NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
MSC DATA SCIENCE & MACHINE LEARNING

ONASSIS FOUNDATION

Accelerating SIVIA (Set Inversion via Interval Analysis). An Interval Set Membership Technique to Evaluate the Generalization of Neural Classifiers.

DIPLOMA THESIS

of

KONSTANTINOS NASIOTIS

Supervisor: Dimitrios Soudris
Professor, NTUA

Athens, October 2023

National Technical University of Athens
School of Electrical and Computer Engineering
MSc Data Science & Machine Learning

ONASSIS FOUNDATION

Accelerating SIVIA (Set Inversion via Interval Analysis). An Interval Set Membership Technique to Evaluate the Generalization of Neural Classifiers.

Diploma Thesis

of

KONSTANTINOS NASIOTIS

Supervisor: Dimitrios Soudris
Professor, NTUA

Approved by the examination committee on 02/11/2023.

(Signature)	*(Signature)*	*(Signature)*
....	....	....
Dimitrios Soudris	Sotirios Xydis	Adam Stavros
Professor, NTUA	Assistant Professor, NTUA	Assistant Professor, UOI

Athens, October 2023

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
MSC DATA SCIENCE & MACHINE LEARNING

ONASSIS FOUNDATION

Copyright © – All rights reserved.
Konstantinos Nasiotis, 2023.

The copying, storage and distribution of this diploma thesis, exall or part of it, is prohibited for commercial purposes. Reprinting, storage and distribution for non - profit, educational or of a research nature is allowed, provided that the source is indicated and that this message is retained.

The content of this thesis does not necessarily reflect the views of the Department, the Supervisor, or the committee that approved it.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

....
Konstantinos Nasiotis
19/10/23

Abstract

Set Inversion via Interval Analysis (SIVIA) is a mathematically rigorous Branch-and-Bound technique, capable of deterministically providing guaranteed inner and outer approximations, given a bounded output interval and an inclusion function. Neural Networks (NNs), when performing a forward pass, in their most basic form are described by addition, multiplication and trigonometric operations. This means that a neural network can be quite portable to develop and should theoretically be easy to combine with other algorithms and hardware. In a classification context, using a Neural Network with SIVIA can provide us with guaranteed approximations of the input space recognized by a given class. This is useful in the visual sense, similarly to the way a fitted line is visualized in a Linear Regression problem, as measures like R^2 do not constitute an absolute indicator of the quality of the fit. In addition, this information can be used to extract new metrics to help us understand the quality of a Network's training, such as its generalization performance. Combining SIVIA with NNs, however, results in a very demanding problem, as both large input spaces and computationally intensive functions (if the NN is large enough) are required to obtain a solution. This is why a parallel approach is proposed in this thesis, utilizing the massive amount of threads embedded into Graphical Processing Units, is proposed. The effectiveness of the proposed parallel algorithm is demonstrated on three different problems, with different hardware configurations and different data management strategies. Results indicate performance ranging from slowdowns to 8000 times the speedup compared to the sequential algorithm for the test problems and problem sizes used. Analysis shows that the GPU accelerated implementation is very sensitive to memory transfer and synchronization operations. It is concluded that for problems with large computational intensity, given sufficient available memory and proper memory management, use of the method presented has the potential of yielding significant speedups.

Περίληψη

Η Αντιστροφή Συνόλου μέσω Ανάλυσης Διαστημάτων (Set Inversion Via Interval Analysis-SIVIA) είναι μία απαιτητική μαθηματική τεχνική που συνήθως χρησιμοποιείται σε προβλήματα βελτιστοποίησης και ανάγεται στις τεχνικές Branch-and-Bound, μία κατηγορία αλγορίθμων που χρησιμοποιούν στρατηγική Διαίρει και Βασίλευε με στόχο την εξερεύνηση - συνήθως αρκετά μεγάλων- χώρων εισόδου. Αυτή η τεχνική έχει την δυνατότητα να προσφέρει εξασφαλισμένες εσωτερικές και εξωτερικές προσεγγίσεις ενός πεδίου ορισμού, δεδομένων ενός διαστήματος του πεδίου τιμών και μιας συνάρτησης εγκλεισμού. Τα Νευρωνικά Δίκτυα, κατά το εμπρόσθιο πέρασμα, αποτελούνται από προσθέσεις, πολλαπλασιασμούς και τριγωνομετρικές συναρτήσεις. Αυτό σημαίνει πως θεωρητικά η χρήση και η μεταφορά ενός Νευρωνικού Δικτύου με άλλες τεχνικές και υπολογιστικούς εξοπλισμούς είναι εύκολη. Πρακτικά, σε προβλήματα κατηγοριοποίησης, η χρήση ενός Νευρωνικού Δικτύου σε συνδυασμό με τη τεχνική SIVIA μπορεί να προσφέρει εξασφαλισμένες προσεγγίσης του χώρου εισόδου δεδομένης μιας κατηγορίας. Αυτό είναι χρήσιμο από την οπτική σκοπιά, όπως αντίστοιχα σε προβλήματα Γραμμικής Παλινδρόμησης χρειάζεται να οπτικοποιήσουμε την γραμμή παλινδρόμησης καθώς μετρικές όπως το R^2 δεν αποτελούν απόλυτη ένδειξη ποιότητας προσαρμογής. Επιπρόσθετα, η πληροφορία που παράγεται μέσω αυτής της τεχνικής μπορεί να χρησιμοποιηθεί για την δημιουργία νέων μετρικών οι οποίες μπορούν να συμβάλλουν στην βαθύτερη κατανόηση της μάθησης ενός νευρωνικού δικτύου. Ο συνδυασμός της τεχνικής SIVIA με ένα Νευρωνικό Δίκτυο μπορεί να αποτελέσει ένα αρκετά απαιτητικό πρόβλημα, καθώς χρειάζονται υπολογιστικά απαιτητικοί υπολογισμοί (στην περίπτωση χρήσης ενός μεγάλου δικτύου) σε συνδυασμό με τεράστιους χώρους εισόδου για την εύρεση λύσης. Συνεπώς, σε αυτή τη διπλωματική εργασία προτείνεται μία παράλληλη προσέγγιση, εκμεταλλευόμενη τον γιγαντιαίο αριθμό νημάτων που εμπεριέχονται σε μία μονάδα επεξεργασίας γραφικών (GPU). Η αποδοτικότητα της προτεινόμενης παράλληλης προσέγγισης παρουσιάζεται σε τρία διαφορετικά προβλήματα, σε διαφορετικά υπολογιστικά συστήματα και διαφορετικές στρατηγικής διαχείρισης δεδομένων. Τα αποτελέσματα ανέδειξαν επιδόσεις από επιβραδύνσεις μέχρι επιτάχυνση κατά 8000 φορές. Η περαιτέρω ανάλυση ανέδειξε πως η προτεινόμενη παράλληλη υλοποίηση είναι ευαίσθητη στις μεταφορές δεδομένων καθώς και στον συγχρονισμό με την κεντρική μονάδα επεξεργασίας (CPU). Συμπερασματικά, για προβλήματα με υψηλές υπολογιστικές απαιτήσεις, δεδομένου επαρκούς διαθέσιμης μνήμης και κατάλληλης διαχείρισης, η χρήση της μεθόδου έχει την προοπτική για περαιτέρω επιταχύνσεις.

To the person reading this.

Acknowledgements

I would like to express my sincere gratitude to Professors Dimitrios Soudris and Stavros Adam for their supervision and mentorship. Continuing on, I am extremely grateful to the ONASSIS Foundation for providing me with a scholarship for postgraduate studies, a valuable resource which allowed me to remain a full-time student. I would also like to express my gratitude towards Dr. Dimosthenis Masouros for his valuable suggestions and comments during this whole year as well as Dr. Georgios Zervakis and Professor Panagiotis Hadjidoukas for granting me access to expensive high-performant graphical computational hardware. Finally, I want to thank my network of friends and family that supported me throughout this whole year, emotionally and materially. Things would be way harder without everyone's encouragement.

Athens, October 2023

Konstantinos Nasiotis

Contents

Abstract **1**

Περίληψη **3**

Acknowledgements **7**

List of Abbreviations **17**

1 Introduction **19**

1.1 English . . . 19

1.2 Ελληνικά . . . 20

2 Theoretical Background **23**

2.1 Interval Analysis . . . 23

2.1.1 Motivation . . . 23

2.1.2 History . . . 24

2.1.3 Basic Concepts . . . 25

2.1.4 Set Inversion via Interval Analysis . . . 29

2.2 Neural Networks . . . 32

2.2.1 Overview . . . 32

2.2.2 Modelizing Neurons . . . 34

2.2.3 Supervised Learning . . . 36

2.2.4 Multilayer Perceptron . . . 37

2.2.5 Training Techniques . . . 38

2.3 Parallel Computing . . . 39

2.3.1 Overview . . . 39

2.3.2 Parallel Design Paradigms . . . 39

2.3.3 Instruction-level Parallelism . . . 39

2.3.4 Hardware Multithreading . . . 40

2.3.5 Classifications of Parallel Computers . . . 40

2.3.6 Parallel Computing on CUDA GPUs . . . 42

2.3.7 CUDA Warps . . . 46

2.3.8 CUDA Memory Architecture . . . 46

2.3.9 Memory Coalescing . . . 46

2.4 Discussion . . . 47

3 Implementation 49
3.1 Branch & Bound Algorithms . . . 49
3.2 GPU Parallelization of the Parameter Estimation problem . . . 50
3.2.1 Previous Work . . . 51
3.3 Parallelization Proposal . . . 52
3.3.1 Parallel Bisection . . . 52
3.3.2 Parallel Evaluation . . . 53
3.3.3 Parallel Reduction . . . 54
3.3.4 Single and Multiple GPU(s) . . . 55
3.4 Problems . . . 57
3.4.1 Problems 1 & 2: 2D Torus and Griewank functions . . . 57
3.4.2 Problem 3: Estimating the Generalization Performance of a Neural Classifier . . . 57
3.5 Test Environment . . . 59
3.6 Results . . . 60
3.6.1 CPU Implementation . . . 60
3.6.2 Problem size . . . 61
3.6.3 Speedup . . . 62
3.6.4 Throughput . . . 62
3.6.5 Number of Runs . . . 63
3.6.6 Profiling . . . 63
3.7 Discussion . . . 63

4 Closing Words 75
4.1 Reproducibility . . . 75
4.2 Conclusion . . . 75
4.3 Future Work . . . 76

Bibliography 80

A A more detailed look on Problem 3 81

B Detailed Results 85

List of Abbreviations 89

List of Figures

2.1 A box [x] of $\mathbb{IR}^n$, with $n = 2$ and $[x] = [x_1] \times [x_2]$ 28

2.2 Regular paving of a box. The boxes in grey form a regular subpaving 29

2.3 Bracketing of the set $\mathbb{S} = \{(x, y) \mid x^2 + y^2 \in [1, 2]\}$. The frame corresponds to the box $[-2, 2] \times [-2, 2]$; precision increases from left to right. 29

2.4 Tree associated with the regular subpaving of Figure 2.2 30

2.5 Four situations encountered by SIVIA. 31

2.6 Nonlinear model of a neuron. 35

2.7 Affine transformation produced by the presence of a bias. 35

2.8 A threshold activation function. 36

2.9 A sigmoid activation function. 36

2.10 A multilayer perceptron with multiple hidden layers and outputs. 38

2.11 Instruction Level Parallelism, I F/D/E stand for Instruction Fetch/Decode/Execute, Mem stands for Memory Access and WB for register Write-Back. . . . 40

2.12 Flynn's Taxonomy of Computer Architectures. 41

2.13 A Collection of Parallel Systems. 42

2.14 A Simplified block diagram of a GPU. 43

2.15 Simplified block diagram of a HOST and a DEVICE. 44

2.16 An example of a kernel call. partialBisect happens to be the name of the device function to be executed. 45

2.17 A simplified schematic of Grid, Block, Thread and Memory hierarchies of a DEVICE and the interconnection with the HOST. 47

3.1 A Binary Tree split by 4 time periods. Each node is a subproblem or a box. Each time period represents a Bisection operation (or the result of one). T_4 is dependent on T_3, T_3 is dependent on T_2 etc. Boxes of the same level are independent problems which can be solved very easily in parallel. 53

3.2 A benchmark comparing the Bisection operation processing time between a sequential implementation on a core i7 5820k and a parallel one with an RTX 3060Ti. At 1024/2048 boxes (red square area) the GPU is faster. The GPU used is from my home setup; jitter is produced as the DEVICE is also being utilized by the OS and can be noticed in 1 and 8192 boxes. 54

3.3 A sequential code snippet using the Ibex C++ library. Some parts have been removed for simplicity. 55

3.4 A code snippet from the CUDA C++ Parallel proposed implementation. *fxy* has a different value for every thread. 55

3.5 Conflict-free sequential addressing parallel reduction which guarantees coalesced memory accesses. 56
3.6 The Input space of the 2D Torus function produced by the sequential SIVIA algorithm. The boxes have different sizes because of a dynamic tree expansion strategy. 57
3.7 The Input space of the 2D Torus function produced by the proposed parallel algorithm. Smaller epsilon values provide more accurate approximations. . . 58
3.8 The Input space of the 2D Griewank function produced by the proposed parallel algorithm. Smaller epsilon values provide more accurate approximations. 59
3.9 **Problems 1 & 2**: Number of Boxes Generated (Figure 3.9a). Rate of Box Generation (Figure 3.9b). 61
3.10 **Problems 1 & 2**: Number of Boxes Generated (Figure 3.9a). Rate of Box Generation (Figure 3.9b). 61
3.11 The speedup plot of the Torus inclusion function using float variables. $\epsilon = [1e-2, 1e-5]$ 62
3.12 The speedup plot of the Torus inclusion function using half variables. $\epsilon = [1e-2, 1e-5]$ 63
3.13 The speedup plot of the Griewank inclusion function using float variables. $\epsilon = [1e-2, 1e-5]$ 64
3.14 The speedup plot of the Griewank inclusion function using half variables. $\epsilon = [1e-2, 1e-5]$ 65
3.15 The speedup plot of the G_{net} partial sum inclusion function using float variables. $\epsilon = [0.1, 0.06]$ 65
3.16 The speedup plot of the G_{net} partial sum inclusion function using half variables. $\epsilon = [0.1, 0.06]$ 66
3.17 The throughput plot of the Torus inclusion function using float variables. $\epsilon = [1e-2, 1e-5]$ 66
3.18 The throughput plot of the Torus inclusion function using half variables. $\epsilon = [1e-2, 1e-5]$ 67
3.19 The throughput plot of the Griewank inclusion function using float variables. $\epsilon = [1e-2, 1e-5]$ 67
3.20 The throughput plot of the Griewank inclusion function using half variables. $\epsilon = [1e-2, 1e-5]$ 68
3.21 The throughput plot of the G_{net} partial sum inclusion function using float variables. $\epsilon = [0.1, 0.03]$ 68
3.22 The throughput plot of the G_{net} partial sum inclusion function using half variables. $\epsilon = [0.1, 0.03]$ 69
3.23 **Problem 1**: Number of Kernel runs. 70
3.24 **Problem 2**: Number of Kernel runs. 71
3.25 **Problem 3**: Number of Kernel runs. 72
3.26 **Problem 1**: The profiler output with $\epsilon = 1e-4$ 72
3.27 **Problem 2**: The profiler output with $\epsilon = 1e-3$ 73

3.28 **Problem 3**: The profiler output with $\epsilon = 0.06$ 73

A.1 An artificial dataset of two classes. 82
A.2 Depiction of the β cut affecting the domain of validity. 82
A.3 Depiction of training affecting the domain of validity using the same β value. 83

List of Tables

3.1 A summary of the configurations used in the benchmarks. 60

B.1 **Problem 1**: GPU Execution time(ms) using **FP32** Floating-point variables. 85
B.2 **Problem 1**: GPU Execution time(ms) using **FP16** Floating-point variables. 85
B.3 **Problem 2**: GPU Execution time(ms) using **FP32** Floating-point variables. 85
B.4 **Problem 2**: GPU Execution time(ms) using **FP16** Floating-point variables. 86
B.5 **Problem 1 & 2** : CPU Execution time(ms) using the sequential algorithm. . 86
B.6 **Problem 3**: GPU Execution time(ms) using **FP32** Floating-point variables. 86
B.7 **Problem 3**: GPU Execution time(ms) using **FP16** Floating-point variables. 86
B.8 **Problem 3**: CPU Execution time(ms) of the sequential algorithm. 86
B.9 **Problem 1**: Number of kernel executions using **FP32** Floating-point variables. 87
B.10 **Problem 1**: Number of kernel executions using **FP16** Floating-point variables. 87
B.11 **Problem 2**: Number of kernel executions using **FP32** Floating-point variables. 87
B.12 **Problem 2**: Number of kernel executions using **FP16** Floating-point variables. 87
B.13 **Problem 3**: Number of kernel executions using **FP32** Floating-point variables. 88
B.14 **Problem 3**: Number of kernel executions using **FP16** Floating-point variables. 88

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
BB	Branch and Bound
CPU	Central Processing Unit
CU	Compute Unit
DL	Deep Learning
GPGPU	General Purpose Graphical Processing Units
GPU	Graphical Processing Unit
GUI	Graphical User Interface
IA	Interval Arithmetic
ILP	Instruction-Level Parallelism
MIMD	Multiple Instructions Multiple Data
MISD	Multiple Instructions Single Data
MLP	Multilayer Perceptron
ML	Machine Learning
MP	Multiprocessor
NN	Neural Network
NUMA	Non-Uniform Memory Access
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SISD	Single Instruction Single Data
SIVIA	Set Inversion Via Interval Analysis
SM	Streaming Multiprocessor
SP	Streaming Processor
TLP	Thread-Level Parallelism
UMA	Uniform Memory Access
VRAM	Video Random Access Memory

Chapter 1

Introduction

1.1 English

At the core of many engineering problems is the solution of sets of equations and inequalities, and the optimization of cost functions. Unfortunately, except in special cases, such as when a set of equations is linear in its unknowns or when a convex cost function has to be minimized under convex constraints, the results obtained by conventional numerical methods are only local and cannot be guaranteed. This means, for example, that the actual global minimum of a cost function may not be reached, or that some global minimizers of this cost function may escape detection. By contrast, Interval Analysis makes it possible to obtain guaranteed approximations of the set of all the actual solutions of the problem being considered[1]. One IA technique to achieve this is Set Inversion via Interval Analysis (or SIVIA) which was introduced by Luc Jaulin and Eric Walter in 1993[2], where an initial Bounded Space gets divided into smaller spaces or subpavings until a threshold ϵ is reached. This threshold exists because otherwise this algorithm would bisect the input infinitely. The input space can be multidimensional, and is called a box when that is the case, whereas the threshold represents the width of a box, generally its largest dimension. The only pieces missing from this explanation, are the Interval Inclusion Function and the Solution Set, where the inclusion(meaning that the inclusion function belongs entirely into the solution set) of the two result in the aforementioned guaranteed solution approximations. The input space is formed by a finite set of boxes that are classified into the solution set after their evaluation of an Interval Inclusion Function. Depending on the ϵ threshold and the width of the initial input space, the problem size (number of boxes) can grow exponentially large and consequently become extremely time consuming or even impossible to compute. When using this technique for optimization, we have the privilege of existing bounds, which imply a prioritization sequence of the problems to be explored. When trying to solve a different problem where bounds are missing, as is the case in this thesis, exhaustive search of the input space is required. This is where mass parallelization, commonly found in GPU devices, can help reduce the time needed to explore these very large problem spaces.

A GPU is a device which was initially created to accelerate computer graphics. Since 2006, however, NVIDIA provides the CUDA parallel computational platform and programming model, enabling the expansion of the aforementioned parallelization benefits to

applications beyond computer graphics[3]. Today, CUDA software is used to accelerate problems such as Simulations, Deep Learning models, Image Recognition, Reinforcement Learning and is also being used in Large Language Models with platforms like ChatGPT appearing and becoming the center of public attention. A GPU consists of many Streaming Multiprocessors(SMs) or Compute Units(CUs) which are capable to run hundreds if not thousands of threads in parallel, the specific configuration depending on the GPU hardware architecture. This very characteristic is what made these devices so valuable and is the reason so many ML applications managed to grow so much in scale.

This thesis explores and exploits the capabilities by accelerating SIVIA using single and multiple GPUs, enabling the exploration of larger input spaces, for more precise approximations and to challenge larger problems in the future. The main focus is on a recent development, which uses SIVIA to estimate the generalization performance of Neural Networks in a deterministic manner, without requiring a train/test split of the data[4]. The model used is a Multi-Layer Perceptron provided by the author of that paper. This technique when deployed with a sequential algorithm requires days of execution time, therefore it could really benefit from an alternative parallel technique. The parallel algorithm proposed is a coarse-grained approach, meaning that it parallelizes operations on the data.

1.2 Ελληνικά

Στον πυρήνα πολλών προβλημάτων μηχανικής φύσεως βρίσκεται η λύση συνόλων εξισώσεων και ανισώσεων, καθώς και η βελτιστοποίηση συναρτήσεων κόστους. Δυστυχώς, τα αποτελέσματα που εξάγονται από τυπικές αριθμητικές μεθόδους είναι μόνο τοπικά ή μπορεί και να μην υπάρχουν. Αυτό σημαίνει, για παράδειγμα, πως η πραγματική βέλτιστη ελάχιστη τιμή μιας συνάρτησης κόστους μπορεί να μην βρεθεί, ή ότι κάποιοι καθολικοί βελτιστοποιητές αυτής της συνάρτησης κόστους μπορεί να μην την εντοπίσουν καθόλου. Αντιθέτως, η Ανάλυση Διαστημάτων κάνει εφικτή την εξασφαλισμένη προσέγγιση του συνόλου όλων των πιθανών λύσεων για κάθε πρόβλημα. Μία τεχνική Ανάλυσης Διαστημάτων που μπορεί να επιφέρει τέτοια αποτελέσματα είναι η Αντιστροφή Συνόλου μέσω Ανάλυσης Διαστημάτων (Set Inversion Via Interval Analysis-SIVIA) των Luc Jaulin & Eric Walter(1993), όπου ένας αρχικός οριοθετημένος χώρος διαιρείται σε μικρότερους επιμέρους χώρους μέχρι να ξεπεραστεί μία τιμή ϵ. Αυτή η τιμή υπάρχει ώστε ο αλγόριθμος να μην διαιρεί τον χώρο εισόδου εις το άπειρο. Αυτός ο χώρος εισόδου μπορεί να είναι πολυδιάστατος, και αναφέρεται ως κουτί σε αυτή τη περίπτωση και η τιμή ϵ αφορά το εύρος του κουτιού, πιο συγκεκριμένα το εύρος της μεγαλύτερής του διάστασης. Το μόνο που λείπει από αυτή τη περιγραφή, είναι η συνάρτηση εγκλεισμού διαστημάτων και το πεδίο τιμών. Η εκχώρηση των δύο, εννοώντας την πλήρη συμπερίληψη του διαστήματος της συνάρτησης εγκλεισμού στο εύρος του πεδίου τιμών, οδηγεί στην δημιουργία της εξασφαλισμένης προσέγγισης που επιλύει το πρόβλημα. Ο χώρος εισόδου αποτελείται από ένα πεπερασμένο πλήθος κουτιών, τα οποία κατηγοριοποιούνται ως λύσεις (εμπεριέχονται στο σύνολο λύσεων) αφότου εκτιμηθούν από μία συνάρτηση εγκλεισμού διαστημάτων. Δεδομένου μιας οριακής τιμής ϵ και ενός εύρους του αρχικού χώρου εισόδου, το μέγεθος του προβλήματος (αριθμός κουτιών) μπορεί να μεγαλώσει εκθετι-

κά έχοντας ως συνέπεια το να απαιτείται πολύς μεγάλος υπολογιστικός χρόνος ή ακόμη και να είναι αδύνατο να εκτιμηθεί. Όταν χρησιμοποιείται αυτή η τεχνική σε προβλήματα βελτιστοποίησης, υπάρχει το προτέρημα του να υπάρχουν ανώτατα και κατώτατα όρια, τα οποία συνήθως οδηγούν σε κάποιο κανόνα προτεραιότητας σχετικά με το ποια υπο-προβλήματα θα προτιμηθούν προς επίλυση. Προσπαθώντας να χρησιμοποιήσουμε τη τεχνική σε διαφορετικά προβλήματα, χωρίς την ύπαρξη τέτοιων ορίων, απαιτείται η εξαντλητική αναζήτηση του χώρου εισόδου. Ως συνέπεια, χρειάζονται τεχνικές μαζικής παραλληλοποίησης, που συνήθως χαρακτηρίζουν συσκευές επεξεργασίας γραφικών (ή κοινώς Κάρτες Γραφικών-GPUs), οι οποίες μπορούν να συμβάλλουν στην μείωση του απαιτούμενου χρόνου υπολογισμού και να επιτρέψουν την εξερεύνηση ακόμη μεγαλύτερων χώρων εισόδου.

