mathbf{x},\mathbf{y}) = \begin{bmatrix} h_{1,1} & h_{1,2} & \cdots & h_{1,n} \\ h_{2,1} & h_{2,2} & \cdots & h_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n,1} & h_{n,2} & \cdots & h_{n,n} \end{bmatrix} \tag{3.12}$$

$$\begin{aligned} h_{j,k}(\mathbf{p},\mathbf{x},\mathbf{y}) &= \frac{\partial^2 f}{\partial p_j \partial p_k}(\mathbf{p},\mathbf{x},\mathbf{y}) = \frac{\partial g_k}{\partial p_j}(\mathbf{p},\mathbf{x},\mathbf{y}) \\ &= 2\left(\left(\frac{\partial \mathbf{e}}{\partial p_j}(\mathbf{p},\mathbf{x},\mathbf{y})\right)^{\mathrm{T}} \frac{\partial \mathbf{m}}{\partial p_k}(\mathbf{p},\mathbf{x}) - \left(\frac{\partial^2 \mathbf{m}}{\partial p_j \partial p_k}(\mathbf{p},\mathbf{x})\right)^{\mathrm{T}} \mathbf{e}(\mathbf{p},\mathbf{x},\mathbf{y})\right) \\ &= 2\left(\left(\frac{\partial \mathbf{m}}{\partial p_j}(\mathbf{p},\mathbf{x})\right)^{\mathrm{T}} \frac{\partial \mathbf{m}}{\partial p_k}(\mathbf{p},\mathbf{x}) - \left(\frac{\partial^2 \mathbf{m}}{\partial p_j \partial p_k}(\mathbf{p},\mathbf{x})\right)^{\mathrm{T}} \mathbf{e}(\mathbf{p},\mathbf{x},\mathbf{y})\right) \\ &= \underline{\underline{2\left(\mathbf{a}_j(\mathbf{p},\mathbf{x})^{\mathrm{T}}\mathbf{a}_k(\mathbf{p},\mathbf{x}) - \mathbf{b}_{j,k}(\mathbf{p},\mathbf{x})^{\mathrm{T}}\mathbf{e}(\mathbf{p},\mathbf{x},\mathbf{y})\right)}} \end{aligned} \tag{3.13}$$

where $\mathbf{b}_{j,k}(\mathbf{p},\mathbf{x}) = \left[\frac{\partial^2 m}{\partial p_j \partial p_k}(\mathbf{p},\mathbf{x}_1), \frac{\partial^2 m}{\partial p_j \partial p_k}(\mathbf{p},\mathbf{x}_2), \ldots, \frac{\partial^2 m}{\partial p_j \partial p_k}(\mathbf{p},\mathbf{x}_\Phi)\right]^{\mathrm{T}}$ is a vector containing the $(j,k)^{\text{th}}$ element of the Hessian of $m(\mathbf{p},\mathbf{x})$, with respect to $\mathbf{p}$ evaluated at the points in $\mathbf{x}$.

It is clear from (3.9)-(3.13) that the computations required to evaluate the functions can be performed in parallel. The operations required are evaluations of the model function, model gradient, model Hessian and vector dot products of length Φ.

As Algorithm 3.3 requires the evaluation of the objective function in (3.9) before evaluating the gradient, and requires the evaluation of the gradient in (3.10) before the Hessian in (3.13), the recurring vectors $\mathbf{e}$ and $\mathbf{a}_j$ can conveniently be reused, possibly saving a large number of computations when Φ is high or evaluation of the model functions is costly.

Additionally Algorithm 3.3 uses the the gradient computed a point $\mathbf{p}$ in the interior of the box $\mathbf{P}$ currently being processed in the interval Gauss-Seidel method. This requires the computation of the $\mathbf{e}$ vector at $\mathbf{p}$. To make full use of the time spent on computing $\mathbf{e}$ at $\mathbf{p}$, it is used to compute the objective function at $\mathbf{p}$, possibly resulting in a better best known upper bound (denoted b in Algorithm 3.3) than what is already known. As $\mathbf{p}$ is typically chosen as the midpoint of $\mathbf{P}$, this is simply a computationally cheap way of sampling of the objective function at $\text{mid}(\mathbf{P})$.

3.5 Summary

This chapter gave a description on how interval analysis is used to construct an algorithm for global optimization. The Interval Newton method, for enclosing zeros of a function, was presented and described. Further, method to accelerated the convergence using different measures were described. When applied to the parameter estimation problem, some of the steps in each iteration of the optimization algorithm were shown to be highly parallel. It was shown that data can be reused between steps, lowering the computational cost of each iteration. Further, reuse of data allows for a low-cost sampling of the objective function at a point in the current interval box, which may yield an improved upper bound for use in the algorithm.

Chapter 4

The CUDA Platform

This chapter describes the Nvidia CUDA GPU architecture and the CUDA programming model. The architecture of the Fermi GPU family is described, followed by a description of the CUDA programming model. After that the different types of memory used when programming CUDA are described followed by a conclusion in Section 4.5.

4.1 GPU Architecture

General Purpose GPU computing has been studied for many years. The first GPU architecture designed specifically for GPGPU computing was released, along with the release of CUDA C programming language, by Nvidia in 2006. In 2007 the first GPU cards designated for high performance computing was released, the Tesla series. In 2008 the Khronos Group specified *Open Computing Language* (OpenCL). This made parallel programming platform independent since the OpenCL code can be ported to a wide range of architectures, including among others Nvidia GPUs, AMD GPUs, Intel CPUs and Altera FPGAs. The next move forward was the introduction of the Nvidia Fermi architecture, which added fused multiply add support. In this work the focus will be on the CUDA programming model, since it, by the authors, is considered the most mature language at the time of writing.

4.1.1 The Fermi Architecture

CUDA capable GPUs of the Fermi architecture consist of up to 16 *Streaming Multiprocessors* (SMs) organized around a bank of DRAM, a cache and a host interface [49]. Each SM consists of 32 cores and three types of memory; a chache, registers and shared memory [49]. A core consists of a integer *arithmetic logic unit* (ALU) and a *floating point unit* (FPU). The cores in each SM shares the registers, cache and shared memory.

The cores in each SM is organized in what Nvidia calls a *Single Instruction Multiple Threads* (SIMT) architecture [50], much akin to the SIMD[1] architecture in Flynn's taxonomy [51]. The overall picture is a form of SIMD within MIMD[1], where the SMs can be regarded as MIMD processors with an internal SIMD structure.

In Figure 4.1 a simplified version of the architecture is shown. The parallel architecture of the GPU makes it well suited for massively data parallel programs. Compared to a CPU the amount of cores in a GPU is much higher, but the clock

[1]By Flynn's taxonomy, SIMD and MIMD refer to *Single Instruction stream - Multiple Data stream* and *Multiple Instruction stream - Multiple Data stream* [51].

frequency of each core is lower than the CPU cores. A typical high end CPU core has a clock frequency around 2-4 GHz, where a typical high end GPU core has clock frequency around 0.7-1.2 GHz.

The DRAM is divided into four; global-, local-, constant- and texture-memory, each with different properties. The different types of memory in the SMs and the DRAM are described further in Section 4.3. The host interface connects the CPU (called the *host*) and the GPU (called the *device*) via a PCI-Express bus. The theoretical peak bandwidth is 8 GB/s on the PCIe x16 Gen2, but practical transfer rates are limited to around 5 GB/s [52]. Measurements using Nvidia samples bandwidth test, shows a bandwidth of 6 GB/s for a GeForce GTX 465[2]. The latency of the PCI-Express bus is approximately $1 - 6\mu s$ [53].

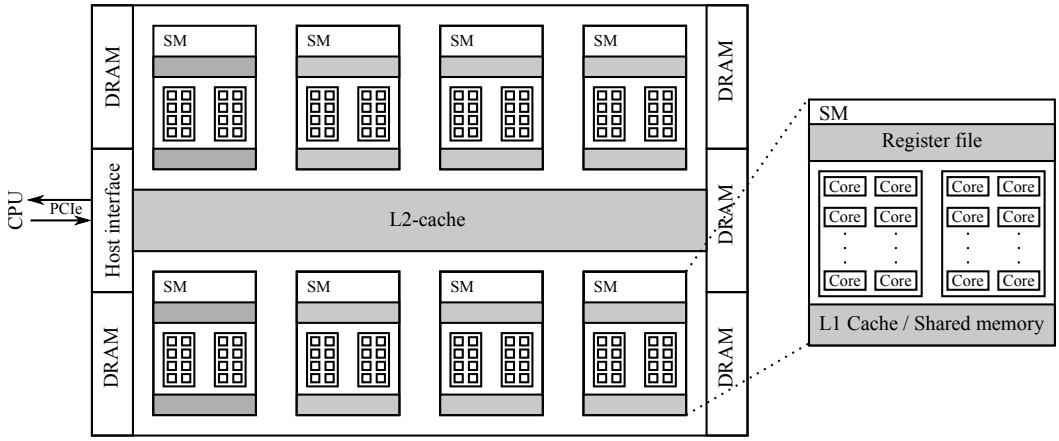


Figure 4.1: Simplified Nvidia GPU architecture, based on the Fermi architecture [49].

4.2 CUDA Programming Model

Apart from denoting the GPU architecture, CUDA is also an extension to the C programming language using a parallel programming model, illustrated in Figure 4.2, designed to facilitate the use of CUDA capable hardware.

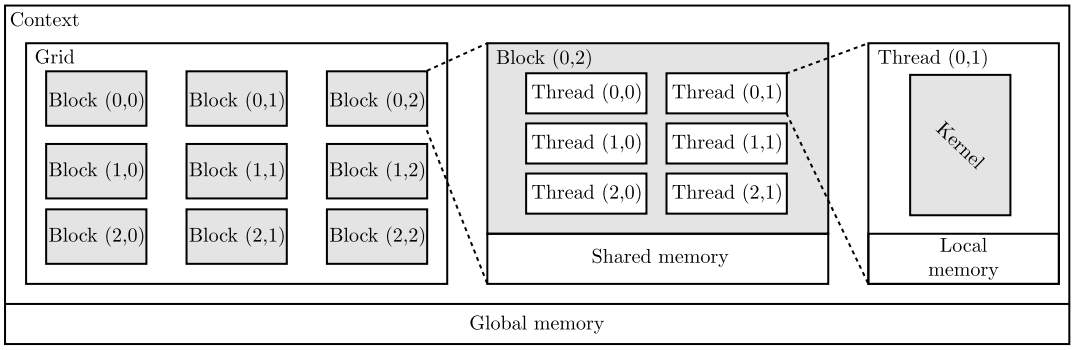


Figure 4.2: Illustration of the CUDA programming model.

The data used in parallel algorithms is often conceptually organized in vectors and 2 or 3 dimensional matrices , and the programming model is designed to mirror this organization. Parallel threads are organized in equally shaped one-, two- or three-dimensional arrays, called *blocks*, which are in turn organized in a one-, two-

[2]System: Supermicro X8SAX motherboard, PCI-Express 2.0

or three-dimensional arrays called the *launch grid* or just the *grid*. The dimensions of the grid, subject to certain bounds, are specified by the developer.

Functions executed on a grid are called *kernels*. Each thread in the grid executes an instance of the kernel. As each thread is aware of its position in the grid, the kernel can be written such that each thread processes different data.

As indicated in the figure, each thread has access to local memory, threads in each block share access to a block of memory while global- constant and texture memory is accessible by all threads in the context.

The threads of each block are executed by the SMs in groups of 32 adjacent threads. Such a group of threads is called a *warp*. When the threads in a warp follow the same execution path, the cores run synchronously. When the execution path of one or more threads in a warp differs from that of the other threads, the threads in the warp are arranged in sub-groups of threads that follow the same path. Each sub-group are then executed in turn. This effect, called *warp divergence*, leaves all other threads in the warp idle, which is undesirable.

Streams are used to manage concurrency of CUDA operations such as memory copy and kernel launch. Streams are defined as *"A sequence of operations that execute in issue-order on the GPU"* [50]. If multiple streams are used, overlapping between kernel launch and memory transfer can some times be utilized to hide data transfer time.

4.3 Memory

As described earlier, the GPU holds different types of memory. To best utilize the GPU the developer must understand these different kinds of memory, and how the use of these can affect performance. The description in this section is based on [49] and [52].

4.3.1 Memory Hierarchy

The different types of memory have different scopes and lifetimes. Some are cached, some are on-chip and others are off-chip. These features are key when programming GPUs for maximum utilization and performance.

The registers, located in each SM, are the fastest type of memory, but are limited in size. For each thread the needed amount of registers are allocated. The allocated registers are only accessible for the thread it has been allocated to. There is a total of 32k 32-bit registers per SM for compute capability 2.x and a total of 64k 32-bit registers per SM for compute capability 3.x. When a thread access a register there is a latency of approximately 24 clock cycles, but the latency can be hidden if there are 768 active threads (32 cores per SM times 24 clock cycles is 768 threads).

Each of the SMs has 64 kiB of on-chip memory. The on-chip memory can be configured in two different ways; 48 kiB shared/16 kiB L1 cache or 16 kiB shared/48 kiB L1 cache [49]. The L1 cache is divided into 128B cache lines. Shared memory can be accessed by all threads in a block. It is located on-chip and has high bandwidth and low latency compared to the global and local memory. The size of the shared memory is limited to 48 KiB per SM. Shared memory is divided into 32 equally sized memory modules denoted *banks*. The shared memory banks can be accessed simultaneously by the threads in an SM. Each of the shared memory banks has a bandwidth of 4B per two clock cycle. The bandwidth of the shared memory is given

as

$$B = b \cdot a \cdot S \cdot f \tag{4.1}$$

where B is the shared memory bandwidth, b is the bandwidth of each shared memory bank, a is the number of banks, S is the number of SMs and f is the GPU clock speed. The Nvidia Geforce GTX 580 has a clock speed of 1544 MHz [54] and 16 SMs. The theoretical shared memory bandwidth can be calculated as

$$B = 2[\text{B}] \cdot 32 \cdot 16 \cdot 1544 \text{ [MHz]} \tag{4.2}$$

$$\approx 1.58 \text{ [TB/s]} \tag{4.3}$$

Local memory is located off-chip and thus has a lower bandwidth and higher latency than the registers, for a Nvidia Tesla M2075 the maximum memory bandwidth is 150 GB/s. The size of the local memory is much larger; 512 KiB per thread. As for the registers, the scope of local memory is limited to one thread. The local memory is used by the compiler to place variables that are to large to reside in registers.

Constant memory is read-only and cached. The size of the constant memory is 64 kiB. Since it is cached, all threads of a half-warp[3], can read from the constant cache as fast as if reading from registers, as long all the threads read from the same address.

Global memory is located off-chip and is the largest compared to the other types of memory, typically several gigabytes. As the local memory the global memory has a lower bandwidth and a high latency compared to registers and shared memory. The global memory is as the name indicates visible to all threads.

Texture memory is read-only and is allocated out of global memory. Texture memory uses one-, two, or three-dimensional caching. For example, if an element in a 2 dimensional matrix is accessed, the adjacent elements of the matrix are cached to allow fast access to these elements afterwards.

The features of the different kinds of memory are summarized in Table 4.1.

Mem. type	**On-/off-chip**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	On	N/A	R/W	1 thread	Thread
Local	Off	Yes	R/W	1 thread	Thread
Shared	On	N/A	R/W	All threads in block	Block
Global	Off	Yes	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Table 4.1: Properties of different types of memory. Based on [52, Table 1, p. 24].

4.3.2 Coalesced Memory Access

Nvidia describes coalesced memory access as *"perhaps the single most important performance consideration"* [52]. Threads of the same warp accessing memory, is divided into a number of coalesced transactions. The number of transactions are equal to the number of cache lines necessary. An example where all 32 threads of a warp access adjacent 4-byte words, is shown in Figure 4.3. In this example the elements accessed are aligned with the 128-byte L1 cache line and therefore only one cache line is used, and one coalesced transaction.

[3] A half-warp is either the first or last 16 threads of a warp

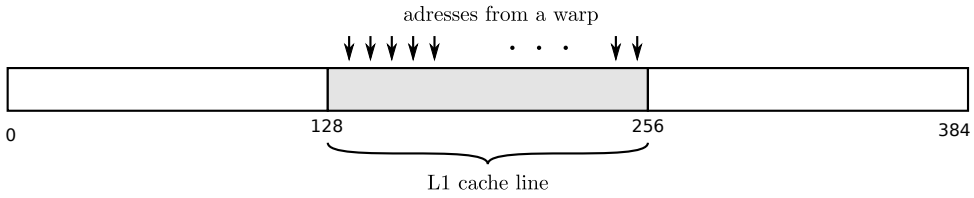


Figure 4.3: One cache line is accessed by one warp, resulting in one coalesced memory transaction. Based on [52, Figure 3, p. 25].

If the elements had not been aligned with the cache line, more cache line would be need and thus more coalesced transaction. An example of unaligned access to memory is illustrated in figure Figure 4.4.

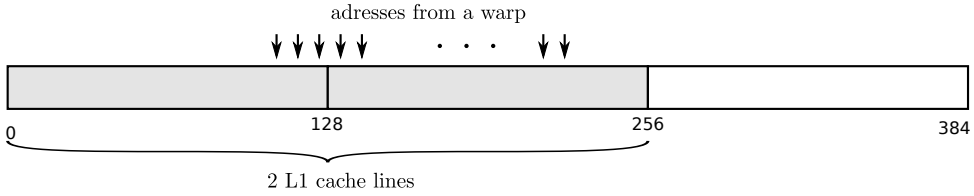


Figure 4.4: Two cache lines are accessed by the threads of one warp. In this example the access is not aligned with the cache lines and thus result in two coalesced accesess. Based on [52, Figure 4 p. 26].

The worst case scenario, with respect to memory access, is the case where all 32 threads in a warp needs its own cache line and therefor 32 coalesced transactions. This case is illustrated in figure 4.5

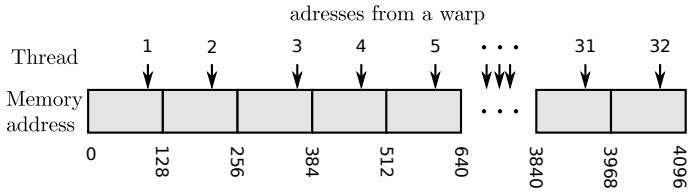


Figure 4.5: Worst case access to memory. Each thread from one warp access memory with its own cache line.

The effective bandwidth effects of strided memory access is illustrated in Figure 4.7. Stride is the length between the elements accessed by threads in a warp, illustrated in Figure 4.6. In Figure 4.7 Already with a stride of 2, only half the bandwidth is achieved [52].

4.4 CUDA for Interval Computations

As described in Section 2.1.5 on page 9, hardware support for directed rounding improves performance when doing computational interval arithmetic. CUDA GPUs of compute capability 2.0 and above support directed rounding for the following functions via compiler intrinsics: Addition, multiplication, fused multiply-add (FMA), division, square root and reciprocal value [50]. Contrary to x86 CPUs, where changing rounding mode is expensive with respect to performance, the CUDA GPUs do not require changing of the rounding mode of the SM to perform directed rounding, which gives CUDA GPUs an advantage over a conventional x86 CPU [55].

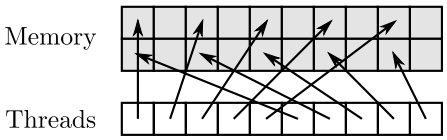


Figure 4.6: Strided memory accesss with a stride of 2. Based on [52, Figure 7, p. 28]

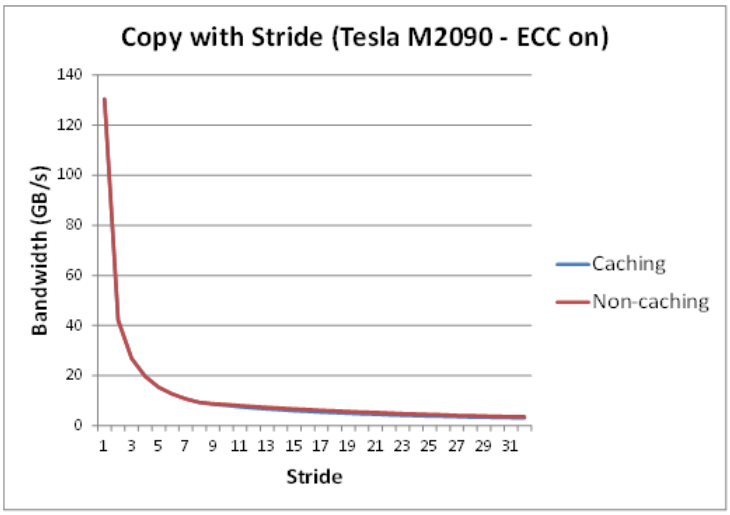


Figure 4.7: The effect of strided memory access. *Non-caching*: only L2 cache is used. *Caching*: L1 cache used. From [52, Figure 8, p. 29].

A crude measurement on the performance of performing interval computations is made. The performance of the CUDA GPU and x86 CPU is measured and the speedup is found. These measurements are not exact but gives an indication of the possible speedup when doing interval computations on a CUDA GPU.

On the GPU and the CPU a single-thread program runs a loop that performs a number of interval additions, multiplications and divisions in double precision. The time taken to execute the program is measured, and the number of interval operations per second (IOPS) is measured. As mentioned, both programs only use a single thread. Therefore, the IOPS measures for the GPU is scaled by the number of double precision units and the CPU measures are by the number of cores.