Μία κάρτα γραφικών είναι μία συσκευή η οποία είχε δημιουργηθεί αρχικά για την επιτάχυνση του των γραφικών στους ηλεκτρονικούς υπολογιστές. Η NVIDIA, ωστόσο, το 2006 ανέπτυξε την CUDA πλατφόρμα παράλληλης επεξεργασίας, επιτρέποντας την χρήση των ήδη ισχυρών δυνατοτήτων παραλληλισμού για εφαρμογές πέρα από τα γραφικά ηλεκτρονικών υπολογιστών. Σήμερα, λογισμικά βασισμένα στην CUDA έχουν αναπτυχθεί για την επιτάχυνση εφαρμογών Προσομοιώσης, Βαθιάς Μάθησης, Αναγνώρισης Εικόνας, Ενισχυτικής Μάθησης. Επιπρόσθετα, έχει χρησιμοποιηθεί για εφαρμογές Μεγάλων Γλωσσικών Μοντέλων, με αποτέλεσμα την ίδρυση πλατφόρμων όπως το chatGPT, οι οποίες βρίσκονται στο κέντρο της επικαιρότητας. Μία κάρτα γραφικών αποτελείται από πολυεπεξεργαστές(Streaming Multiprocessors-SMs) ή υπολογιστικές μονάδες (CUs) οι οποίες έχουν τη δυνατότητα να εκτελέσουν εκαντοντάδες αν όχι χιλιάδες επεξεργαστικά νήματα παράλληλα, των οποίων ο αριθμός εξαρτάται από την εκάστοτε αρχιτεκτονική. Αυτό χαρακτηριστικό αποτελεί και τον βασικότερο λόγο που αυτές οι συσκευές είναι τόσο πολύτιμες καθώς επέτρεψε την ανάπτυξη εφαρμογών Μηχανικής Μάθησης (ML) μεγάλης κλίμακας.

Σε αυτή τη διπλωματική εργασία εξερευνούνται οι προαναφερθείσες δυνατότητες, επιταχύνοντας τη τεχνική SIVIA χρησιμοποιώντας μία ή περισσότερες κάρτες γραφικών μέρος διαφορετικών υπολογιστικών συστημάτων. Στόχος των οποίων η επίτευξη ταχύτερων χρόνων επεξεργασίας προσφέροντας τη δυνατότητα να εξερευνηθούν μεγαλύτερα προβλήματα στο μέλλον. Ο κύριος στόχος της εργασίας είναι σε μία σχετικά πρόσφατη εξέλιξη, η οποία χρησιμοποιεί την τεχνική SIVIA για την εκτίμηση της ικανότητας γενίκευσης ενός Νευρωνικού Δικτύου με ντετερμινιστικό τρόπο, χωρίς να απαιτείται διαχωρισμός συνόλου εκπαίδευσης και επαλήθευσης στα δεδομένα. Το μοντέλο που χρησιμοποιείται είναι ένα πολυεπίπεδο Perceptron το οποίο το παρείχε προ-εκπαιδευμένο ο συγγραφέας της πρωτότυπης εργασίας. Αυτή η τεχνική όταν εκτελείται σειριακά χρειάζεται ημέρες υπολογισμού για να επιστρέψει την λύση, συνεπώς θα μπορούσε να κερδίσει από μια εναλλακτική παράλληλη τεχνική. Ο παράλληλος αλγόριθμος που προτείνεται σε αυτήν την εργασία αποτελεί μία προσέγγιση «χονδρού-κόκκου» (coarse-grained), εννοώντας την παραλληλοποίηση των διαδικασιών στα δεδομένα.

Chapter 2

Theoretical Background

2.1 Interval Analysis

2.1.1 Motivation

In elementary mathematics, a problem is "solved" when we write down an exact solution. We solve the equation

$$x^2 + x - 6 = 0$$

by factoring and obtaining the roots $x_1 = -3$ and $x_2 = +2$. Few high school algebra teachers would be satisfied with an answer of the form "One root lies between -4 and -2, while the other lies between 1 and 3". We need not look far, however, to find even elementary problems where answers of precisely this form are appropriate. The quadratic equation

$$x^2 - 2 = 0$$

has the positive solution $\sqrt{2}$. The number it designates cannot be represented exactly with a finite number of digits. Indeed, the notion of irrational number entails some process of approximation from above and below. Archimedes (287-212 BCE) was able to bracket π by taking a circle and considering inscribed and circumscribed polygons. Increasing the numbers of polygonal sides, he obtained both an increasing sequence of lower bounds and a decreasing sequence of upper bounds for this irrational number. Aside from irrational numbers, many situations involve quantities that are not exactly representable. In machine computation, representable lower and upper bounds are requires to describe a solution rigorously. The need to enclose a number also arises in the physical sciences. Since an experimentally measured quantity will be known with only limited accuracy, any calculation involving this quantity must begin with inexact initial data. Newton's law

$$F = ma$$

permits us to solve for the acceleration a of a body exactly only when the force F and mass m are known exactly. If the latter quantities are known only to lie in certain ranges like

$$F_0 - \Delta F \leq F \leq F_0 + \Delta F$$

$$m_0 - \Delta m \leq m \leq m_0 + \Delta m$$

then a can only be bounded above and below:

$$a_l \leq a \leq a_u$$

It is easy to determine how a_l and a_u depend on F_0, m_0, ΔF, and Δm. For more complicated relations ordinary algebra can be cumbersome. In Interval Analysis, we phrase inequality statements in terms of *closed intervals* on the real line. We think of an interval as a set of numbers, which are commonly represented as an ordered pair. Instead of a, we write

$$a \in [a_l, a_u]$$

The interval $[a_l, a_u]$ is called an *enclosure* of a. The essence is that we would like to know F and m exactly so that we can get a exactly. In other circumstances, however, we might wish to treat F and m as *paremeters* and intentionally vary them to see how a varies. The act of merely enclosing a solution might seems rather weak, since it fails to yield *the solution* itself. While this is true, the degree of satisfaction involved in enclosing a solution can depend strongly on the *tightness* of the enclosure obtained. Returning to the initial example, we might be more satisfied with answers of the form

$$x_1 \in [-3.001, -2.999], \; x_2 \in [1.999, 2.001]$$

In addition, if we obtain something like

$$x \in [0.66666, 0.66667]$$

then we *do know x to four decimal places*. There are times where we can and should be satisfied with rather loose bounds on a solution. It might be better to know that $y \in [59, 62]$ *rigorously* than to have an “answer” of the form $y \approx 60$ with no idea of how much error might be present. If we can compute an interval $[a, b]$ containing an exact solution x to some problem, then we can take a *midpoint* $m = (a + b)/2$ of the interval as an approximation to x and have $|x - m| \leq w/2$, where $w = b - a$ is the *width* of the interval. Hence we obtain both an approximate solution *and* error bounds on the approximation.

2.1.2 History

The story of IA methods begin in 1962 by Moore, who presented his doctorate on the use of intervals to analyze and control numerical errors[5]. His next step was to publish his first book Interval Analysis in 1966, which remains a reference to this day and is also the reference which was used in the previous section of this thesis[6]. During the same period, Hansen studied interval manipulation in linear algebra[7], and a group of German researches including Alefeld, Krawczyk and Nickel developed many aspects of computer implementation[1]. During the first twenty years, the spreading of the interval methodology remained relatively confined to the periphery of the initial seeds, notably in

Germany within Karlsruhe University.[8] Among the new adepts who brought important advances, one of the most significant ones were Neumaier on the solution of sets of linear and non-linear equations (1985), Ratschek and Rokne (1984) as well as Kearfott(1989) on optimization. During the 1990s, interval analysis has recruited a larger community. It now has its own journal Interval Computations, created in 1991 and renamed Reliable Computing in 1995 as well as other regular international conferences, one of which is SWIM (Summer Workshop on Interval Methods), where I also participated in 2019[9].

2.1.3 Basic Concepts

As the title of this section denotes, many IA definitions and operations will follow. It should be noted that only the most important concepts concerning later sections will be described. This is an intentional move to avoid distracting the reader from the focus of this thesis. The problem IA methods will be applied to in this thesis requires a bounded-error set estimation technique. Hence, more details on open-ended unbounded Intervals and the process that resulted in the following definitions can be found in [1] as well as [10].

Intervals

First and foremost, an interval real $[x]$ is a connected subset of $\mathbb{R}$. The *lower bound* $lb([x])$ of an interval $[x]$, also denoted by $\underline{x}$, is defined as

$$\underline{x} = lb([x]) \triangleq sup\{a \in \mathbb{R} \cup \{-\infty, \infty\} \mid \forall x \in [x], a \leq x\}. \tag{2.1}$$

Its *upper bound* $ub([x])$, also denoted by $\overline{x}$, is defined as

$$\overline{x} = ub([x]) \triangleq inf\{b \in \mathbb{R} \cup \{-\infty, \infty\} \mid \forall x \in [x], x \leq b\}. \tag{2.2}$$

For instance, if $[x] =]-3, 7]$ then $\underline{x} = -3$ and $\overline{x} = 7$; if $[x] =]-\infty, \infty[$ then $\underline{x} = -\infty$ and $\overline{x} = \infty$.

The *width* of *any* non-empty interval $[x]$ is

$$w([x]) \triangleq \overline{x} - \underline{x}, \tag{2.3}$$

so $w(]3, \infty[) = \infty$. The *midpoint* or *center* of any bounded and non-empty interval $[x]$ is defined as

$$mid([x]) \triangleq \frac{\underline{x} + \overline{x}}{2}. \tag{2.4}$$

We should also define the set-theoretic operations which can be applied to intervals. The *intersection* of two intervals $[x]$ and $[y]$, defined by

$$[x] \cap [y] \triangleq \{z \in \mathbb{R} \mid z \in [x] \text{ and } z \in [y]\}, \tag{2.5}$$

is always an interval, This is not the case for their *union*

$$[x] \cup [y] \triangleq \{z \in \mathbb{R} \mid z \in [x] \text{ or } z \in [y]\}. \tag{2.6}$$

To make the set of intervals closed with respect to union, an *interval hull* of a subset $\mathbb{X}$ of $\mathbb{R}$ is defined as the smallest interval $[\mathbb{X}]$ that contains it ([1] p.16). For instance, the interval hull of $]2,3] \cup [5,7]$ is the interval $]2,7]$. The *interval union* or $[x]$ and $[y]$, denoted by $[x] \sqcup [y]$ is defined as the interval hull of $[x] \cup [y]$, i.e.,

$$[x] \sqcup [y] \triangleq [[x] \cup [y]]. \tag{2.7}$$

In the same manner,

$$[x]\ [\backslash]\ [y] = [[x] \setminus [y]] = [\{x \in [x] \mid x \notin [y]\}] \tag{2.8}$$

The Cartesian product of two intervals is not an interval but a box of $\mathbb{R}^2$

Closed Intervals

Denote by $\mathbb{IR}$ the set of closed intervals, since $\mathbb{R}$ and $\emptyset$ are both open and closed, they both belong to $\mathbb{IR}$ and any element of $\mathbb{IR}$ can be written in one of the following forms: $[a,b],]-\infty,a], [a,\infty[,]-\infty,\infty[$ or $\emptyset$, where a and b are real numbers such that $a \le b$. Any $[x]$ of $\mathbb{IR}$ can be specified in a unique way by its lower bound $\underline{x}$ and its upper bound $\overline{x}$. It will be notated from now on as $[x] = [\underline{x}, \overline{x}]$ even if bounds are infinite. Thus, $[0,\infty]$ should be interpreted as $[0,\infty[$. Intervals may be viewed as *sets* or as *couples of elements of* $\mathbb{R}$ on which arithmetic can be built. Couples of the form $[\infty,\infty], [-\infty,-\infty]$ and $[a,b]$ with $a > b$ do not correspond to intervals. When $\underline{x}$ and $\overline{x}$ are equal, the interval $[x]$ is said to be *punctual* (or degenerate). Any real number could thus be represented as a punctual interval and vice versa.

Interval Operations of Closed Intervals

The interval union of two non-empty closed intervals $[x]$ and $[y]$ satisfies

$$\forall [x] \in \mathbb{IR}, \forall [y] \in \mathbb{IR}, [x] \sqcup [y] = \left[min\{\underline{x}, \underline{y}\}, max\{\overline{x}, \overline{y}\} \right]. \tag{2.9}$$

The intersection of two non-empty closed intervals $[x]$ and $[y]$ satisfies

$$\begin{aligned} [x] \cap [y] = \left[max\{\underline{x}, \underline{y}\}, min\{\overline{x}, \overline{y}\} \right] \text{ if } max\{\underline{x}, \underline{y}\} \le min\{\overline{x}, \overline{y}\}, \\ = \emptyset \text{ otherwise} \end{aligned} \tag{2.10}$$

If a is a real number and $[x]$ a non-empty interval, then the interval

$$a[x] \triangleq \{ax \mid x \in [x]\} \tag{2.11}$$

is given by

$$a[x] = [a\underline{x}, a\overline{x}] \ \textit{if} \ a \geq 0 \\ = [a\overline{x}, a\underline{x}] \ \textit{if} \ a < 0. \tag{2.12}$$

For non-empty closed intervals, we can perform addition(+), subtraction(−), multiplication(∗) and division(/) as follows

$$[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}], \tag{2.13}$$

$$[x] - [y] = [\underline{x} - \overline{y}, \ \overline{x} - \underline{y}] \tag{2.14}$$

$$[x] * [y] = [min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}.\overline{x}\overline{y}\}, max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}]. \tag{2.15}$$

$$\begin{aligned} 1/[y] &= \emptyset & &\textit{if} [y] = [0, 0], \\ &= [1/\overline{y}, 1/\underline{y}] & &\textit{if} 0 \notin [y], \\ &= [1/\overline{y}, \infty[& &\textit{if} \underline{y} = 0 \text{ and } \overline{y} > 0, \\ &=] - \infty, 1/\underline{y}] & &\textit{if} \underline{y} < 0 \text{ and } \overline{y} = 0, \\ &=] - \infty, \infty[& &\textit{if} \underline{y} < 0 \text{ and } \overline{y} > 0, \end{aligned} \tag{2.16}$$

$$\text{and} \quad [x]/[y] = [x] * (1/[y]).$$

The product of two intervals can be denoted indifferently by $[x] * [y]$ or $[x][y]$. When applied to punctual intervals $[x]$ and $[y]$, the previous rules simplify into the usual rules of real arithmetic, which is why interval arithmetic can claim to be an extension of the latter.
Elementary interval functions can also be expressed in terms of bounds. For any non-empty [x],

$$[exp]([x]) = [exp(\underline{x}), exp(\overline{x})]. \tag{2.17}$$

However, for non-monotonic functions things are different. $[sin]([0, \pi]) = [0, 1]$ differs from the interval $[sin(0), sin(\pi)] = (0, 0)$. Specific algorithms have been built for those cases. An example algorithm is given by [1]:

ALGORITHM 2.1: *sin(in: [x]; out:[r])*

1: **if** $\exists k \in \mathbb{Z} \mid 2k\pi - \pi/2 \in [x]$ **then**
2: $\quad \underline{r} = -1;$
3: **else**
4: $\quad \underline{r} = min(sin\underline{x}, sin\overline{x});$
5: **end if**
6: **if** $\exists k \in \mathbb{Z} \mid 2k\pi + \pi/2 \in [x]$ **then**
7: $\quad \underline{r} = 1;$
8: **else**
9: $\quad \overline{r} = min(sin\underline{x}, sin\overline{x});$
10: **end if**

Interval Vectors

An *interval real vector* $[x]$ is a subset of $\mathbb{R}^n$ that can be defined as the Cartesian product of n closed intervals. When there is no ambiguity, $[x]$ is be called an interval vector, or a *box*. It is written as

$$[x] = [x_1] \times [x_2] \times \ldots \times [x_n], \text{ with } [x_i] = [\underline{x_i}, \overline{x_i}] \text{ for } i = 1, \ldots, n. \tag{2.18}$$

Its ith *interval component* $[x_i]$ is the projection of $[x]$ onto the ith axis. The empty set of $\mathbb{R}^n$ should be written as $\varnothing \times \ldots \times \varnothing$ because all of its interval components are empty. Expressions such as $[x] = \varnothing \times [0, 1]$ are prohibited, because $[0, 1]$ is not the projection of $[x]$ onto the second axis. This guarantees the uniqueness of notation of a given box. The set of all n-dimensional boxes will be denoted by $\mathbb{IR}^n$. Non-empty boxes are n-dimensional axis-aligned parallelepeds. Many of the aforementioned notions described in

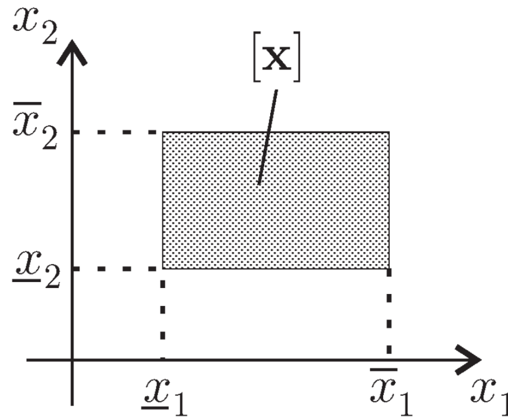


Figure 2.1: *A box [x] of* $\mathbb{IR}^n$*, with* $n = 2$ *and* $[x] = [x_1] \times [x_2]$

previous sections apply to boxes. For instance, a box will be said to be punctual if *all* its interval components are. From the list of box-wide operations, only one is needed in its purest sense (meaning that it performs the operation given a box as the input instead of an interval), which is the width of a box given by

$$w([x]) \triangleq max_{1 \leq i \leq n} w([x_i]). \tag{2.19}$$

The rest of the proper definitions, which are extensions of the ones provided in previous sections, can be found in [1].

Inclusion Functions

Consider a function f from $\mathbb{R}^n$ to

A regular subpaving can also be represented as a binary tree. A binary tree contains a finite set of *nodes*. This set may be empty, may contain a single node, the *root* of the tree, or may contain two binary trees with an empty intersection, namely the *left* and *right subtrees*. Thus, we can describe Figure 2.2 as the binary tree of Figure 2.4. On this

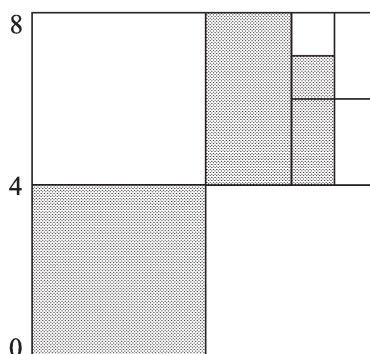


Figure 2.2: *Regular paving of a box. The boxes in grey form a regular subpaving*

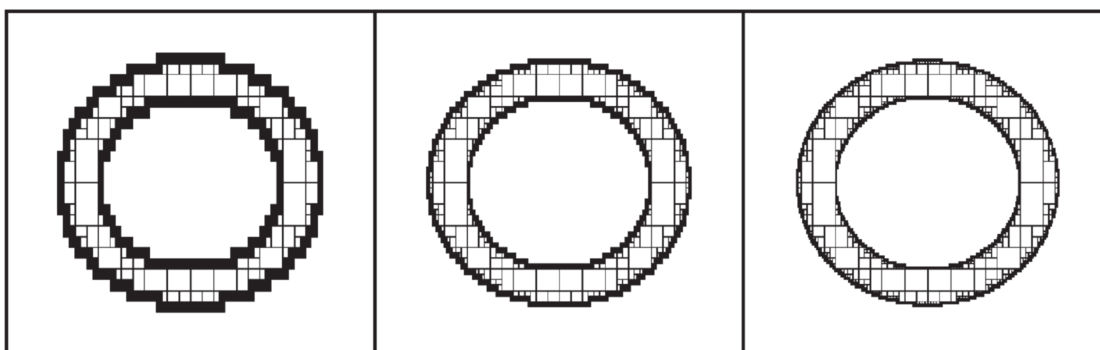


Figure 2.3: *Bracketing of the set* $\mathbb{S} = \{(x, y) \mid x^2 + y^2 \in [1, 2]\}$. *The frame corresponds to the box* $[-2, 2] \times [-2, 2]$*; precision increases from left to right.*

figure, A is the root of the tree, B and C are respectively its left and right children. They are siblings as they have the same parent node A. A has a left subtree and a right subtree, the right subtree of B is empty. Finally D has no subtree and it is colled a degenerate node of *leaf*. The growth of a binary tree's branches is defgined by how the initial box $[x_0]$, which corresponds to the root of the tree, is bisected. Any leaf indicates that the box it stands for belongs to the subpaving. The *depth* of a box is the number of bisections necessary to get it from the root box. Notice the recursiveness of this structure. We can perform 4 basic operations on regular subpavings. We can unite sibling subpavings, take the union, intersect and test whether a box is included in a subpaving. In this thesis, operations occur on an interval basis, therefore describing those operations on the subpaving level is beyond the required scope. If it is of interest, the reader can find more on pages 52-54 of [1].

2.1.4 Set Inversion via Interval Analysis

First introduced by Jaulin and Walter in 1993[2]. This algorithm utilizes all the aforementioned definitions to compute an unknown input set, using the known image $[Y]$ and an inclusion function $[f]([x])$. It is formally described as:
Let f be a possibly non-linear function from $\mathbb{R}^n$ to $\mathbb{R}^m$ and let $\mathbb{Y}$ be a subset of $\mathbb{R}^m$ (it can

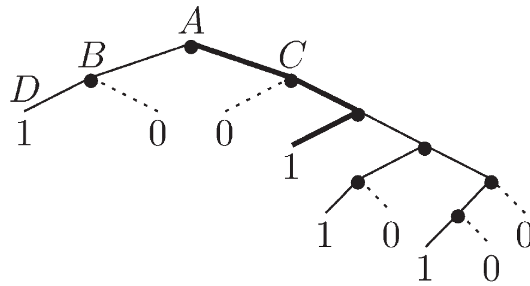


Figure 2.4: *Tree associated with the regular subpaving of Figure 2.2*

be a subpaving). Set inversion is the characterization of

$$\mathbb{X} = \{x \in \mathbb{R}^n \mid f(x) \in \mathbb{Y}\} = f^-1(\mathbb{Y}). \tag{2.20}$$

For any $\mathbb{Y} \subset \mathbb{R}^m$ and for any function f admitting a convergent inclusion function $[f](.)$, two regular subpavings $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ that satisfy $\underline{\mathbb{X}} \subset \mathbb{X} \subset \overline{\mathbb{X}}$ can be obtained with the SIVIA algorithm[2]. **SIVIA** requires a search box X_0 to which $\overline{\mathbb{X}}$ is guaranteed to belong. This nice figure 2.5 from page 57 of [1] describes the basic steps of this algorithm quite well, assuming that $\mathbb{Y}$ is a regular subpaving. To facilitate the steps to perform this algorithm we must first list the four cases which can be encountered.

- If $[f]([x])$ has a non-empty intersection with $\mathbb{Y}$, but is not entirely in $\mathbb{Y}$, then $[x]$ may contain a part of the solution set (Figure 2.5a). When this is the case $[x]$ is said to be *undertermined*. If it has a width greater than an arbitrary (usually small) precision parameter ϵ, then it should be bisected, creating two or more offspring boxes out of $[x]$. Then, the test is recursively applied to these newly generated boxes.

- If $[f](x)$ has an empty intersection with $\mathbb{Y}$, then $[x]$ does not belong to $\mathbb{X}$ and can be cut off from the solution tree (Figure 2.5b).

- If $[f]([x])$ is entirely in $\mathbb{Y}$, then $[x]$ belongs to the solution subpaving $\mathbb{X}$, and is stored in $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ (Figure 2.5c).

- If the box is considered undertermined and its width is lower than ϵ, then it is deemed too small and is stored in the outer approximation $\overline{\mathbb{X}}$ of $\mathbb{X}$ (Figure 2.5d).

To sum up, the SIVIA algorithm starts with an initial box X_0, is then sent to an inclusion function $[f]([x])$ which can be non-linear, the aforementioned tests are performed to the resulting $[y]$ and the solution set is progressively created until all generated $[x]$ boxes have been evaluated. The predefined ϵ threshold is set since the algorithm would run infinitely otherwise. Algorithm 2.2 demonstrates the above.
The boxes that remain undetermined at the end of an algorithm execution comprise the *uncertainty layer*.