The performance is computed as:

$$p = \frac{r \cdot o}{t_e} \tag{4.4}$$

where $r = 100$ is the number of times the experiment was repeated, $o = 10^6$ is the number of times the given operation was performed in each experiment and t_e is the accumulated time taken to perform the experiments. The scaling factors used are the number of cores in the GPU and CPU, respectively. The measurements are shown in Table 4.2. It can be seen that when the performance is scaled the GPU performs significantly better than the GPU. Especially for the multiplication the GPU outperforms the CPU.

		Single core			Scaled by # of cores		
		GPU	CPU	$\frac{\text{GPU}}{\text{CPU}}$	GPU	CPU	$\frac{\text{GPU}}{\text{CPU}}$
Addition	[IOPS]	14.08M	67.16M	0.2096	37.84G	0.2687G	140.8
Multiplication	[IOPS]	8.751M	61.96M	0.1412	23.52G	0.2478G	94.91
Division	[IOPS]	5.088M	59.55M	0.08544	13.68G	0.2382G	57.41

Table 4.2: Crude measurements and extrapolation of the number of interval operations per second (IOPS) performed by the GPU and CPU. (The measurements are made for the PyInt_light and CuInt libraries, described in Section 5.5 on page 48.) The GPU used is a Nvidia GTX TITAN with 2688 single precision cores [56] and the CPU is a 3.07GHz Intel Core i7-950 with 4 cores. For full specifications on the test system, see Appendix A. The Python code for this experiment can be found in `code/misc/IOPS`

.

4.5 Summary

In this chapter the most important features of the GPU were described. When programming a GPU one must understand the limitation and possibilities of the architecture, to best utilize the massive parallelism the GPU introduces.

The GPU has different types of memory. It is important to use these different kinds of memory correctly to maximize performance. The on-chip memory is generally fast and has low latency, but is limited in size, where the off-chip memory is slow and has a high latency but a larger size. It is shown that when accessing off-chip memory it is important to do this coalesced. Further it was show that a CUDA GPU performs significantly better at performing interval operations, than a CPU.

It has been shown that the GPU has an advantage over the CPU when doing interval computations, this is due to the fact that x86 architecture most change rounding mode to do directed rounding.

Chapter 5

Implementation

This chapter treats the parallelization and implementation of Algorithm 3.3 and some variations thereof. The implementations are tested and benchmarked Chapter 6.

In Section 5.1 a short and non-exhaustive survey of parallelization strategies for BB-type algorithms and algorithms for global optimization using interval analysis is given, followed by a proposed scheme for parallelization of Algorithm 3.3 applied to the least-squares parameter estimation problem.

The algorithm is implemented in a number of different versions, some using only the CPU and utilizing the GPU as a co-processor for acceleration. An implementation utilizing the GPU is described in a basic form in Section 5.2.2 and in a more advanced and improved form in Section 5.2.3. In Section 5.2.4, a set of algorithmic variations to the implementation in Section 5.2.3 are described. The base CPU-only implementation and variations thereof is described in Section 5.3. Some implementation details, such as how the work pool is handled and how data is reused in different steps of the algorithm is described in Section 5.4. The implementations make use of interval analysis software libraries for GPU and CPU. These libraries, described in Section 5.5, are extended and adapted versions of already existing libraries.

5.1 Parallelization Strategies

Much work has been done in the area of parallel Branch & Bound-type algorithms on various types of architectures. In [14] a survey and classification of different parallel Branch & Bound approaches until 1994 is given. The classification divides parallelism into three types, as described below.

- **Type 1:** Parallelism in the operations performed on the sub-problems (inner iteration), e.g. in the bounding method. This type of parallelism does not alter the structure of the overall algorithm and is therefore easily implemented. It is well suited for fine-grained parallel systems, i.e. systems where each processor only processes a small amount of data.
- **Type 2:** Parallel construction and processing of the search tree (outer iteration). Because the search tree is processed in parallel, some boxes may be processed that would not have been processed using a sequential algorithm. This causes unnecessary computations to be performed. This type of parallelism is well suited for coarse-grained parallel systems where each processor handles large amounts of data, and is according to [14] the most studied type of parallelism.

- **Type 3:** Parallel construction of several search trees that use different methods for branching, bounding, testing, etc, and exchange information. The idea is that the different approaches may complement each other in order to solve the problem faster. This approach requires extensive communication and handling if the same parts of the search area should not be processed more than once. According to [14] this is the least used and least studied type.

Parallelization of type 2 and 3 results in several nodes of the search tree being processed simultaneously. Some considerations that must be taken into account when doing this are:

- **Processing time:** The time taken to process a node (one outer iteration) may differ between nodes.
- **Communication:** Parallel processing of nodes requires a strategy for communication of bounds in order to lessen the amount of unnecessary work.
- **Work distribution:** Unused computational resources result in sub-optimal performance. Therefore a strategy to keep the parallel resources occupied is important. Furthermore, performance can be gained by implementing a strategy that promotes processing of promising regions of the search space.

Type 2 parallelism has been implemented in many different ways on many different types of architectures [30, 32, 34, 33, 14]. The survey in [14] covers many different implementations, and draws some relevant conclusions on that background. It is concluded that the type 2 form of parallelism is only suitable on SIMD architectures if the operations performed in each iteration are trivial and run in constant time. On MIMD architectures, for which type 2 parallelism is better suited, the approaches used when implementing it are classified by two parameters: whether they use synchronous or asynchronous parallelism and whether they use a single work pool[1] or have multiple pools [14]. Implementations that use different combinations of the parameters are called *Synchronous/Asynchronous Single Pool* (SSP/ASP) or *Synchronous/Asynchronous Multiple Pool* (SMP/AMP). For SSP and SMP, it is important that the processing needed for each sub-problem is approximately equal to avoid idle time. ASP is concluded to be suited only for *"problems with a nontrivial bounding operation, and parallel architectures having a relatively small number of processors"* [14]. For AMP strategies it is concluded that a dynamical load balancing must be employed in order to achieve high efficiency.

Most of the previous work on parallel Branch & Bound methods either concerns general frameworks or methods specifically designed for combinatorial optimization problems. Methods designed for interval global optimization (IGO) algorithms are more sparse [14, 33]. Some examples of work focusing on global optimization using interval analysis are [30, 32, 34] and [33]. The approaches, detailed below, all use type-2 parallelization.

1. **Henriksen & Madsen, [30]:**
   - Parallelization type: Type-2.
   - Work pool organization: ASP.

[1] Work pool: Storage containing the interval boxes to be processed.

- Control and communication strategy: A master-slave structure, where the master keeps the work pool from where the parallel workers request boxes to process and send back resulting subboxes.
- Box selection strategy: Both depth-first and best-first.

2. **Berner, [32]:**
   - Parallelization type: Type-2.
   - Work pool organization: AMP.
   - Control and communication strategy: The system consists of a central mediator and a number of worker nodes. The central mediator keeps track of the best known upper bound and performs work load balancing between the worker nodes.
   - Box selection strategy: Best-first.
3. **Ibraev, [34]:**
   - Parallelization type: Type-2.
   - Work pool organization: AMP.
   - Control and communication strategy: A *Challenge Leadership* model, where the role of master node is shifted between the parallel worker nodes, depending on which has most recently updated the globally best known upper bound. When a worker node has no boxes to process, it requests one or more from the master node.
   - Box selection strategy: Best-first.
4. **Casado et al., [33]:**
   - Parallelization type: Type-2.
   - Work pool organization: Both ASP and AMP.
   - Control and communication strategy (ASP): A number of worker nodes process interval boxes in parallel. The best known upper bound and the work pool is kept in shared memory, which is accessed by the worker nodes by use of a semaphore.
   - Control and communication strategy (AMP): A number of worker nodes, each with its own work pool, process interval boxes in parallel. The best known upper bound is kept in shared memory and accessed by use of a semaphore. At each iteration, each worker node checks for any idle workers. If a busy worker node with more than two boxes in its work pool detects an idle worker, it moves every other box in its work pool to the idle workers pool.
   - Box selection strategy: Best-first.

Interestingly in [33], which is the only of the four that compares different methods for parallelization on the exact same optimization algorithm and the same test problems, it is concluded that the ASP and AMP approaches are approximately equally fast.

Work within the area of parallel Branch & Bound-type algorithms on GPUs appears equally sparse and seems only to cover combinatorial problems. A number of papers focus on problems where the bounding operations are sufficiently trivial to be implemented using a type 2 scheme on the GPU architecture by assigning one thread

per node of the search tree [57, 58, 59, 60, 61]. This approach for parallelization is not relevant to the problem at hand, as the operations performed in each sub-problem are significant more complicated.

5.1.1 GPU Parallelization for the Parameter Estimation Problem

As stated in Section 4.1.1, the CUDA GPU architecture is two-layered, with a SIMD architecture on the lower (SM) level and a MIMD architecture on the upper layer (Stream). Therefore a parallelization strategy designed for either MIMD or SIMD is not a perfect fit for a CUDA GPU. The possibilities of implementing different types of parallelism for the problem at hand are identified as:

The interval arithmetic operation level (Type 1): Parallelization on this level consists of parallelizing inner operations of the individual interval arithmetic operations. For example, an interval multiplication can be performed by doing 8 parallel floating point multiplications followed by 4 parallel comparisons in turn followed by two parallel comparisons. The comparisons require synchronization of the threads and communication via shared memory. Relative the the amount of computations being performed, the amount of communication and synchronization between the threads is large. Further, as the steps in the example use a decreasing number of threads and because of the SIMD architecture of each CUDA SM, a number of threads remain idle in the second and third step.

Inner iteration level (Type 1): As discussed in Section 3.4, parallelization can be introduced on the inner iteration level by evaluating the objective, gradient and Hessian functions and the Interval Newton step in parallel. The tasks mainly consist of computing vectors with independent elements and summing the elements. The number of elements in the vectors used in the computation of the objective, gradient and Hessian functions is equal to the number of measurements, Φ. Thus when Φ is high the problem becomes of computing the vector elements becomes massively parallel. The computations used to compute each element are the same, and therefore this problem maps well to the architecture of the CUDA SMs by assigning a thread to each element. Summation of the vector elements can performed in parallel in a series of reduction steps. As such summation is well suited for parallelization, but similar to the method described above for parallelizing on the interval arithmetic level it leaves some threads idle. Parallel reduction is described further in Section 5.2.1.

Overlapping iteration steps (Type 1): Notice in Section 3.4 on page 25, that the vectors $\mathbf{e}$, $\mathbf{a}$ and $\mathbf{b}$, and the gradient at the midpoint of the box being processed can be computed independently, as they do not reuse previously computed data. The gradient at the midpoint of the box being processed is computed for use in the Newton step in Algorithm 3.3. As noted in Section 3.4, this computation also makes a computationally cheap evaluation of the objective function, at the midpoint of the box, possible. As these computations do not reuse any data, they can be executed in parallel. However, if one test fails, the vectors used in subsequent tests are not used. For example, if $\mathbf{e}$, $\mathbf{a}$ and $\mathbf{b}$ are computed in parallel and the objective function range test fails, then only $\mathbf{e}$ is used. Therefore this method of parallelization can introduce considerable amounts of unused computations compared to a sequential approach to performing the iteration steps.

Outer iteration level (Type 2/3): Each interval box in the work queue can be processed independently in parallel. CUDA streams, each with a controlling thread on the CPU, can be used as asynchronous worker nodes. In order to share information, and distribute boxes to be processed, this type of parallelization introduces a communication overhead between the CUDA streams, through the controlling CPU threads.

To summarize the above, the following conclusions are drawn: Parallelization on the interval arithmetic operation level does not fit the CUDA GPU architecture well. The resources are more easily and efficiently used by introducing parallelism on the inner iteration level, where less communication and synchronization is required and less idle time is introduced. For both approaches, no unnecessary computations are performed. Overlapping of iteration steps can be used to speed up each iteration, as long as unused resources are available. When this is not the case, this approach for parallelization merely introduces unnecessary computations, slowing down the algorithm. The same is the case for parallelization on the outer iteration level.

Following the conclusion in [33] that ASP and AMP schemes perform approximately equally well an ASP scheme is used. This results in a simpler scheme with no need for dynamic load balancing. This work is limited to a single-GPU implementation. As streams residing in the same CUDA context (on the same GPU) share access to global GPU memory, the work pool can potentially be stored there. If the implementation was to be extended to a multi-GPU system, the work pool would either have to reside on the CPU, or an AMP scheme with dynamic load balancing would be necessary.

The proposed parallelization scheme, designed for a single-GPU system, resembles that of the master-slave scheme used in [30] and the ASP scheme in [33]. The work pool is kept in shared memory on the CPU. A number of asynchronous parallel worker nodes, each with a controlling CPU thread and a GPU stream, pick boxes for processing from the shared memory as needed and return the processed results. When a worker node updates the best known upper bound, it is communicated to the other worker nodes via shared memory. The scheme is illustrated in Figure 5.1. Because of time limitations for the present work, parallelization on the inner iteration level is prioritized. The implementations described below therefore only use a single worker node.

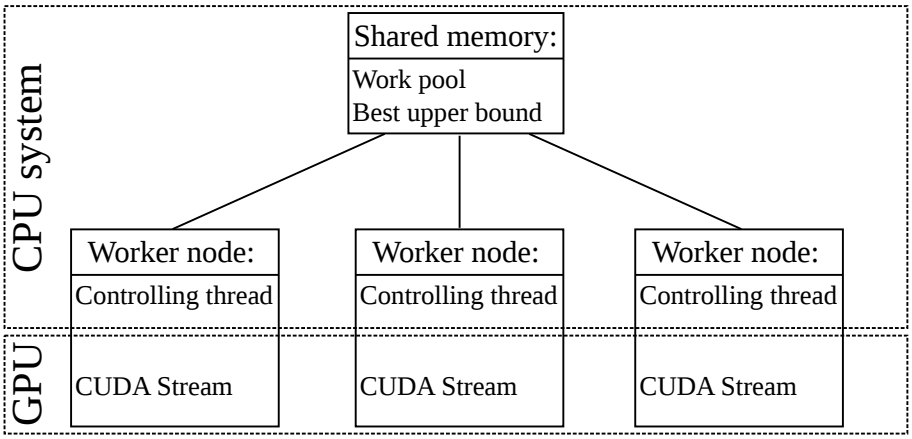


Figure 5.1: Structure of proposed parallelization scheme. The dashed boxes indicate the separation of the GPU and the CPU system.

5.2 GPU Accelerated Implementations

In this section the different GPU accelerated implementations developed in the present work, are presented. Two implementations are presented; 1) a basic version using type 1 parallelization, and 2) an improved version with reduced communication and synchronization between the CPU and the GPU. The improved implementation can be extended with a number of different algorithmic improvements in different combinations. A number of variations of these improvements are implemented to test their impact on performance.

5.2.1 Parallel reduction

A central element in the several of the steps in the algorithm is parallel reduction of a vector of elements, e.g. in the form of a dot product.

An example of a parallel reduction of 8 elements is illustrated in Figure 5.2. In each step the number of data elements, and consequently the number of threads used for the reduction, is reduced by half until only the result remains. Assuming that the number of elements n is a power of two, a parallel reduction requires $\log_2(n)$ steps, while a sequential reduction requires $n-1$ steps. The first step of the reduction requires $\frac{n}{2}$ operations, and in each successive step the number of operations is halved. This means that in step i, $n(1/2 - 2^{-i})$ threads are idle. In practice (not illustrated in Figure 5.2) , the data elements are often computed using n threads in the same kernel as the reduction, thus leaving $n(1 - 2^{-i})$ threads idle in step i.

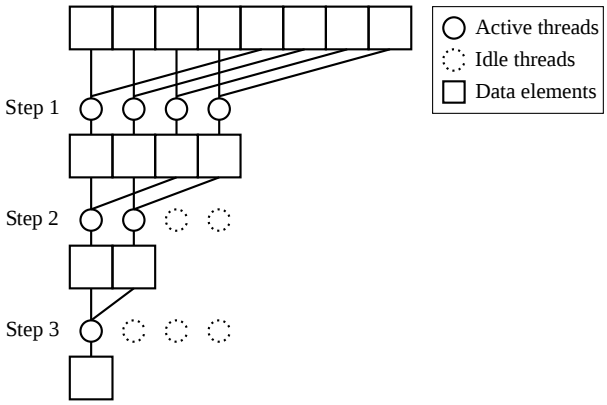


Figure 5.2: Illustration of a parallel reduction with 8 input data elements and 1 output element.

As described in Chapter 4 each block holds a certain amount of threads. Therefore, if the size of the vector to be reduced is larger than twice the number of threads in each block, the vector must be split and processed by several blocks. The partial result from each block is then written to global memory and a new kernel that reduces these partial results is then launched. Figure 5.3 illustrates the entire structure of several blocks performing an element wise function on a split vector, and then performing a partial reduction, the result of which is further reduced by a second kernel.

To compute each element in the objective function, gradient and Hessian, the method shown in Figure 5.3 is used. This results in a structure as illustrated in Figure 5.4. For example, to compute the Hessian Ψ^2 instances of method illustrated in Figure 5.3 are used, where Ψ denotes the number of parameters to in the problem. Normally the computation of the Hessian could be reduced, due to the fact

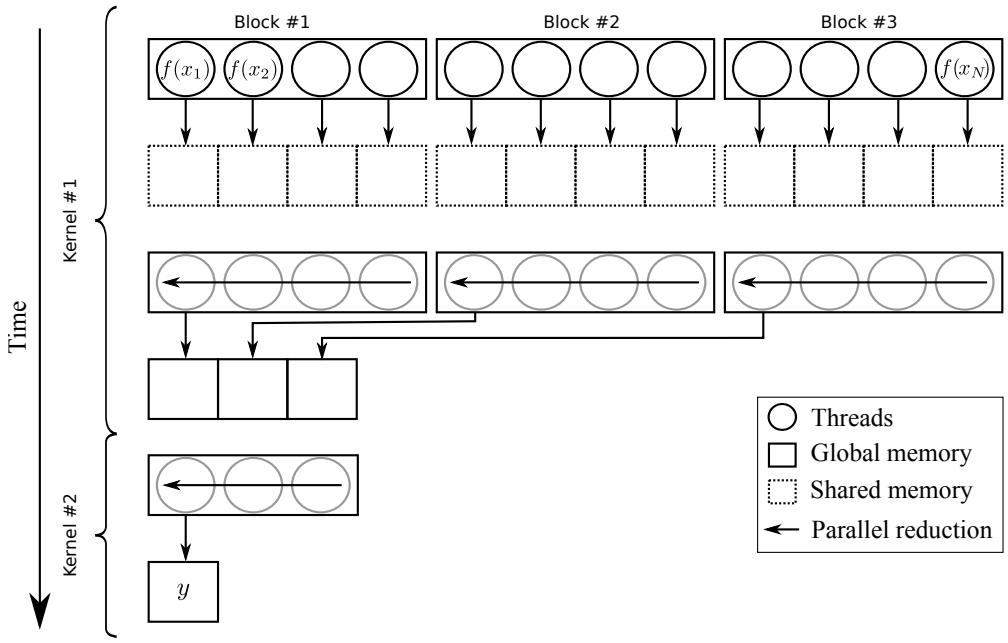


Figure 5.3: Illustration of three blocks first performing a function $f()$ on each of the elements a vector $\mathbf{x} = x_1, x_2, ..., x_N$ and then reducing using two parallel reductions.

that the Hessian is symmetric, but in this implementation, the method described in Section 3.3.3 on page 24, is used resulting in a non-symmetric Hessian.

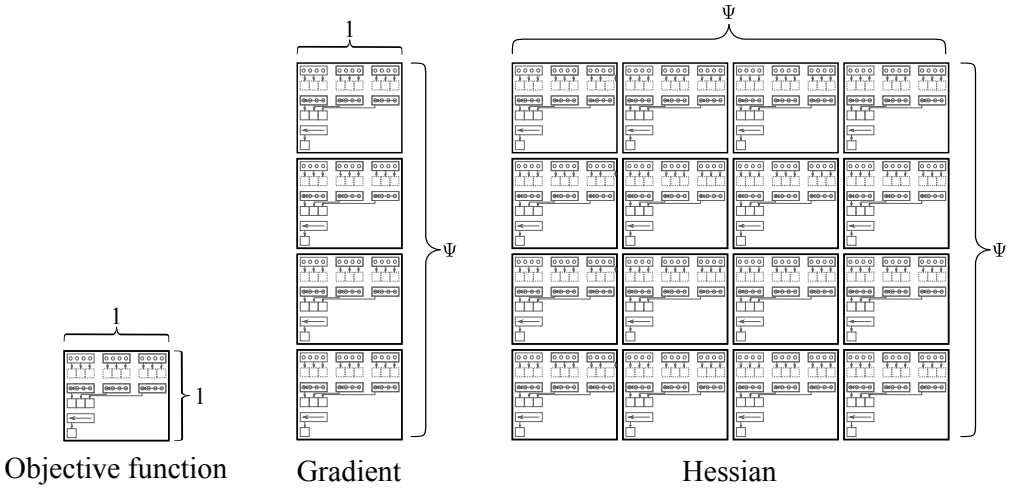


Figure 5.4: Illustration of how several instances of the structure illustrated in Figure 5.3 are used to compute the value of the objective-, gradient- and Hessian functions. In the example shown here, the number of parameters is $\Psi = 4$.

5.2.2 Implementation 1 - Basic Type 1 Parallelization

In Algorithm 3.3 on page 20 the algorithm was divided into a number of tests. These tests consist of two parts, namely a computation of a result and a check on this result. In this implementation the algorithm is accelerated by parallelizing the inner iterations in the computation of the objective function, gradient, Hessian and Interval Newton step. The general structure of the used to compute the objective function, gradient and Hessian is as illustrated in Figure 5.4. By storing intermediate results

in global memory, data is reused as described in Section 3.4. The results of the computations are transferred to the CPU where the outcome checks are performed. For each of these transfers a synchronization between the CPU and the GPU occurs. A synchronization of the GPU and CPU requires all work on the GPU to be completed, and thus no work will execute on the GPU before the synchronization is complete and a new kernel is launched from the CPU. In each check it is determined whether to proceed to the next test for the current box, or proceed with a new box. The implementation uses the inverse-midpoint preconditioning scheme in the Interval Newton test. Due to lack of a GPU implementation for matrix inversion, the preconditioning is done on the CPU. The Interval Newton computation is run on the GPU.