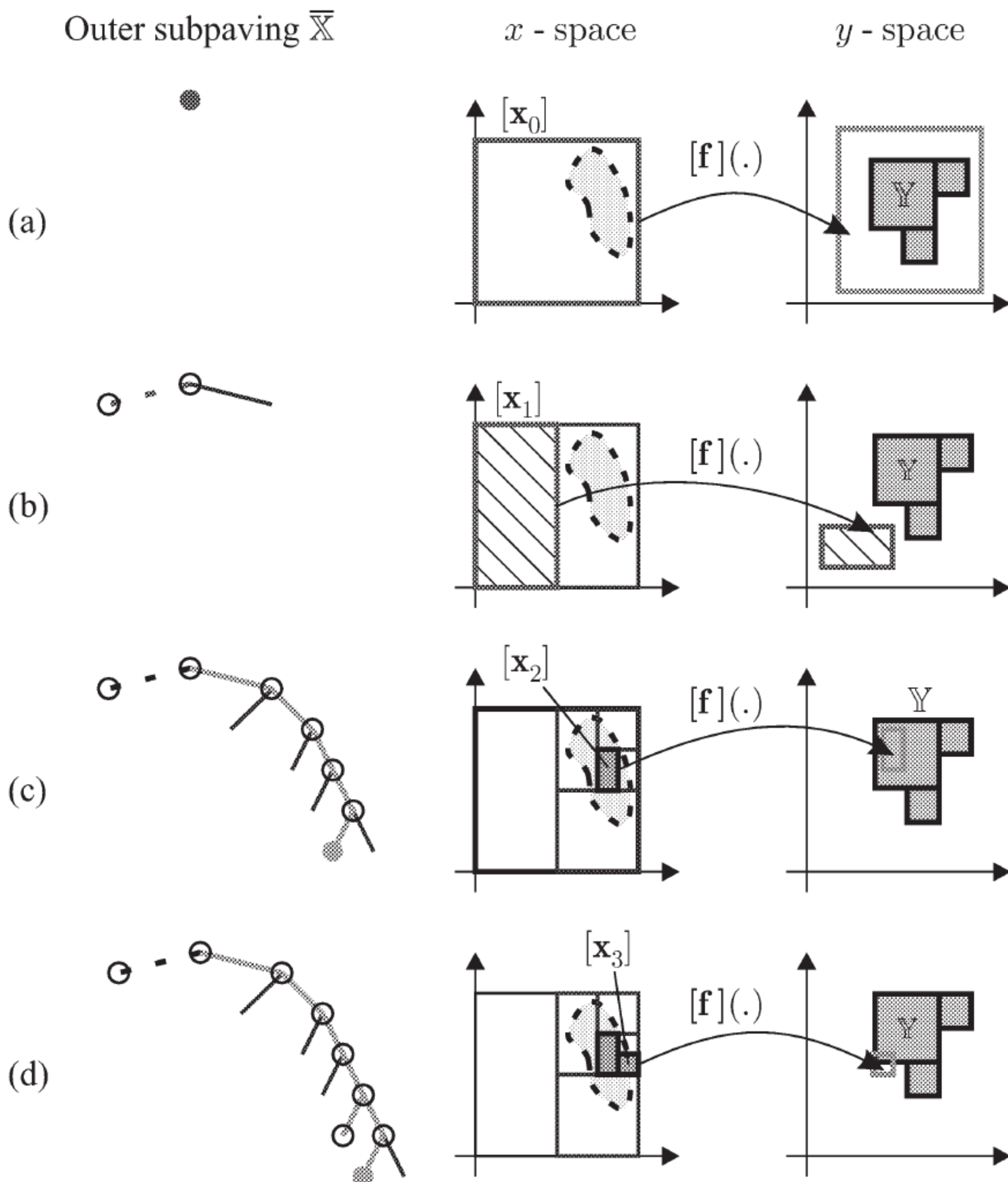


Figure 2.5: *Four situations encountered by SIVIA.*

Applications

SIVIA can be applied to any problem that requires the discovery of an input(or validation) set. Given that it can work with non-linear functions, this means that its usage extends to any problem which was previously deemed unsolveable. The technique is deterministic and guarantees a solution, proving extremely useful in many engineering and optimization problems. And this is exactly the case as [11],[12],[13],[14] use this algorithm for robotics-related problems, such as inverse kinematics and path planning. While it has been traditionally used for global optimization problems [15],[16],[17] and [18], the technique does not shy away from being useful to an even bigger variety of problems. Here [19] it is used for the estimation of electrochemical parameters and here [20] it was

ALGORITHM 2.2: *SIVIA(in: $f, \mathbb{Y}, [x], \epsilon$; in/out:$\underline{\mathbb{X}}, \overline{\mathbb{X}}$)*

1: **if** $[f]([x]) \cap \mathbb{Y} = \emptyset$ **then** return;
2: **end if**
3: **if** $[f]([x]) \subset \mathbb{Y}$ **then** {
4: $\quad \overline{\mathbb{X}} \leftarrow \overline{\mathbb{X}} \cup [x]$;
5: $\quad \underline{\mathbb{X}} \leftarrow \underline{\mathbb{X}} \cup [x]$;
6: $\quad$ *return*; }
7: **end if**
8: **if** $w([x]) < \epsilon$ **then** {
9: $\quad \overline{\mathbb{X}} \leftarrow \overline{\mathbb{X}} \cup [x]$;
10: $\quad$ *return*; }
11: **end if**
12: SIVIA($f, \mathbb{Y}, L[x], \epsilon, \underline{\mathbb{X}}, \overline{\mathbb{X}}$);
13: SIVIA($f, \mathbb{Y}, R[x], \epsilon, \underline{\mathbb{X}}, \overline{\mathbb{X}}$);

used among other problems for the colorimetric determination of formaldehyde. Finally, closer to the topic of this thesis, S.P. Adam uses this technique for the inversion of Neural Networks, for purposes such as bounding the search space of parameters[21], estimation of the generalization capability of a NN[4] or simply for the input space mapping of a classifier[22].

2.2 Neural Networks

2.2.1 Overview

Work on artificial neural networks, commonly referred to as “neural networks,” has been motivated right from its inception by the recognition that the human brain computes in an entirely different way from the conventional digital computer. The brain is a highly complex, nonlinear, and parallel computer. It has the capability to organize its structural constituents, known as neurons, so as to perform certain computations (e.g., pattern recognition, perception, and motor control) many times faster than the fastest digital computer in existence today. A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects, one being that knowledge is acquired via a learning process and that interneuron synaptic weights are used to store the acquired that knowledge. The procedure used to perform the learning process is called a learning algorithm, and it is a function that is used to modify the synaptic weights of the network in an orderly fashion in order to attain a desired design objective. What is very important for a neural network is its capacity to *generalize*, meaning the production of reasonable outputs for inputs not encountered during training (learning). This should not imply that a neural network has the ability to generalize each and every problem, in practice, a complex problem requires multiple simple solutions, in a system engineering fashion. A neural network offers the following useful properties and capabilities:

- **Nonlinearity**. An artificial neuron can be linear or nonlinear. A neural network, made up of an interconnection of nonlinear neurons, is itself nonlinear. Moreover, the nonlinearity is of a special kind in the sense that it is distributed throughout the network. This property is very important if the input signal is inherently nonlinear.

- **Input-Output Mapping**. A popular paradigm of learning, called supervised learning, involves modification of the synaptic weights of a neural network by applying a set of labeled training examples. Each example consists of a unique input signal and a corresponding desired (target) response.The network is presented with an example picked at random from the set, and the synaptic weight of the network are modified to minimize the difference between the desired response and the actual response of the network produced by the input signal in accordance with an appropriate statistical criterion. The training of the network is repeated for many examples in the set, until the network reaches a steady state where there are no further significant changes in the synaptic weights.The previously applied training examples may be reapplied during the training session, but in a different order. Thus the network learns from the examples by constructing an *input-output mapping* for the problem at hand.

- **Adaptivity**. Neural networks have a built-in capability to adapt their synaptic weights to changes in the surrounding environment. In particular, a neural network trained to operate in a specific environment can be easily retrained to deal with minor changes in the operating environmental conditions. Moreover, when it is operating in a nonstationary environment (i.e., one where statistics change with time), a neural network may be designed to change its synaptic weights in real time.The natural architecture of a neural network for pattern classification, signal processing, and control applications, coupled with the adaptive capability of the network, makes it a useful tool in adaptive pattern classification, adaptive signal processing, and adaptive control.

- **Evidential Response**. In the context of pattern classification, a neural network can be designed to provide information not only about which particular pattern to select, but also about the confidence in the decision made.This latter information may be used to reject ambiguous patterns, should they arise, and thereby improve the classification performance of the network.

- **Contextual Information**. Knowledge is represented by the very structure and activation state of a neural network. Every neuron in the network is potentially affected by the global activity of all other neurons in the network. Consequently, contextual information is dealt with naturally by a neural network.

- **Uniformity of Analysis and Design**. neural networks enjoy universality as information processors.We say this in the sense that the same notation is used in all domains involving the application of neural networks. Neurons, in one form or other represent an ingredient *common* to all neural networks. This commonality makes it

possible to *share* theories and learning algorithms in different applications of neural networks. Therefore, modular networks can be built through a seamless integration of modules.

More information about those properties can be found in [23].

2.2.2 Modelizing Neurons

A neuron is an information-processing unit that is fundamental to the operation of a neural network. The three basic elements of a neural model are

1. A **set of synapses**, or connecting links, each of which is characterized by a weight or strength of its own.
2. An **adder** for summing the input signals, weighted by the respective synaptic strengths of the neuron.
3. An **activation function** for limiting the amplitude of the output of a neuron.The activation function is also referred to as a squashing function, in that it squashes (limits) the permissible amplitude range of the output signal to some finite value.

In mathematical terms, we can describe the model by a set of equations (Figure 2.7).

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{2.21}$$

$$y_k = \phi(u_k + b_k) \tag{2.22}$$

$x_1, x_2, ..., x_m$ are the input signals, $w_k1, w_k2, ..., w_{km}$ are the synaptic weights of the neuron k, v_k is the linear combiner, b_k is the bias, $\phi(\cdot)$ is the activation function and y_k is the output signal of the neuron. The use of the bias b_k has the effect of applying an affine transformation to the output u_k of the linear combiner as shown by Figure 2.6.

$$v_k = u_k + b_k \tag{2.23}$$

Activation functions

The activation function, denoted by $\phi(v)$, defines the output of a neuron in terms of the induced local field v. In what follows, we identify two basic types of activation functions:

- **Threshold Function**. (Figure 2.8) This type of function is described by

$$\phi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases} \tag{2.24}$$

$$y_k = \begin{cases} 1 & \text{if } v_k \geq 0 \\ 0 & \text{if } v_k < 0 \end{cases} \tag{2.25}$$

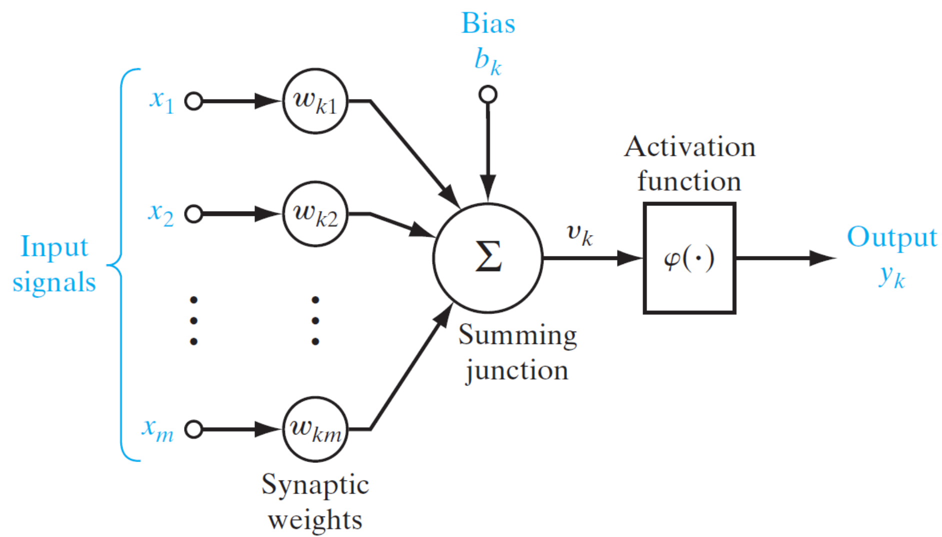


Figure 2.6: *Nonlinear model of a neuron.*

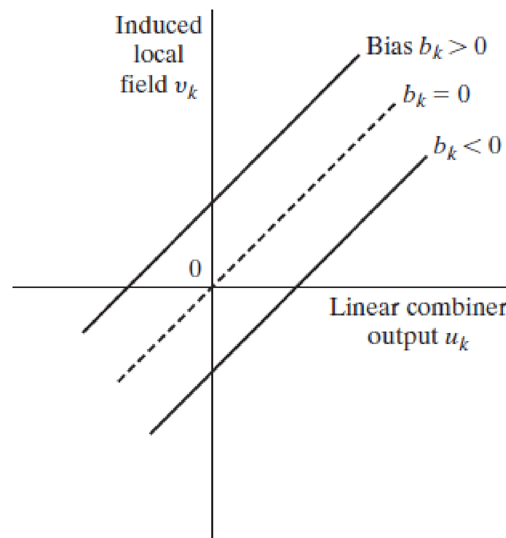


Figure 2.7: *Affine transformation produced by the presence of a bias.*

$$v_k = \sum_{j=1}^{m} w_{kj} x_j + b_k \tag{2.26}$$

- **Sigmoid Function.**(Figure 2.9) The sigmoid function, whose graph is "S"-shaped, is by far the most common form of activation function used in the construction of neural networks. It is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior. An example of the sigmoid function is the logistic function, defined by:

$$\frac{1}{1 + exp(-av)} \tag{2.27}$$

where a is the slope parameter. By varying this parameter we can obtain sigmoid

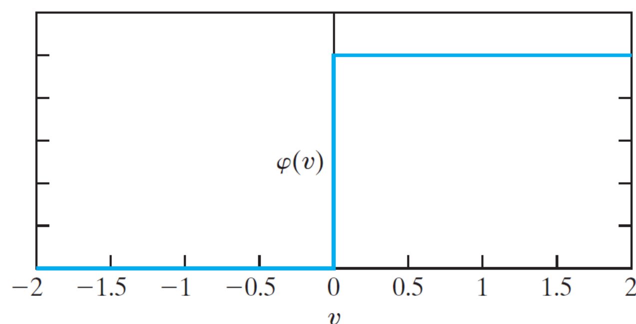


Figure 2.8: *A threshold activation function.*

functions of different slopes. as the slope parameter approaches infinity, the sigmoid function becomes simply a threshold function. In addition, it is sometimes desirable to have activation functions range from -1 to +1. This is commonly referred to as the signum function:

$$\phi(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v = 0 \\ -1 & \text{if } v < 0 \end{cases} \quad (2.28)$$

For the corresponding form of a sigmoid function, we may use the hyperbolic tangent function, defined by

$$\phi(v) = tanh(v) \quad (2.29)$$

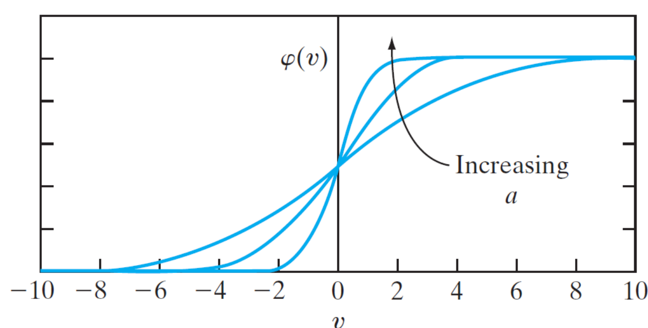


Figure 2.9: *A sigmoid activation function.*

2.2.3 Supervised Learning

In supervised learning, knowledge is represented by sets of input-output examples. We, as the supervisor or the teacher, can provide the neural network with a desired response for a given training vector. Then, the network parameters are adjusted under the combined influence of the training vector and an error signal, which is defined as the difference between the desired response and the actual response of the network. This adjustment is carried out iteratively in a step-by-step fashion with the aim of eventually making the neural network emulate the teacher. In this way, knowledge of the environment available to the teacher is transferred to the neural network through training and

stored in the form of "fixed" synaptic weights, representing long-term memory.When this condition is reached, we may then dispense with the teacher and let the neural network deal with the environment completely by itself. As a performance measure for the system, we may think in terms of the mean square error, or the sum of squared errors over the training sample, defined as a function of the free parameters (i.e., synaptic weights) of the system. This function may be visualized as a multidimensional error-performance surface, or simply error surface, with the free parameters as coordinates. For the system to improve performance over time and therefore learn from the teacher, the operating point has to move down successively toward a minimum point of the error surface; the minimum point may be a local minimum or a global minimum. A supervised learning system is able to do this with the useful information it has about the gradient of the error surface corresponding to the current behavior of the system. Reading this section, given the aforementioned description of SIVIA (Section 2.1.4), should give interesting ideas to the reader, as SIVIA is used to provide guaranteed approximations given a parameter space. Perhaps this algorithm could be useful during the training of Neural Networks. Unfortunately, this is not the topic explored in this thesis, but the interested reader can refer to [21].

2.2.4 Multilayer Perceptron

Before we move to explain the multilayer part of the title, we have to begin with the basics. The Perceptron was the first algorithmically described neural network, first proposed by Rosenblatt in 1958. It is basically a single-layer neural network and it is limited to the classification of linearly separable patterns. To overcome this limitation a neural network structure called a multilayer perceptron was proposed. The basic characteristics of this structure are:

- The model of each neuron in the network includes a nonlinear activation function that is *differentiable*.
- The network contains one or more layers that are *hidden* from both the input and output nodes.
- The network exhibits a high degree of *connectivity*, the extent of which is determined by synaptic weights of the network.

A popular method for the training of multilayer perceptrons is the **back-propagation** algorithm. The training proceeds in two phases:

1. The **forward phase**, in which the synaptic weights of the network are fixed and the input signal is propagated through the network, layer by layer, until it reaches the output. Thus, in this phase, changes are confined to the activation potentials and outputs of the neurons in the network.
2. The **backward phase**, where an error signal is produced by comparing the output of the network with a desired response.The resulting error signal is propagated through

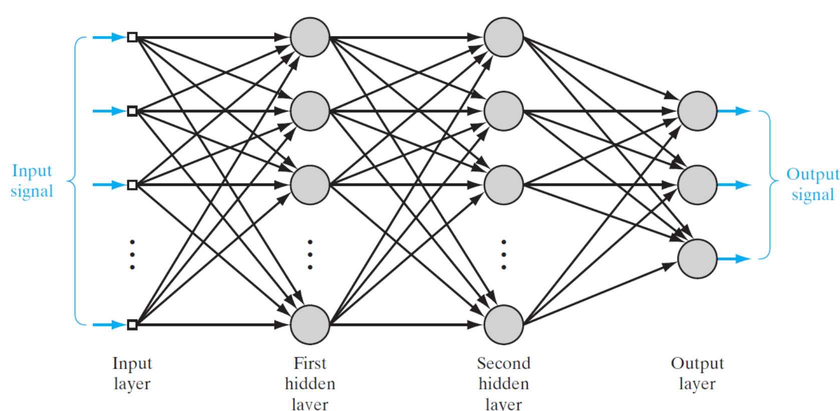


Figure 2.10: *A multilayer perceptron with multiple hidden layers and outputs.*

the network, again layer by layer, but this time the propagation is performed in the backward direction. In this second phase, successive adjustments are made to the synaptic weights of the network.

2.2.5 Training Techniques

The essence of back-propagation learning is to encode an input–output mapping (represented by a set of labeled examples) into the synaptic weights and thresholds of a multilayer perceptron. The hope is that the network becomes well trained so that it learns enough about the past to *generalize* to the future. More precisely, **generalization** refers to the network's ability to produce correct outputs, given inputs which were not provided during the training process. From such a perspective, the learning process amounts to a choice of network **parameterization** for a given set of data. We may view the network selection problem as choosing, within a set of candidate model structures (parameterizations), the "best" one according to a certain criterion. One way to do that is to split the data set into a training set and a validation set, the later serves the purpose of validating the generalization capabilities of the network. However, if the categories represented by the data are not split in a uniform manner, the model will adapt to this artificial bias. In addition, the data size may be very small to be able to afford this *fair* split. To counter this, we utilize a statistical technique, called **cross-validation**[24]. The data is partitioned into a training and a test set. Then, the training set is further partitioned into two disjoint subsets, an *estimation* subset, used to select the model and a *validation* subset to test or validate this model. Thus, it is possible to validate the models during training. To guard against overfitting, the model is finally verified by the initial test set. In some cases, this test set is merged into the validation splits of the training sample. This is generally not a good practice if the purpose is to maximize the generalization capability of the model. However, many problems are not accompanied by large data sets and each decision is ultimately made based on those material circumstances. There is another method for evaluating generalization proposed recently[4] which claims to enable the usage of the whole data set during training, without instilling

bias into the model. This thesis explores an accelerated revision of this proposition.

2.3 Parallel Computing

2.3.1 Overview

Until a bit over decade ago, people held the notion that the more they waited before purchasing new computational hardware, the more performance they would gain. This was true because between 1986 to 2003, the performance of microprocessors increased, on average, more than 50% per year. From 2003 and onwards this performance increase started to decline to the point that between 2015 to 2017, the increase was on average less than 4% per year[25]. This difference in performance increase has been associated with a dramatic change in processor design. By 2005, most of the major manufacturers of microprocessors had decided that the road to rapidly increasing performance lay in the direction of parallelism. Rather than trying to continue to develop ever-faster monolithic processors, manufacturers started putting multiple complete processors on a single integrated circuit. That lead software developers to re-examine their methods as their **serial** programs would not see any performance improvement throughout the years by itself.

2.3.2 Parallel Design Paradigms

The two most widely used approaches to parallelism[26] are:

- **Task Parallelism**. A task required to solve a problem is partitioned among the processor's cores.
- **Data Parallelism**. The data required to solve a problem is partitioned and thus each core carries the same operations but on a part of the data.

In reality, it is very common for a mixed parallelization strategy to be used. Depending on the problem, the partitioning of a task is not exclusionary to partitioning the data.

2.3.3 Instruction-level Parallelism

Instruction-level parallelism (or **ILP**) attempts to improve processor performance by having multiple processor components to simultaneously execute instructions. One can imagine this level of parallelism as a factory assembly line. Imagine the production of a mechanical keyboard; it requires a control board, a casing (or shell), switches, keycaps and a cable. Let's assume that each machine can only produce one type of component and that all of the required machines are located in the same factory. Since each component assembly is *independent* from the production done on other machines, to produce a keyboard we would assign a worker to a machine so that the production would be performed in parallel. The factory of our example is the processor and the machines are the individual components inside it, arithmetic & logic units, control units, registers etc. Figure 2.11 depicts this process.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 2.11: *Instruction Level Parallelism, I F/D/E stand for Instruction Fetch/Decode/Execute, Mem stands for Memory Access and WB for register Write-Back.*

2.3.4 Hardware Multithreading

ILP requires that the statements are independent between one another, and unfortunately that is not always the case. This is where Thread-level parallelism (or **TLP**) plays its hand. A thread is a mechanism provided to programmers, that offers the ability to divide a program into smaller independent tasks, with the property that when one thread is blocked, another can be run. In addition, the switch between threads happens faster than in processes, as threads are contained within the same process, therefore avoiding the performance hit of unnecessary system calls. **TLP** attempts to provide parallelism through the simultaneous execution of different threads, providing a **coarser-grained** parallelism than ILP. This means that the program units, that are simultaneously executed, are larger (or coarser) than the **finer-grained** individual instructions. Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled—for example, if the current task has to wait for data to be loaded from memory. Instead of looking for parallelism in the currently executing thread, it may make sense to simply run another thread.

2.3.5 Classifications of Parallel Computers

Two important independent classifications of parallel computers is **Flynn's taxonomy**[27] and the distinction between **shared memory** and **distributed memory** systems. Flynn's taxonomy works by classifying a parallel computer according to the number of instruction streams and the number of data streams it can simultaneously manage.

Flynn's Taxonomy

The categories in bold are the most commonly found parallel systems[26]:

1. SISD. A classical von Neumann system is an example of a Single Instruction, Single Data stream as it executed a single instruction at a time and, computes a single data value at a time.

2. **SIMD**. Single Instruction, Multiple Data systems are parallel systems. As the name

suggests, SIMD systems operate on multiple data streams by applying the same instruction to multiple data items. An example SIMD system is a GPU, however it is not a pure one.

3. MISD. Multiple Instructions operate on a Single Data stream. This is an uncommon architecture.

4. **MIMD**. Multiple Instruction, Multiple Data systems support multiple simultaneous instruction streams operating on multiple data streams. They usually consist of a collection of fully independent processing units (or cores), each of which has its own control unit and its own datapath. Unlike SIMD systems, MIMD systems are usually asynchronous, meaning that the processors can operate at their own pace. There is usually no global clock and there may be no relation between the system times of two different processors. At any given time two processors may be executing different statements, even if they were given the same sequence of instructions.

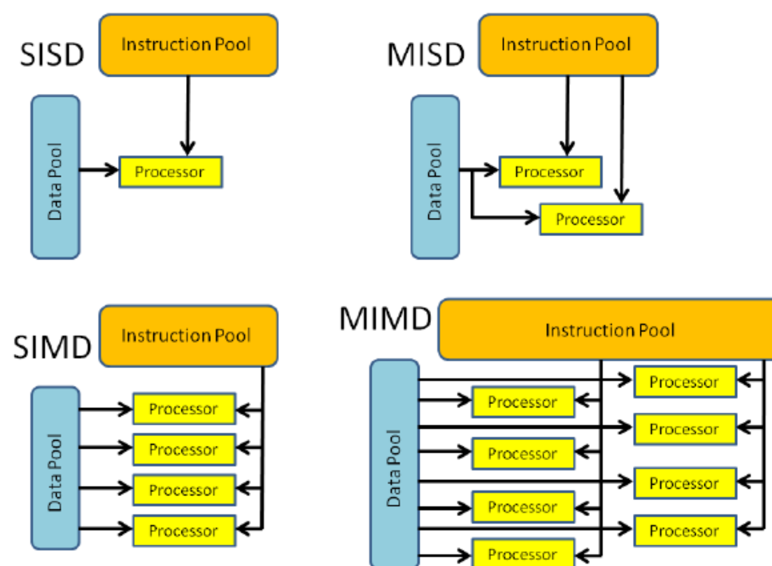


Figure 2.12: *Flynn's Taxonomy of Computer Architectures.*

Shared-Memory Systems

A shared-memory system is a collection of autonomous processors that is connected to a memory system via an interconnection network, and each processor can access each memory location. In a shared-memory system, the processors usually communicate implicitly by accessing shared data structures. The most widely available shared-memory systems use one or more multicore processors. In shared-memory systems with multiple multicore processors, the interconnect can either connect all the processors directly to main memory, or each processor can have a direct connection to a block of main memory, and the processors can access each other's blocks of main memory through special

hardware built into the processors. This constitutes another sub-categorization of these systems:

- **UMA**. In Uniform Memory Access shared-memory systems each processor has direct access to all available memory.
- **NUMA**. In Non-Uniform Memory Access systems each processor access its own memory. If the system enables the memory access of another processor (usually via a BUS), it is generally slower to do so.

UMA systems are usually easier to program, since the programmer doesn't need to worry about different access times for different memory locations. This advantage can be offset by the faster access to the directly connected memory in NUMA systems. Furthermore, NUMA systems have the potential to use larger amounts of memory than UMA systems.

Distributed-Memory Systems

In a distributed-memory system, each processor is paired with its own *private* memory, and the processor-memory pairs communicate over an interconnection network. In distributed-memory systems, the processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor.

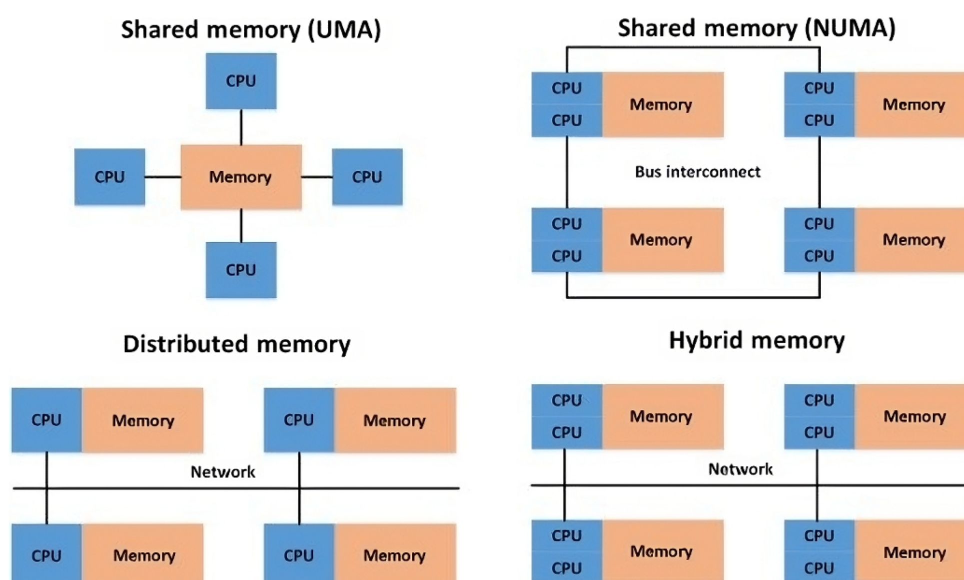


Figure 2.13: *A Collection of Parallel Systems.*

2.3.6 Parallel Computing on CUDA GPUs

History

During the late 90s, video games started becoming popular and players were demanding more realistic graphics. The computer industry responsed to that demand by developing extremely powerful graphics processing units, or as we know them, GPUs.

These processors, as their name suggests, are designed to improve the performance of programs that need to render many detailed images. The existence of this computational power was a temptation to programmers who didn't specialize in computer graphics, and by the early 2000s they were trying to apply the power of GPUs to solving general computational problems, problems such as searching and sorting, rather than graphics. This became known as General Purpose computing on GPUs or GPGPU. One of the biggest difficulties faced by the early developers of GPGPU was that the GPUs of the time could only be programmed using computer graphics APIs, such as Direct3D and OpenGL. So programmers needed to reformulate algorithms for general computational problems so that they used graphics concepts, such as vertices, triangles, and pixels. This added considerable complexity to the development of early GPGPU programs, and it wasn't long before several groups started work on developing languages and compilers that allowed programmers to implement general algorithms for GPUs in APIs that more closely resembled conventional, high-level languages for CPUs. Currently the most widely used APIs are CUDA and OpenCL. CUDA was developed for use on Nvidia GPUs. OpenCL, on the other hand, was designed to be highly portable.

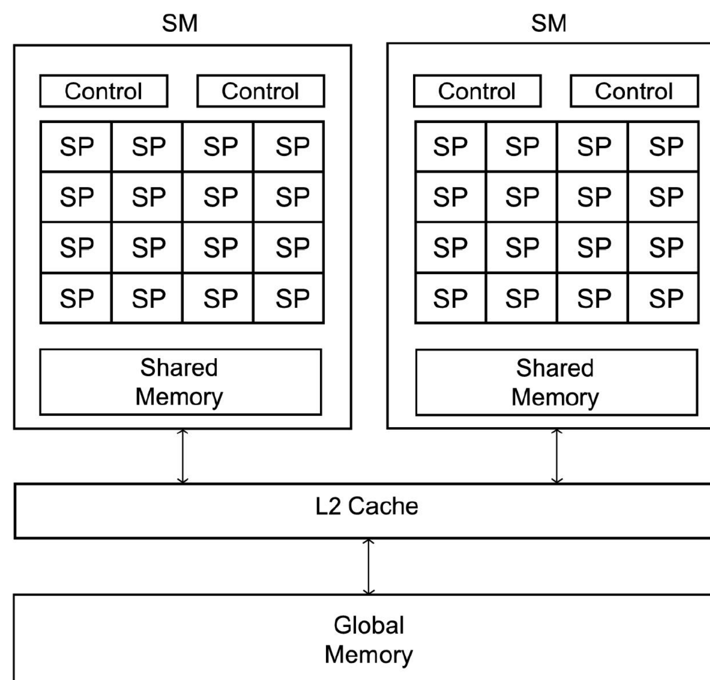


Figure 2.14: *A Simplified block diagram of a GPU.*

GPU Architectures

A typical GPU can be thought of as being composed of one or more SIMD processors. Nvidia GPUs are composed of Streaming Multiprocessors or SMs. One SM can have several control units and many more Streaming Processors or SPs. So an SM can be thought of as consisting of one or more SIMD processors. In addition, the SMs operate asynchronously. To put things into perspective, my current desktop GPU, namely the

RTX 3060 Ti has 38 SMs and each SM has 128 SPs[1] for a total of 4864 SPs. It is noteworthy to mention that Nvidia uses the term **SIMT** instead of SIMD. SIMT stands for Single Instruction Multiple Thread, and the term is used because threads on an SM that are executing the same instruction may not execute simultaneously. Some threads may block while memory is accessed and other threads, that have already accessed the data, may proceed with execution. This is done to hide memory access latency. Each SM has a relatively small block of memory that is shared among its SPs. This memory can be accessed very quickly by the SPs. All of the SMs on a single chip also have access to a much larger block of memory that is shared among all the SPs. Accessing this memory is relatively slow (Figure 2.14).

In Nvidia documentation, the CPU together with its associated memory is often called the **HOST**, and the GPU together with its memory is called the **DEVICE** (Figure 2.15). In earlier systems the physical separation of host and device memories required that data was usually explicitly transferred between CPU memory and GPU memory. That is, a function was called that would transfer a block of data from host memory to device memory or vice versa. However, in more recent Nvidia systems (Compute Capability ≥ 3.0), the explicit transfers in the source code aren't needed for correctness, although they may be able to improve overall performance.

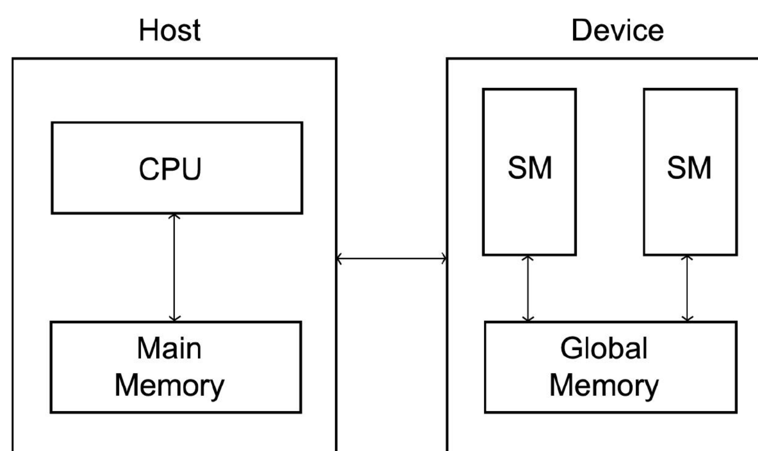


Figure 2.15: *Simplified block diagram of a HOST and a DEVICE.*

Heterogeneous Computing

Writing a program that runs on a GPU is an example of heterogeneous computing. The reason is that the programs make use of both a host processor, meaning a conventional CPU, and a device processor, namely a GPU. The two processors have different architectures. This means that the program will have functions for intended for conventional CPUs as well as functions explicitly defined to run on GPUs. Heterogeneous computing has become much more important in recent years[26]. Programmers are leaving no stone unturned in their search for ways to bolster performance, and one possibility is to make

[1]The Ampere architecture of the RTX 3060 Ti mentions SPs as Multiprocessors or CUDA Cores.

use of other types of processors, processors other than CPUs. Other possibilities for heterogeneous computing include Field Programmable Gate Arrays or **FPGAs**, and Digital Signal Processors or **DSPs**. FPGAs contain programmable logic blocks and interconnects that can be configured prior to program execution. DSPs contain special circuitry for manipulating (e.g., compressing, filtering) signals, especially *real-world* analog signals.

CUDA Grids, Blocks and Threads

An Nvidia GPU consists of a collection of streaming multiprocessors (SMs), and each streaming multiprocessor consists of a collection of streaming processors (SPs). When a CUDA kernel runs, each individual **thread** will execute its code on an SP. CUDA organizes threads into blocks and grids. A **thread block** (or simply block) is a collection of threads that run on a single SM. In a **kernel call** the first value in the angle brackets specifies the number of thread blocks. The second value is the number of threads in each thread block (Figure 2.16). When the kernel is started, each block is assigned to an SM, and the threads in the block are then run on that SM. A **grid** is the collection of thread blocks started by a kernel. So a thread block is composed of threads, and a grid is composed of thread blocks. There are several built-in variables that a thread can use to get information on

```
partialBisect<<<blocks, threads>>>(d_boxes, i, eps, dims);
CHECKED_CALL(cudaGetLastError());
CHECKED_CALL(cudaDeviceSynchronize());
```

Figure 2.16: *An example of a kernel call. partialBisect happens to be the name of the device function to be executed.*

the grid started by the kernel. The following four variables are structs that are initialized in each thread's memory when a kernel begins execution:

- **threadIdx**: The rank or index of the thread in its thread block
- **blockDim**: The dimensions, shape, or size of the thread blocks.
- **blockIdx**: The rank or index of the block within the grid.
- **gridDim**: The dimensions, shape, or size of the grid.

These structs have three fields x, y, z and are declared as unsigned integers. The fields are often convenient for applications. For example, an application that uses graphics may find it convenient to assign a thread to a point in a 2 or 3-dimensional space, and the fields in *threadIdx* can be used to indicate the point's position. An application that makes extensive use of matrices may find it convenient to assign a thread to an element of a matrix, and the fields in *threadIdx* can be used to indicate the column and row of the element. Finally, all the blocks must have the same dimensions. More importantly, CUDA requires that thread blocks be independent. So one thread block must be able to complete its execution, regardless of the states of the other thread blocks: the thread blocks can be executed sequentially in any order, or they can be executed in parallel. This

ensures that the GPU can schedule a block to execute solely on the basis of the state of that block: it doesn't need to check on the state of any other block.

2.3.7 CUDA Warps

In CUDA a warp is a set of threads with consecutive ranks belonging to a thread block. The number of threads in a warp is currently 32, although Nvidia has stated that this could change[26]. The threads in a warp operate in a SIMD fashion. Threads in different warps can execute different statements with no penalty, while threads within the same warp must execute the same statement. When the threads within a warp attempt to execute different statements, the threads are said to have **diverged**. When divergent threads finish executing different statements, and start executing the same statement, they are said to have **converged**.

2.3.8 CUDA Memory Architecture

We can think of the GPU memory as a hierarchy with three *levels*. At the bottom, is the slowest but larger, **global memory**. In the middle is the **shared memory**, which is of smaller size but faster than the global memory. At the top are the fastest, of even smaller size, the registers. All threads of a SM have access to the shared and global memory, but threads individually only have access to their respective registers. In Compute Capability ≥ 3 there are functions called **warp shuffles** which allow threads of a block to access variables stored by other threads in the warp. It takes on the order of 1 cycle to copy a 4-byte int from one register to another. Depending on the system it can take up to an order of magnitude more time to copy from one shared memory location to another, and it can take from two to three orders of magnitude more time to copy from one global memory location to another. Depending on the GPU architecture, there are usually three levels of cache to minimize the latency penalties induced by reading and writing to the slowest memories.

2.3.9 Memory Coalescing

CUDA Best Practices

The performance guidelines and best practices described in [28] and [29] apply to all CUDA-capable GPU architectures. Programmers must primarily focus on following those recommendations to achieve the best performance[30]. The high-priority recommendations from those guides are as follows:

- Find ways to parallelize sequential code.
- Minimize data transfers between the host and the device.
- Adjust kernel launch configuration to maximize device utilization.
- Ensure global memory accesses are coalesced.
- Minimize redundant accesses to global memory whenever possible.

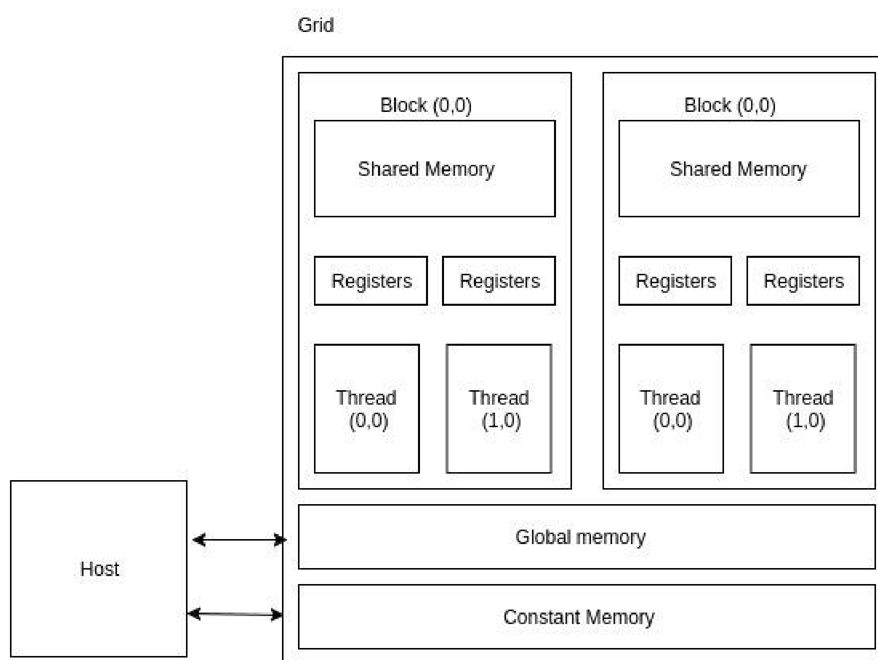


Figure 2.17: *A simplified schematic of Grid, Block, Thread and Memory hierarchies of a DEVICE and the interconnection with the HOST.*

- Avoid long sequences of diverged execution by threads within the same warp.

2.4 Discussion

This chapter was about Interval Arithmetic, Neural Networks, Parallelization methods and hardware. Interval Analysis is a field of mathematics which was initially designed to help with engineering problems that require calculations with the inclusion of an error, as the physical world is not always forgiving. In addition, these techniques are shown to be very similar to operations on scalars and the benefits of a technique named Set Inversion via Interval Analysis was showcased. SIVIA is able to provide, in a deterministic notion, a guaranteed approximation for non-linear functions, namely, inclusion functions. IA methods are also very reliant on set operations. Afterwards, an overview of neural networks was provided, with the focus being on multilayer perceptrons for their utility in classification problems, and, finally, an overview in parallel computing was provided. This included types of parallel hardware, parallelization strategies with the spotlight aimed at CUDA GPGPU software design and the best practices for that purpose. The next chapter is directly related to the implementation of this thesis, background work on other attempts are explored and will be followed with the design choices of the proposed parallelization attempt.

Chapter 3

Implementation

3.1 Branch & Bound Algorithms

Branch-and-bound (BB) methods are well-known algorithmic tools for solving NP-hard optimization problems. For many of these inherently difficult problems, only small instances can be solved in a reasonable amount of time on sequential computers[31]. Consequently, the use of parallelism to speed up the execution of BB algorithms has emerged as a way to solve larger problem instances. BB methods are more commonly deployed in optimization problems where a problem is not tractable and a divide-and-conquer approach may be required for its solution. As the name suggests, this method consists of a branching and a bounding operation. **Branch** refers to the decomposition of a problem while **Bound** refers to operations aimed at eliminating resulting subproblems. Therefore we can describe these methods as the process of building a tree. The root of this tree is the original problem while the leaves are the subproblems obtained through the decomposition of the root. Now let's replace problems and subproblems with pavings and subpavings, branching with bisection and Bounding with the set evaluation of subpavings. SIVIA of section 2.1.4 now seems like a BB technique and it is in fact the case. However, there is an important distinction to be made; most applications of SIVIA have been applied to optimization problems and not on neural networks as you will see further in this chapter. Contrary to those problems, where a tree is branched out based on an upper bound and a lower bound test, in our case the tree is branched whenever a subproblem cannot provide a definitive answer and results in what is called **exhaustive search** as the criteria to prevent the branching of a subproblem (besides a minimum threshold ϵ) are missing. The similarity of SIVIA to BB algorithms is important because parallel work done on the latter can be seen as a template for the parallelization of the former. We can now proceed with the classification of parallel BB algorithms[31]:

- **Type 1**. Introduces parallelism when performing the operations on generated subproblems. It consists, for example, of executing the bounding operation in parallel for each subproblem to accelerate the execution.
- **Type 2**. Consists of building the BB tree in parallel by performing operations on several subproblems simultaneously.
- **Type 3**. Several BB Trees are built in parallel.

Furthermore, Type 2 parallel algorithms are further classified on the existence of synchronization between processes and the number of **work pools**. A work pool is the data structure that contains the subproblems that are waiting to be examined, its properties rely on the implementation. The classification goes:

- **Synchronous Single Pool**.
- **Asynchronous Single Pool**.
- **Synchronous Multiple Pool**.
- **Asynchronous Multiple Pool**.

Gendron concluded that the type 2 parallelization scheme 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 or have multiple pools. 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". For AMP strategies it is concluded that a *dynamic load balancing* must be deployed in order to achieve high efficiency.

3.2 GPU Parallelization of the Parameter Estimation problem

[20] contains a very good summary of the parallelization schemes in a CUDA GPGPU context. 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 exclusively for either MIMD or SIMD is not a perfect fit for a CUDA GPU. Given the fact, these are some of the main options when designing parallel software:

- **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 to 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. A fine-grained scheme.
- **Inner iteration level (Type 1)**: This generally means parallelizing the inclusion function mentioned in section 2.1.3. These functions generally involve vectors,

therefore a thread can be allocated to each element. This is also a fine-grained scheme.

- **Outer iteration level (Type 2/3)**: Each interval box in the work queue can be processed independently in parallel. CUDA streams, each with 1 or more controlling threads 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. This scheme is coarse-grained.

It is generally considered that 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. This also has the benefit to minimize the search tree, or the input space area, which makes the approach very memory efficient. Most research on the Parallel BB algorithms has been on MIMD systems, in single or multiple CPU settings and for a good reason as, most of the time, the inner iterations are not so expensive to include a GPU architecture. Even most SIMD research was not performed in a GPGPU context. In the recent years however, GPUs have become more massively parallel than ever[1] thanks to the AI boom[2] and to experimenting with breadth-first approaches sounds plausible.

3.2.1 Previous Work

Most parallelization research of BB methods on SIMD architectures has been done in combinatorial or global optimization problems[32]. On the other hand, research in a GPGPU context is sparse[20]. Work done on SIMD architectures is usually of Type-1 with fine-grained interval arithmetic operations[33][34][35][36]. These approaches are generally implemented using a mixed MIMD approach where the the BB tree is managed on the CPU-level(usually MIMD implementations) and only the processing functions (Type-1) (bounding, interval-newton etc.) are parallelized[20][37]. Of the exclusively MIMD approaches, very notable is the work of Casado at al.[38], where it is showcased that ASP and AMP approaches are almost equally fast. In addition, all of the aforementioned approaches are not relevant to the task at hand and this matters because the existence of bounds can is generally used to direct the prioritization of the box processing. That is not the case in Section 3.4.2. Work with breadth-first approaches is even more sparse, which is understandable, as the general notion is to minimize the search tree in order to avoid redundant operations and save on memory utilization. Lastly, in my undergraduate years, as part of my ERASMUS+ internship and undergraduate thesis, I tried to solve this problem by implementing Casado's AMP algorithm using the C-XSC C++ library[39] but due to time constraints I was not able introduce a neural network in the design, thus

[1] https://www.forensicfocus.com/news/the-new-nvidia-rtx-3080-has-double-the-number-of-cuda-cores-but-is-there-a-2x-performance-gain/

[2] https://www.reuters.com/technology/nvidia-shares-rise-ai-boom-lifts-hopes-another-strong-revenue-forecast-2023-08-22/

the problems solved were of trivial nature. My work, however, validated the algorithm's scaling capabilities[9][40].

3.3 Parallelization Proposal

The base code for this implementation was provided by NVIDIA via the open-source Cuda Samples Github repository[41]. This repository provides a CUDA GPGPU IA implementation based on the design of the C++ Boost Library. This library provides a few basic operations $(+, -, *, /)$ which are designed for float, double and integer variables as well as the calculation of the width of an interval and some other functions beyond the scope of this thesis. My implementation provides additional features such as set operators (subset,disjoint, intersection) trigonometric operations and extending all the pre-included operations to 16bit half-variables. The implementation on the design level consists of three parts, two of which are required for problems 1&2 and all three of them are required for problem 3, which is also the main focus of the thesis. The proposed implementation inverts the sequence of operations. Instead of bisecting a box when it cannot provide a solution to a problem and its width is greater than *epsilon*, the bisection occurs indiscriminately at the beginning of the algorithm, expanding the tree in regards to the *epsilon* threshold value. The total amount of boxes that will be generated is given by

$$Number\ of\ Boxes = \prod_{i=1}^{N} 2^{ceil\left(log_2\left(\frac{width([X_0]_i)}{\epsilon}\right)\right)}, \tag{3.1}$$

with N as the number of dimensions $[X_0]$. Depending on the input space, meaning the size of the width of each dimension as well as the number of dimensions, expanding the tree aggressively might require large amounts of available memory. However, this also results in a very predictable problem structure that can be easily parallelized in a grid-stride manner, as work is evenly distributed between threads and coalesced global memory accesses are guaranteed. The slight exception is the Bisection operation itself as each level of the binary BB tree is characterized by the bisection performed on the previous one, meaning that some sort of synchronization is required. The proposed implementation overall, can be categorized as an SSP algorithm in the sense that the GPU is used as a slave with a single work pool, while the multi-GPU variation can be described as an AMP as multiple CPU threads *own* a GPU, from the thread's perspective the algorithm is ASP. Thus, the multi-GPU implementation is AMP/SSP.