Algorithm 5.1 shows the flow and division of tasks between the CPU and the GPU for the implementation. The Interval Newton check in line 16 of Algorithm 5.1 os detailed in Algorithm 5.2.

Algorithm 5.1 Pseudo code describing Implementation 1 - Basic Type 1 Parallelization

Input:

Search space $\mathbf{P}_0 \in \mathbb{IR}^{\Psi}_*$.
Model function M, model gradient $\mathbf{G}_m$ and model Hessian $\mathbf{H}_m$.
Objective function width and box width thresholds $\epsilon_f \in \mathbb{R}_+$ and $\epsilon_p \in \mathbb{R}_+$.
Initial best known upper bound $b \in \mathbb{R}+$.

1:	$\mathcal{L} \leftarrow \{\}$ (Solution list)	*CPU*
2:	$\mathcal{W} \leftarrow \{\mathbf{P}_0\}$ (Work queue)	*CPU*
3:	$\rho_\tau \leftarrow 0.6$ (Bisection threshold)	*CPU*
4:	**while** $\mathcal{W} \neq \{\}$ **do**	*CPU*
5:	$\mathbf{P} \leftarrow$ take new box from $\mathcal{W}$	*CPU*
6:	$F(\mathbf{P}) \leftarrow$ Objective function bound computation	***GPU***
7:	Objective function bound check on $F(\mathbf{P})$	*CPU*
8:	$\mathbf{G}(\mathbf{P}) \leftarrow$ Gradient computation	***GPU***
9:	Monotonicity check on $\mathbf{G}(\mathbf{P})$	*CPU*
10:	$\mathbf{H}(\mathbf{P}) \leftarrow$ Hessian computation	***GPU***
11:	Convexity check on $\mathbf{H}(\mathbf{P})$	*CPU*
12:	Solution test on $\mathbf{P}$ and $F(\mathbf{P})$	*CPU*
13:	$\mathbf{G}_{\text{mid}} \leftarrow$ Gradient at mid($\mathbf{P}$)	***GPU***
14:	$\hat{\mathbf{G}}_{\text{mid}}, \hat{\mathbf{H}} \leftarrow$ Preconditioning of $\mathbf{G}_{\text{mid}}$ and $\mathbf{H}$	*CPU*
15:	$\mathbf{P}_{\text{new}} \leftarrow$ Interval Newton computation using $\hat{\mathbf{G}}_{\text{mid}}$ and $\hat{\mathbf{H}}$	***GPU***
16:	Interval Newton check on $\mathbf{P}_{\text{new}}$	*See Algorithm 5.2*
17:	**end while**	*CPU*
18:	Cleanup $\mathcal{L}$	*CPU*
19:	**return** $\mathcal{L}$	*CPU*

Notes: Instead of specifying the search space as a single box $\mathbf{P}_0$ it can be specified as a list of boxes.

Algorithm 5.2 Interval Newton Check

1:	$\rho = \frac{\text{vol}(\mathbf{P}_{\text{new}})}{\text{vol}(\mathbf{P})}$	*CPU*
2:	**if** $\rho >$ Bisection threshold **then**	*CPU*
3:	$\quad \delta \leftarrow \underset{i=0,\ldots,\Psi-1}{\text{argmax}} \ (\text{wid}(G_i) \cdot \text{wid}(P_i))$	*CPU*
4:	$\quad \mathbf{P}_{\text{a}}, \mathbf{P}_{\text{b}} \leftarrow$ bisect $\mathbf{P}_{\text{new}}$ at dimension δ	*CPU*
5:	$\quad$ Append $\mathbf{P}_{\text{a}}$ and $\mathbf{P}_{\text{b}}$ to $\mathcal{W}$	*CPU*
6:	**else**	*CPU*
7:	$\quad$ Append $\mathbf{P}_{\text{new}}$ to $\mathcal{W}$	*CPU*
8:	**end if**	*CPU*

This implementation is simple and serves as a basic presentation of the algorithm. The purpose of this implementation is to locate performance bottlenecks and present the basic parallelization of the objective function, gradient and Hessian computations.

A major disadvantage, of this implementation, is the many synchronizations and data transfers. The amount data transferred is small and thus most of the transfer time is latency.

To investigate the time between kernels are run, a profiling of GPU activity is made using the Nvidia Visual Profiler (NVVP) tool [62]. A part of the profiling is shown in Figure 5.5a. Here, the idle time between kernel launches, caused by the need for synchronization, is clearly visible. It should be noted that a profiling of this implementation is very problem specific and should therefor be treated as such. A reason why it is so problem specific is the fact that the evaluations of the objective function, gradient and Hessian are all dependent on the amount of measurements and the number of parameters, and thus more or less work are executed on the GPU.

5.2.3 Implementation 2 - Reduced Synchronization and Communication

In Implementation 1 the computational part of the tests is done on the GPU and the checking part done on the GPU. In this implementation most of the checks are moved to the GPU in order to reduce the need for synchronization and data transfer between the CPU and the GPU. As for Implementation 1, the base version of this implementation uses the inverse-midpoint preconditioner for the Interval Newton step. The implementation scheme is shown in Algorithm 5.3.

As mentioned above it is desired to perform both the computational part and outcome check for each test on the GPU. If a test fails, the subsequent tests should not be run. However the GPU cannot launch kernels on its own and therefore the decision on whether a kernel should be launched must still be taken on the CPU, requiring synchronization. This problem is solved by introducing a status flag on the GPU. The CPU then queues the kernels to perform all the tests on the GPU, but before performing the actual computation each kernel first checks the status flag and only performs the computation if the flag is clear. If a test fails (or the box qualifies as a solution) the kernel sets the status flag ans the subsequent kernels do nothing other than check the status flag and return. After queuing the kernels that perform the tests, the CPU queues a synchronization point and waits for the GPU to signal that this point has been reached. In this way all the kernels are run but does not do any actual computation unless necessary, and unnecessary communication with the CPU is avoided.

In order to save time on data transfers, page locked memory is used to transfer

the status flag, the Hessian, the gradient at the midpoint of the current box, the objective function value and the result of the Newton step from the GPU to the CPU. Page locked is a scarce memory resource that resides on the host system, but can be addressed by the GPU, eliminating the need for explicit data transfer. Further, when using this type of memory, data transfers overlap with code execution [50].

Algorithm 5.3 Pseudo code describing the second implementation

Input:
Search space $\mathbf{P}_0 \in \mathbb{IR}^{\Psi}_*$.
Model function M, model gradient $\mathbf{G}_m$ and model Hessian $\mathbf{H}_m$.
Objective function width and box width thresholds $\epsilon_f \in \mathbb{R}_+$ and $\epsilon_p \in \mathbb{R}_+$.
Initial best known upper bound $b \in \mathbb{R}+$.

1:	$\mathcal{L} \leftarrow \{\}$ (Solution list)	*CPU*
2:	$\mathcal{W} \leftarrow \{\mathbf{P}_0\}$ (Work queue)	*CPU*
3:	$\rho_\tau \leftarrow 0.6$ (Bisection threshold)	*CPU*
4:	**while** $\mathcal{W} \neq \{\}$ **do**	*CPU*
5:	$\mathbf{P} \leftarrow$ take new box from $\mathcal{W}$	*CPU*
6:	Enqueue Objective function bound test	***GPU***
7:	Enqueue $\mathbf{G}(\mathbf{P}) \leftarrow$ Monotonicity test	***GPU***
8:	Enqueue $\mathbf{H}(\mathbf{P}) \leftarrow$ Convexity test	***GPU***
9:	Enqueue status_flag $\leftarrow$ Solution test	***GPU***
10:	Enqueue $\mathbf{G}_{\text{mid}}(\mathbf{P}) \leftarrow$ Gradient at mid($\mathbf{P}$)	***GPU***
11:	Synchronize CPU and GPU	*CPU*
12:	**if** status_flag = discard **then**	*CPU*
13:	Discard $\mathbf{P}$	*CPU*
14:	Break	*CPU*
15:	**else if** status_flag = solution **then**	*CPU*
16:	Append $\mathbf{P}$ to $\mathcal{L}$	*CPU*
17:	Break	*CPU*
18:	**end if**	*CPU*
19:	$\mathbf{q}(\mathbf{P}) \leftarrow \text{mid}(\mathbf{H}(\mathbf{P}))^{-1}$	*CPU*
20:	$\hat{\mathbf{H}}(\mathbf{P}) \leftarrow \mathbf{q}(\mathbf{P}) \cdot \mathbf{H}(\mathbf{P})$	*CPU*
21:	$\hat{\mathbf{G}}_{\text{mid}}(\mathbf{P}) \leftarrow \mathbf{q}(\mathbf{P}) \cdot \mathbf{G}_{\text{mid}}(\mathbf{P})$	*CPU*
22:	$\mathbf{P}_{\text{new}} \leftarrow$ Interval Newton computation using $\hat{\mathbf{G}}_{\text{mid}}$ and $\hat{\mathbf{H}}$	***GPU***
23:	**if** Status flag not clear **then**	*CPU*
24:	Discard $\mathbf{P}$	*CPU*
25:	Break	*CPU*
26:	**end if**	*CPU*
27:	wid($\mathbf{G}$) $\leftarrow$ Transfer from GPU.	*CPU*
28:	Interval Newton check (using wid($\mathbf{G}$))	*See Algorithm 5.2*
29:	**end while**	*CPU*
30:	Cleanup $\mathcal{L}$	*CPU*
31:	**return** $\mathcal{L}$	*CPU*

Notes: Instead of specifying the search space as a single box $\mathbf{P}_0$ it can be specified as a list of boxes.

As for Implementation 1, the GPU activity is profiled using NVVP. The result is shown in Figure 5.5b. From this profiling it is evident that the idle time between kernel executions significantly reduced compared to Implementation 1, resulting in

improved performance. Since these profilings are done on the same test problem, it is expected to be a fair comparison and the it is clear that the reduced time between kernel execution benefits the implementation.

5.2.4 Variations of Implementation 2

As described in Section 3.3 and Section 3.4 a number of algorithmic modifications can be applied to attempt to accelerate the convergence. In order to investigate the effect of applying these modifications in different combinations a number of variations of Implementation 2 are implemented.

The base algorithm, consisting of the objective function range test, monotonicity test and convexity test, is used with the following algorithmic elements in different combinations:

Interval Newton with inverse midpoint preconditioner (labeled `I`). As described in Section 3.3.

Interval Newton with pivoting preconditioner (labeled `P`). As described in Section 3.3.

Objective function sample at box midpoint (labeled `F`). As described in Section 3.4.

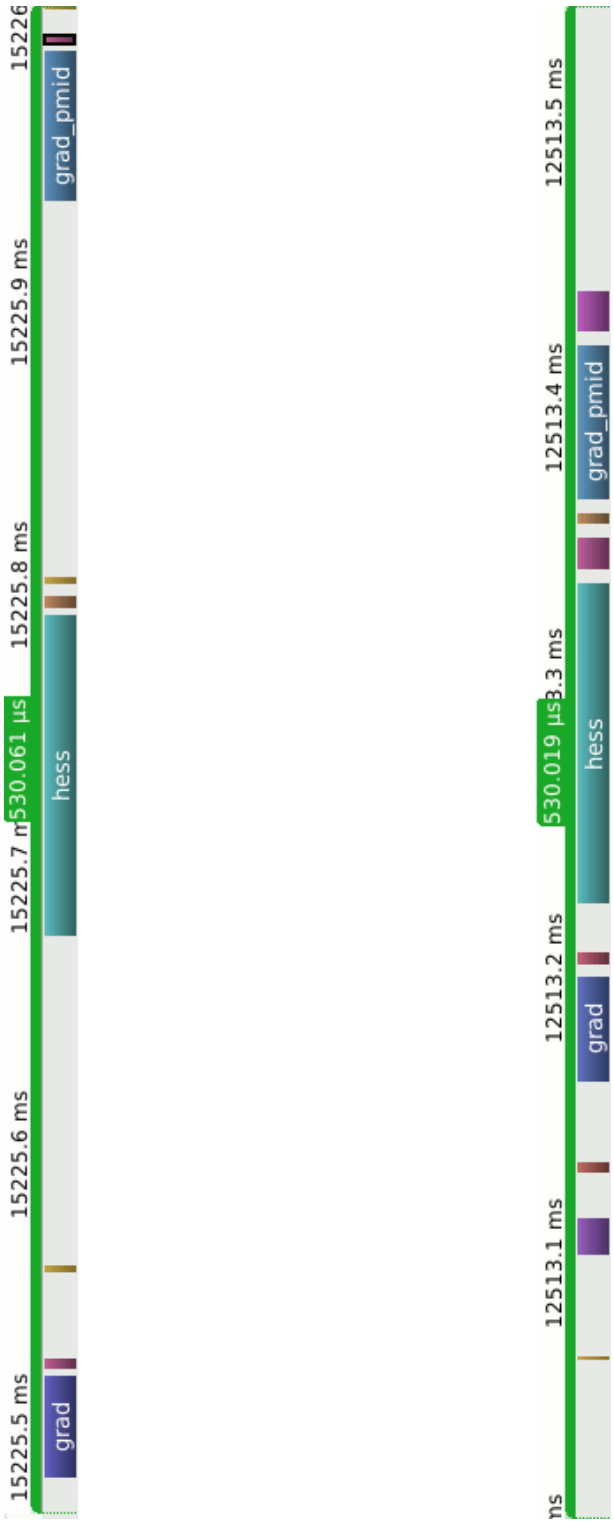
No additional algorithmic elements (labeled `N`). Only the objective function range test, monotonicity test and convexity test, followed by a bisection step of the tests are passed.

The different combinations of these elements are: `IPF`, `IF`, `IP`, `PF`, `I`, `P`, `F` and `N`. All of the variations are benchmarked in Chapter 6 to evaluate the performance of the different methods and different combinations.

The pivoting preconditioner is implemented as described in Section 3.3.2 using a grid structure with Ψ blocks with each $\Psi \times \Psi$ threads. For each element of the resulting box, Ψ pivot positions must be tried and compared to get the best result. Each of the Ψ blocks computes one element of the resulting box and each row, of threads in a block, computes a candidate for the resulting box for one pivot position, the best of which is saved in global memory. All communication between threads happens in shared memory.

5.3 CPU Implementation

In order to measure the speedup achieved by using GPU acceleration, a CPU reference version of the algorithm is implemented for comparison. To make the comparison of the GPU and CPU implementations as fair as possible, the CPU implementation is implemented as a combination of Python and C-code. Most of the computationally heavy parts of the algorithm, the evaluation of the model-, objective-, gradient- and Hessian functions and the Interval Newton steps are implemented in C, while the flow of the algorithm is controlled by Python. The matrix inversion in the Interval Newton step using the inverse midpoint preconditioner is done in Python, using a Fortran LAPACK library (see Appendix A). This resembles the structure of the GPU implementations closely. The structure of the CPU implementation is listed in Algorithm 5.4. For comparison, the same set of variations of the base algorithm as described in Section 5.2.4 are implemented for the CPU version.



(a) Implementation 1 **(b)** Implementation 2

Figure 5.5: Profiling time slice of GPU activity for implementations 1 and 2. Each colored block illustrates execution of a kernel on the GPU. Blocks of the same color are instances of the same kernel. Grey areas are GPU idle time. The profiling is done using the test problem FDEHYDE, described in Section 6.2 on page 54, using 4096 simulated measurements.

Algorithm 5.4 Pseudo code describing the CPU reference implementation

Input:
Search space $\mathbf{P}_0 \in \mathbb{IR}^{\Psi}_*$.
Model function M, model gradient $\mathbf{G}_m$ and model Hessian $\mathbf{H}_m$.
Objective function width and box width thresholds $\epsilon_f \in \mathbb{R}_+$ and $\epsilon_p \in \mathbb{R}_+$.
Initial best known upper bound $b \in \mathbb{R}+$.

```
1: L ← {} (Solution list)
2: W ← {P_0} (Work queue)
3: ρ_τ ← 0.6 (Bisection threshold)
4: while W ≠ {} do
5:     P ← take new box from W
6:     Objective function range test
7:     Monotonicity test
8:     Convexity test
9:     Solution test
10:    Gradient at mid(P)
11:    Interval Newton test with inverse midpoint preconditioner
12: end while
13: Cleanup L
14: return L
```

Notes: Instead of specifying the search space as a single box $\mathbf{P}_0$ it can be specified as a list of boxes.

5.4 Implementation Details

This section describes some of the common details, that apply for all the implementations.

Model function: The algorithm requires interval versions of the model function m, its gradient $\mathbf{g}_m$ and its Hessian, $\mathbf{h}_m$. These are provided by the user, implemented with a standard interface in CUDA-C for the GPU versions or in C/C++ for the CPU version described in Appendix B.

Inverse midpoint preconditioning, non-invertible matrices: In some cases the matrix mid($\mathbf{H}$), which is inverted to compute the preconditioner, is singular or contains non-real elements ($\pm\infty$), such that the matrix is non-invertible and preconditioner cannot be computed. In such cases, the Interval Newton step using this preconditioner is skipped.

Choice of numerical precision: All the implementations described use single precision (32 bit) floating point for all computations. This choice comes out in favor of the GPU used, as it is primarily a 32bit architecture with 2866 single precision cores against 896 double precision units, whereas the CPU used is a 64bit architecture (see Appendix A). From a mathematical perspective it can be argued that double precision is preferable as it gives tighter bounds. In practice it may be more efficient to use a mixed precision approach, switching to double precision when little or no improvement is achieved using single precision. Unfortunately the double precision library supplied by Nvidia in the version of CUDA used is concluded to be unstable at the time of writing. For more on this, see Appendix C.

Numerical limits: For some problems, the threshold ϵ_f may not be attainable with the numerical precision used for some boxes. For example the algorithm may reach a point where a box cannot be split any further. Even though the threshold is not reached, the box may still contain a global minimum. Therefore, in these situations, the box is added to the list of possible solutions with a flag indicating that the threshold has not been reached.

Limitation on number of measurements, Φ: In the GPU implementations, the number of measurements Φ is limited by the number of parameters Ψ to $\Phi_{\max} = \frac{d_{x,\max}}{\Psi}$, where $d_{x,\max}$ is the maximum x-dimension of a grid of threads. The value of $d_{x,\max}$ depends on the Compute Capability of the GPU (65535 for CC < 3.0 and $2^{31} - 1$ for CC ≥ 3.0 [50]). This limitation can trivially be removed by adding additional layers of reduction in the parallel reduction scheme illustrated in Figure 5.3. This would leave $\Phi_{\max}$ limited only by the amount of memory on the GPU.

5.5 Libraries

The CPU and GPU implementations both rely on libraries for interval computations, both of which are based on the interval arithmetic library within the Boost C++ Libraries [37, 63]. The libraries are denoted $\text{Boost}_{\text{CPU}}$ and $\text{Boost}_{\text{GPU}}$

For development purposes in the present work, a Python wrapper to $\text{Boost}_{\text{CPU}}$ with a few extensions has been developed. The software package, denoted PyInt, can be found in the `code/libs/pyint` folder on the enclosed CD. To utilize the CPU resources more efficiently, a simpler and faster version of this wrapper, denoted PyInt-light was developed, where most of the functionality was moved from Python to C++ . The main reason that PyInt-light is faster is that in PyInt, each interval is stored as a Python object. This means that when an array of intervals is created, what is stored in memory is an array containing pointers to interval objects that may be scattered in memory. This makes operations on arrays of intervals inefficient. Contrary to PyInt, in PyInt-light the data is stored contiguously in C++ arrays, making operations more efficient. The fast implementation can be found in `code/libs/PyInt-light` on the enclosed CD. For the GPU implementations, a CUDA version of the Boost interval library is used. The library is originally presented in [55, ch. 9] and is now included in the Nvidia CUDA Samples pack. The library is extended to provide more functionality and higher precision. The extended library is denoted CuInt and can be found in `code/libs/cuint`. This section describes the extensions to $\text{Boost}_{\text{CPU}}$ and $\text{Boost}_{\text{GPU}}$.

5.5.1 CUDA Interval Library – CuInt

The $\text{Boost}_{\text{GPU}}$ provides most of the functionality of $\text{Boost}_{\text{CPU}}$. The library provides an interval type, support the basic operations $(+, -, \times, /)$ on two intervals, intersection of intervals and squaring of intervals. Some features are missing for the library to be useful for this project and therfore the library has been extended with the functionalities listed below.

- **Basic operations, $+, -, \times$:** Extends the original implementation with support for operations on `double`, `int` and `float` type variables. (There is no need to extend the $\text{Boost}_{\text{GPU}}$ library for division, since this feature is already implemented).

- **Compound assignment operators,** `+=`,`-=`,`×=`: Implemented to simplify coding and improve readability.
- **Contains,** `contains(`x`,`$\mathbf{Y}$`)`: Function to test whether a x is contained in the given interval $\mathbf{Y} \in \mathbb{IR}^n_*$.
- **Hull,** `hull(`$\mathbf{X}, \mathbf{Y}$`)`: The `hull` function returns the smallest box that $\mathbf{Z} \in \mathbb{IR}^n_*$ that contains both boxes $\mathbf{X}, \mathbf{Y} \in \mathbb{IR}^n_*$:

$$\texttt{hull}(X,Y)_i = \left[\min(\underline{X_i}, \underline{Y_i}),\ \max(\overline{X_i}, \overline{Y_i})\right] \text{ for } i = 0, \ldots, n-1$$

- **One-part division,** `X/Y`: The CUDA interval library shipped with the CUDA Samples supports, division by intervals that do not contain zero and two-part division (see Section 2.1.4). A simple one-part division operator, which returns the union of the results of a two-part division was implemented in the present work.. Therefore, if zero is contained in the interior of the denominator, $[-\infty, \infty]$ is returned.
- **Various non-basic math functions:**
  - `sin(X)`: Computes $\sin(X) \mid X \in \mathbb{IR}_*$
  - `cos(X)`: Computes $\cos(X) \mid X \in \mathbb{IR}_*$
  - `exp(X)`: Computes $\exp(X) \mid X \in \mathbb{IR}_*$
  - `log(X)`: Computes $\log(X) \mid X \in \mathbb{IR}_*$
  - `pow(X,i)`: Computes $X^i \mid X \in \mathbb{IR}_*, i \in \mathbb{Z}_+$

The extension of the supported types for the basic operations, the compound assignment operators and the `contains` function, are trivial implementations and are not be described further. For further details, see the source code in the folder `code/libs/cuint/`.

As stated above, the library is extended with a set of non-basic functions (sin, cos, exp, etc.). Directionally rounded versions of these do not exist in the CUDA library, and therefore, they are was implemented, in the present work, using the normal version of the given function combined the `nextafter(x,d)` C-function. The `nextafter(x,d)` function returns the next number from x in the direction of the number d that is representable by the floating point standard used. The functionality of `nextafter(x,d)` differs between Compute Capability < 3.0 and Compute Capability ≥ 3.0, affecting the results where it is used. For details, see Appendix C.

The non-basic functions above fall into two main categories; 1) those that are monotonic (exp, log, pow) and 2) those that are not (sin, cos). Enclosures of the monotonic functions are fairly simple to compute, while the non-monotonic functions require more handling of special cases.

Consider for example the real exponential function in the CUDA math library which is guaranteed to have a maximum error of 1 ULP[2] [50]. The true value y of $\exp(x)$ for some x lies somewhere in the interval between two values a, b, separated by 1 ULP, that can be represented using floating point. The value $\tilde{y}$ returned by the CUDA `exp` function can be any of these two values, a and b. To return an interval that encloses y with certainty, 1 ULP is added to either side of $\tilde{y}$. The principle is

[2]ULP: *Unit in the Last Place* – the smallest distance between two numbers that can be represented in floating point.

illustrated in Figure 5.6. The method described is trivially extended to take interval inputs in stead of real inputs. The resulting method is:

$$\exp(X) \leftarrow [\texttt{nextafter(}\exp(\underline{X})\texttt{,}-\infty\texttt{),nextafter(}\exp(\overline{X})\texttt{,}\infty\texttt{)]} \tag{5.1}$$

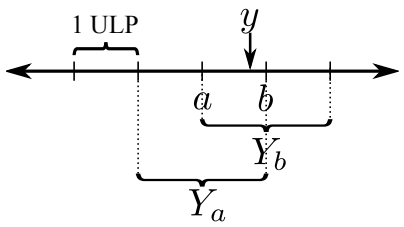


Figure 5.6: The true value y is rounded to either a or b. To assure that x is contained in the resulting interval, Y_a or Y_b is returned if y is rounded to a or b respectively.

The implementation of the non-monotonous sine and cosine functions is more complex. The implementation is based on inclusion function for the cosine, described in [64][3]. As the functions are very similar, only the implementation of the cosine is described here.

The cosine function repeats a pattern of piecewise monotonicity. It is non-increasing in the intervals $x \in [2n\pi, (2n+1)\pi]$ and non-increasing in the intervals $x \in [(2n+1)\pi, (2n+2)\pi], n \in \mathbb{Z}$. It attains its maximum and minimum values of ± 1 respectively at $x = 2n\pi$ and $x = (2n+1)\pi, n \in \mathbb{Z}$. Based on this, the inclusion function is given as:

$$\cos(X) = \begin{cases} [-1,\ 1\,] & \text{if } 1 + \left\lceil \frac{\underline{X}}{\pi} \right\rceil \leq \frac{\overline{X}}{\pi} \\ [-1,\ b\,] & \text{if } \left\lceil \frac{\underline{X}}{\pi} \right\rceil \leq \frac{\overline{X}}{\pi} \text{ and } \left\lceil \frac{\underline{X}}{\pi} \right\rceil \text{ is uneven} \\ [\ a,\ 1\,] & \text{if } \left\lceil \frac{\underline{X}}{\pi} \right\rceil \leq \frac{\overline{X}}{\pi} \text{ and } \left\lceil \frac{\underline{X}}{\pi} \right\rceil \text{ is even} \\ [\ a,\ b\,] & \text{otherwise} \end{cases} \tag{5.2}$$

$$a = \min\left(\cos(\underline{X}),\ \cos(\overline{X})\right)$$

$$b = \max\left(\cos(\underline{X}), \cos(\overline{X})\right)$$

The cos function has a guaranteed maximum error of 1 ULP [50], and so inclusion isotonicity is guaranteed in a way similar to the exp function in (5.1).

Exponentiation ($X^n, X \in \mathbb{IR}_*, n \in \mathbb{N}_0$) is implemented for non-negative integer exponents n. As specified in (2.7) on page 6, an interval exponentiation method must handle a set of special cases. These cases use real exponentiation, which is implemented based on the method described in [65], listed in Algorithm 5.5, in downward and upward rounded versions. When the base is non-negative, this is easily implemented by using either downward or upward rounding multiplication in line 4 and 6 of Algorithm 5.5. When the base is negative and the exponent is uneven, the value of y in Algorithm 5.5 changes sign and thus requires the rounding mode, used in the multiplication in line 4, to be changed to obtain a correct end result.

[3] In the formulation given here, use of the modulo functions is replaced with checks for evenness or unevenness, as the behavior of the modulo function for negative numbers is implemented differently in different programming languages, which may lead to confusion. For an example code showing how the modulo function differs between Python and C++ , see the code in the folder `code/misc/modulo/` on the enclosed CD

Using different rounding modes in different iterations of the loop is inconvenient. Instead, logic is implemented in the interval exponentiation method that avoids this. The implementation of the interval exponentiation method, which closely resembles the implementation in the Boost CPU interval library, is listed in Algorithm 5.6. As squaring is often used, square(X) is implemented as a fast special case of the pow function.

Algorithm 5.5 Real-valued exponentiation method for non-negative integer exponents [65].

Input:
Base $x \in \mathbb{R}$.
Exponent $n \in \mathbb{N}_0$.

$y \leftarrow 1$
$p \leftarrow x$
while n > 0 **do**
  if n is uneven **then** $y \leftarrow y \cdot p$ **end if**
  $n \leftarrow \lfloor n/2 \rfloor$
  $p \leftarrow p \cdot p$
end while
return y

Algorithm 5.6 Interval exponentiation method for non-negative integer exponents.

Input:
Base $X = [x_\ell, x_u] \in \mathbb{IR}_*$.
Exponent $n \in \mathbb{N}_0$.

if $x_\ell < 0$ and $x_u < 0$ **then**
  $a \leftarrow$ `pow_dn`$(-x_u, n)$
  $b \leftarrow$ `pow_up`$(-x_\ell, n)$
  if n is uneven **then**
    $y \leftarrow [-b, -a]$
  else
    $y \leftarrow [a, b]$
  end if
else if $x_\ell < 0$ and $x_u > 0$ **then**
  $x_{\max} \leftarrow \max(-x_\ell, x_u)$
  $a \leftarrow$ `pow_up`$(x_{\max}, n)$
  $y \leftarrow [0, a]$
else
  $a \leftarrow$ `pow_dn`(x_ℓ, n)
  $b \leftarrow$ `pow_up`(x_u, n)
  $y \leftarrow [a, b]$
end if
return y

Notes: The `pow_dn` and `pow_up` functions denote the method listed in Algorithm 5.5 using downward and upward rounding multiplication, respectively.

5.5.2 Python Interval Library – PyInt

The PyInt library, wraps the interval class from the $\text{Boost}_{\text{CPU}}$ library to the PyInt interval class. To support matrices and vectors the Numpy[4] `ndarray` class is subclassed to an interval array class. Further, some support functions such as a special interval array constructor function and a special dot function are developed. The structure of PyInt is illustrated in Figure 5.7.

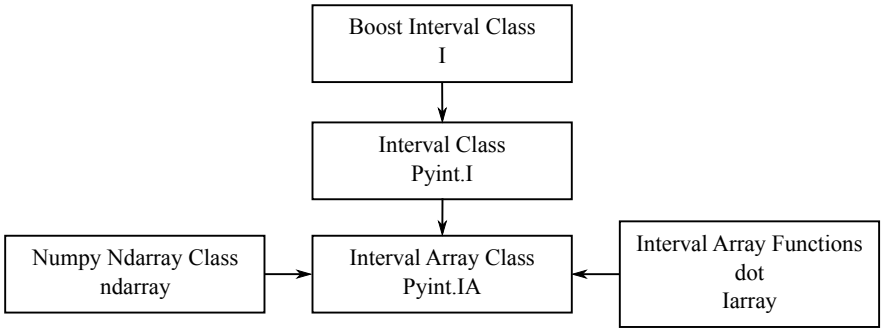


Figure 5.7: Structure of PyInt. The interval class PyInt.I wraped from the Boost interval class. An interval array class, extends PyInt.I to arrays, and subclasses the Numpy Ndarray class.

The Boost interval class contains interval arithmetic functionality and the PyInt interval class wraps this functionality to Python. The operations and comparisons available are the same as in the CUDA interval library in Section 5.5.1.

The PyInt interval array (`Indarray`) class subclasses the Numpy `ndarray` class. This exposes all the functionality of the `ndarray` class to the `Indarray` class. This functionality is extended with some interval specific functions for computing the midpoint, volume, width, etc. of an interval array. Matrix-vector and matrix-matrix dot products are implemented to handle multiplication between interval matrices/vectors and real valued matrices/vectors. (see `code/libs/pyint/pyint.py`)

A problem with PyInt is that the data in interval arrays is not stored contiguously. This results in operations such as the dot product being unnecessarily slow because the CPU cache is not used efficiently. Therefore, PyInt is functionally adequate and convenient for algorithm development, but too inefficient for a fair comparison with GPU code. To create a more fair CPU library for use in performance comparisons between the CPU and GPU, a lightweight version of PyInt, named PyInt-light was implemented in the present work. In PyInt-light the basic operations $(+, -, \times, /)$ and the non-basic math functions listed in Section 5.5.1 are unavailable from within Python. Instead the functions used in the global optimization algorithm is programmed in C++ and called from Python, much like the CUDA implementations. Data in interval arrays is stored contiguously in PyInt-light making the overall optimization algorithm faster.

5.6 Summary

In this chapter, different strategies for parallelization of the global optimization were analyzed with focus on the parameter estimation problem. It was selected to focus on parallelizing the inner loops of the sub-problems in the branch & bound tree. Based on this, a GPU accelerated implementation, a sequential CPU implementation, some variations on these implementations and the interval software libraries used were described.

[4]Numpy is a Python package for scientific computing [66].

Chapter 6

Benchmarking

This chapter describes the test suite used to benchmark the implementations presented in Chapter 5 and the results of the benchmarks with the purpose of comparing the speedup gained by using the GPU accelerated implementation over the CPU reference implementation. General considerations regarding the benchmarks are treated in Section 6.1. In Sections 6.2-6.5 each test problem and the benchmarks run on the problem are described along with the results of the benchmarks, a comparison of the results for the GPU and CPU and a discussion of the results. Finally a summary is given in Section 6.6.

6.1 Test Suite

In order to construct a meaningful scheme for testing the performance of the implementations, a couple of considerations must be taken into account:

a) The time taken to solve a specific problem[1] consists of two factors; a base execution time t_b and an uncertainty in the measurement, t_n, such that the total execution time is $t_\mathrm{e} = t_\mathrm{b} + t_\mathrm{n}$. For problems that are solved quickly, t_n is expected to make a relatively larger contribution to t_e than for problems that take a long time to solve. Because of this, and because the time available for testing is limited in this work, more measurements of the execution time are taken for the problems where t_e is small.

b) For problems of the same type, but with different data sets and thereby different solutions, the execution time may vary significantly. This is because the scheme used for picking an interval to process may turn out to be especially efficient (or inefficient) at producing a low upper bound of the objective function, as discussed in Section 3.3.1. Therefore, the time taken to solve similar problems with different solutions may not be comparable.

The benchmarking has two main objectives: 1) To compare the variations of the GPU implementation, described in Section 5.2.3, with each other and compare the variations of the CPU implementation, described in Section 5.3, with each other in order to find the fastest variation. 2) To compare the fastest variation of the GPU implementation with the fastest variation of the CPU implementation and measure the speedup.

[1] A specific problem is a problem of a specific structure and with specific input data and thereby a specific solution.

To find the fastest among the variations of the CPU and GPU implementations, the execution time of each variation is measured for a number of instances of a set of test problems. For some of these problems, the number of parameters, Ψ, is fixed, and for other it can be varied. For all the problems, the number of measurements, Φ, can be varied. Each specific problem is solved a number of times, such that the accumulated execution time is at least 100 seconds, and the mean value of the execution time is computed. As limited time is available for benchmarking, a time limit of 10 minutes for solving each test problem is set. Additionally, the number of measurements used for testing the CPU and GPU accelerated implementations are different.

Note that on plots of results in Sections 6.2-6.5, lines are drawn between data points. These lines are not necessarily representative for the execution time/speedup between the data points, and should therefore be regarded as a help for the eye rather than a clear indication of the execution time/speedup between data points. For some of the variations of the implementations, the benchmark does not complete within the 10 minute time limit. In these cases, the timeout data points are not plotted. Tables containing the data points for each of the figures in Sections 6.2-6.5 can be found in Appendix F.

6.2 Colorimetric Determination of Formaldehyde (FDEHYDE)

6.2.1 Problem

This test problem, a four parameter chemistry problem from the collection in [4], originates from [67] and concerns optimization of the absorbance response for formaldehyde as a function of the amount of two other substances added.

The true value of the parameters is selected to be the same as in [4], and the measurements are simulated using measurement points (representing substance volumes) that lie in the same ranges as in [4]. The search region used in [4] is chosen such that all parameters are in the range $[10^{-4}, 10^{5}]$. This choice seems unreasonably large, as the starting values are all below 20. Using the search region from [4], the execution time is prohibitively long, and therefore a somewhat smaller region is used here.

As the measurements are simulated, noise can be added. The purpose of this test problem is to evaluate the performance of the implementations on a simulated real world problem with a varying number measurements, Φ. In Table 6.1 the details of this test problem and the experiments for this problem is specified.

FDEHYDE		
Model:	$m(\mathbf{p}, \mathbf{x}) = p_0 \exp\left(-\frac{\left(\frac{x_1}{x_0+x_1+2} - p_1\right)^2}{2p_2^2}\right) \frac{2 - 2\exp(-p_3 x_0)}{x_0 + x_1 + 2}$	
Experiment(s):		
		Experiment 1 - Variable Φ
	Ψ	4
	$\mathbf{p}_{\text{true}}$	$[1.55, 0.57, 0.07, 22.00]^{\text{T}}$
	Φ_{GPU}	100, 2000, 4000,...,16000
	Φ_{CPU}	100, 200,300,...,900
	Measurements are simulated. Measurement points are equally distanced in the ranges $x_0 \in [0.1, 3.0]$ and $x_1 \in [0.2, 6.0]$.	
Parameter bounds:	$\left[[10^{-4}, 10^{1}], [10^{-4}, 10^{1}], [10^{-4}, 10^{1}], [10^{-4}, 10^{2}]\right]^{\text{T}}$	
Tolerances:	$\varepsilon_p = 0.001$ $\varepsilon_f = 0.001$	
Noise:	Gaussian i.i.d., $\sigma = 0.001$	
Initial upper bound:	$b = \infty$	
Details/notes:	A test problem from the collection in [4]. Originally from [67].	

Table 6.1: FDEHYDE - model function, parameters and search range.

6.2.2 Experiment 1 - Variable number of measurements, Φ

GPU: In Figure 6.1 the results for the variations of the GPU implementation are shown. From Figure 6.1 it is seen that the `PF` implementation gives the best performance. It is also evident that variations that include the `F` element perform significantly better than those that do not. The scaling of the execution time, with respect the number of measurements, is approximately linear.

CPU: In Figure 6.2 the CPU results are plotted. The variations `IPF`, `PF` and `IF` perform approximately well, with the `PF` variation being slightly faster than the other two. Again it is evident that the variations that include the `F` element perform significantly better than those that do not. The scaling of the execution time for these four variations is approximately linear.

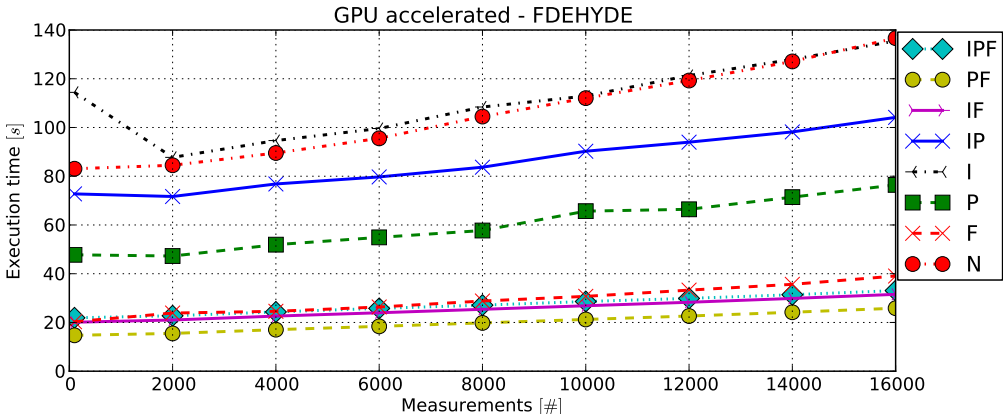


Figure 6.1: FDEHYDE: Experiment 1. Execution time for GPU variations. Variable number of measurements, $\Phi = 100, 2000, 4000..., 16000$. Fixed number of parameters, $\Psi = 4$. Detailed results can be found in Appendix F on page 86 in Table F.1.

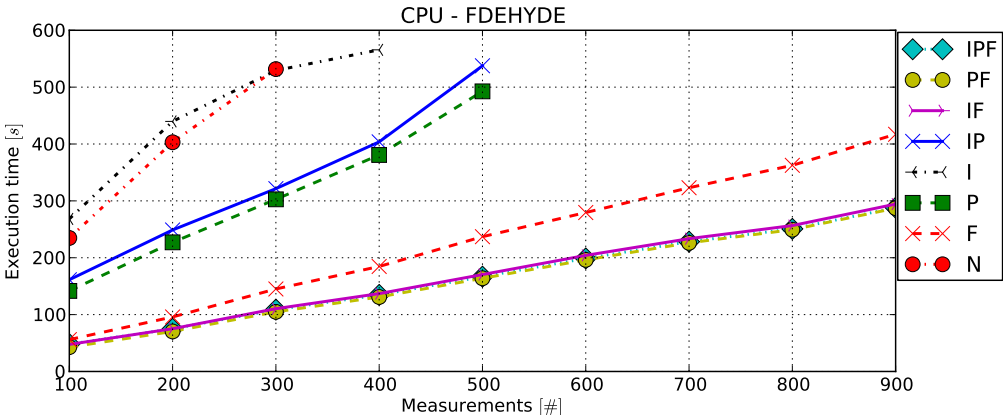


Figure 6.2: FDEHYDE: Experiment 1. Execution time for CPU variations. Variable number of measurements, $\Phi = 100, 200, ..., 900$. Fixed number of parameters, $\Psi = 4$. Missing data points for some of the variations indicate that the benchmark did not complete within the 10 minute time limit. Detailed results can be found in Appendix F on page 86 in Table F.1.

6.2.3 GPU and CPU Comparison

The fastest variation for both the GPU and CPU implementations is `PF`. In Figure 6.3 and in Figure 6.4 the execution times and speedup of the GPU implementation over the CPU implementation is plotted for $\Phi = 100, 200, ..., 1800$. The time used by the GPU to solve the problems is nearly constant, while the time used by the CPU increases approximately linearly with the number of measurements. The minimum speedup, at $\Phi = 200$, is approximately 5 times and the maximum speedup, achieved at $\Phi = 1800$ is approximately 37 times.

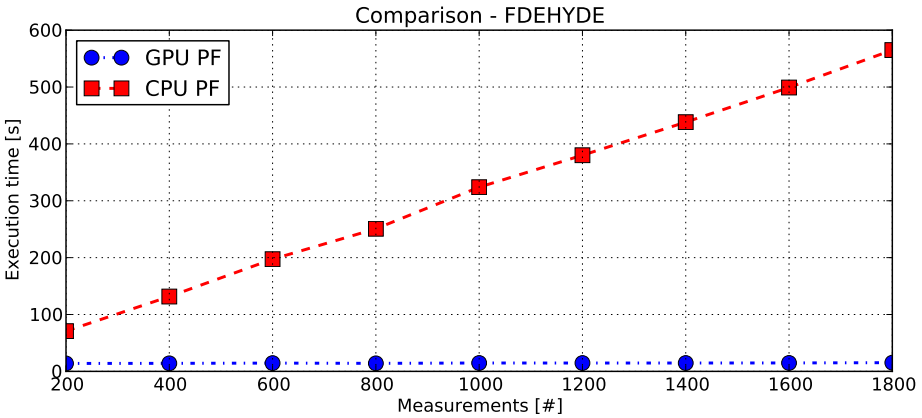


Figure 6.3: FDEHYDE: Experiment 1. Execution time for GPU-`PF` and CPU-`PF` variations. Variable number of measurements, $\Phi = 100, 200, ..., 900$. Fixed number of parameters, $\Psi = 4$.