3.3.1 Parallel Bisection

Most (if not all) SIVIA implementations bisect a box at the final step of a loop, after the inclusion function and the set operations that determine whether the box is part of the solution. It is assumed in this implementation that only 1 split occurs in a given box. The boxes on a given BB tree level are considered independent problems (Figure 3.1). The main idea for the parallelization of the bisection operation is inspired from the amount of GPU threads available in the recent years. Using the CPU as a master (or primary) node

and the GPU as a slave (or *secondary* in that manner) we can send commands to the GPU to bisect all boxes of a given tree level. This means that the first commands do not utilize all available resources as the number of boxes is very small, however theoretically there is a certain level where the GPU is faster than the CPU as the number of boxes for bisection is large enough. Notice this very idea on Figure 3.2 where that certain number of boxes exists and acts as a threshold in which a GPU starts to perform better than the CPU. There was also the idea to pre-bisect the boxes on the CPU before the GPU takes over but that would include memory transfer latency and there was not any significant benefit in implementing the bisection that way. The number of Bisection commands the HOST will send to the DEVICE is trivially given by

$$Bisection\ Commands = log_2(Number\ of\ Boxes) \tag{3.2}$$

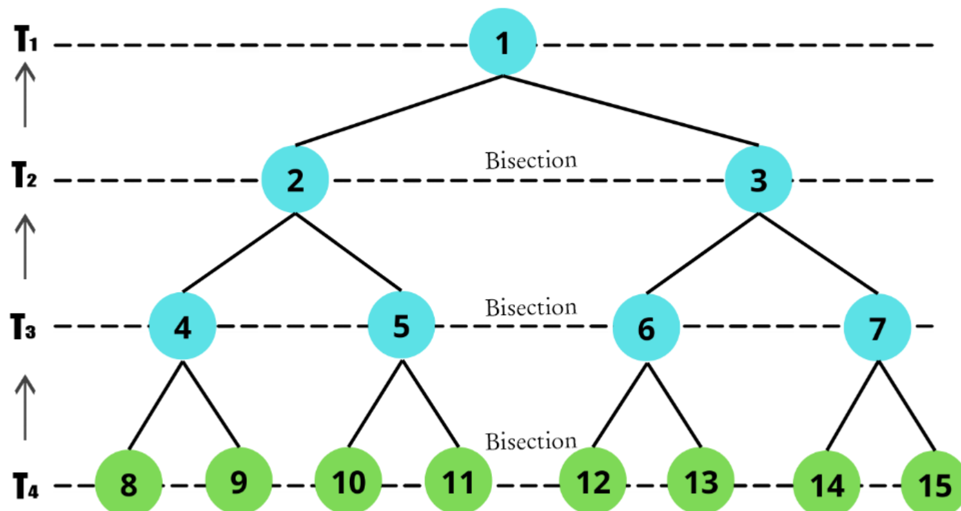


Figure 3.1: *A Binary Tree split by 4 time periods. Each node is a subproblem or a box. Each time period represents a Bisection operation (or the result of one). T_4 is dependent on T_3, T_3 is dependent on T_2 etc. Boxes of the same level are independent problems which can be solved very easily in parallel.*

3.3.2 Parallel Evaluation

This is as straightforward as it sounds. In a coarse-grained algorithmic design, a vector of boxes is generated by the previous -bisection- phase and each box is processed in parallel. The evaluation here includes two phases:

1. The interval inclusion function

2. The subset,intersection and epsilon check

The first phase can be simply described as sending a box to a function and return another box or a simple interval. This function as will be mentioned further here can be in the form

```
> GPU Device has Compute Capabilities SM 8.6

Bisection Benchmark
Initial Box ([-1.500000,1.500000],[-1.500000,1.500000])
Epsilon: 0.001 Dimensions: 2
Problem size: 16777216 boxes
Boxes: 1 | CPU Duration: 0us GPU Duration: 230us
Boxes: 2 | CPU Duration: 0us GPU Duration: 40us
Boxes: 4 | CPU Duration: 0us GPU Duration: 43us
Boxes: 8 | CPU Duration: 0us GPU Duration: 49us
Boxes: 16 | CPU Duration: 0us GPU Duration: 44us
Boxes: 32 | CPU Duration: 1us GPU Duration: 43us
Boxes: 64 | CPU Duration: 2us GPU Duration: 54us
Boxes: 128 | CPU Duration: 5us GPU Duration: 47us
Boxes: 256 | CPU Duration: 9us GPU Duration: 75us
Boxes: 512 | CPU Duration: 19us GPU Duration: 48us
Boxes: 1024 | CPU Duration: 39us GPU Duration: 43us
Boxes: 2048 | CPU Duration: 77us GPU Duration: 40us
Boxes: 4096 | CPU Duration: 216us GPU Duration: 40us
Boxes: 8192 | CPU Duration: 451us GPU Duration: 127us
Boxes: 16384 | CPU Duration: 916us GPU Duration: 43us
Boxes: 32768 | CPU Duration: 1693us GPU Duration: 43us
Boxes: 65536 | CPU Duration: 3398us GPU Duration: 46us
Boxes: 131072 | CPU Duration: 6843us GPU Duration: 56us
Boxes: 262144 | CPU Duration: 13761us GPU Duration: 201us
Boxes: 524288 | CPU Duration: 28239us GPU Duration: 236us
Boxes: 1048576 | CPU Duration: 55536us GPU Duration: 307us
Boxes: 2097152 | CPU Duration: 111522us GPU Duration: 433us
Boxes: 4194304 | CPU Duration: 225368us GPU Duration: 688us
```

Figure 3.2: *A benchmark comparing the Bisection operation processing time between a sequential implementation on a core i7 5820k and a parallel one with an RTX 3060Ti. At 1024/2048 boxes (red square area) the GPU is faster. The GPU used is from my home setup; jitter is produced as the DEVICE is also being utilized by the OS and can be noticed in 1 and 8192 boxes.*

of a neural network. In the latter case, it is simply a forward pass. The second phase is usually split into a three-if clause that determines whether the resulting interval(or box) is part of the solution. If the second phase is left the way it is usually implemented it would create warp divergence in the GPU and that would be a performance hit. The solution proposed for efficient parallelization calculates the inclusion with boolean operations, thus every GPU thread in a warp perform the same operation at any given time and no warp rescheduling is needed. Figures 3.3 and 3.4 illustrate a code comparison for the box set evaluation between a sequential algorithm with the Ibex C++ Interval Library[42] and a parallel non divergent one. *fxy* represents an interval object which was returned by an inclusion function.

3.3.3 Parallel Reduction

The main ideas for the parallel reduction are derived from Mark Harris's *Optimizing Parallel Reduction in CUDA* presentation[43]. This step is required only by the 3rd problem described in Section 3.4.2. The implementation was provided by the CUDA Samples repository[41], more precisely, of the eight variations provided, the third one was picked

```
if (fxy.is_subset(Ybox[0])) {
}
//Rejection
else if (!fxy.intersects(Ybox[0])) {
}
//Width threshold satisfied
else if (box.max_diam()<=eps){
}

//Otherwise we bisect.
int i=box.extr_diam_index(false);
pair<IntervalVector,IntervalVector> p=box.bisect(i);
```

Figure 3.3: *A sequential code snippet using the Ibex C++ library. Some parts have been removed for simplicity.*

```
//labels= 0:in, 1:epsilon, 2:out
labels[i]= (!fxy.is_subset(yb) + !fxy.intersects(yb));
```

Figure 3.4: *A code snippet from the CUDA C++ Parallel proposed implementation. fxy has a different value for every thread.*

for its efficiency and simplicity of the code structure. The other variations did not provide any significant improvement given the vector size needed for the problem. To explain more, the 3rd problem requires the calculation of a sum and each CUDA thread calculates and stores its own partial sum. For the parallel reduction technique to function properly, the vector that holds the summations has to be of size n that is a power of 2. Since a CUDA GPU does not (yet) contain threads by a factor of millions, the vector is not large enough to benefit from more complex techniques. A simple reduction algorithm is good enough. Figure 3.5 depicts the parallel algorithm deployed given a vector with size $n = 16$.

3.3.4 Single and Multiple GPU(s)

One important detail of the proposed implementation is that it is designed to work with both single and multiple GPU setups. This is possible because of the strategy deployed for sending work (boxes) towards the GPU(s). Two limitations are involved when sending work; the first involves the memory transfer times to and from the selected GPU DEVICE. To minimize transfer times one only needs to send a single box to the DEVICE at the beginning of the algorithm. Problems 1&2 of Section 3.4.1 require all generated boxes to be transferred back to the CPU. Unfortunately this part has not been implemented more efficiently and the performance hit is visible on Section 3.6.3; On the other hand, Problem 3 of Section 3.4.2 does not require that transfer operation. The other limitation is the memory capacity of the GPU. This has been mitigated by pre-bisecting the initial box on the HOST until the problem can fit in the global memory of the DEVICE. The

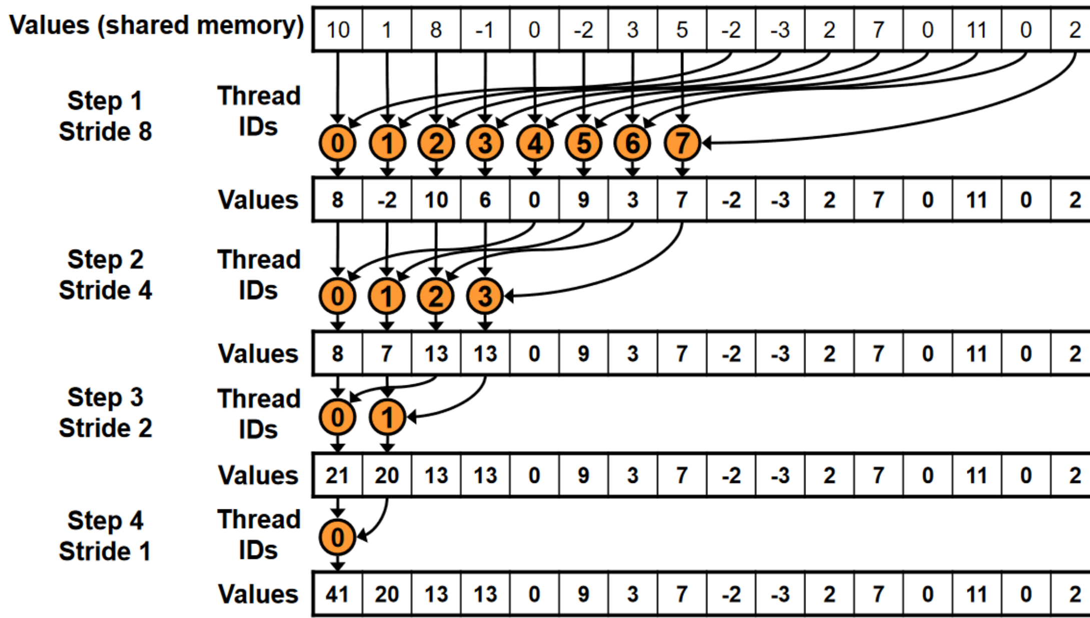


Figure 3.5: *Conflict-free sequential addressing parallel reduction which guarantees coalesced memory accesses.*

time needed for this pre-bisection is trivial since only a small number of boxes need to be generated. This results in multiple GPU executions. The same idea can be applied to multiple GPU devices. There are two ways to handle a multi-GPU implementation. A single or multiple CPU threads can be used to handle the communication between the HOST and the DEVICES; the latter is used in this thesis. The initial box is pre-bisected until the number of boxes is equal or greater to the number of threads (usually no more than 8). Then they are evenly distributed between threads and the algorithm functions in a MIMD fashion. If the number of available GPU devices is a multiple of 2 then work can be easily evenly distributed, otherwise boxes are distributed in a round-robin manner resulting in some threads to process more work. Using Equation 3.3, we can calculate the number of pre-bisections -and executions on the GPU- required so that the problem fits a GPU's global memory.

$$Number\ of\ PreBisections = ceil\left(log_2\left(\frac{Number\ of\ Boxes}{GPU\ Capacity} + 1\right)\right) \tag{3.3}$$

It should be noted that the *GPU Capacity* in this context is the amount of available global memory **in boxes**. To calculate it we need amount of available global memory in bytes which is returned by the CUDA context. The amount of memory required by a box **in bytes** is calculated by multiplying the amount of memory each variable requires times 2 (as an interval is defined by two variables) times the number of dimensions N of the interval vector. Equation 3.4 demonstrates this notion using 4-Byte floating point variables.

$$GPU\ Capacity(Boxes) = \frac{GPU\ Capacity(Bytes)}{2 * 4(bytes) * N} \tag{3.4}$$

3.4 Problems

3.4.1 Problems 1 & 2: 2D Torus and Griewank functions

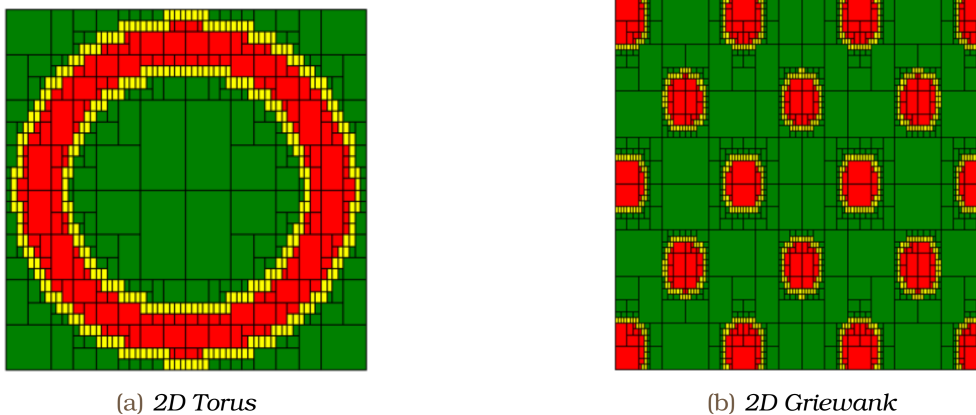
The first two problems are very trivial (they require a very few execution cycles per box) and were used mainly for benchmarking purposes during the development of this algorithm. For them to be of essence however, both problems will be considered *solved* when all generated boxes are labeled and transferred back to the HOST. That way it is verifiable that the program returned valid results. Notice that in Figures 3.7 & 3.8 the boxes are of equal size compared to Figure 3.6. The 2-dimensional Torus interval inclusion function is described by

$$[f]([x]) = [x]^2 + [y]^2 \tag{3.5}$$

and the initial box X_0 is bound between $[x_0] = [-1.5, 1.5], [-1.5, 1.5]$. The solution set Y was set $[y] = [1, 2]$ and different ϵ values were tested. Continuing on, the 2-dimensional Griewank interval inclusion function is defined as

$$[f]([x]) = \sum_{i=1}^{2} \frac{[x]_i^2}{4000} - \prod_{i=1}^{2} cos\left(\frac{[x]_i}{\sqrt{i}}\right) + 1 \tag{3.6}$$

and the bounded input space X_0 was set in $[x_0] = [-10, 10], [-10, 10]$. Y was set between $[y] = [1.5, 3]$ and, as with the Torus function, different ϵ thresholds were tested. The figures of this section were created with the VIBES GUI toolkit[44].

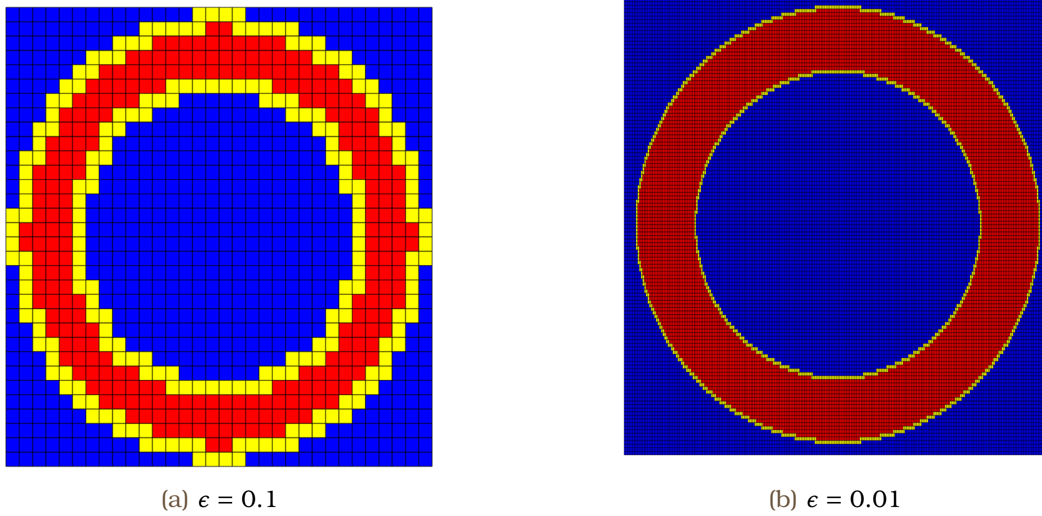


(a) *2D Torus* (b) *2D Griewank*

Figure 3.6: *The Input space of the 2D Torus function produced by the sequential SIVIA algorithm. The boxes have different sizes because of a dynamic tree expansion strategy.*

3.4.2 Problem 3: Estimating the Generalization Performance of a Neural Classifier

As we saw in Section 2.2.5, the generalization of a network architecture is measured by estimating the error of classification on previously unseen data. A very common technique used for its estimation is cross-validation. Adam et al.[22][45][4] claim that



(a) $\epsilon = 0.1$ (b) $\epsilon = 0.01$

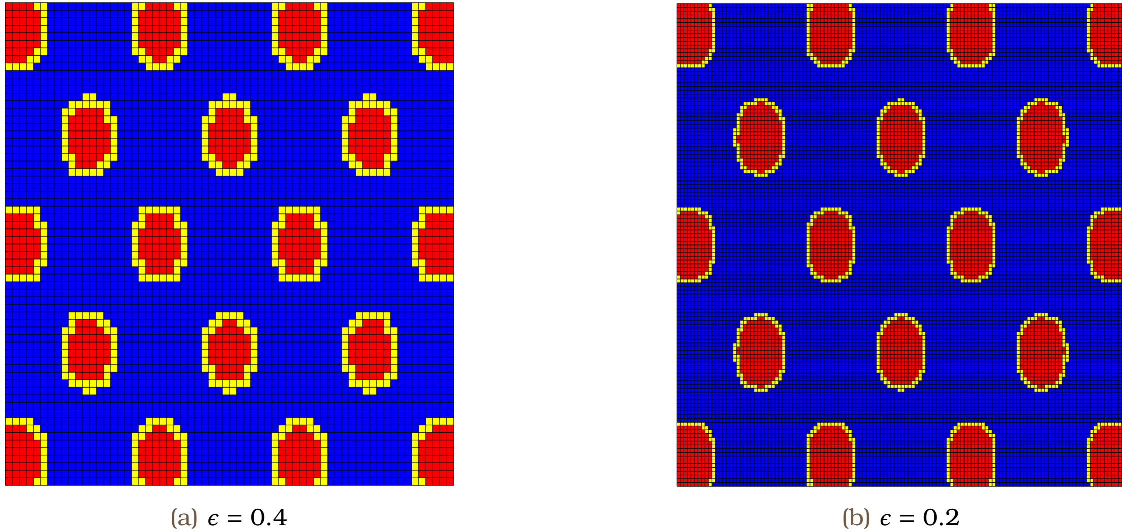
Figure 3.7: *The Input space of the 2D Torus function produced by the proposed parallel algorithm. Smaller epsilon values provide more accurate approximations.*

despite the fact that attempts have been made to prove that cross-validation results in a consistent estimator of the learning algorithm's generalization, it seems that this approach does not sufficiently define generalization. Several causes for this problem are related to the difference of the distribution generating test patterns during cross-validation from the distribution of the off-training set defined by real-world processes. Another reason for questioning the efficiency and unbiasedness of cross-validation is the stochastic splitting into folds, while, to a lesser extent, one may consider that cross-validation is computationally intensive since it requires the training process to be repeated several times. They instead propose an alternative solution using IA methods, mainly SIVIA. The proposal was in the form of a sequential algorithm (as SIVIA is traditionally implemented) and this means that it either required large amounts of execution times or lower-quality approximations (high ϵ values) had to be generated. [4] introduces three metrics, namely G_{net}, E_{net} and M_{net}. The latter of the three is a combination of the first two. In this thesis only G_{net} is used as the purpose is the demonstration of a parallel alternative for its calculation. The thought behind G_{net} is the assumption that the larger the domain of validity of a classifier, the bigger its volume and so the higher the probability for some unknown pattern to be in this area and be classified. Hence, the necessary condition for some unknown pattern to be classified by the network is to lie within the domain of validity of its respective class. It is computed with the following equations:

$$G_{net} = \frac{V_{net}}{V_{input}} - \frac{l}{P}, \tag{3.7}$$

where l is the number of misclassified or unclassified patterns and P is the total number of patterns. The volume of the solution set for all classes C is given by

$$V_{net} = \sum_{i=1}^{M} V_i, \tag{3.8}$$



(a) $\epsilon = 0.4$ (b) $\epsilon = 0.2$

Figure 3.8: *The Input space of the 2D Griewank function produced by the proposed parallel algorithm. Smaller epsilon values provide more accurate approximations.*

with V_i as the partial volume of the set of boxes that classified into class C_i (part of the solution set), given an output interval $Y_i = [1 - \beta, 1]$. β is an arbitrary scalar usually set between $[0.1, 0.2]$ (should not be confused with intervals). V_{input} is the volume of the X_0 hyperbox. Given a classification problem it can be described with the size of each feature of the training data.

$$V_{input} = \prod_{i}^{N} |x_i^{max} - x_i^{min}| \tag{3.9}$$

Given all the above, to test the performance and scalability of the parallel proposal one only needs to calculate a partial sum V_i as it is the only variable in the equation directly dependent on the generated boxes. V_{input} only needs to be computed once for a given NN; the same goes for $\frac{l}{P}$. Finally, the Neural Network used as the inclusion is a pre-trained **6-30-2** MLP, using the Levenberg-Marquardt algorithm with early stopping, in the Vertebral Column[3] dataset[46] and was provided by Professor Adam himself. The β value used is 0.2, resulting in a window $Y = [0.8, 1]$. More details of the model can be found in Section 5.1.5 of [4].

3.5 Test Environment

For the performance tests, 4 different systems have been deployed, provided by different entities. Reading Table 3.1 from top to bottom; The first system is my home desktop setup, the second, namely the DaVinci system, was provided by Microlab of ECE-NTUA; the 3rd is owned by the Laboratory for Computing of the CEID department at the University of Patras. The final setup is a multi-configuration system part of the Google

[3]The biomedical data of 310 patients are used for two possible classification tasks. The second task was retained, where the categories Disk Hernia and Spondylolisthesis of the first task are merged into a single category labeled as *abnormal*. So, this task consists in classifying patients as belonging to class *Normal* (100 patients) or *Abnormal* (210 patients).

Collaboratory's subscription service.

CPU	Memory	GPU
Intel Core i7 5820K @ 4.3Ghz (OC)	30GiB DDR4	NVIDIA RTX 3060 Ti PCIe 8GB
2x Intel Xeon Gold 5218R @ 2.10Ghz	128GiB DDR4	NVIDIA Tesla V100 PCIe 32GB
2x AMD EPYC 7742 @ 2.25Ghz	1026GiB DDR4	8x NVIDIA A100 PCIe 32GB
Unknown (depends on the session)	≥ GPU Memory	NVIDIA Tesla V100 PCIe 16GB
		NVIDIA A100 PCIe 32GB

Table 3.1: *A summary of the configurations used in the benchmarks.*

3.6 Results

The comparison is made between intervals of FP32 (float) and CUDA FP16 (half) variables. The Epsilon values for problem 1 and 2 range between $1e-2$ and $1e-4$, while for problem 3 it ranges between $[0.1, 0.03]$. The smallest possible number a half variable can store is 5.96×10^8 and is way smaller than the ϵ values used in the tests. Thus, even if some accuracy is missing from some operations, the approximations will still become more accurate as ϵ decreases. In addition, the primary purpose of problem 3 was to use the method to compare different neural networks, so all of them would get *rated* with the same accuracy. The tests measure the speedup.

$$Speedup = T_s/T_p \ , \tag{3.10}$$

where T_s is the real execution time, in seconds, of the sequential algorithm and T_p of the parallel one; the throughput, which in our case will be given by

$$Throughput = \frac{Boxes\ Processed}{Execution\ Time}, \tag{3.11}$$

and finally various causes will be explored using the profiler of the NVIDIA Nsight Compute suite[4] in order to explain the behavior of the proposed parallel algorithm in all three problems. While the general good practice is to perform multiple executions and average the measured time, intensive tests were run instead so that delays attributed to system calls and synchronization take a trivial percentage of the execution time, far from reducing the integrity of the measurements. The plots were created using the Seaborn[5] Python module.

3.6.1 CPU Implementation

The sequential implementation used on the CPU has been written with the Ibex Numerical and Interval Arithmetic C++ library[42]. This implementation uses double precision variables for the intervals and is written to perform well in a sequential context. The BB tree is expanding dynamically, depth-first, as only boxes with *width* $> \epsilon$ which

[4] https://developer.nvidia.com/nsight-compute
[5] https://seaborn.pydata.org/

are not part of the inner/outer approximation are bisected. So the problem sizes between implementations will be vastly different.