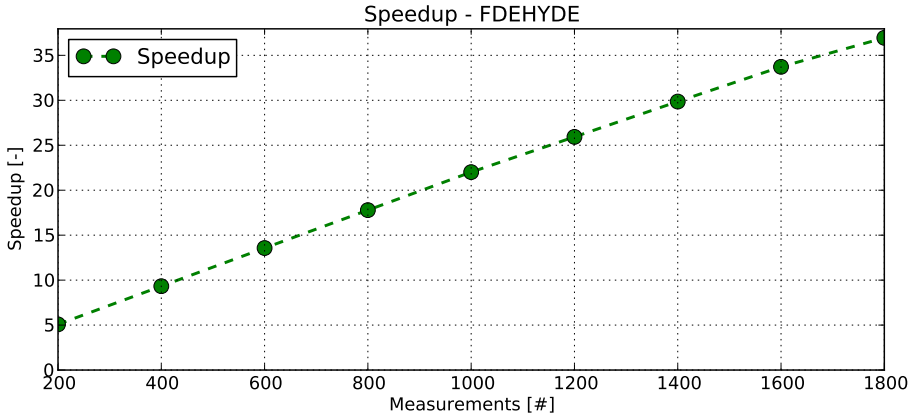


Figure 6.4: FDEHYDE: Experiment 1. Speedup, GPU vs. CPU for GPU-`PF` and CPU-`PF` variations. Variable number of measurements, $\Phi = 100, 200, ..., 1800$. Fixed number of parameters, $\Psi = 4$.

6.2.4 Discussion

It shows from the comparison that the GPU implementation has a significant advantage over the GPU implementation for large numbers of measurements. It is expected that the speedup converges to a stable level at some point, but in the range shown this effect is not evident. The fact that the execution time for the GPU implementation is almost constant indicates that the parallel resources are not being occupied. A measurement of the GPU compute utilization using the Nvidia Visual Profiler at $\Phi = 1800$ shows an compute utilization of 15.8 %, which supports this.

Common for the CPU and GPU implementations is that the variations that include the `F` element are all among the best performing.

6.3 Synthetic Sinusoidal/Polynomial Problem (POLYCOS)

6.3.1 Problem

This problem is designed by the authors to be variable in the number of parameters. It exhibits a number of local minima and one global minimum. The measurements are simulated. The purpose is to have a test problem where both the number of parameters and the number of measurements can be swept. In Table 6.2 the details of this test problem and the experiments for this problem are specified.

POLYCOS			
Model:	$m(\mathbf{p}, x) = \sum_{k=0}^{\Psi-1} \left(\alpha \cdot (x - p_k)^2 - \cos(\beta \cdot (x - p_k))\right)$ where $\alpha = 0.05$ and $\beta = 3$		
Experiment(s):			
		Experiment 1	**Experiment 2**
	Ψ	2,3,4	3
	$\mathbf{p}_{\text{true}}$	$[-4, -2, 2, 4]^T$	$[-4, -2, 2]^T$
	Φ_{GPU}	1024	$100, 2000, 4000, ..., 16000$
	Φ_{CPU}	1024	$500, 1000, 1500, ..., 4500$
	Measurements are simulated with measurement points equally distanced in the range $x \in [-2\pi, 2\pi]$.		
Parameter bounds:	$[-2\pi, 2\pi]$ for all parameters		
Tolerances:	$\varepsilon_p = 0.001$ $\varepsilon_f = 0.001$		
Noise:	$\sigma = 0$		
Initial upper bound:	$b = \infty$		
Details/notes:	The constant $\beta \in \mathbb{R}_+$ controls the objective function value in the local minima and $\alpha \in \mathbb{R}_+$ controls the spacing of the local minima. The initial search region for this test problem is pre-processed as described in Appendix E.		

Table 6.2: POLYCOS - model function, parameters and search range.

6.3.2 Experiment 1 - Variable number of parameters, Ψ

GPU: The results of experiment 1 are plotted in Figure 6.5. In this experiment, the `P`, `I` and `N` variations are not able to solve the problem with more than two parameters, within the time limit of 10 minutes. Again, the fastest variation is `PF` closely followed by the `IPF` variation.

CPU: In Figure 6.6 the results of experiment 1 of the CPU implementations are shown. It can be observed that none of the variations are able to solve for more than 3 parameters within the 10 minute time limit. The fastest variation is `IPF`.

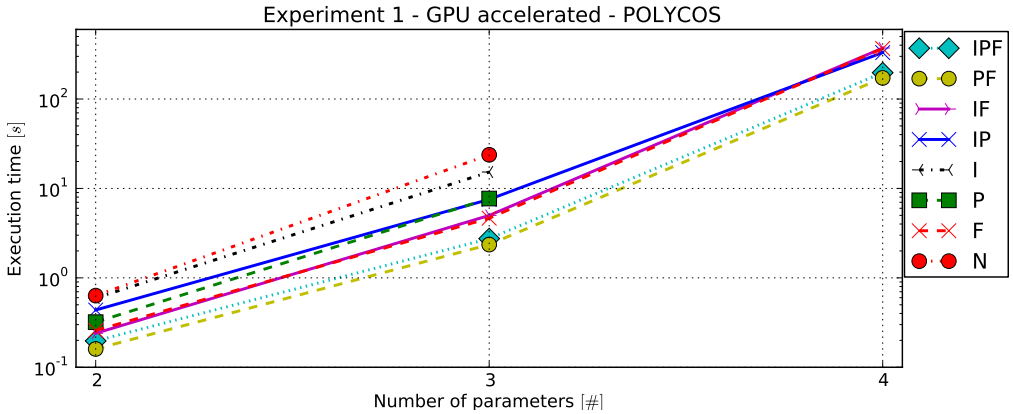


Figure 6.5: POLYCOS: Experiment 1. Execution time for GPU variations. Fixed number of measurements, $\Phi = 1024$. Variable number of parameters, $\Psi = 2, 3, 4$. The variation that does not have measurement points for all parameters has reached time out of 10 minutes. Detailed results can be found in Appendix F on page 86 in Table F.3.

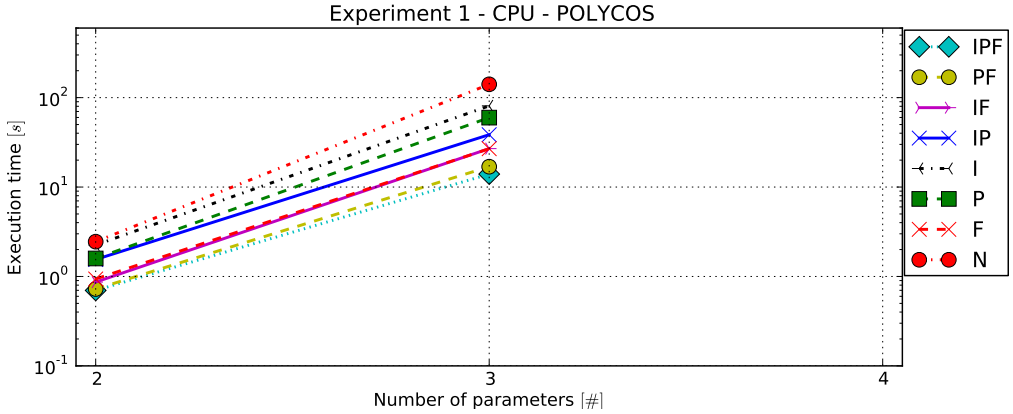


Figure 6.6: POLYCOS: Experiment 1. Execution time for CPU variations. Fixed number of measurements, $\Phi = 1024$. Variable number of parameters, $\Psi = 2, 3, 4$. The variation that does not have measurement points for all parameters has reached time out of 10 minutes. Detailed results can be found in Appendix F on page 86 in Table F.4.

6.3.3 Experiment 2 - Variable number of measurements, Φ

GPU: In this experiment, the results for the GPU variations are illustrated in Figure 6.7. It can again be seen that `PF` is the fastest variation, closely followed by `IPF`. It is worth noting that the simple `N` variation outperforms the `I` variation for problem sizes 12000, 14000 and 16000. Again in this experiment the variations including element `F` outperforms the others.

CPU: In Figure 6.8 the results of experiment 2 of the CPU variations are illustrated. It can be seen that the `IPF` variation is the fastest in this case, closely followed

by the PF variation. The N variation is not able to solve the problem for more than 3500 measurements within the 10 minute time limit.

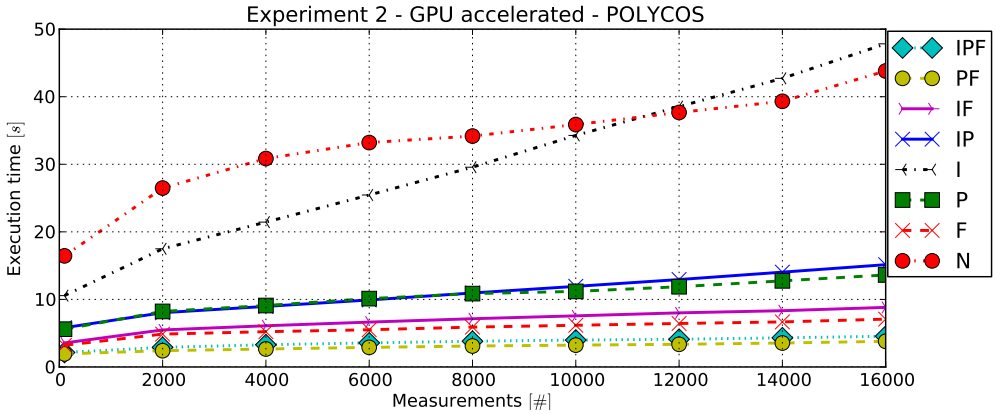


Figure 6.7: POLYCOS: Experiment 2. Execution time for GPU variations. Variable number of measurements, $\Phi = 100, 2000, 4000, ..., 16000$. Fixed number of parameters, $\Psi = 4$. Detailed results can be found in Appendix F on page 86 in Table F.5.

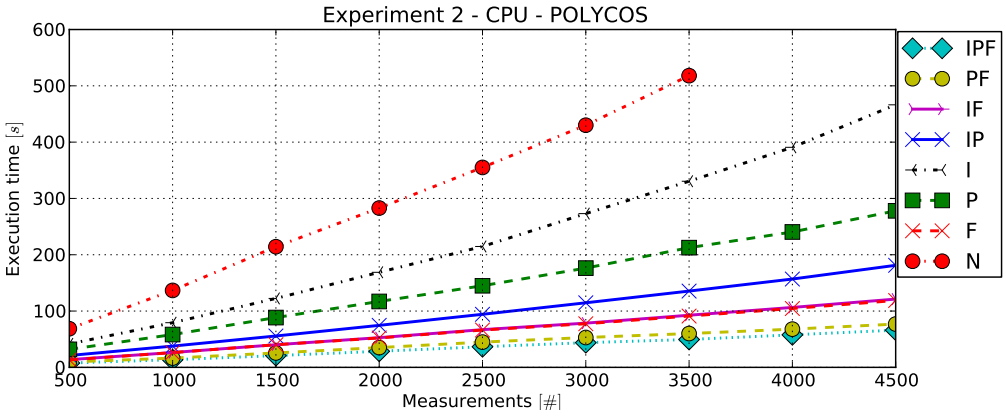


Figure 6.8: POLYCOS: Experiment 2. Execution time for CPU variations. Variable number of measurements, $\Phi = 500, 1000, 1500, ..., 4500$. Fixed number of parameters, $\Psi = 3$. Missing data points for some of the variations indicate that the benchmark did not complete within the 10 minute time limit. Detailed results can be found in Appendix F on page 86 in Table F.6.

6.3.4 GPU and CPU Comparison

For the POLYCOS problem, two comparisons are made; one for varying number of parameters and one for a varying number of measurements. In both cases, the PF variation is the fastest among the GPU variations and the IPF variation is the fastest among the CPU variations.

The results for varying number of parameters, plotted in Figures 6.9 and 6.10, show speedups of 4.36 to 5.93 times for 2 and 3 parameters, respectively. In Figures 6.11 and 6.12 the results for varying number of measurements are shown. The execution time for the GPU implementation is close to constant compared to the execution time for the CPU, which increases approximate linearly with the number of measurements. The measured speedup is 1.43 times for $\Phi = 100$ and increases to 60.4 times for $\Phi = 12000$.

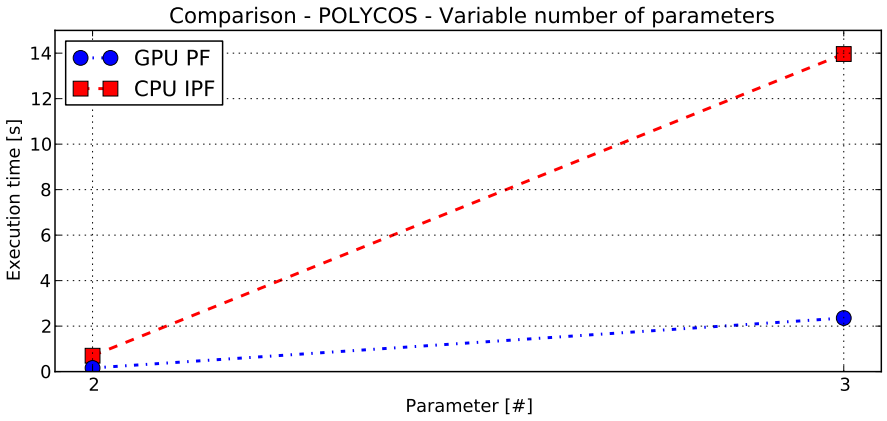


Figure 6.9: POLYCOS: Experiment 1. Execution time for GPU-`PF` and CPU-`IPF` variations. Fixed number of measurements, $\Phi = 1024$. Variable number of parameters, $\Psi = 2, 3$.

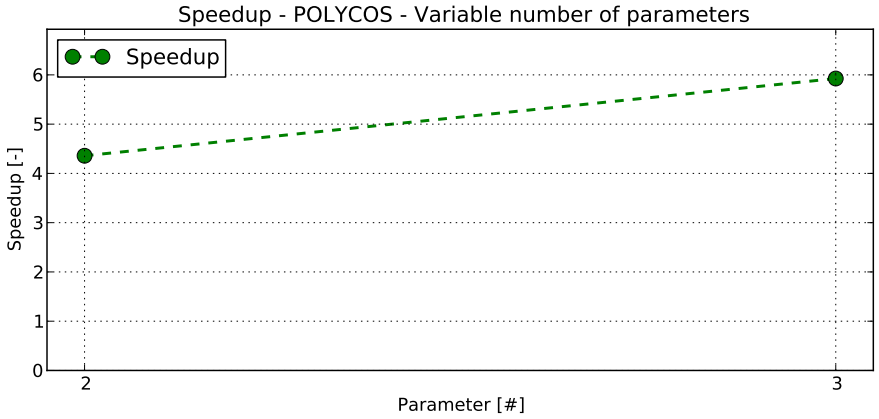


Figure 6.10: POLYCOS: Experiment 1. Speedup, GPU vs. CPU for GPU-`PF` and CPU-`IPF` variations. Fixed number of measurements, $\Phi = 1024$. Variable number of parameters, $\Psi = 2, 3$.

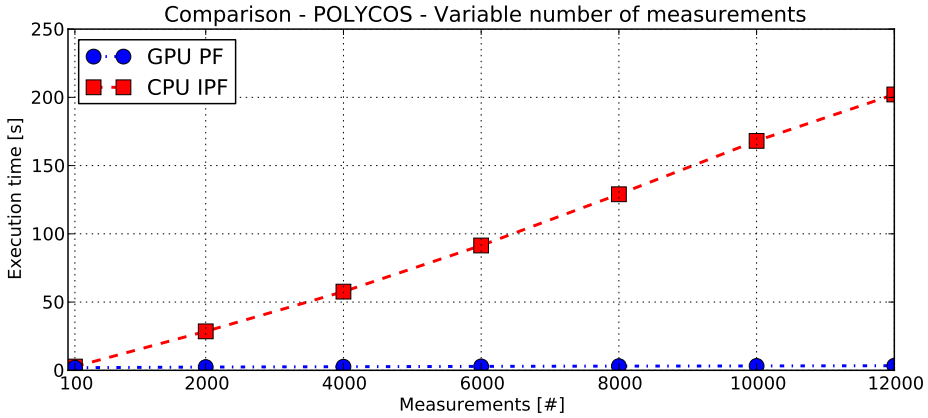


Figure 6.11: POLYCOS: Experiment 2. Execution time for GPU-`PF` and CPU-`IPF` variations. Variable number of measurements, $\Phi = 100, 2000, 4000, ..., 12000$. Fixed number of parameters, $\Psi = 3$.

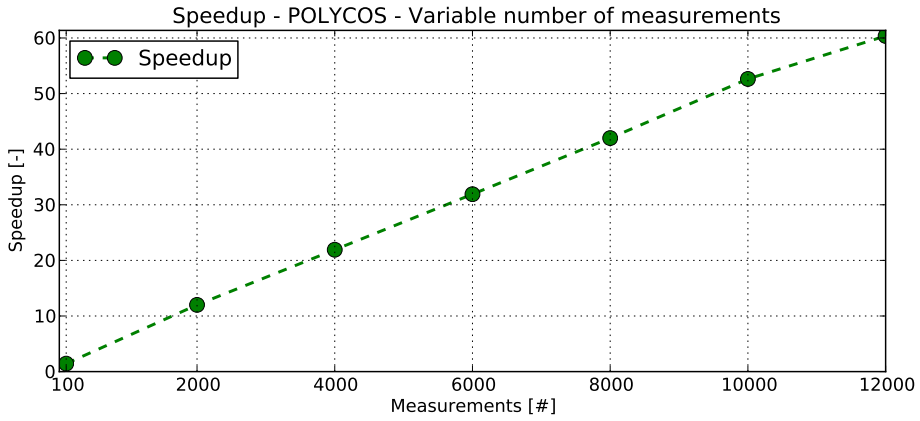


Figure 6.12: POLYCOS: Experiment 2. Speedup, GPU vs. CPU for GPU-`PF` and CPU-`IPF` variations. Variable number of measurements, $\Phi = 100, 2000, 4000, ..., 12000$. Fixed number of parameters, $\Psi = 3$.

6.3.5 Discussion

For this test problem the execution time increases significantly for more than 3 parameters. The results for the GPU implementation indicate exponential growth of the execution time. This is also evident in the fact that none of the CPU implementations were able to solve for 4 parameters within the 10 minute time limit. The comparison for variable number of parameters therefore only includes $\Psi = 2$ and $\Psi = 3$, which makes it difficult to conclude on the result, other than the fact that a small speedup is attained.

For the comparison for varying number of measurements, the situation is the same as for the FDEHYDE problem. The execution time for the GPU is almost constant and thus the speedup increases approximately linearly with the number of measurements. A profiling of the GPU implementation at $\Phi = 12000$ and $\Psi = 3$ reveals a compute utilization of only 26.8 %. This shows that the GPU resources are not being fully utilized.

6.4 Synthetic Polynomial Problem (2D_POLY)

6.4.1 Problem

This is a simple two-parameter synthetic model function, designed by the authors such that the objective function has saddle points and two global minima. The purpose of this test problem is to illustrate a simple 2 dimensional problem. In Table 6.3 the details of this test problem and the experiments for this problem is specified.

2D_POLY	
Model:	$m(\mathbf{p}, x) = -p_0^2 x - p_1^2 x + 0.01 p_0^4 x + 0.01 p_1^4 x + 10 p_1$
Experiment(s):	
	Experiment 1
	Ψ 2
	$\mathbf{p}_{\text{true}}$ $[3, 2]^{\text{T}}$
	Φ 1000, 2000, 4000,...,16000
	Measurements are simulated. Inputs are equally distanced in the range $x \in [-8, 8]$.
Parameter bounds:	$[[-8, 8], [-8, 8]]^{\text{T}}$
Tolerances:	$\varepsilon_p = 0.001$ $\varepsilon_f = 0.001$
Noise:	$\sigma = 0$
Initial upper bound:	$b = \infty$

Table 6.3: 2D_POLY - model function, parameters and search range

6.4.2 Experiment 1 - Variable number of measurements, Φ

GPU: In Figure 6.13 the results for the variations of the GPU implementation are shown. It can be seen that the execution times for this problem are significantly lower than for the other test problems. It can be observed that `PF` and `IPF` are almost equal in performance. `PF` is slightly faster than `IPF` for under 10000 measurements, and the opposite is seen for the higher numbers.

CPU: In Figure 6.14 the results of the CPU implementations are plotted. For this test problem, the `IPF` variation is the fastest, followed by the `IP` variation. The scaling of execution time is approximately linear with respect to the number of measurements.

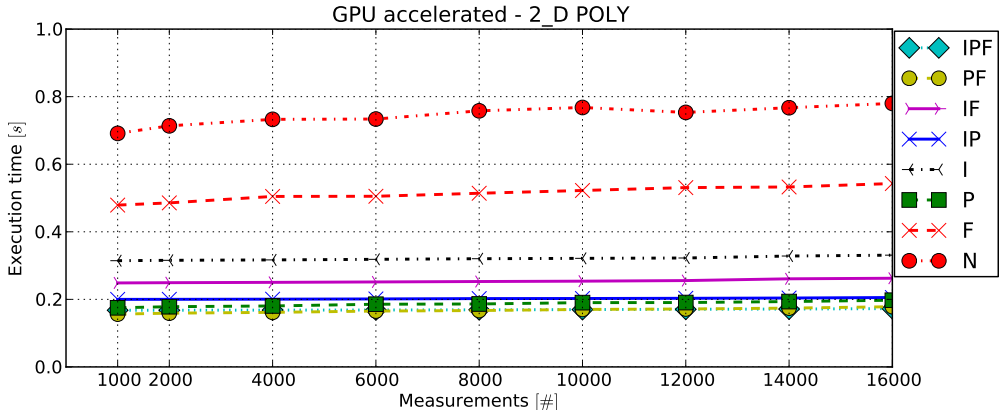


Figure 6.13: 2D_POLY: Experiment 1. Execution time for GPU variations. Variable number of measurements, $\Phi = 1000, 2000, 4000..., 16000$. Fixed number of parameters, $\Psi = 2$. Detailed results can be found in Appendix F on page 86 in Table F.7.

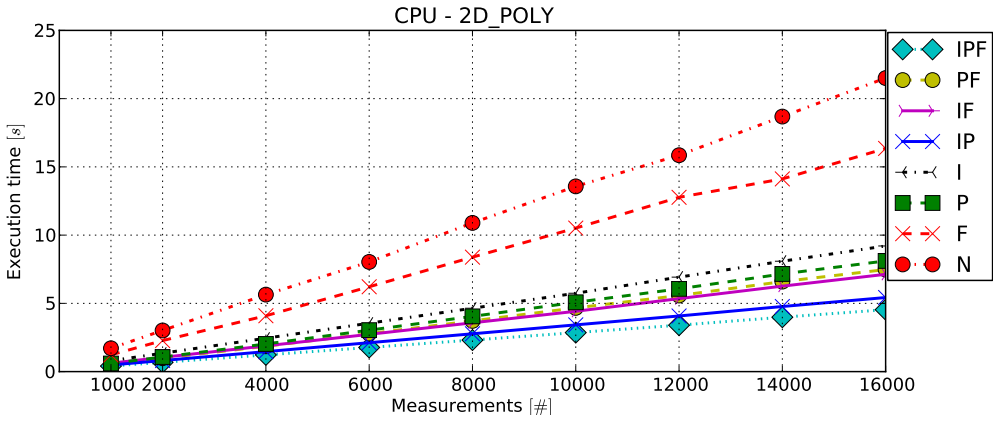


Figure 6.14: 2D_POLY: Experiment 1. Execution time for CPU variations. Variable number of measurements, $\Phi = 1000, 2000, 4000..., 16000$. Fixed number of parameters, $\Psi = 2$. Detailed results can be found in Appendix F on page 86 in Table F.8.

6.4.3 GPU and CPU Comparison

For the GPU implementation, the `IPF` and `PF` variations perform approximately equally well. Which one is slightly faster differs with the number of measurements. For this comparison, the `IPF` variation is selected. For the CPU implementation the situation is not as ambiguous. Here the `IPF` variation is the fastest, and is thus used in the comparison.

In Figures 6.15 and 6.16 execution times and speedup is shown. The execution time for the GPU implementation is approximately constant, while it increases approximately linearly for the CPU implementation. The measured speedup ranges from 2.42 times at $\Phi = 1000$ to 26.3 at $\Phi = 16000$.

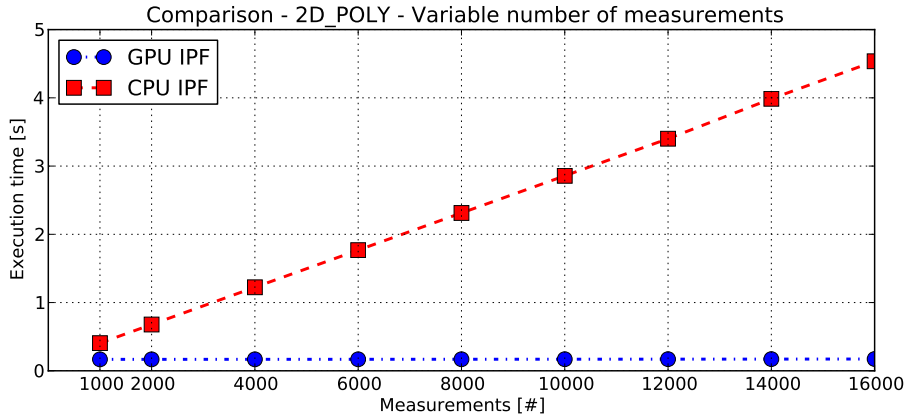


Figure 6.15: 2D_POLY: Experiment 1. Execution time for GPU-`IPF` and CPU-`IPF` variations. Variable number of measurements, $\Phi = 1000, 2000, ..., 16000$. Fixed number of parameters, $\Psi = 2$.

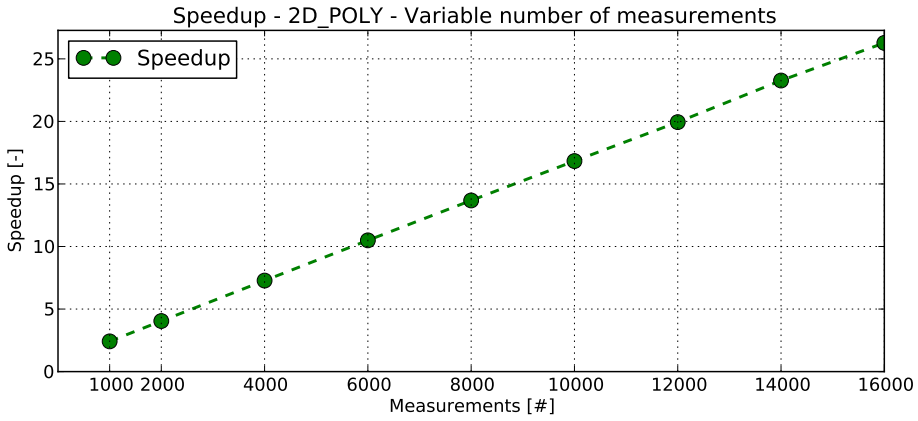


Figure 6.16: 2D_POLY: Experiment 1. Speedup, GPU vs. CPU for GPU-`IPF` and CPU-`IPF` variations. Variable number of measurements, $\Phi = 1000, 2000, ..., 16000$. Fixed number of parameters, $\Psi = 2$.

6.4.4 Discussion

This test problem has proven very easy for the algorithm to solve, and both the CPU and GPU implementation are able to do so below 5 seconds for the best variations. Once again, like for the FDEHYDE and POLYCOS test problems, the execution time for the GPU implementations increases very little with the number of measurements, which points to low utilization of the GPU resources. A profiling of the GPU implementation for $\Phi = 16000$ shows a compute utilization of 8.3%.

6.5 Sinusoidal Modeling (SINUSOIDAL)

6.5.1 Problem

Sinusoidal modeling is widely used within speech analysis and synthesis e.g. in [20, 21]. The number parameters can be varied by adding extra tones to the model. For each tone added to the model, three parameters are added. The data used is simulated.

In [18] it is stated that: "*For a highly oscillatory function f, our algorithm could be prohibitively slow*". This is the case for this test problem, and since the algorithm used in the present work resembles the algorithm in [18], it is expected that the problem is hard to solve. In Table 6.4 the details of this test problem and the experiments for this problem is specified.

SINUSOIDAL

Model:

$$m(\mathbf{p}, x) = \sum_{k=0}^{\Omega-1} p_{(3k+0)} \sin\left(p_{(3k+1)}x + p_{(3k+2)}\right)$$

where Ω is the model order (number of tones).

Experiment(s):

	Experiment 1	**Experiment 2**
Ψ	3,6	6
$\mathbf{p}_{\text{true}}$	$[0.3, 25.13, 4,$ $0.8, 37.70, 2]^T$	$[0.3, 25.13, 4,$ $0.8, 37.70, 2]^T$
Φ_{GPU}	256	$200, 300, ..., 600$
Φ_{CPU}	200	$150, 200, ...350$

Measurements are simulated. Sampling frequency $f_s = 400$. Inputs are equally distanced in the range $x \in \left[0,\ \Phi \cdot \frac{1}{f_s}\right]$.

Parameter bounds:

$[[0.1,\ 1]\,,[0.628,\ 62.83]\,,[0,\ 6.28]\,,[0.1,\ 1]\,,$
$[0.63,\ 62.83]\,,[0,\ 6.28]]^{\mathrm{T}}$

Tolerances:

$\varepsilon_p = 0.001$
$\varepsilon_f = 0.001$

Noise:

$\sigma = 0$

Initial upper bound:

$b = \infty$

Details/notes:

The problem in it self does not specify the order of the tones. This means that the algorithm will find $\Omega!$ solutions where the only difference is the ordering of the tones. As it is not desirable to spend time searching for several solutions that are basically equal, the search region is preprocessed as described in Appendix E.

Table 6.4: SINUSOIDAL - model function, parameters and search range.

6.5.2 Experiment 1 - Variable number of parameters, Ψ

GPU Table F.9 and Figure 6.17 show the results of experiment 1 for the variations of the GPU implementation. The `PF` variation performs better than the other implementations. The variations `IP`, `P`, `N` and `I` are not able to solve for 6 parameters within the time limit of 10 minutes. Once again it is evident that the variations that include the `F` element outperform those that do not.

CPU: In Table F.10 and Figure 6.18 the results of experiment 1 of the CPU implementations are detailed and illustrated. Only the `IPF` and `PF` implementations are able to solve the 6 parameter problem within the time limit. Among these two, the `PF` variation is the fastest.

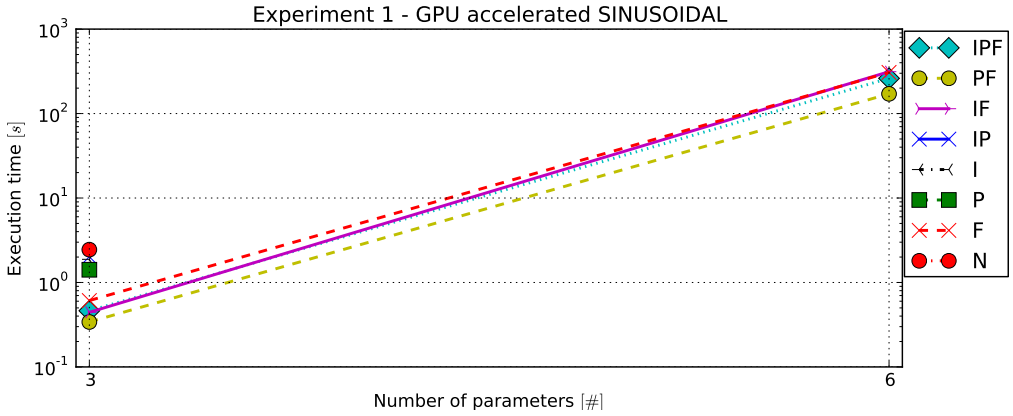


Figure 6.17: SINUSOIDAL: Experiment 1. Execution time for GPU variations. Fixed number of measurements, $\Phi = 256$. Variable number of parameters, $\Psi = 3, 6$. The variation that does not have measurement points for all parameters has reached time out of 10 minutes. Detailed results can be found in Appendix F on page 86 in Table F.9.

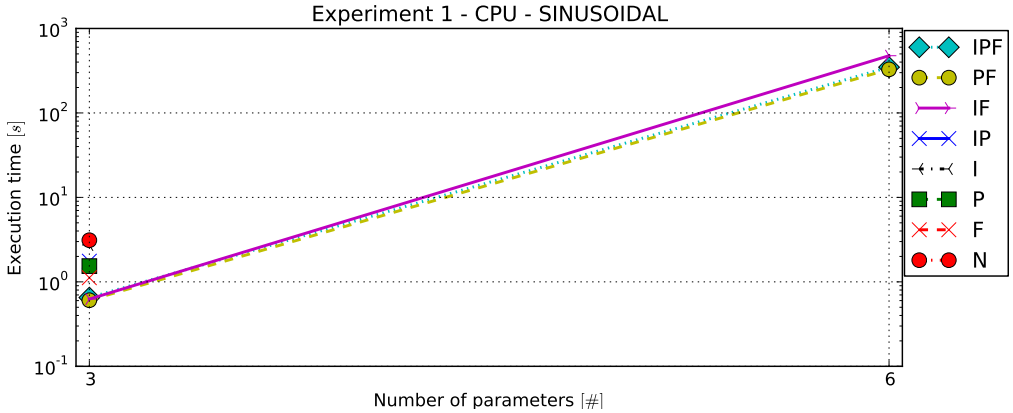


Figure 6.18: SINUSOIDAL: Experiment 1. Execution time for CPU variations. Fixed number of measurements, $\Phi = 200$. Variable number of parameters, $\Psi = 3, 6$. The variation that does not have measurement points for all parameters has reached time out of 10 minutes. Detailed results can be found in Appendix F on page 86 in Table F.10.

6.5.3 Experiment 2 - Variable number of measurements, Φ

GPU: In Table F.11 and Figure 6.19 the results of experiment 2 for the variations of the GPU implementation are shown. It can be seen that none of the variations

`IP`, `P`, `N` and `I` are able to solve the problem within the time limit for any of the measurement sizes used. Again `PF` is the fastest implementation.

CPU: In Table F.12 and Figure 6.20 the results of experiment 2 for the variations of the CPU implementation are detailed and illustrated. Only `IPF` and `PF` are able to solve for any of these problem sizes. Again, the `PF` variation performs best.

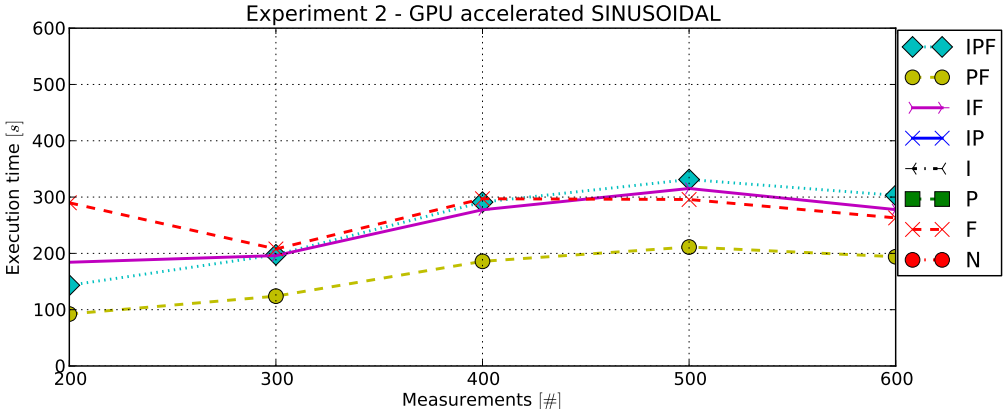


Figure 6.19: SINUSOIDAL: Experiment 2. Execution time for GPU variations. Variable number of measurements, $\Phi = 200, 300, ..., 600$ Fixed number of parameters, $\Psi = 6$. Missing data points for some of the variations indicate that the benchmark did not complete within the 10 minute time limit. Detailed results can be found in Appendix F on page 86 in Table F.11.

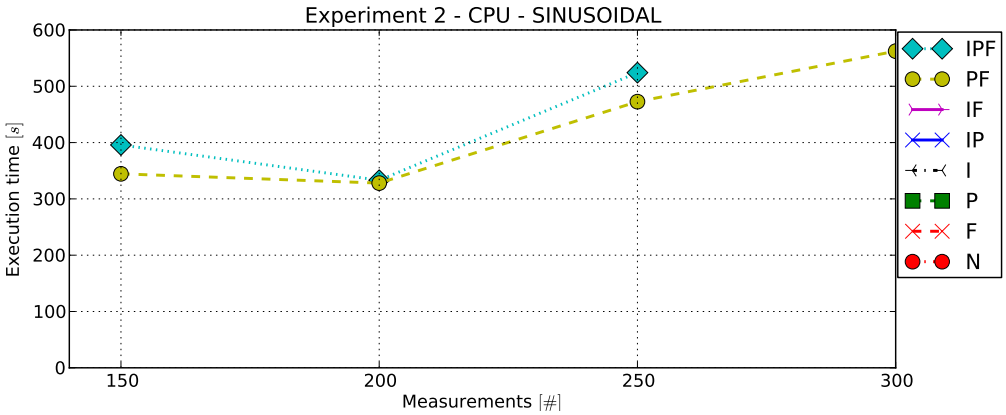


Figure 6.20: SINUSOIDAL: Experiment 2. Execution time for GPU variations. Variable number of measurements, $\Phi = 150, 200, ..., 350$ Fixed number of parameters, $\Psi = 6$. Missing data points for some of the variations indicate that the benchmark did not complete within the 10 minute time limit. Detailed results can be found in Appendix F on page 86 in Table F.12.

6.5.4 GPU and CPU Comparison

For both experiment 1 and 2, the `PF` variation was found to be the fastest for both the GPU and CPU implementations. Figures 6.21 and 6.22 show the execution times and speedups measured for experiment 1 (variable number of parameters). Speedups of 1.89 and 3.56 are measured for $\Psi = 3$ and $\Psi = 6$, respectively.

The execution times and speedups for experiment 2 (variable number of measurements) are shown in Figures 6.23 and 6.24 for $\Phi = 150, 200, ..., 300$ and $\Psi = 6$

parameters. For both the GPU and CPU, the algorithm is faster for 200 measurements than for 150. The speedup ranges between 2.74 times for $\Phi = 150$ to 4.55 times for $\Phi = 300$.

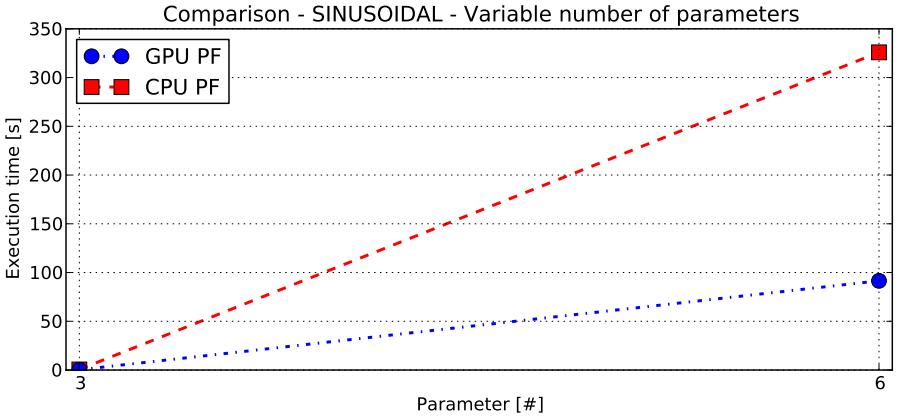


Figure 6.21: SINUSOIDAL: Experiment 1 Execution time for GPU-PF and CPU-PF variations. Fixed number of measurements, $\Phi = 200$. Variable number of parameters, $\Psi = 3, 6$.

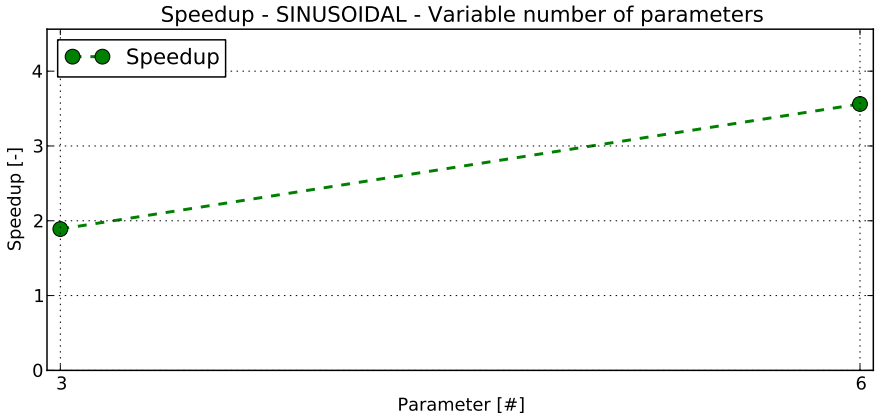


Figure 6.22: SINUSOIDAL: Experiment 2. Speedup, GPU vs. CPU for GPU-PF and CPU-PF variations. Fixed number of measurements, $\Phi = 200$. Variable number of parameters, $\Psi = 3, 6$.

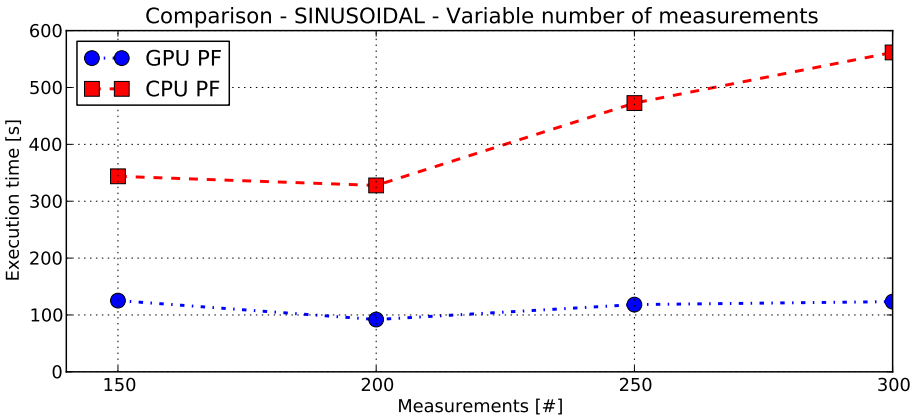


Figure 6.23: SINUSOIDAL: Experiment 2 Execution time for GPU-PF and CPU-PF variations. Variable number of measurements, $\Phi = 150, 200, ..., 300$. Fixed number of parameters, $\Psi = 6$.