3.6.2 Problem size

Figures 3.9, 3.10 showcase the amount of boxes generated as well as the rate of increase by each inclusion function as ϵ decreases. There are missing values in Figures 3.10a, 3.10b because the CPU execution did not exit successfully due to time constraints. As the nature of the sequential implementation the dynamic generation of the BB tree, it is not possible (to my knowledge) to mathematically calculate the number of boxes a priori. However, a very similar trend can be noticed when compared to Figures 3.9a and 3.10b.

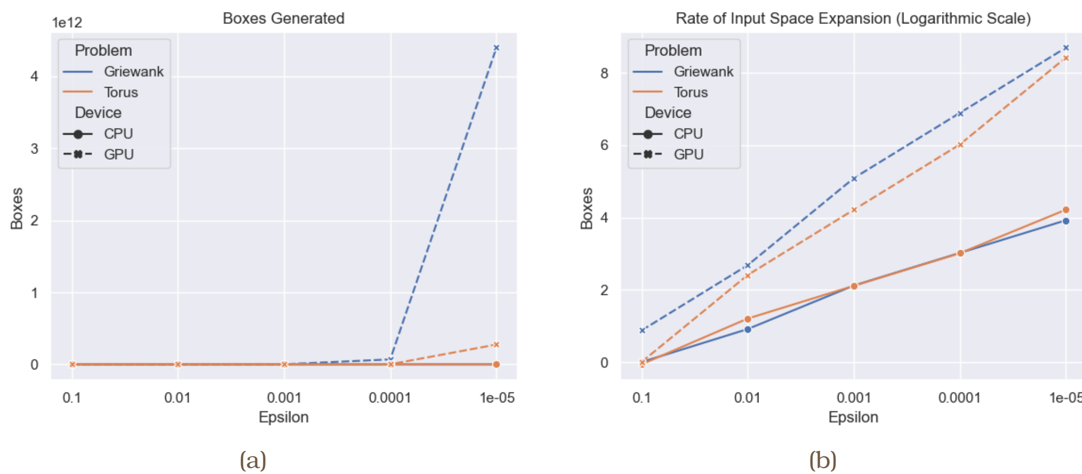


Figure 3.9: ***Problems 1 & 2***: *Number of Boxes Generated (Figure 3.9a). Rate of Box Generation (Figure 3.9b).*

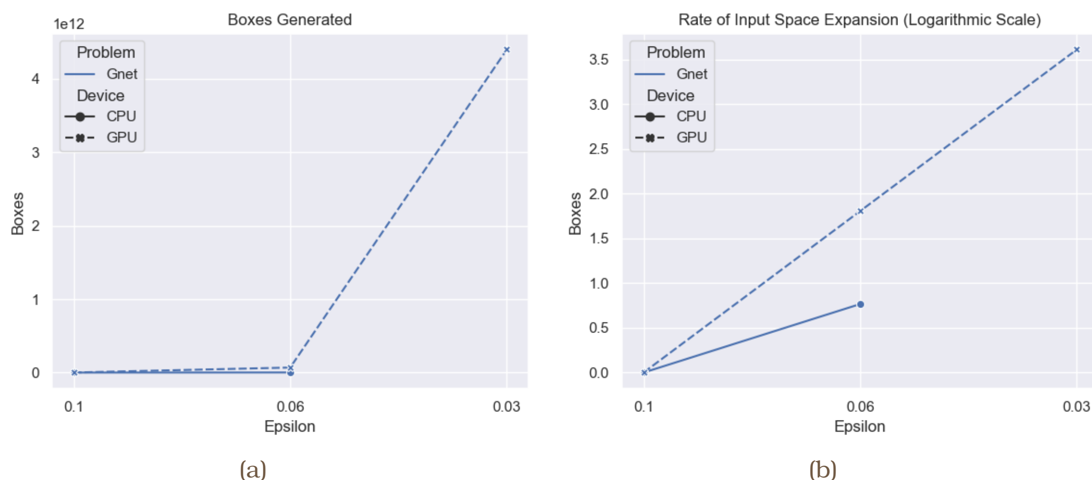


Figure 3.10: ***Problems 1 & 2***: *Number of Boxes Generated (Figure 3.9a). Rate of Box Generation (Figure 3.9b).*

3.6.3 Speedup

The speedup figures of Problems 1 and 2 show a very disheartening result, depicting slowdowns (Figures 3.11, 3.12, 3.13 & 3.14) . We can notice, however, an instance where the parallel algorithm initially performs better (Figure 3.12). Problem 3 on the other hand shows a completely different picture (Figures 3.15 and 3.16) with speedup values by a factor of thousands including cases where the performance ceiling has not been reached (4x & 8x A100). GPU models with less than 40GB of VRAM seem to have reached that performance ceiling. This might, however, be a consequence of the implementation and not a property of the respective architectures. One more test was conducted on Problem 3 with $\epsilon = 0.03$ but the sequential algorithm execution required more than 270 hours. Including this measurement without the program exiting properly would hinder the reliability of the results. That test, using 8 A100 GPUs required approximately 4 minutes, therefore we can be sure that the speedup value would not vary by a lot compared to the already depicted outcomes.

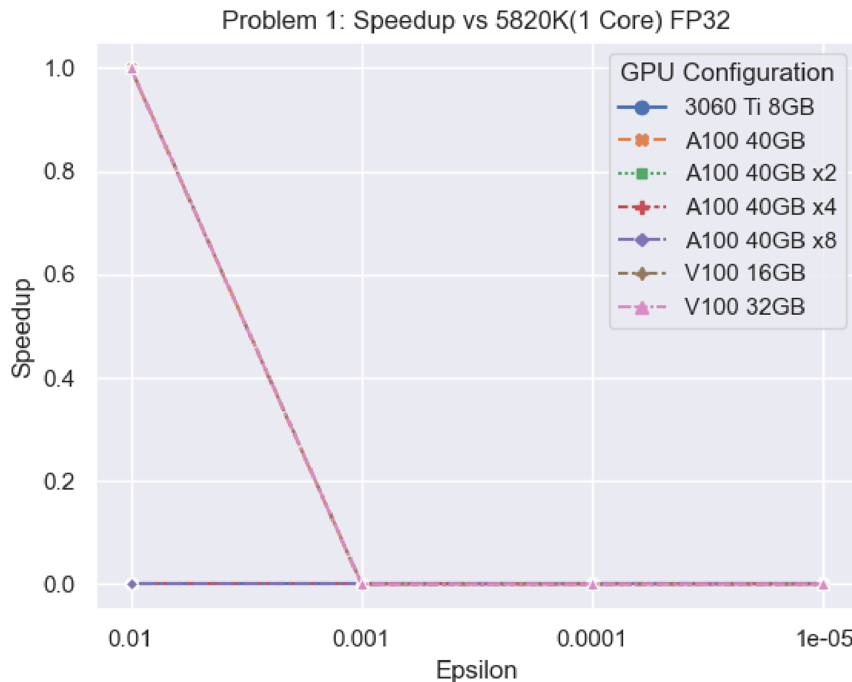


Figure 3.11: *The speedup plot of the Torus inclusion function using float variables.* $\epsilon = [1e-2, 1e-5]$

3.6.4 Throughput

Contrary to the worst scenarios depicted on the speedup figures, the throughput, meaning the boxes evaluated compared to the total time required, shows that performance improvements in fact do exist. Increases might seem marginal in many cases due to the figures being in logarithmic scale. Figures 3.21, 3.22 lack the $\epsilon = 0.03$ CPU time measurement. Multi-GPU executions display an overhead with smaller problem sizes.

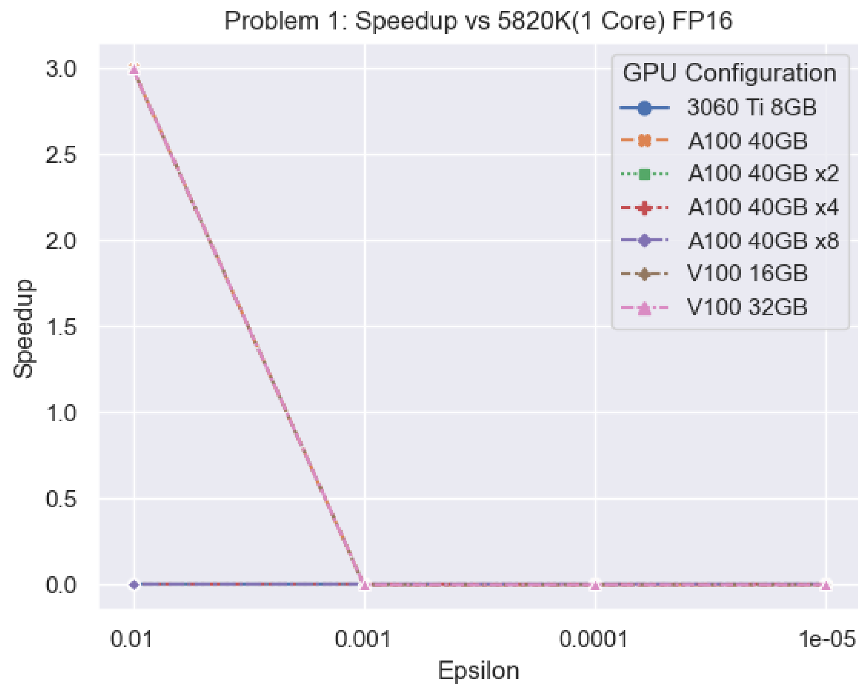


Figure 3.12: *The speedup plot of the Torus inclusion function using half variables.* $\epsilon = [1e-2, 1e-5]$

3.6.5 Number of Runs

In this section the amount of GPU executions required per problem and per variable are depicted. The main trend all figures depict is that the more memory available -by a multiple of 2- the lesser the amount of DEVICE executions required. This is in line with the bisection strategy deployed. Using half variables twice as many boxes can fit into the Global Memory.

3.6.6 Profiling

The final step to expose the culprit for the slowdowns encountered in Problems 1 & 2 is to use a profiler. We can finally determine (Figures 3.23a, 3.24a) that memory transfer bottlenecks is what caused the slowdowns (Figures 3.26, 3.27) as cudaMemcpy takes more than 85% of the total execution time. Problem 3, on the other hand, is bottlenecked by host-device synchronizations, but as we saw previously (Figures 3.15 & 3.16) the performance improvements are enormous.

3.7 Discussion

In this chapter, we noticed how SIVIA can be categorized as a Branch-and-Bound algorithm due to the way it generates and explores subproblems. We saw how the problems of this thesis do not provide any information that can lead to processing prioritization, resulting in the exhaustive search of the input space. General methods of parallelization

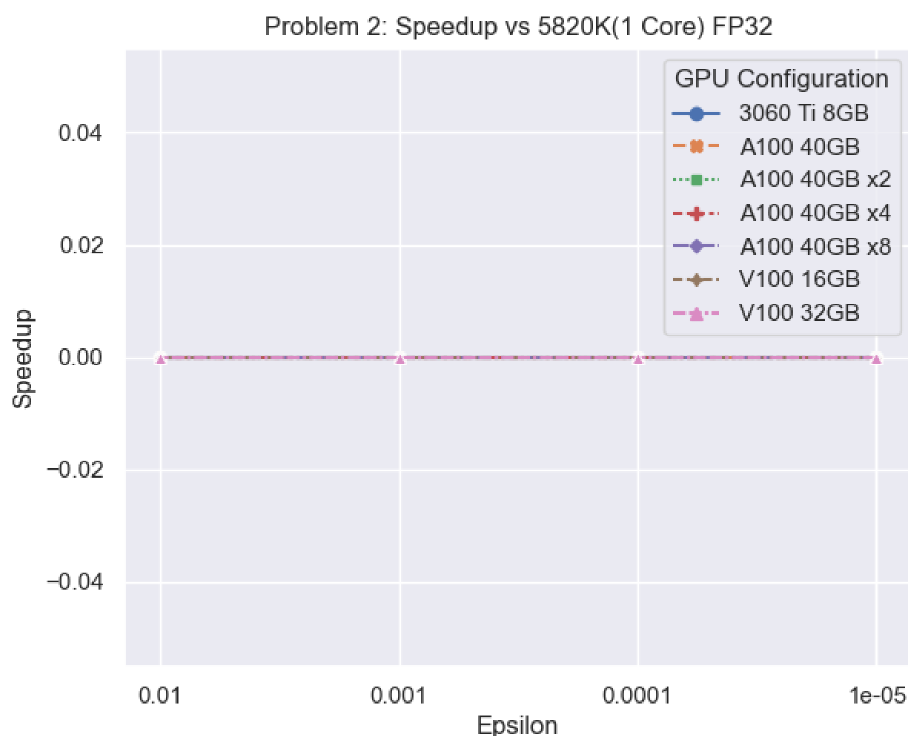


Figure 3.13: *The speedup plot of the Griewank inclusion function using float variables.* $\epsilon = [1e-2, 1e-5]$

followed, with GPUs as the primary focus and related background literature was explored. Finally, the parallelization structure proposed of this thesis was unveiled, using three different inclusion functions. The first two problems included intensive memory operations that resulted in slowdowns but excluding them on the 3rd problem resulted to executions more than eight thousand times faster when compared to the sequential algorithm. In the discussion several other figures and metrics were introduced, including the problem size and its rate of expansion, the throughput of each configuration as well as results from an execution profiling software which exposed the aforementioned bottlenecks.

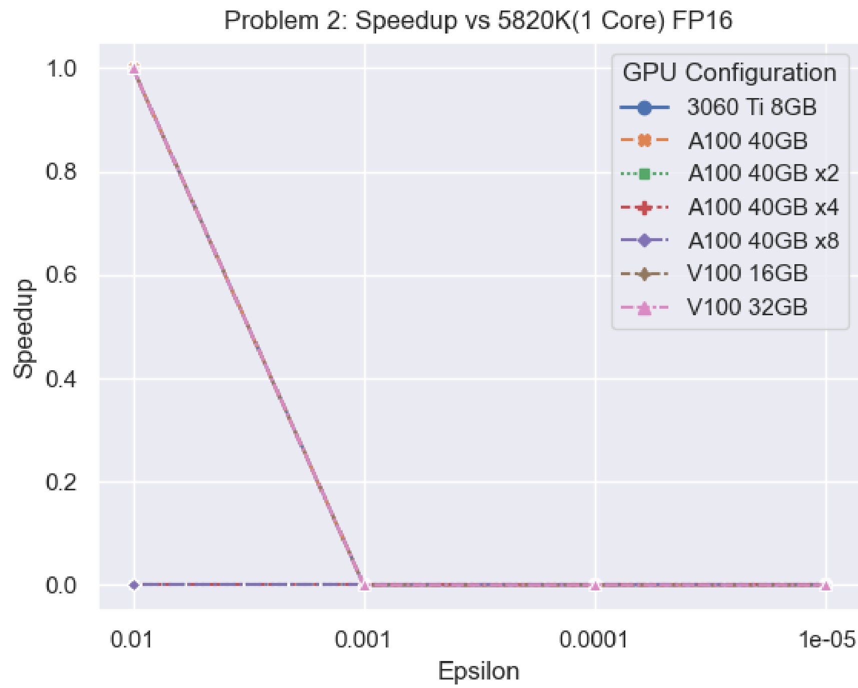


Figure 3.14: *The speedup plot of the Griewank inclusion function using half variables.* $\epsilon = [1e-2, 1e-5]$

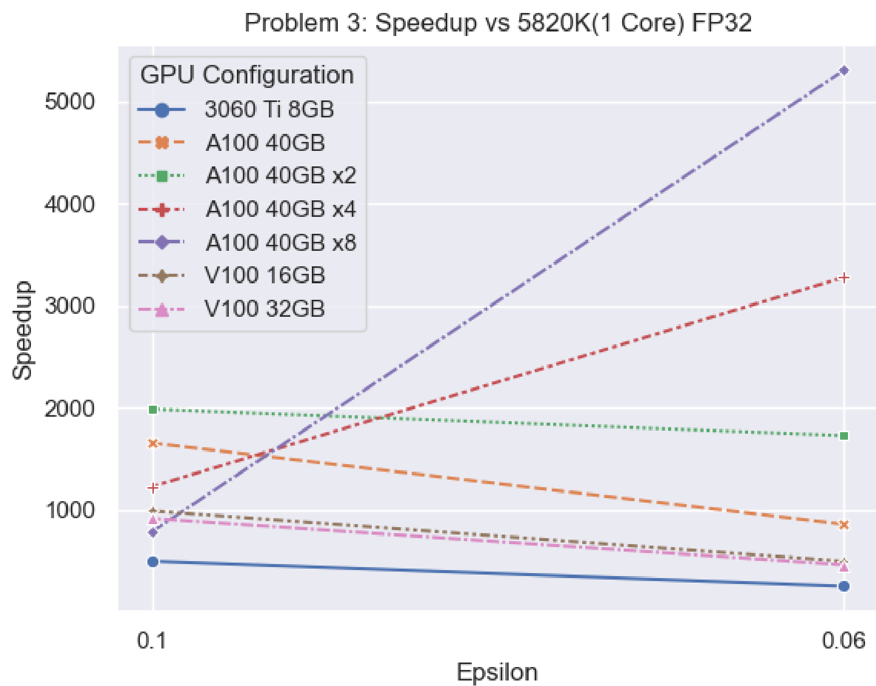


Figure 3.15: *The speedup plot of the* G_{net} *partial sum inclusion function using float variables.* $\epsilon = [0.1, 0.06]$

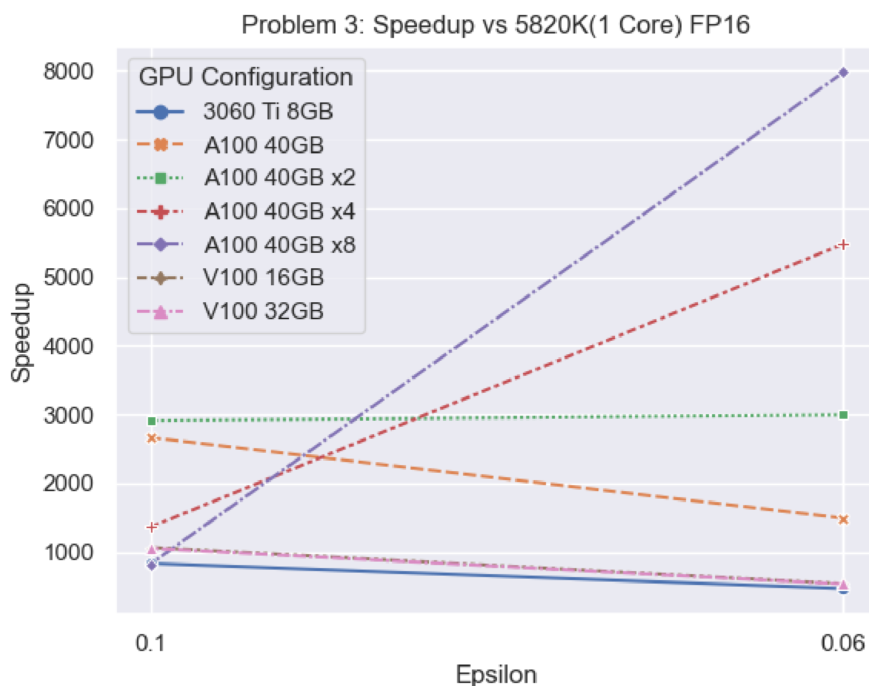


Figure 3.16: *The speedup plot of the G_{net} partial sum inclusion function using half variables.* $\epsilon = [0.1, 0.06]$

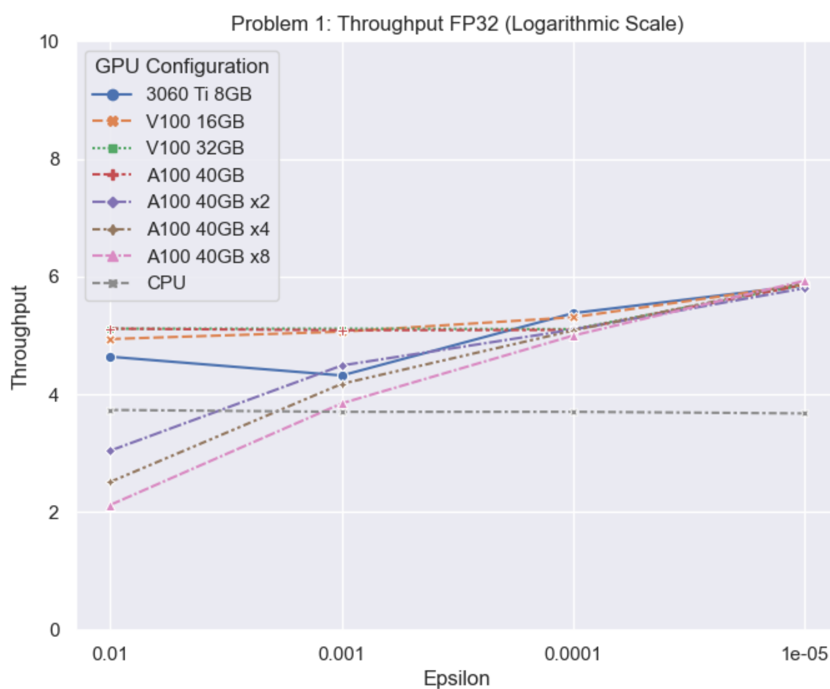


Figure 3.17: *The throughput plot of the Torus inclusion function using float variables.* $\epsilon = [1e-2, 1e-5]$

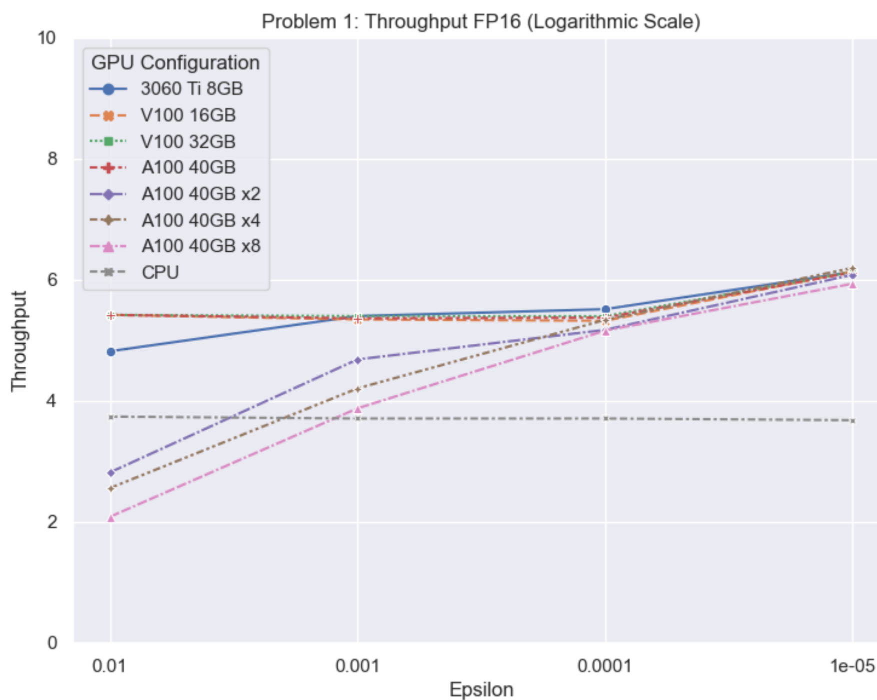


Figure 3.18: *The throughput plot of the Torus inclusion function using half variables.* $\epsilon = [1e-2, 1e-5]$

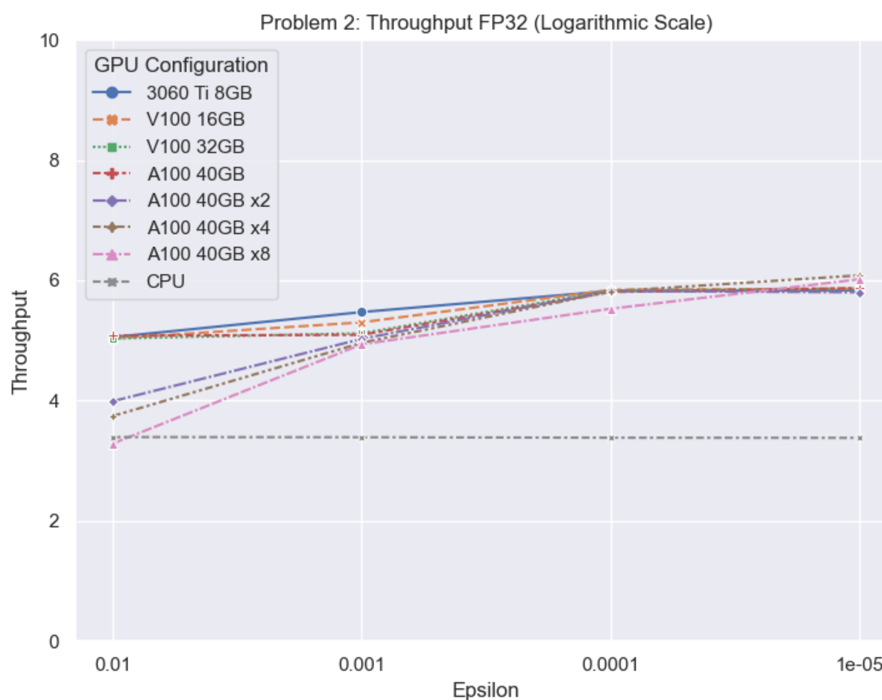


Figure 3.19: *The throughput plot of the Griewank inclusion function using float variables.* $\epsilon = [1e-2, 1e-5]$