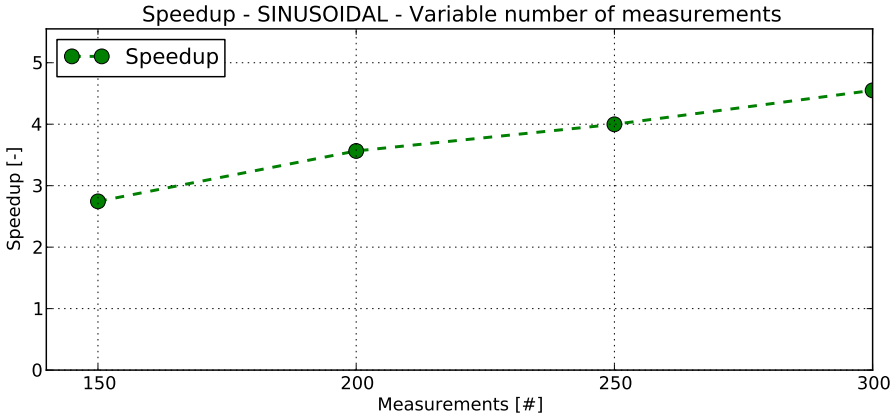


Figure 6.24: SINUSOIDAL: Experiment 2. Speedup, GPU vs. CPU for GPU-PF and CPU-PF variations. Variable number of measurements, $\Phi = 150, 200, ..., 300$. Fixed number of parameters, $\Psi = 6$.

6.5.5 Discussion

As expected, this test problem has proven to be difficult for the algorithm to solve when the number of parameters is high. The highest number of measurements for which the fastest CPU variation was able to solve the problem within the time limit was $\Phi = 300$. As neither the GPU or CPU implementation were able to solve the problem for more parameters than 6, the comparison for varying number of parameters only contains two data points, making it difficult to conclude on the results, other than concluding that low speedups are observed. However, the speedup is expected to increase for higher numbers of measurements. The fact that the execution time for the GPU implementation is almost constant indicates that the parallel resources are not being occupied. A measurement of the GPU compute utilization using the Nvidia Visual Profiler at $\Phi = 300$ and $\Psi = 6$ shows an compute utilization of 10.0 %, which supports this.

6.6 Summary and Overall Discussion

In this chapter the test suite and four test problems were presented. For each test problem, one or two experiments were performed to determine which among the variations of the GPU and CPU implementations. The best performing variation for the GPU was compared with the best performing variation for the GPU. The results show some general tendencies:

1. A general tendency is that the comparisons indicate that the resources of the GPU are not utilized by the implementation. This indication is supported by profilings of GPU activity, revealing low compute utilization of GPU resources. This leads to speedup graphs that do not show convergence toward a constant level. A conclusion that may be drawn from this is, that type-1 parallelization is not sufficient to occupy the GPU resources, unless the number of measurements is very high.

2. For the test problems FDEHYDE, POLYCOS and SINUSOIDAL, the GPU implementation is able to solve problems with significantly higher number of measurements within the time limit of 10 minutes than the CPU implementation. Additionally, in the case of the POLYCOS test problem, the GPU implementation is able to solve the problem for 4 parameters which the CPU implementation is not.

3. Another tendency is that variations including the `F` element perform well. The `F` element performs a computationally cheap sampling of the objective function at the midpoint of the box being processed. This indicates that further point sampling of the objective function may be beneficial.

Chapter 7

Conclusion and Future Perspectives

This chapter provides a conclusion on the present work followed by some considerations on future perspectives for further development and research.

7.1 Conclusion

The present work has treated acceleration of global optimization methods using interval analysis. The focus is on the least-squares parameter estimation problem, which has been accelerated by utilization of the parallel processing capabilities of Graphical Processing Units (GPUs).

Through analysis of well known Interval Global Optimization (IGO) methods applied to the parameter estimation problem, it has been concluded that the problem is well suited for parallelization. This is especially the case for parameter estimation problems with many data points (measurement points and measurements). Further, the suitableness of CUDA capable GPUs for interval arithmetic computations has been evaluated and concluded to be well suited for the purpose.

Previous work on parallel IGO has, to the best knowledge of the authors, only covered parallelization on the outer-loop level[1]. The present work has taken a different approach and focused on parallelization on the inner-loop level[2]. Based on this, methods for parallelization and efficient use of computations, on the inner-loop level of the algorithm have been presented. These involve: 1) Parallel computation of the objective function, the gradient and the Hessian, combined with reuse of data between the computations of the three. 2) A method for parallelization of the Interval Newton step by use of the Interval Gauss-Seidel method and the pivoting preconditioning scheme presented in [28]. 3) A method for obtaining a computationally cheap point sample of the objective function has been proposed. This is done in conjunction with computations related to the Interval Newton step by reuse of data.

An global optimization algorithm for parameter estimation using these methods has been specified. The applicability of the algorithm used is limited to continuously differentiable real-valued model functions.

Two parallel implementations of the global optimization algorithm utilizing the GPU have been presented, along with a sequential reference implementation utilizing only the CPU. The fastest of the two GPU implementations is selected for compar-

[1] Parallelization of type 2, following the classification in [14]
[2] Parallelization of type 1, following the classification in [14]

ison with the CPU implementation. To evaluate the effect of different algorithmic elements, a number of variations of the base GPU and CPU implementations have been implemented. In connection with the development of the implementations, extensions for existing GPU- and CPU software libraries for interval analysis, have been presented.

The GPU and CPU implementations have been benchmarked in two steps using a series of four test problems. In the first step, the fastest among the variations of the GPU implementation and the fastest among the variations of the CPU implementation was found. In the second step these two were compared against each other in order to measure the speedup of the GPU implementation over the CPU implementation. The benchmarks have shown speedups in the range 1.43 to 60.4 times for various instances the test problems used. A general observation is that the speedup increases with the number of data points used for the parameter estimation problem. Analysis of GPU activity for the test cases reveals that the GPU implementation does not fully utilize the computational resources of the GPU.

7.2 Future Perspectives

It is important to stress that what is compared in the present work is a multi-core parallel GPU accelerated implementation, implemented specifically for the hardware used, against a somewhat less specialized sequential single-core CPU implementation. In [35] a software library that uses the Streaming SIMD Extensions (SSE) on Intel CPUs to parallelize individual interval arithmetic instructions is presented. Thereby a significantly more efficient interval library can be implemented than what is available in the Boost C++ libraries, which was used in the present work. Therefore an extension of the interval library presented in [35] to include full support for extended interval arithmetic, used in a CPU implementation of the optimization algorithm poses an interesting topic of further research.

As stated in the conclusion, the GPU implementation suffers from low utilization of the computational resources of the GPU. This may possibly be remedied by introducing parallelism on the outer iteration level in order to occupy more resources. A second possible improvement to the GPU implementation is the use of the Dynamic Parallelism capabilities of newer[3] Nvidia GPUs. Dynamic Parallelism allows kernels running on the GPU to launch other kernels, making it possible to move the entire execution of the algorithm to the GPU. Thereby communication and synchronization between the GPU and the CPU can be avoided, thus increasing performance.

Lastly, a third possible improvement is the use of a mixed precision scheme. Typical GPUs have significantly more processing power for single precision computations than for double precision computations. However, at some point the optimization algorithm may reach its numerical limits using only single precision or the convergence might be accelerated at some points by using double precision arithmetic to obtain sharper bounds. Therefore an implementation using a scheme to switch between different levels of precision may enhance performance. For a general treatment of mixed precision algorithms, see [69].

[3] Dynamic Parallelism is available on Nvidia GPUs with Compute Capability 3.5 and above [68].

Bibliography

[1] L.-P. Tian, L. Mu, and F.-X. Wu, "Iterative linear least squares method of parameter estimation for linear-fractional models of molecular biological systems," in *2010 4th International Conference onBioinformatics and Biomedical Engineering (iCBBE)*, pp. 1–4, 2010.

[2] L. Granvilliers, J. Cruz, and P. Barahona, "Parameter estimation using interval computations," *SIAM Journal on Scientific Computing*, vol. 26, no. 2, pp. 591–612, 2004.

[3] W. R. Esposito and C. A. Floudas, "Global optimization for the parameter estimation of differential-algebraic systems," *Industrial & Engineering Chemistry Research*, vol. 39, no. 5, pp. 1291–1310, 2000.

[4] K. Schittkowski, *Data fitting and experimental Design in Dynamical Systems with Easy-Fit Model Design.* Department of Computer Science, University of Bayreuth, 5.1 ed., 2009.

[5] W. R. Esposito and C. A. Floudas, "Parameter estimation in nonlinear algebraic models via global optimization," *Computers & Chemical Engineering*, vol. 22, Supplement 1, no. 0, pp. S213 – S220, 1998.

[6] C. Gau and M. Stadtherr, "Deterministic global optimization for error-in-variables parameter estimation," *AIChE Journal*, vol. 48, no. 6, pp. 1192–1197, 2004.

[7] A. Neumaier, "Complete search in continuous global optimization and constraint satisfaction," *Acta Numerica*, vol. 13, no. 1, pp. 271–369, 2004.

[8] G. Chen, G. Li, S. Pei, and B. Wu, "High performance computing via a GPU," in *2009 1st International Conference on Information Science and Engineering (ICISE)*, pp. 238–241, IEEE, 2009.

[9] J. Nickolls and W. J. Dally, "The GPU computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010.

[10] A. Törn and A. Zilinskas, *Global optimization.* Lecture notes in computer science, Springer-Verlag, 1989.

[11] E. R. Hansen and G. W. Walster, *Global Optimization using Interval Analysis, Revised and Expanded.* Marcel Dekker, 2nd ed., 2004.

[12] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization: an overview," *Swarm intelligence*, vol. 1, no. 1, pp. 33–57, 2007.

[13] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to interval analysis*. Society for Industrial Mathematics, 2009.

[14] B. Gendron and T. G. Crainic, "Parallel branch-and-branch algorithms: Survey and synthesis," *Operations Research*, vol. 42, no. 6, pp. 1042–1066, 1994.

[15] S. Skelboe, "Computation of rational interval functions," *BIT Numerical Mathematics*, vol. 14, no. 1, pp. 87–95, 1974.

[16] E. Hansen, "A globally convergent interval method for computing and bounding real roots," *BIT Numerical Mathematics*, vol. 18, no. 4, pp. 415–424, 1978.

[17] E. R. Hansen, "Global optimization using interval analysis: the one-dimensional case," *Journal of Optimization Theory and Applications*, vol. 29, no. 3, pp. 331–344, 1979.

[18] E. Hansen, "Global optimization using interval analysis—the multi-dimensional case," *Numerische Mathematik*, vol. 34, no. 3, pp. 247–270, 1980.

[19] C.-Y. Gau, J. F. Brennecke, and M. A. Stadtherr, "Reliable nonlinear parameter estimation in vle modeling," *Fluid Phase Equilibria*, vol. 168, no. 1, pp. 1–18, 2000.

[20] R. McAulay and T. Quatieri, "Speech analysis/synthesis based on a sinusoidal representation," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 34, no. 4, pp. 744–754, 1986.

[21] S. Ramamohan and S. Dandapat, "Sinusoidal model-based analysis and classification of stressed speech," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 14, no. 3, pp. 737–746, 2006.

[22] T. L. Jensen and T. Larsen, "Robust computation of error vector magnitude for wireless standards," *IEEE Transactions on Communications*, vol. 61, no. 2, pp. 648–657, 2013.

[23] N. I. of Standards and Technology, "Statistical reference datasets, nonlinear regression." `http://www.itl.nist.gov/div898/strd/nls/nls_main.shtml`, 2013.

[24] S. Amaran and N. Sahinidis, "Global optimization of nonlinear least-squares problems by branch-and-bound and optimality constraints," *TOP*, vol. 20, no. 1, pp. 154–172, 2012.

[25] E. Hansen, "Publications related to early interval work of RE Moore." `http://interval.louisiana.edu/Moores_early_papers/bibliography.html`, 2001.

[26] E. R. Hansen and S. Sengupta, "Bounding solutions of systems of equations using interval analysis," *BIT Numerical Mathematics*, vol. 21, no. 2, pp. 203–211, 1981.

[27] C.-Y. Gau and M. A. Stadtherr, "Reliable nonlinear parameter estimation using interval analysis: error-in-variable approach," *Computers & Chemical Engineering*, vol. 24, no. 2, pp. 631–637, 2000.

[28] C.-Y. Gau and M. A. Stadtherr, "New interval methodologies for reliable chemical process modeling," *Computers & chemical engineering*, vol. 26, no. 6, pp. 827–840, 2002.

[29] R. B. Kearfott, "Preconditioners for the interval gauss-seidel method," *SIAM Journal on Numerical Analysis*, vol. 27, no. 3, pp. pp. 804–822, 1990.

[30] T. Henriksen and K. Madsen, "Parallel algorithms for global optimization.," *Interval Computations*, vol. 3, no. 5, pp. 88–95, 1992.

[31] J. Eriksson and P. Lindström, "A parallel interval method implementation for global optimization using dynamic load balancing," *Reliable Computing*, vol. 1, no. 1, pp. 77–91, 1995.

[32] S. Berner, "Parallel methods for verified global optimization practice and theory," *Journal of Global Optimization*, vol. 9, no. 1, pp. 1–22, 1996.

[33] L. Casado, J. Martinez, I. García, and E. Hendrix, "Branch-and-bound interval global optimization on shared memory multiprocessors," *Optimization Methods & Software*, vol. 23, no. 5, pp. 689–701, 2008.

[34] S. Ibraev, *A new parallel method for verified global optimization.* PhD thesis, Fachbereich Mathematik der Bergischen Universität Gesamthochschule Wuppertal genehmigte, 2001.

[35] B. Lambov, "Interval arithmetic using sse-2," *Reliable Implementation of Real Number Algorithms: Theory and Practice*, pp. 102–113, 2008.

[36] S. Rump, "INTLAB - INTerval LABoratory," in *Developments in Reliable Computing* (T. Csendes, ed.), pp. 77–104, Dordrecht: Kluwer Academic Publishers, 1999. `http://www.ti3.tuhh.de/rump/`.

[37] G. Melquiond, S. Pion, and H. Brönnimann, *Boost C++ Libraries Documentation, Interval Arithmetic Library*, 2006. `http://www.boost.org/doc/libs/release/libs/numeric/interval/doc/interval.htm`.

[38] NVIDIA, *NVIDIA GPU Computing Software Development Kit, CUDA SDK Release Notes Version 3.2 Release.* `http://developer.download.nvidia.com/compute/cuda/3_2_prod/sdk/docs/CUDA_SDK_Release_Notes.txt`.

[39] M. Lerch, G. Tischler, J. W. von Gudenberg, W. Hofschuster, and W. Krämer, *The Interval Library filib++ 2.0, Design Features and Sample Programs*, 2001.

[40] S. Microsystems, *Fortran 95 Interval Arithmetic Programming Reference*, 2005. `http://docs.oracle.com/cd/E19422-01/819-3695/`.

[41] Y. Lin and M. A. Stadtherr, "Advances in interval methods for deterministic global optimization in chemical engineering," *Journal of Global Optimization*, vol. 29, no. 3, pp. 281–296, 2004.

[42] E. Kreyszig, *Advanced engineering mathematicss.* Wiley, 1999.

[43] E. Hansen, "Interval arithmetic in matrix computations, part i," *Journal of the Society for Industrial & Applied Mathematics, Series B: Numerical Analysis*, vol. 2, no. 2, pp. 308–320, 1965.

[44] R. E. Moore, "On computing the range of a rational function of n variables over a bounded region," *Computing*, vol. 16, no. 1, pp. 1–15, 1976.

[45] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004. Available at `http://www.stanford.edu/~boyd/cvxbook/`.

[46] C. Schnepper and M. Stadtherr, "Robust process simulation using interval methods," *Computers & chemical engineering*, vol. 20, no. 2, pp. 187–199, 1996.

[47] L. Casado, J. Martínez, I. García, and Y. D. Sergeyev, "New interval analysis support functions using gradient information in a global minimization algorithm," *Journal of Global Optimization*, vol. 25, no. 4, pp. 345–362, 2003.

[48] E. R. Hansen, "On solving systems of equations using interval arithmetic," *Mathematics of Computation*, vol. 22, no. 102, pp. 374–384, 1968.

[49] NVIDIA, *Fermi Computer Architecture White paper*, v1.1 ed., 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[50] NVIDIA, *CUDA C Programming Guide*, "pg-02829-001_v5.0" ed., 2012. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[51] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[52] NVIDIA, *CUDA C Best Practieces Guide*, dg-05603-001_v5.0 ed., 2012. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`.

[53] M. J. Koop, W. Huang, K. Gopalakrishnan, and D. K. Panda, "Performance analysis and evaluation of PCIe 2.0 and quad-data rate infiniband," in *High Performance Interconnects, 2008. HOTI'08. 16th IEEE Symposium on*, pp. 85–92, IEEE, 2008.

[54] NVIDIA, *Tesla M2075 dual-slot computing processor module*. `http://www.nvidia.com/docs/IO/43395/BD-05837-001_v01.pdf`.

[55] W. mei W Hwu, *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2011.

[56] NVIDIA, *Kepler, GK110*, v1.0 ed., 2012. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[57] I. Chakroun and N. Melab, "An adaptative multi-GPU based branch-and-bound. a case study: the flow-shop scheduling problem," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 389–395, IEEE, 2012.

[58] T. Carneiro, A. E. Muritiba, M. Negreiros, and G. A. Lima de Campos, "A new parallel schema for branch-and-bound algorithms using GPGPU," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pp. 41–47, IEEE, 2011.

[59] V. Boyer, D. El Baz, and M. Elkihel, "Solving knapsack problems on GPU," *Computers & Operations Research*, vol. 39, no. 1, pp. 42–47, 2012.

[60] A. Boukedjar, M. E. Lalami, and D. El-Baz, "Parallel branch and bound on a CPU-GPU system," in *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pp. 392–398, IEEE, 2012.

[61] M. E. Lalami and D. El-Baz, "GPU implementation of the branch and bound method for knapsack problems," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 1769–1777, IEEE, 2012.

[62] NVIDIA, *Profiler User's Guide*, "du-05982-001_v5.0" ed., 2012. `http://http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf`.

[63] S. Collange, J. Flórez, D. Defour, *et al.*, "A gpu interval library based on boost. interval," in *Proceedings of the 8th Conference on Real Numbers and Computers*, pp. 61–71, 2008.

[64] J. M. Snyder, "Interval analysis for computer graphics," *ACM SIGGRAPH Computer Graphics*, vol. 26, no. 2, pp. 121–130, 1992.

[65] H. S. Warren, *Hacker's Delight*. Addison-Wesley Professional, second ed., 2012.

[66] N. Developers, "Scientific computing tools for python-numpy." `http://numpy.org`, 2010.

[67] A. S. Olansky and S. N. Deming, "Optimization and interpretation of absorbance response in the determination of formaldehyde with chromotropic acid," *Analytica Chimica Acta*, vol. 83, pp. 241–249, 1976.

[68] NVIDIA, *CUDA Dynamic Parallelism Programming Guide*, 2012. `http://docs.nvidia.com/cuda/pdf/CUDA_Dynamic_Parallelism_Programming_Guide.pdf`.

[69] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526 – 2533, 2009.

[70] NVIDIA, "CUDA API Reference Manual Version 5.0," October 2012. `http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Reference_Manual.pdf`.

Appendix A

Test Platform Specifications

The hardware- and software specifications of the platform used are given below in Table A.1 and Table A.2, respectively.

Hardware:	
CPU:	Intel Core i7 950 3.07 GHz (4 cores, Hyper Threading enabled)
RAM:	12 GB, 1333 MHz (6 blocks of 2 GB)
GPU #1:	ASUSTeK Nvidia GTX TITAN (6 GB mem.)
GPU #2:	ASUSTek Nvidia GTX 465 (1 GB mem.)
Motherboard:	Supermicro X8SAX v2.0 (PCI-E 2.0)

Table A.1: Hardware specifications for the test platform.

Software:	
Operating system:	Linux Mint 13 Maya (Kernel 3.2.0-23-generic)
CUDA:	v5.0.32
GCC:	v4.6
Python:	v2.7.3
Pycuda:	v2012.1
BLAS:	ATLAS BLAS 3.8.4
LAPACK:	NetLib LAPACK 3.3.1
Boost C++ Libraries:	v1.49

Table A.2: Software specifications for the test platform.

Appendix B

The Model Function

For both the GPU and CPU implementations, the model, model gradient and model Hessian functions must be specified by the user. The functions are implemented in CUDA-C for the GPU and C/C++ for the CPU and included in the code the files `model.cu` and `model.cpp` respectively. Since the GPU versions are implemented as single-thread device functions, the CPU and GPU versions are basically the same, only with a slight difference. The interfaces for the functions are listed in Table B.1.

$m(\mathbf{p}, \mathbf{x})$	`_d_ I model(const I* _R_ p, const double* _R_ x)`
$g_j(\mathbf{p}, \mathbf{x})$	`_d_ I modelg(const I* _R_ p, const double* _R_ x, const int j)`
$H_{j,k}(\mathbf{p}, \mathbf{x})$	`_d_ I modelH(const I* _R_ p, const double* _R_ x, const int j, const int k)`

Table B.1: Interface specification for the model, model gradient and model Hessian functions. `_d_` denotes the `__device__` qualifier, which is only used in CUDA-C. `_R_` denotes the `__restrict__` qualifier, which tells the compiler that the data referenced by the pointer does not overlap with data referenced by any other pointers.

To get as tight bounds as possible the user most be aware of the effects of dependence described in 2.1.3 on page 7 and the method for obtaining narrower bounds, described in 3.3.3 on page 24 when implementing the model.

The availability of transcendental functions and other math functions, other than the basic operators is described in 5.5 on page 48. In the CPU code, these functions reside in the `pyint` namespace, and must therefore be prefixed with `pyint::` (e.g. `pyint::sin(x)`). In the GPU code they reside in the default namespace and this no prefix is necessary.

Appendix C

Note on the CUDA Nextafter Function

This is a note on the `nextafter(x,d)` function and the CUDA double precision library, shipped by Nvidia with the CUDA SDK.

`nextafter(x,d)` function in the CUDA library shipped by Nvidia with the CUDA SDK. In the CUDA API Reference Manual, the description of the function is:

> *Calculate the next representable double-precision floating-point value following x in the direction of y. For example, if y is greater than x, nextafter() returns the smallest representable number greater than x.* [70]

However, if one studies the library header files, it turns out that there exists two different implementations of the functions. Which of the two is used when the code is compiled depends on the targeted Compute Capability. For Compute Capability < 3.0 the implementation is (from `math_functions_dbl_ptx1.h` included in the Nvidia CUDA toolkit v5.0):

```
static __forceinline__ double nextafter(double a, double b)
{
  return (double)nextafterf((float)a, (float)b);
}
```

For Compute Capability ≥ 3.0 the implementation is (from `math_functions_dbl_ptx3.h` included in the Nvidia CUDA toolkit v5.0):

```
static __forceinline__ double nextafter(double a, double b)
{
  unsigned long long int ia;
  unsigned long long int ib;
  ia = __double_as_longlong(a);
  ib = __double_as_longlong(b);
  if (__isnand(a) || __isnand(b)) return a + b; /* NaN */
  if (((ia | ib) << 1) == 0ULL) return b;
  if ((ia + ia) == 0ULL) {
    return __internal_copysign_pos(CUDART_MIN_DENORM, b); /* crossover */
  }
  if ((a < b) && (a < 0.0)) ia--;
  if ((a < b) && (a > 0.0)) ia++;
  if ((a > b) && (a < 0.0)) ia++;
  if ((a > b) && (a > 0.0)) ia--;
  a = __longlong_as_double(ia);
  return a;
}
```

Observe that in the first code sample, the double precision inputs are typecast to single precision and the single precision version of the nextafter function, `nextafterf(x,f)`. This function is then used to compute a single precision result which is converted to double precision. This means that the return value is in fact not the next representable double precision value, but the next representable single precision value - possibly a many times larger step.

In the second code sample, used when compiling for Compute Capability < 3.0, the function takes a step of the correct magnitude. For this implementation however, an abnormality was observed. In some cases when performing a double precision multiplication using directed rounding with the `__dmul_[rd/ru](x,y)` intrinsic, the result would come out alltogether wrong, and of a very large magnitude. The behaviour was observed to occur at seemingly random times when using the global optimization algorithm and has at the time of writing not been reproduced using simple code. When using single precision, the abnormal behaviour does not occur.

Appendix D

Containment Sets for Basic Operations Over the Extended Real Numbers

$x+y$		y		
		$y=-\infty$	$y\in\mathbb{R}$	$y=+\infty$
x	$x=-\infty$	$-\infty$	$-\infty$	$\mathbb{R}_*$
	$x\in\mathbb{R}$	$-\infty$	$x+y$	$+\infty$
	$x=+\infty$	$\mathbb{R}_*$	$+\infty$	$+\infty$

Table D.1: Addition over the extended real numbers [11, Table 4.1]

$x-y$		y		
		$y=-\infty$	$y\in\mathbb{R}$	$y=+\infty$
x	$x=-\infty$	$\mathbb{R}_*$	$-\infty$	$-\infty$
	$x\in\mathbb{R}$	$+\infty$	$x-y$	$-\infty$
	$x=+\infty$	$+\infty$	$+\infty$	$\mathbb{R}_*$

Table D.2: Subtraction over the extended real numbers [11, Table 4.2]

$x\cdot y$		y				
		$y=-\infty$	$y\in\,]-\infty,0[$	$y=0$	$y\in\,]0,\infty[$	$y=+\infty$
x	$x=-\infty$	$+\infty$	$+\infty$	$\mathbb{R}_*$	$-\infty$	$-\infty$
	$x\in\,]-\infty,0[$	$+\infty$	$x\cdot y$	0	$x\cdot y$	$-\infty$
	$x=0$	$\mathbb{R}_*$	0	0	0	$\mathbb{R}_*$
	$x\in\,]0,+\infty[$	$-\infty$	$x\cdot y$	0	$x\cdot y$	$+\infty$
	$x=+\infty$	$-\infty$	$-\infty$	$\mathbb{R}_*$	$+\infty$	$+\infty$

Table D.3: Multiplication over the extended real numbers [11, Table 4.3]

$\frac{x}{y}$		y				
		$y=-\infty$	$y\in\,]-\infty,0[$	$y=0$	$y\in\,]0,+\infty[$	$y=+\infty$
x	$x=-\infty$	$[0,\,+\infty]$	$+\infty$	$\{-\infty,+\infty\}$	$-\infty$	$[-\infty,\,0]$
	$x\in\,]-\infty,0[$	0	x/y	$\{-\infty,+\infty\}$	x/y	0
	$x=0$	0	0	$\mathbb{R}_*$	0	0
	$x=+\infty$	$[-\infty,\,0]$	$-\infty$	$\{-\infty,+\infty\}$	$+\infty$	$[0,\,+\infty]$

Table D.4: Division over the extended real numbers [11, Table 4.4]

Appendix E

Search Region Preprocessioning

For some model functions, the objective function of the parameter estimation problem is known to be symmetric. An example of this is the model function

$$m(x, \mathbf{p}) = sin(p_0 x) + sin(p_1 x) \tag{E.1}$$

Denoting the true values of the parameters $\hat{p_0}$ and $\hat{p_1}$, and assuming that these are sufficiently separated, the global optimization algorithm will enclose two solutions to the problem:

$$\tilde{\mathbf{p}}_{01} = [\hat{p_0}, \hat{p_1}]^{\mathrm{T}} \quad \text{and} \quad \tilde{\mathbf{p}}_{10} = [\hat{p_1}, \hat{p_0}]^{\mathrm{T}}$$

This is inefficient, as the algorithm spends time finding two solutions that are basically equal. Therefore, the initial search region $\mathbf{P}$ is preprocessed to remove the mirrored solution. This is done by removing a number of subregions where $p_0 > p_1$. This is illustrated in Figure E.1

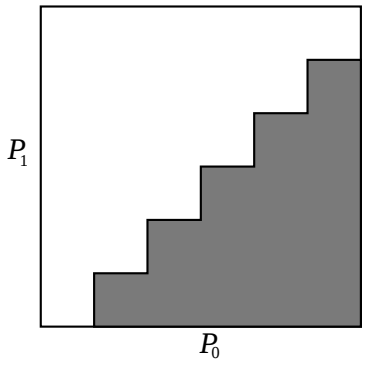


Figure E.1: Illustration of preprocessed search region. The grayed area shows sub-regions discarded by the preprocessing method.

Appendix F

Detailed Results

F.1 FDEHYDE

FDEHYDE - GPU results									
Φ [#]:	100	2000	4000	6000	8000	10000	12000	14000	16000
$t_{\texttt{IPF}}$ [s]	21.903	22.741	24.324	25.772	27.105	28.599	29.883	31.384	33.077
$t_{\texttt{PF}}$ [s]	14.732	15.503	17.049	18.405	19.818	21.233	22.645	24.193	25.850
$t_{\texttt{IF}}$ [s]	20.097	21.008	22.617	23.978	25.393	26.850	28.283	29.860	31.534
$t_{\texttt{IP}}$ [s]	72.755	71.684	76.828	79.716	83.702	90.258	93.979	98.184	104.15
$t_{\texttt{I}}$ [s]	114.42	87.739	94.637	99.578	108.41	112.91	121.33	127.93	135.51
$t_{\texttt{P}}$ [s]	47.762	47.263	51.922	54.917	57.727	65.674	66.433	71.421	76.447
$t_{\texttt{F}}$ [s]	20.306	23.862	24.584	26.346	28.800	30.710	33.274	35.641	39.057
$t_{\texttt{N}}$ [s]	83.069	84.500	89.533	95.544	104.49	112.04	119.26	127.11	136.66

Table F.1: FDEHYDE: Experiment 1. Execution time for GPU variations. Variable number of measurements, $\Phi = 100, 2000, 4000..., 16000$. Fixed number of parameters, $\Psi = 4$. T/O indicates the time out limit of 10 minutes was reached.

FDEHYDE - CPU results									
Φ [#]:	100	200	300	400	500	600	700	800	900
$t_{\texttt{IPF}}$ [s]	48.063	75.051	108.74	134.64	166.75	198.79	228.26	250.73	287.92
$t_{\texttt{PF}}$ [s]	42.758	70.378	104.88	130.96	163.86	196.26	226.18	249.32	287.18
$t_{\texttt{IF}}$ [s]	47.58	75.218	110.44	136.96	170.45	204.15	233.53	256.59	294.33
$t_{\texttt{IP}}$ [s]	161.32	248.83	321.48	403.77	537.56	T/O	T/O	T/O	T/O
$t_{\texttt{I}}$ [s]	268.25	439.74	529.34	565.60	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{P}}$ [s]	141.66	227.08	302.72	380.72	492.57	T/O	T/O	T/O	T/O
$t_{\texttt{F}}$ [s]	56.039	95.885	145.09	184.68	237.02	279.50	323.09	362.52	417.12
$t_{\texttt{N}}$ [s]	234.77	403.09	531.73	T/O	T/O	T/O	T/O	T/O	T/O

Table F.2: FDEHYDE: Experiment 1. Execution time for CPU variations. Variable number of measurements, $\Phi = 100, 200, ..., 1800$. Fixed number of parameters, $\Psi = 4$. T/O indicates the time out limit of 10 minutes was reached.

F.2 POLYCOS

Experiment 1 - POLYCOS GPU results			
Ψ [#]:	2	3	4
$t_{\texttt{IPF}}$ [s]	0.19717	2.7465	196.86
$t_{\texttt{PF}}$ [s]	0.16022	2.3560	172.19
$t_{\texttt{IF}}$ [s]	0.23999	4.9890	371.67
$t_{\texttt{IP}}$ [s]	0.43693	7.5377	330.75
$t_{\texttt{I}}$ [s]	0.59468	15.262	T/O
$t_{\texttt{P}}$ [s]	0.31947	7.7074	T/O
$t_{\texttt{F}}$ [s]	0.25815	4.6490	367.27
$t_{\texttt{N}}$ [s]	0.63277	23.776	T/O

Table F.3: POLYCOS: Experiment 1. Execution time for GPU variations. Fixed number of measurements, $\Phi = 1024$. Variable number of parameters, $\Psi = 2, 3, 4$. T/O indicates the time out limit of 10 minutes was reached.

Experiment 1 - POLYCOS CPU results			
Ψ [#]:	2	3	4
$t_{\texttt{IPF}}$ [s]	0.69828	13.962	T/O
$t_{\texttt{PF}}$ [s]	0.72386	16.892	T/O
$t_{\texttt{IF}}$ [s]	0.86533	26.915	T/O
$t_{\texttt{IP}}$ [s]	1.5481	38.503	T/O
$t_{\texttt{I}}$ [s]	2.2475	80.923	T/O
$t_{\texttt{P}}$ [s]	1.5894	59.758	T/O
$t_{\texttt{F}}$ [s]	0.9376	26.844	T/O
$t_{\texttt{N}}$ [s]	2.4498	140.90	T/O

Table F.4: POLYCOS: Experiment 1. Execution time for CPU variations. Fixed number of measurements, $\Phi = 1024$. Variable number of parameters, $\Psi = 2, 3, 4$. T/O indicates the time out limit of 10 minutes was reached.

Experiment 2 - POLYCOS - GPU results									
Φ [#]:	100	2000	4000	6000	8000	10000	12000	14000	16000
$t_{\texttt{IPF}}$ [s]	2.1483	2.9066	3.2899	3.5666	3.8128	3.9839	4.1046	4.3154	4.5507
$t_{\texttt{PF}}$ [s]	1.9357	2.4116	2.6611	2.8991	3.1049	3.2352	3.3565	3.5475	3.7968
$t_{\texttt{IF}}$ [s]	13.679	26.324	40.182	52.756	67.005	78.54	92.602	106.42	121.26
$t_{\texttt{IP}}$ [s]	5.8132	8.0344	8.9658	9.9190	10.951	11.931	12.940	14.035	15.141
$t_{\texttt{I}}$ [s]	10.562	17.489	21.442	25.453	29.568	34.268	38.535	42.719	47.828
$t_{\texttt{P}}$ [s]	5.5944	8.2228	9.0980	10.119	10.858	11.188	11.876	12.747	13.619
$t_{\texttt{F}}$ [s]	3.2134	4.8658	5.2226	5.5102	5.9018	6.1728	6.4392	6.6705	7.0808
$t_{\texttt{N}}$ [s]	16.445	26.480	30.849	33.214	34.172	35.879	37.666	39.319	43.809

Table F.5: POLYCOS: Experiment 2. Execution time for GPU variations. Variable number of measurements, $\Phi = 100, 2000, 4000, ..., 16000$. Fixed number of parameters, $\Psi = 4$. T/O indicates the time out limit of 10 minutes was reached.

Experiment 2 - POLYCOS - CPU results									
Φ [#]:	500	1000	1500	2000	2500	3000	3500	4000	4500
$t_{\texttt{IPF}}$ [s]	7.6415	13.570	20.831	28.571	36.597	43.838	49.479	57.754	66.055
$t_{\texttt{PF}}$ [s]	9.5561	16.445	25.590	35.129	44.851	53.027	60.041	67.863	76.9
$t_{\texttt{IF}}$ [s]	41.886	79.742	123.09	169.81	215.93	274.78	332.83	393.09	469.19
$t_{\texttt{IP}}$ [s]	21.051	37.757	55.721	74.591	94.232	114.66	135.48	156.86	181.24
$t_{\texttt{I}}$ [s]	41.682	79.314	122.42	168.80	214.68	273.06	330.53	390.80	466.26
$t_{\texttt{P}}$ [s]	31.947	58.258	88.441	117.03	144.96	176.28	212.57	240.44	278.03
$t_{\texttt{F}}$ [s]	13.214	26.201	40.202	52.208	66.267	77.417	91.121	104.51	118.75
$t_{\texttt{N}}$ [s]	68.607	136.65	214.26	282.92	355.07	430.03	518.03	T/O	T/O

Table F.6: POLYCOS: Experiment 2. Execution time for CPU variations. Variable number of measurements, $\Phi = 500, 1000, 1500, ..., 4500$. Fixed number of parameters, $\Psi = 3$. T/O indicates the time out limit of 10 minutes was reached.

F.3 2D_POLY

2D_POLY - GPU results									
Φ [#]:	1000	2000	4000	6000	8000	10000	12000	14000	16000
$t_{\texttt{IPF}}$ [s]	0.16777	0.16777	0.16818	0.16853	0.16903	0.16977	0.17054	0.17130	0.17263
$t_{\texttt{PF}}$ [s]	0.15629	0.15913	0.16156	0.16535	0.16692	0.17019	0.17151	0.17407	0.17827
$t_{\texttt{IF}}$ [s]	0.24864	0.24935	0.25034	0.25156	0.25282	0.25385	0.25563	0.26084	0.26250
$t_{\texttt{IP}}$ [s]	0.20000	0.20013	0.20057	0.20127	0.20232	0.20257	0.20317	0.20396	0.20548
$t_{\texttt{I}}$ [s]	0.31447	0.31559	0.31680	0.31852	0.32032	0.32141	0.32244	0.32827	0.33084
$t_{\texttt{P}}$ [s]	0.17508	0.17828	0.18049	0.18558	0.18635	0.19030	0.19057	0.19349	0.19744
$t_{\texttt{F}}$ [s]	0.47905	0.48555	0.50466	0.50516	0.51406	0.52228	0.53083	0.53268	0.54286
$t_{\texttt{N}}$ [s]	0.69129	0.71364	0.73285	0.73374	0.75807	0.76793	0.75334	0.76727	0.77998

Table F.7: 2D_POLY: Experiment 1. Execution time for GPU variations. Variable number of measurements, $\Phi = 1000, 2000, 4000..., 16000$. Fixed number of parameters, $\Psi = 2$. T/O indicates the time out limit of 10 minutes was reached.

2D_POLY - CPU results									
Φ [#]:	1000	2000	4000	6000	8000	10000	12000	14000	16000
$t_{\texttt{IPF}}$ [s]	0.40544	0.67813	1.2238	1.7682	2.3126	2.8571	3.4009	3.9858	4.5358
$t_{\texttt{PF}}$ [s]	0.54203	0.97963	1.8545	2.7853	3.7160	4.6688	5.5742	6.5936	7.4768
$t_{\texttt{P}}$ [s]	0.59151	1.0691	2.0248	3.0361	4.0464	5.0777	6.0641	7.1547	8.1091
$t_{\texttt{IP}}$ [s]	0.4834	0.81085	1.4643	2.1180	2.7708	3.4237	4.0762	4.7691	5.4274
$t_{\texttt{N}}$ [s]	1.7089	3.0150	5.6405	8.0359	10.895	13.571	15.854	18.686	21.508
$t_{\texttt{I}}$ [s]	0.79717	1.3481	2.4471	3.5455	4.6437	5.7415	6.9214	8.0875	9.2054
$t_{\texttt{IF}}$ [s]	0.61854	1.0414	1.8868	2.7308	3.5742	4.4181	5.3471	6.2614	7.1283
$t_{\texttt{F}}$ [s]	1.2245	2.2729	4.0929	6.2159	8.3857	10.516	12.771	14.118	16.353

Table F.8: 2D_POLY: Experiment 1. Execution time for CPU variations. Variable number of measurements, $\Phi = 1000, 2000, 4000..., 16000$. Fixed number of parameters, $\Psi = 2$. T/O indicates the time out limit of 10 minutes was reached.

F.4 SINUSOIDAL

Experiment 1 - SINUSOIDAL - GPU results		
Ψ [#]:	3	6
$t_{\texttt{IPF}}$ [s]	0.46245	261.32
$t_{\texttt{PF}}$ [s]	0.34064	171.16
$t_{\texttt{IF}}$ [s]	0.44354	313.16
$t_{\texttt{IP}}$ [s]	2.0168	T/O
$t_{\texttt{I}}$ [s]	1.8801	T/O
$t_{\texttt{P}}$ [s]	1.4150	T/O
$t_{\texttt{F}}$ [s]	0.61057	306.77
$t_{\texttt{N}}$ [s]	2.4468	T/O

Table F.9: SINUSOIDAL: Experiment 1. Execution time for GPU variations. Fixed number of measurements, $\Phi = 256$. Variable number of parameters, $\Psi = 3, 6$. T/O indicates the time out limit of 10 minutes was reached.

Experiment 1 - Sinusoidal - CPU results		
Ψ [#]:	3	6
$t_{\texttt{IPF}}$ [s]	0.65133	348.31
$t_{\texttt{PF}}$ [s]	0.60585	328.02
$t_{\texttt{IF}}$ [s]	0.62511	475.87
$t_{\texttt{IP}}$ [s]	1.7574	T/O
$t_{\texttt{I}}$ [s]	2.7362	T/O
$t_{\texttt{P}}$ [s]	1.5364	T/O
$t_{\texttt{F}}$ [s]	1.1185	T/O
$t_{\texttt{N}}$ [s]	3.1107	T/O

Table F.10: SINUSOIDAL: Experiment 1. Execution time for CPU variations. Fixed number of measurements, $\Phi = 200$. Variable number of parameters, $\Psi = 3, 6$. T/O indicates the time out limit of 10 minutes was reached.

Experiment 2 - SINUSOIDAL - GPU results					
Φ[#]	200	300	400	500	600
$t_{\texttt{IPF}}$ [s]	143.37	197.44	290.98	331.18	303.05
$t_{\texttt{PF}}$ [s]	92.284	124.00	185.88	211.29	194.02
$t_{\texttt{IF}}$ [s]	184.24	196.07	277.46	315.24	277.74
$t_{\texttt{IP}}$ [s]	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{I}}$ [s]	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{P}}$ [s]	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{F}}$ [s]	289.84	207.85	297.03	295.71	263.01
$t_{\texttt{N}}$ [s]	T/O	T/O	T/O	T/O	T/O

Table F.11: SINUSOIDAL: Experiment 2. Execution time for GPU variations. Variable number of measurements, $\Phi = 200, 300, ..., 600$ Fixed number of parameters, $\Psi = 6$. T/O indicates the time out limit of 10 minutes was reached.

Experiment 2 - SINUSOIDAL - CPU results					
Φ [#]:	150	200	250	300	350
$t_{\texttt{IPF}}$ [s]	395.78	333.13	524.20	616.48	T/O
$t_{\texttt{PF}}$ [s]	344.50	328.03	472.66	562.36	T/O
$t_{\texttt{IF}}$ [s]	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{IP}}$ [s]	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{I}}$ [s]	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{P}}$ [s]	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{F}}$ [s]	T/O	T/O	T/O	T/O	T/O
$t_{\texttt{N}}$ [s]	T/O	T/O	T/O	T/O	T/O

Table F.12: SINUSOIDAL: Experiment 2. Execution time for GPU variations. Variable number of measurements, $\Phi = 150, 200, ..., 350$ Fixed number of parameters, $\Psi = 6$. T/O indicates the time out limit of 10 minutes was reached.