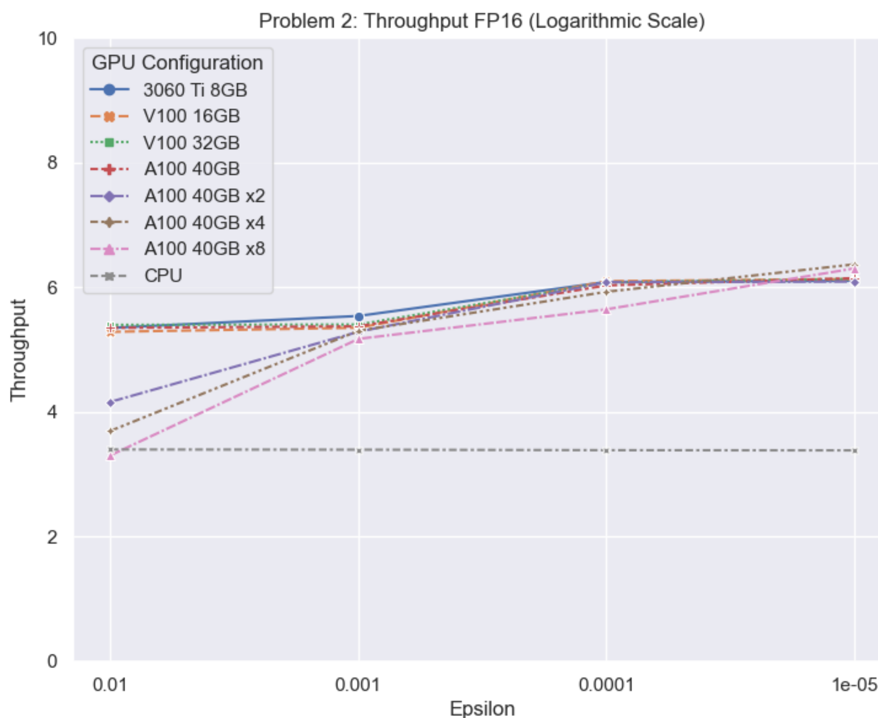


Figure 3.20: *The throughput plot of the Griewank inclusion function using half variables.* $\epsilon = [1e-2, 1e-5]$

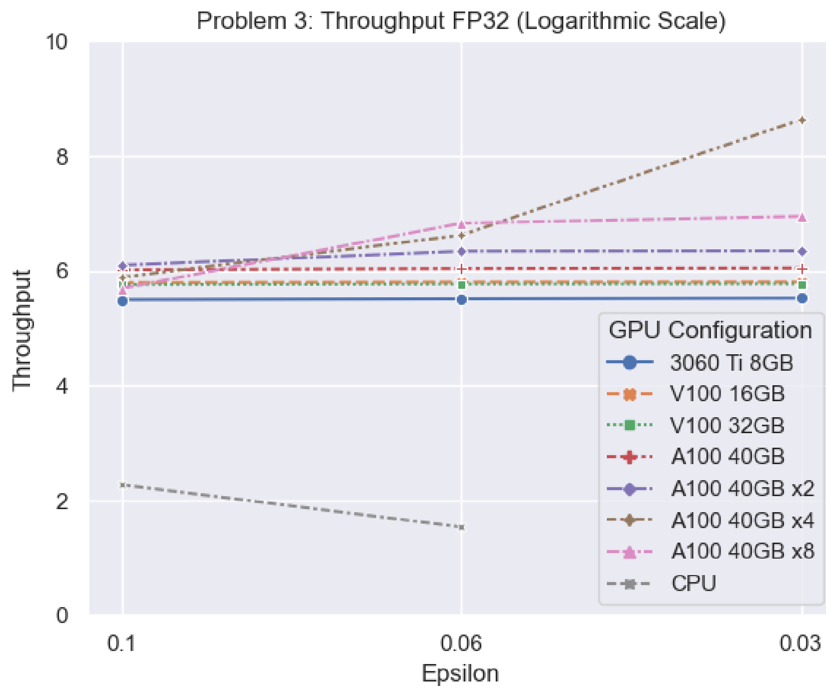


Figure 3.21: *The throughput plot of the* G_{net} *partial sum inclusion function using float variables.* $\epsilon = [0.1, 0.03]$

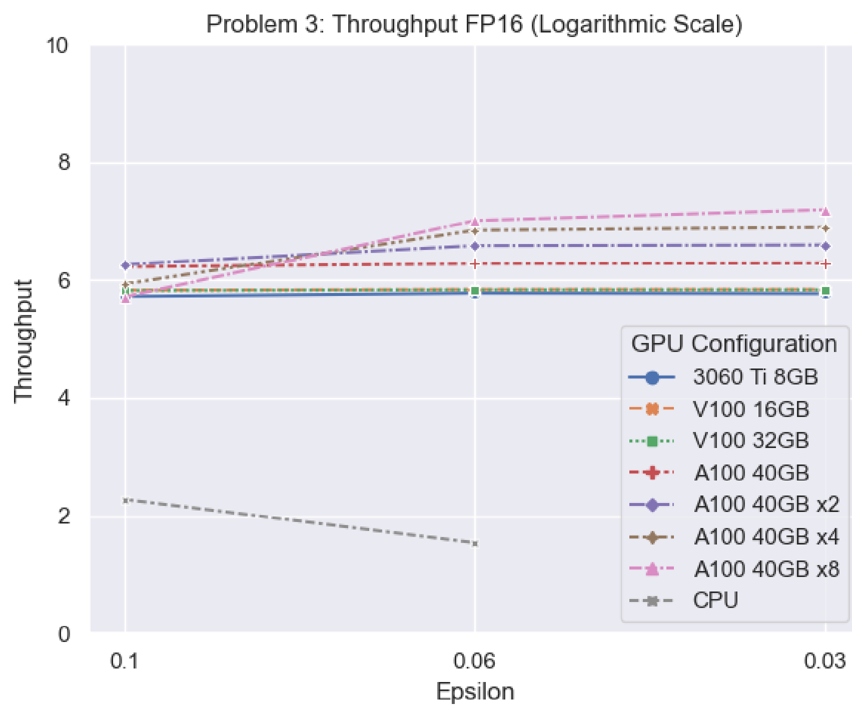
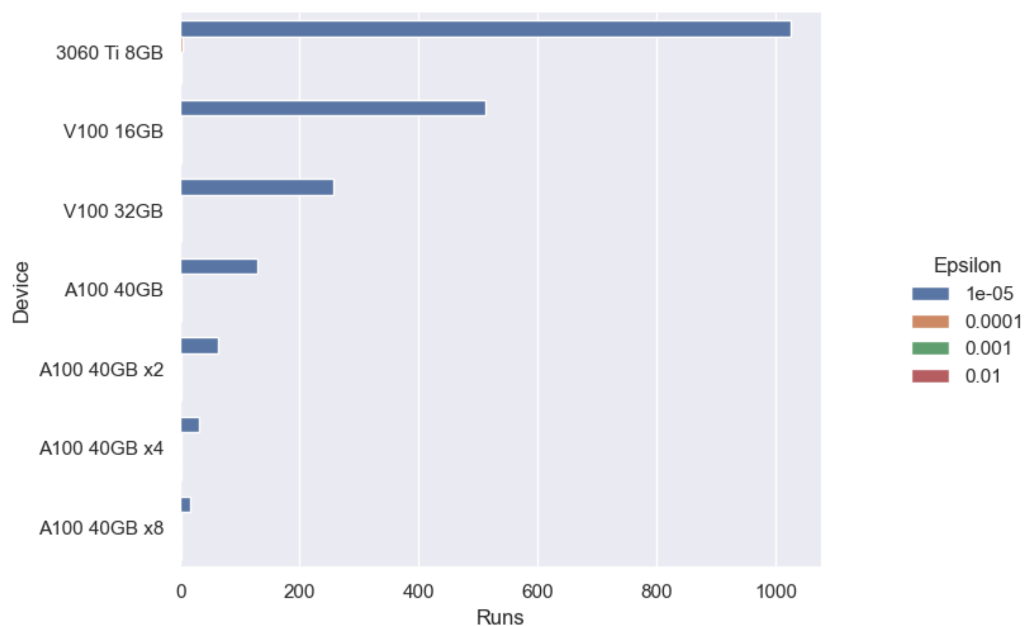
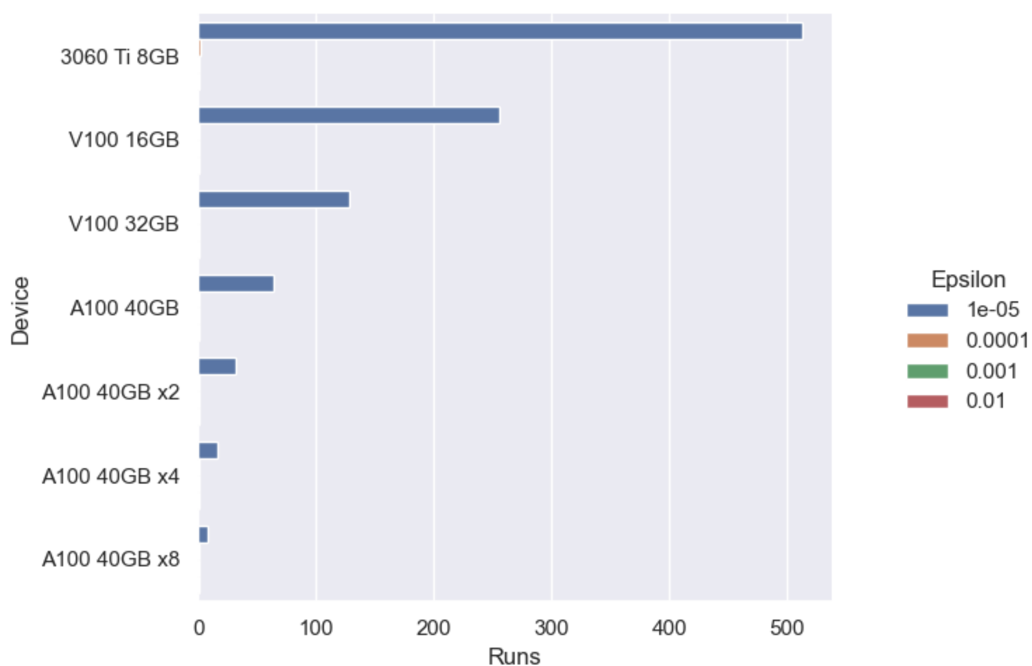


Figure 3.22: *The throughput plot of the G_{net} partial sum inclusion function using half variables.* $\epsilon = [0.1, 0.03]$

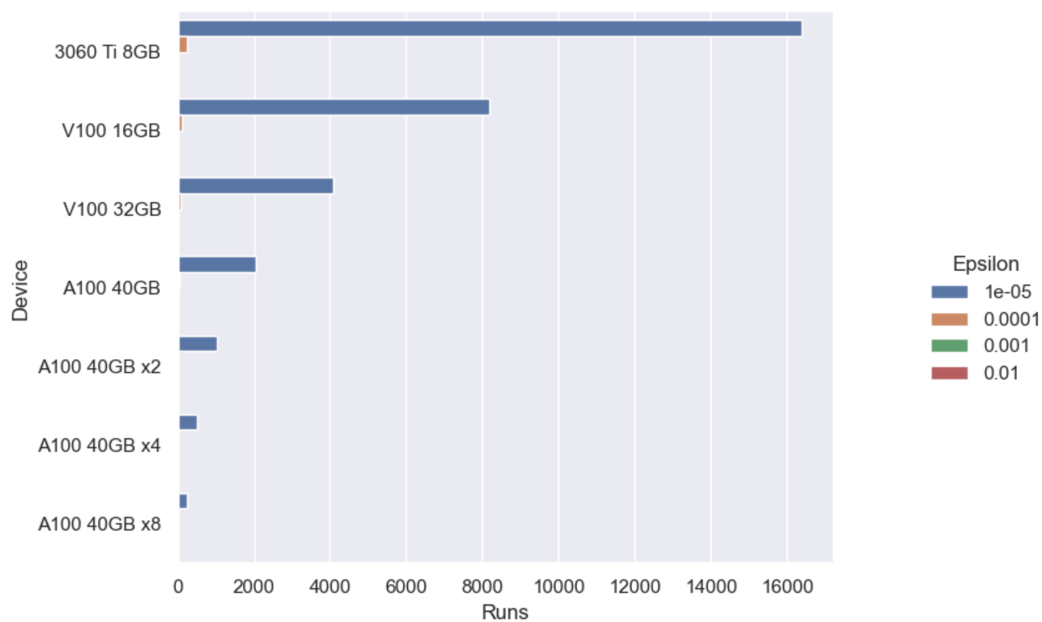


(a) *Float FP32 variables.*

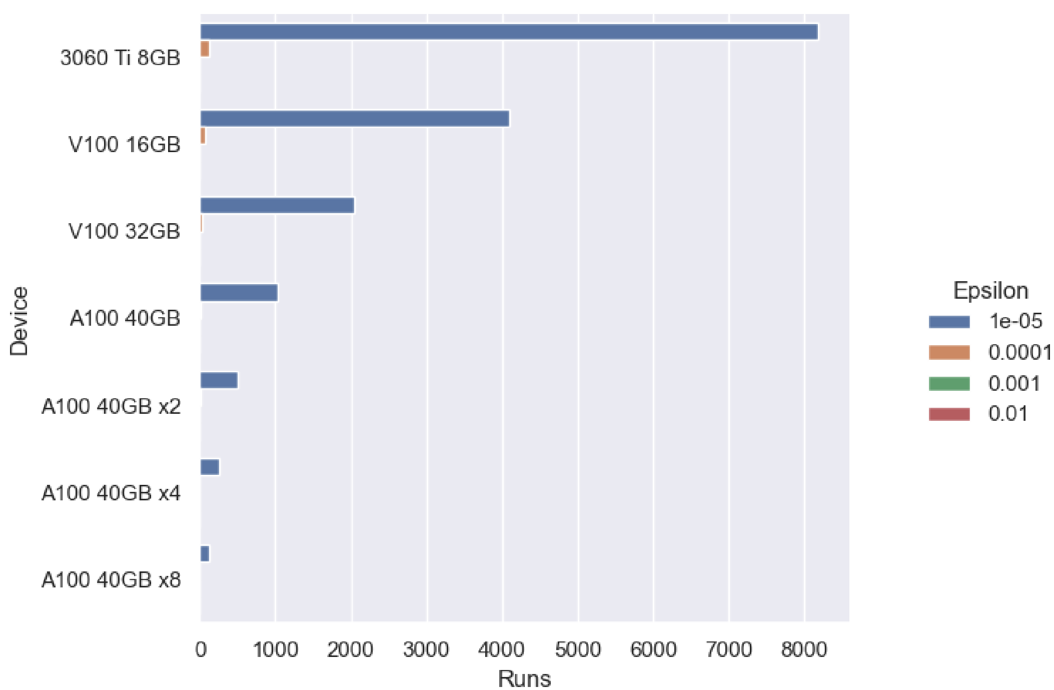


(b) *Half FP16 variables.*

Figure 3.23: ***Problem 1**: Number of Kernel runs.*

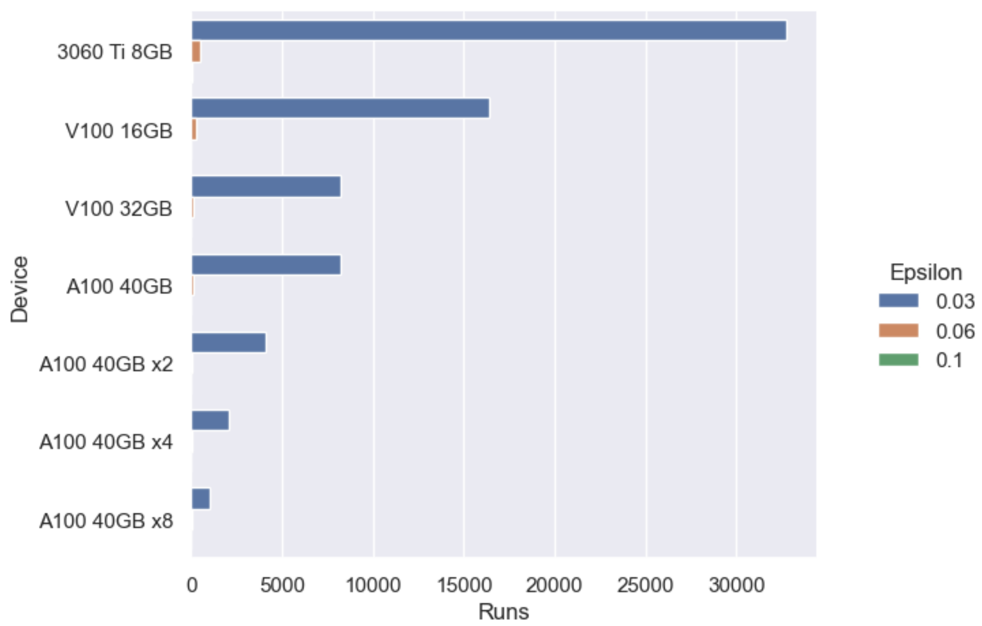


(a) *Float FP32 variables.*

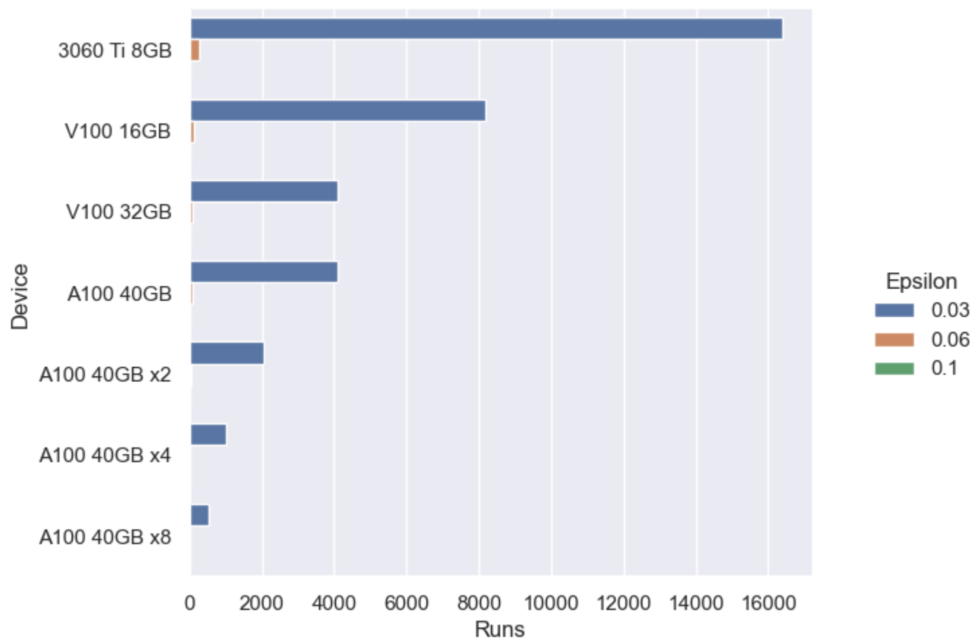


(b) *Half FP16 variables.*

Figure 3.24: ***Problem 2***: *Number of Kernel runs.*



(a) *Float FP32 variables.*



(b) *Half FP16 variables.*

Figure 3.25: ***Problem 3***: *Number of Kernel runs.*

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
94.0	19768397867	9	2196488651.9	83987300.0	61800	18054353560	5957089058.1	cudaMemcpy
3.4	710578802	1	710578802.0	710578802.0	710578802	710578802	0.0	cudaMallocHost
1.4	302196801	60	5036613.4	63350.0	5300	77571200	13790215.0	cudaDeviceSynchronize
0.9	180500301	3	60166767.0	54629700.0	473800	125396801	62645298.1	cudaMalloc
0.2	49644000	3	16548000.0	4892800.0	299300	44451900	24274385.4	cudaFree

Figure 3.26: ***Problem 1***: *The profiler output with* $\epsilon = 1e - 4$

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
85.2	5697886176	9	633098464.0	104344400.0	56500	3964895784	1317343026.4	cudaMemcpy
10.8	722301100	1	722301100.0	722301100.0	722301100	722301100	0.0	cudaMallocHost
2.5	168768499	60	2812808.3	43000.0	20200	51632499	9549282.8	cudaDeviceSynchronize
0.6	43053000	3	14351000.0	4908400.0	293300	37851300	20482257.3	cudaFree
0.6	42401600	3	14133866.7	11234500.0	550600	30616500	15241205.0	cudaMalloc

Figure 3.27: ***Problem 2***: *The profiler output with* $\epsilon = 1e - 3$

** CUDA API Summary (cuda_api_sum):

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
99.1	114912155468	7681	14960572.3	54200.0	2600	374076197	59393960.7	cudaDeviceSynchronize
0.6	684587990	1	684587990.0	684587990.0	684587990	684587990	0.0	cudaMallocHost
0.2	217664907	7680	28341.8	12800.0	6300	4600899	62991.5	cudaLaunchKernel
0.1	58488800	10	5848880.0	3500.0	2100	58446300	18480850.2	cudaFree
0.0	44460599	40	1111515.0	2050.0	1600	43923599	6943062.0	cudaMalloc
0.0	24409100	549	44461.0	40600.0	9500	317700	34166.6	cudaMemcpy

Figure 3.28: ***Problem 3***: *The profiler output with* $\epsilon = 0.06$

Chapter 4

Closing Words

4.1 Reproducibility

The code that produced all the plots and the pre-trained models that was used, is widely available at this GitHub repository. The code provided includes the basic structure of the parallelization for both single and multiple GPU configurations with only the Neural Network missing as it is not my intellectual property. Keep in mind that if the sequential algorithm is ever re-tested, my i7 5820k was overclocked at 4.3GHz and my memory was tuned at 2666MHz with CAS timing 15 at the time of writing this thesis, that would explain some variation between my tests and others.

4.2 Conclusion

This thesis proposes a new parallel Interval Arithmetic algorithm. Contrary to previous work on Branch-and-Bound and other IA implementations, this algorithm is based on a breadth-first approach, resulting in the aggressive expansion of the input space. The IA technique accelerated is named SIVIA and, when combined with a neural network, has the ability to deterministically compute its domain of validity, providing guaranteed approximations. The parallel approach requires large amounts of memory, however, with the massive parallelization GPGPU developments of the recent years, particularly thanks to NVIDIA's contributions, it seems that when there are no intensive memory transfer or synchronization operations the performance improvements are quite significant. It is of importance to mention that the memory bottlenecks introduced in the first two problems were a subjective requirement to demonstrate the performance changes between the two cases. If the storage of the boxes is indeed a requirement, the issue could be mitigated by moving all processing to the DEVICE (e.g. by writing an OpenGL visualization software). In addition, another very recent development was showcased for evaluating and comparing the generalization performance of Neural Networks as a response to flaws surrounding cross-validation statistical methods. That technique initially required weeks of sequential computational time, however, using a modest GPU cluster reduced the runtime to just four minutes, an eight thousand-fold improvement. It is interesting that when this technique is used with lower precision variables, the results are not of lesser quality due to the enormous execution time required to reach that limit. The way a network's generalization

was estimated comply with commonly known theoretical considerations (e.g. Occam's razor) although it is not without its flaws as it is considered a macroscopic measure of the validity domain, lacking the ability to provided any qualitative characteristics. Initially two more metrics were proposed to circumvent this issue, however, retrofitting the algorithm to accommodate them constitutes an open issue for future study. The performance results presented comply with all previous theoretical knowledge of parallel computing as well as NVIDIA's best practices guidelines. Using SIVIA with large scale neural models still remains a challenge due to the exhaustive search required in very large input spaces. Thus, if we want to broaden our knowledge of neural networks, additional mathematical techniques should accompany parallelization efforts.

4.3 Future Work

Research on B&B techniques and SIVIA is far from exhausted. Beyond comparing the results presented here with previous MIMD or finer-grained CUDA approaches, there are many other ways to move on from this work. This thesis, for instance, set the requirement for the storage of the resulting boxes of the first two problems for visualization and, consequently, verification purposes. The problem of the costly memory transfers, in this case, could be easily mitigated by implementing a visualizing tool using common graphics APIs such as OpenGL. Another interesting idea would be to combine past MIMD parallelization methods with the one proposed here. This is far from new, however, as many interval GPU implementations of the past utilized the DEVICE with multiple execution streams to perform finer-grained inclusion function passes, but none of those propositions included a parallel bisection method. This parallel bisection method could be used to generate larger problem sizes efficiently, ideally not as large as the current implementation which brutally expands the BB tree, for equal workload distribution and benefit from smaller problem sizes at the same time, further increasing speedups. Another idea would be to consider a method after the box evaluation that would result in a vector containing only the boxes that require further bisection, shortening the search window with every iteration, minimizing the search tree (in a more lax definition). Finally, modern technologies such as Julia could be explored and compare its efficiency to current implementations. Of course, this is not suggested in a vacuum, as in the SWIM 2023 conference there was a presentation regarding Interval Arithmetic in GPUs using the Julia language[47].

Bibliography

[1] L. Jaulin, M. Kieffer, O. Didrit, E. Walter, L. Jaulin, M. Kieffer, O. Didrit, and É. Walter, *Applied Interval analysis*. Springer, 2001.

[2] L. Jaulin and E. Walter, "Set inversion via interval analysis for nonlinear bounded-error estimation," *Automatica*, vol. 29, no. 4, pp. 1053–1064, 1993.

[3] "Cuda faq." https://developer.nvidia.com/cuda-faq. Accessed: 2023-09-12.

[4] S. P. Adam, A. C. Likas, and M. N. Vrahatis, "Evaluating generalization through interval-based neural network inversion," *Neural Computing and Applications*, vol. 31, no. 12, pp. 9241–9260, 2019.

[5] R. E. Moore, *Interval arithmetic and automatic error analysis in digital computing*. PhD thesis, Department of Mathematics, Stanford University, 1962.

[6] R. E. Moore, *Interval analysis*, vol. 4. Prentice-Hall Englewood Cliffs, 1966.

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

[8] U. W. Kulisch and W. L. Miranker, *Computer arithmetic in theory and practice*. Academic press, 2014.

[9] K. Nasiotis, D. López, S. Adam, and L. Casado, "Set inversion via interval analysis a study on parallel processing implementation," 2019.

[10] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to interval analysis*. SIAM, 2009.

[11] R. Rao, A. Asaithambi, and S. Agrawal, "Inverse kinematic solution of robot manipulators using interval analysis," 1998.

[12] M. Kieffer, L. Jaulin, É. Walter, and D. Meizel, "Robust autonomous robot localization using interval analysis," *Reliable computing*, vol. 6, no. 3, pp. 337–362, 2000.

[13] L. Jaulin, "Path planning using intervals and graphs," *Reliable computing*, vol. 7, no. 1, pp. 1–15, 2001.

[14] A. Piazzi and A. Visioli, "Global minimum-time trajectory planning of mechanical manipulators using interval analysis," *International journal of Control*, vol. 71, no. 4, pp. 631–652, 1998.

[15] E. Hansen and G. W. Walster, *Global optimization using interval analysis: revised and expanded*, vol. 264. CRC Press, 2003.

[16] P. Xu, "A hybrid global optimization method: the multi-dimensional case," *Journal of computational and applied mathematics*, vol. 155, no. 2, pp. 423–446, 2003.

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

[18] A. Piazzi and A. Visioli, "A global optimization approach to trajectory planning for industrial robots," in *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS'97*, vol. 3, pp. 1553–1559, IEEE, 1997.

[19] I. Braems, F. Berthier, L. Jaulin, M. Kieffer, and E. Walter, "Guaranteed estimation of electrochemical parameters by set inversion using interval analysis," *Journal of Electroanalytical Chemistry*, vol. 495, no. 1, pp. 1–9, 2000.

[20] M. B. Eriksen and S. Rasmussen, "Gpu accelerated parameter estimation by global optimization using interval analysis," 2013.

[21] S. P. Adam, G. D. Magoulas, D. A. Karras, and M. N. Vrahatis, "Bounding the search space for global optimization of neural networks learning error: an interval analysis approach," *Journal of Machine Learning Research*, vol. 17, pp. 1–40, 2016.

[22] S. P. Adam, D. A. Karras, G. D. Magoulas, and M. N. Vrahatis, "Reliable estimation of a neural network's domain of validity through interval analysis based inversion," in *2015 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2015.

[23] S. Haykin, *Neural Networks and Learning Machines*. Pearson India, 2008.

[24] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the royal statistical society: Series B (Methodological)*, vol. 36, no. 2, pp. 111–133, 1974.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 6th ed., 2017.

[26] P. Pacheco and M. Malensek, *An introduction to parallel programming*. Morgan Kaufmann, 2021.

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

[28] "Cuda c++ programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2023-10-06.

[29] "Cuda c++ best practices guide." https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/. Accessed: 2023-10-06.

[30] “Ampere turing guide.” https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html. Accessed: 2023-10-06.

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

[32] G. A. P. Kindervater and H. W. J. M. Trienekens, “Experiments with parallel algorithms for combinatorial problems,” *European Journal of Operational Research*, vol. 33, no. 1, pp. 65–81, 1988.

[33] F. Goualard, “Fast and correct simd algorithms for interval arithmetic,” in *PARA'08*, Springer, 2008.

[34] F. Dehne, A. G. Ferreira, and A. Rau-Chaplin, “Parallel branch and bound on fine-grained hypercube multiprocessors,” in *IEEE International Workshop on Tools for Artificial Intelligence*, pp. 616–617, IEEE Computer Society, 1989.

[35] C. Collange, J. Flórez, and D. Defour, “A gpu interval library based on boost. interval,” in *8th conference on real numbers and computers*, pp. 61–71, 2008.

[36] Z. Bagóczki and B. Bánhelyi, “A parallel interval arithmetic-based reliable computing method on a gpu,” *Acta Cybernetica*, vol. 23, pp. 491–501, Jan. 2017.

[37] B. Chrétien, A. Escande, and A. Kheddar, “Gpu robot motion planning using semi-infinite nonlinear programming,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2926–2939, 2016.

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

[39] “C-xsc - a c++ class library for extended scientific computing.” https://www2.math.uni-wuppertal.de/wrswt/xsc/cxsc/apidoc/html/index.html. Accessed: 2023-10-14.

[40] K. N*ασιώτης*, “*Υλοποίηση τεχνικών καθολικής βελτιστοποίησης* (branch and bound) *σε περιβάλλον παράλληλης επεξεργασίας*. εφαρμογή *σε νευρωτικά δίκτυα*.,” 2020.

[41] “Cuda samples.” https://github.com/NVIDIA/cuda-samples. Accessed: 2023-10-14.

[42] “Ibex is a c++ library for constraint processing over real numbers..” https://github.com/ibex-team/ibex-lib. Accessed: 2023-10-14.

[43] M. Harris *et al.*, “Optimizing parallel reduction in cuda,” *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

[44] “Visualizer for intervals and boxes.” https://github.com/ENSTABretagneRobotics/VIBES. Accessed: 2023-10-15.

[45] S. P. Adam, A. C. Likas, and M. N. Vrahatis, “Interval analysis based neural network inversion: a means for evaluating generalization,” in *Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25–27, 2017, Proceedings*, pp. 314–326, Springer, 2017.

[46] G. Barreto and A. Neto, “Vertebral Column.” UCI Machine Learning Repository, 2011. DOI: https://doi.org/10.24432/C5K89B.

[47] L. Gillner and E. Auer, “Interval methods for the gpu,” SWIM 2023.

Appendix A

A more detailed look on Problem 3

The technique mentioned in Section 3.4.2 is only one of the three metrics of the original paper[4]. The G_{net} which was discussed is considered a macroscopic measure as it is based on the whole volume of the validity domain and the total number of misclassified/unclassified patterns. This is not a problem when under-training a model as this can be detected simply by looking at the training error, but it is very tricky when considering over-training. So a second measure namely E_{net} was proposed which takes into account local information of the domain of validity, as it considers for each valid box both its volume and its performance. The performance in this case means the evaluation of training patterns inside a box, whether they are correctly classified or not. It is important to mention that it would potentially require a different parallelization strategy, therefore accelerating this measurement is a subject of a potential future study. This measure tends to consider hyper-boxes based on the density of correctly classified patterns and results in the following effects:

- It deals with overlapping between classes.
- It rejects regions of the domain of validity that do not contain any pattern. These areas are taken into account by G_{net} but they do not contribute to the local behavior of the decision surface between classes.
- It favors the volume of hyper-boxes with classification performance.

The third bullet point means that E_{net} provides lower values in the case of over-training. Consequently, E_{net} acts as a complementary metric to G_{net}.

The third metric is named M_{net} with the purpose of effectively combining both G_{net} and E_{net}. It is computed by

$$M_{net} = G_{net} tanh(E_{net}).$$

The hyperbolic tangent is used to transform E_{net} into the interval $[-1, 1]$ so that they are of the same scale as the G_{net}. In the case where $E_n et$ is negative, wrong classifications are expected to be higher than correct ones, thus the network is poorly trained. This results in negative M_{net} values and is indicative of an unacceptable network.

The original paper has some very interesting figures, visualizing the formation domain of validity in different cases. Consider an artificial dataset of two classes (Figure A.1). Different MLP configurations were trained in this dataset, using logistic sigmoid activation

functions. Figure A.2 illustrates the ways the domain of validity is formed when adjusting the β value of the output interval $[1-\beta, 1]$. Regions in red are regions categorized in one of the two classes. Figure A.3 depicts the formation of the domain of validity in regards to the quality of a network's training. We can conclude that the β value should be chosen carefully, otherwise we would end up with a skewed perception of the input space.

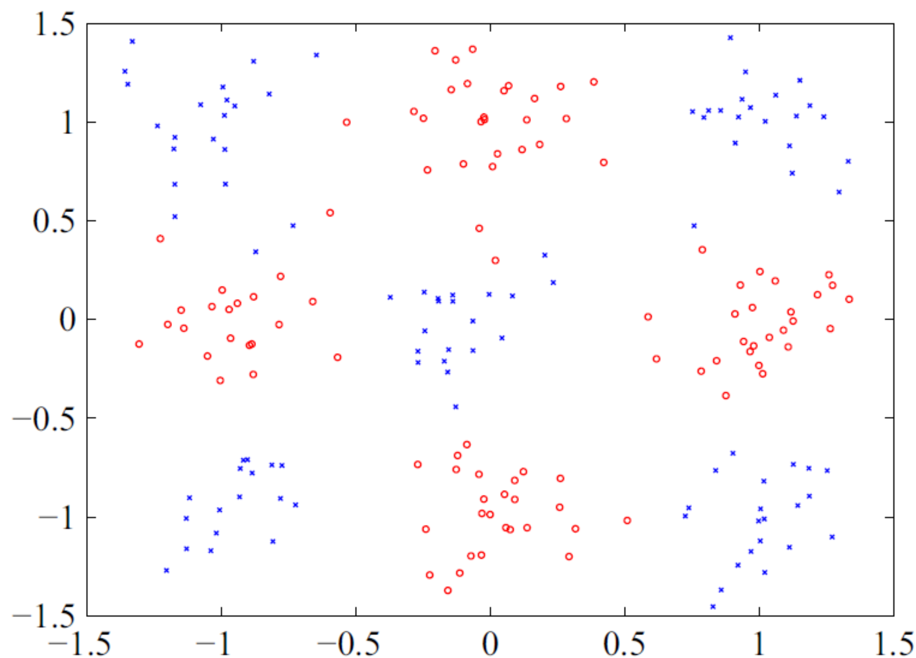
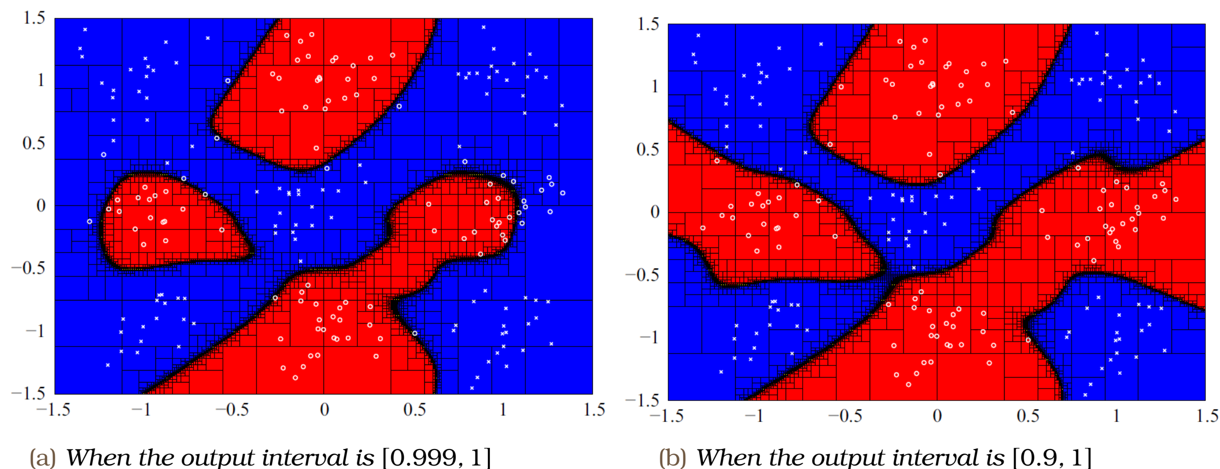


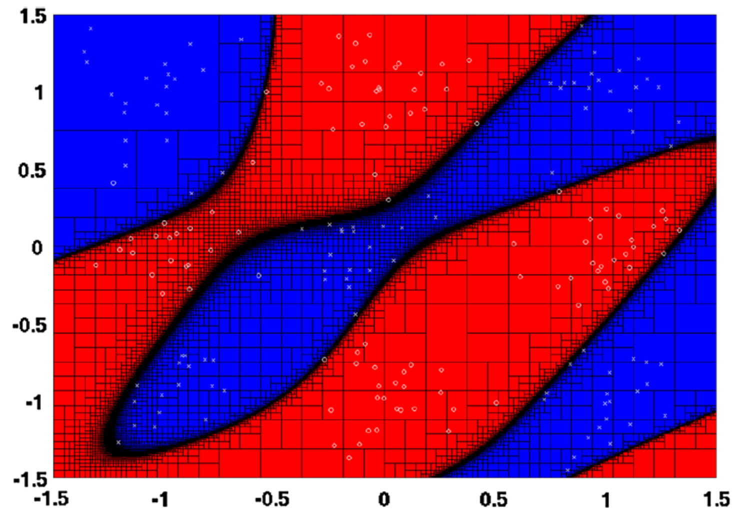
Figure A.1: *An artificial dataset of two classes.*



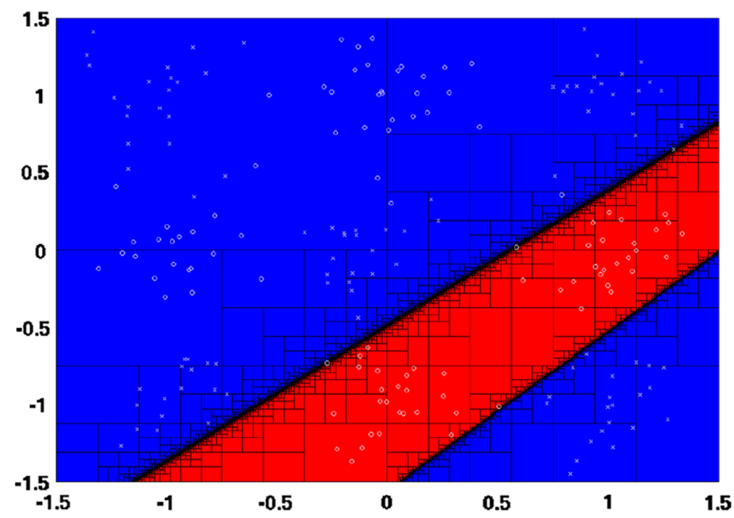
(a) *When the output interval is* [0.999, 1]

(b) *When the output interval is* [0.9, 1]

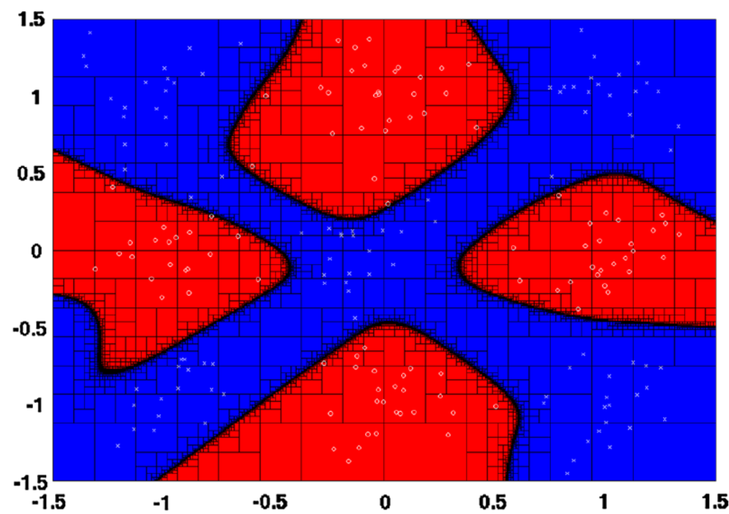
Figure A.2: *Depiction of the β cut affecting the domain of validity.*



(a) *Domain of validity resulting from successfully training a 2-4-2 MLP.*



(b) *Domain of validity resulting from under-training a 2-2-2 MLP.*



(c) *Domain of validity resulting from over-training a 2-25-2 MLP.*

Figure A.3: *Depiction of training affecting the domain of validity using the same β value.*

Appendix B

Detailed Results

DEVICE I Epsilon	1e-2	1e-3	1e-4	1e-5
RTX 3060 Ti 8GB	6	799	4461	383811
TESLA V100 16GB	3	144	5241	372590
TESLA V100 32GB	2	128	8338	375094
A100 40GB	2	136	8616	376657
A100 40GB x2	236	540	8483	429352
A100 40GB x4	800	1107	8808	346272
A100 40GB x8	1971	2376	10730	321001

Table B.1: ***Problem 1***: *GPU Execution time(ms) using* ***FP32*** *Floating-point variables.*

DEVICE I Epsilon	1e-2	1e-3	1e-4	1e-5
RTX 3060 Ti 8GB	4	67	3303	204567
TESLA V100 16GB	1	76	5142	200305
TESLA V100 32GB	1	67	4309	204084
A100 40GB	1	73	4561	208053
A100 40GB x2	403	349	7261	227864
A100 40GB x4	731	1054	4965	175196
A100 40GB x8	2173	2250	7475	319831

Table B.2: ***Problem 1***: *GPU Execution time(ms) using* ***FP16*** *Floating-point variables.*

DEVICE I Epsilon	1e-2	1e-3	1e-4	1e-5
RTX 3060 Ti 8GB	37	3625	103437	6463740
TESLA V100 16GB	39	5373	98936	6071450
TESLA V100 32GB	39	8174	101878	6070690
A100 40GB	35	8661	107348	5940950
A100 40GB x2	430	10120	103124	6964450
A100 40GB x4	760	11784	105390	3635880
A100 40GB x8	2172	12447	204537	4227760

Table B.3: ***Problem 2***: *GPU Execution time(ms) using* ***FP32*** *Floating-point variables.*

DEVICE | Epsilon	1e-2	1e-3	1e-4	1e-5
RTX 3060 Ti 8GB	19	3132	57052	3499940
TESLA V100 16GB	22	4842	55501	3290443
TESLA V100 32GB	17	4298	58348	3271100
A100 40GB	19	4594	64817	3177030
A100 40GB x2	292	5616	57560	3605100
A100 40GB x4	853	5420	82124	1905760
A100 40GB x8	2100	7401	156883	2217960

Table B.4: ***Problem 2***: *GPU Execution time(ms) using* ***FP16*** *Floating-point variables.*

DEVICE & Problem | Epsilon	1e-2	1e-3	1e-4	1e-5
CPU Torus	3	26	208	3546
CPU Griewank	29	468	3818	30732

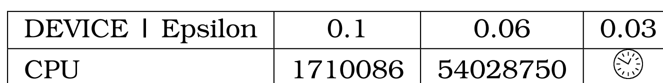
Table B.5: ***Problem 1 & 2*** : *CPU Execution time(ms) using the sequential algorithm.*

DEVICE | Epsilon	0.1	0.06	0.03
RTX 3060 Ti 8GB	3424	211982	13148015
TESLA V100 16GB	1725	108296	6900080
TESLA V100 32GB	1864	116637	7374800
A100 40GB	1032	62783	3939303
A100 40GB x2	860	31260	1972117
A100 40GB x4	1393	16471	10179
A100 40GB x8	2180	10179	494029

Table B.6: ***Problem 3***: *GPU Execution time(ms) using* ***FP32*** *Floating-point variables.*

DEVICE | Epsilon	0.1	0.06	0.03
RTX 3060 Ti 8GB	2041	114540	7463943
TESLA V100 16GB	1599	99614	6373648
TESLA V100 32GB	1618	100648	6441754
A100 40GB	641	36130	2274617
A100 40GB x2	587	18030	1123189
A100 40GB x4	1254	9854	561906
A100 40GB x8	2073	6772	282741

Table B.7: ***Problem 3***: *GPU Execution time(ms) using* ***FP16*** *Floating-point variables.*



DEVICE | Epsilon	0.1	0.06	0.03
CPU	1710086	54028750	

Table B.8: ***Problem 3***: *CPU Execution time(ms) of the sequential algorithm.*

DEVICE I Epsilon	1e-2	1e-3	1e-4	1e-5
RTX 3060 Ti	1	1	4	1024
TESLA V100 16GB	1	1	2	512
TESLA V100 32GB	1	1	1	256
A100 40GB	1	1	1	128
A100 40GB x2	1	1	1	64
A100 40GB x4	1	1	1	32
A100 40GB x8	1	1	1	16

Table B.9: ***Problem 1***: *Number of kernel executions using* ***FP32*** *Floating-point variables.*

DEVICE I Epsilon	1e-2	1e-3	1e-4	1e-5
RTX 3060 Ti	1	1	2	512
TESLA V100 16GB	1	1	1	256
TESLA V100 32GB	1	1	1	128
A100 40GB	1	1	1	64
A100 40GB x2	1	1	1	32
A100 40GB x4	1	1	1	16
A100 40GB x8	1	1	1	8

Table B.10: ***Problem 1***: *Number of kernel executions using* ***FP16*** *Floating-point variables.*

DEVICE I Epsilon	1e-2	1e-3	1e-4	1e-5
RTX 3060 Ti	1	4	256	16384
TESLA V100 16GB	1	1	128	8192
TESLA V100 32GB	1	1	64	4096
A100 40GB	1	1	32	2048
A100 40GB x2	1	1	16	1024
A100 40GB x4	1	1	8	512
A100 40GB x8	1	1	4	256

Table B.11: ***Problem 2***: *Number of kernel executions using* ***FP32*** *Floating-point variables.*

DEVICE I Epsilon	1e-2	1e-3	1e-4	1e-5
RTX 3060 Ti	1	2	128	8192
TESLA V100 16GB	1	1	64	4096
TESLA V100 32GB	1	1	32	2048
A100 40GB	1	1	16	1024
A100 40GB x2	1	1	8	512
A100 40GB x4	1	1	4	256
A100 40GB x8	1	1	2	128

Table B.12: ***Problem 2***: *Number of kernel executions using* ***FP16*** *Floating-point variables.*

DEVICE | Epsilon	0.1	0.06	0.03
RTX 3060 Ti	8	512	32768
TESLA V100 16GB	4	256	16384
TESLA V100 32GB	2	128	8192
A100 40GB	2	128	8192
A100 40GB x2	1	64	4096
A100 40GB x4	1	32	2048
A100 40GB x8	1	16	1024

Table B.13: ***Problem 3***: *Number of kernel executions using* ***FP32*** *Floating-point variables.*

DEVICE | Epsilon	0.1	0.06	0.03
RTX 3060 Ti	4	256	16384
TESLA V100 16GB	2	128	8192
TESLA V100 32GB	1	64	4096
A100 40GB	1	64	4096
A100 40GB x2	1	32	2048
A100 40GB x4	1	16	1024
A100 40GB x8	1	8	512

Table B.14: ***Problem 3***: *Number of kernel executions using* ***FP16*** *Floating-point variables.*

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
BB	Branch and Bound
CPU	Central Processing Unit
CU	Compute Unit
DL	Deep Learning
GPGPU	General Purpose Graphical Processing Units
GPU	Graphical Processing Unit
GUI	Graphical User Interface
IA	Interval Arithmetic
ILP	Instruction-Level Parallelism
MIMD	Multiple Instructions Multiple Data
MISD	Multiple Instructions Single Data
MLP	Multilayer Perceptron
ML	Machine Learning
MP	Multiprocessor
NN	Neural Network
NUMA	Non-Uniform Memory Access
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SISD	Single Instruction Single Data
SIVIA	Set Inversion Via Interval Analysis
SM	Streaming Multiprocessor
SP	Streaming Processor
TLP	Thread-Level Parallelism
UMA	Uniform Memory Access
VRAM	Video Random Access Memory