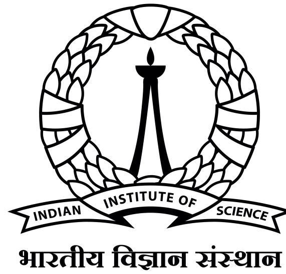
Scalable Computation of Extremum Graphs

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology
IN
Faculty of Engineering

BY

Abhijath Ande



Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

July, 2021

Declaration of Originality

I, **Abhijath Ande**, with SR No. **04-04-00-10-42-19-1-17154** hereby declare that the material presented in the thesis titled

Scalable Computation of Extremum Graphs

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2019-2021**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

© Abhijath Ande
July, 2021
All rights reserved

DEDICATED TO

My parents

Acknowledgements

First, I would like to thank Prof. Vijay Natarajan of the Visualization and Graphics Lab (VGL), under the Department of Computer Science and Automation, IISc Bangalore, for his involvement in this journey from the beginning and providing the necessary support and direction throughout. He was gracious and enthusiastic to accept to mentor and guide me. Regular discussions with him, and his inquisitive questioning at every detail of my work, has propelled me to be a good researcher.

I am very appreciative of the support I received from my friend Varshini Subhash, of the VGL Lab. The progress of my work picked up great pace thanks in parts to the countless discussions and knowledge-sharing sessions I had with her.

I would finally like to express my gratitude to my friends in the VGL Lab and all the loved ones who were always willing to provide a patient ear when I needed it, and constant encouragement to bring this work to the finish line.

Abstract

Extremum graphs have become increasingly important in the field of topology, as they allow for dimensionality reduction and exploratory analysis of high-dimensional scalar fields, while also preserving local geometric structure. As the interesting features of a scalar field are mostly associated with its extrema, this makes extremum graphs an appealing choice for high-dimensional scalar field analysis. But various works define their own proper but different notions of extremum graphs and how to compute them. This causes a lot of research time to be spent on designing and implementing common boilerplate code to compute extremum graphs for their studies. It also causes difficulties in transferring and comparing results across different studies. To solve these issues, we have created scalable library to compute extremum graphs utilizing GPU parallelism.

Contents

Acknowledgements i

Abstract ii

Contents iii

List of Figures v

List of Tables vi

1 Introduction 1

1.1 Motivation 1

1.2 Project Goal 2

2 Related Work 3

2.1 Extremum graphs 3

2.2 Morse-Smale complex 3

2.3 Applications of extremum graphs 4

3 Problem Description 5

3.1 Problem Statement 5

3.2 Solution Overview 5

3.3 Major Contributions 6

4 Background and Definitions 7

4.1 Morse-Smale Complex 8

4.2 Extremum Graphs 8

5 Computation of Extremum Graphs **10**
5.1 Point Classification 10
5.2 Grid Tessellation 12
5.3 Point classification for boundary points 13
5.4 Gradient Path Tracing 14
5.5 Persistence Simplification 14

6 Design and Implementation **16**
6.1 Simulation of Simplicity 16
6.2 Point Classification 17
6.3 Gradient Path Tracing 19
6.4 External Memory Algorithm 21

7 Results **23**
7.1 Internal Memory Computations 23
7.2 External Memory Computations 24
7.3 Strong Scaling 26

8 Conclusions & Future Work **27**
8.1 Conclusions 27
8.2 Future Work 27

Bibliography **28**

List of Figures

5.1 Various classification of a point in a 3D grid. 11
5.2 Grid tessellation in 2D and 3D grids. 13
5.3 Gradient path tracing in a 3D grid 14

6.1 External memory computation: Block division of a 3D dataset. 21

7.1 Internal memory computation experiment results 24
7.2 External memory computation experiment results 25
7.3 Strong Scaling results for point classification 26

List of Tables

7.1 Timings for internal memory computation 24
7.2 Timings for external memory computation 25
7.3 Number of critical points identified for various datasets 26

Chapter 1

Introduction

This section introduces the various challenges faced by researchers in computing extremum graphs, followed by the motivation for the project, and the targeted goal.

1.1 Motivation

Topological analysis has become a key tool in all branches of scientific research. The topology of scalar fields provide great insight into the inner workings of many natural phenomena. There exist various tools to study the topology of scalar fields such as contour trees, Morse-Smale complexes and extremum graphs. Each of these abstractions provide different perspectives to analyze and interact with the same scalar field.

Extremum graphs have become on of the widely used tools in the field of topological analysis. It provides an abstract yet intuitive representation of the underlying scalar field while also preserving extrema connectivity. This key property of extremum graphs makes it very appealing tool and lends itself to a variety of applications. For example in feature tracking for cases where many interesting features of a scalar field are extrema related. Also for time-varying data where extremum graphs are computed for all time steps, the evolution of these graphs over time uncovers insights about various key events within the data.

Extremum graphs are usually computed in two ways. One is to directly extract it from the Morse-Smale complex, which is a super-set of the extremum graph. This method becomes infeasible for higher dimensions as the extremum graph becomes a sparser subset of the MS complex, besides the fact that computation of MS complex in higher dimensions by itself is a highly cumbersome and non-trivial task.

The other method is to perform a flood fill search to directly compute extremum graphs, where one scans all iso-surfaces in decreasing order of function value. Creation of new components in the iso-surfaces represents the birth of new extrema. Moving to lower valued iso-surfaces will allow us to track the descending manifold of all existing extrema. The highest function valued point found among the common boundary shared by the descending manifolds of adjacent extrema correspond to an $(n-1)$-saddle. This method is inefficient in higher dimensions due to the number of data points needed to be tracked at all stages. The work of [7] employs the use of this method to identify Morse cells.

1.2 Project Goal

For an algorithm which directly compute extremum graphs to be scalable, it must be able to divide the total computation task evenly among all available processors. This is a non-trivial task, considering how varied extremum graphs can be depending on the topology of the underlying scalar field. Another non-trivial task is to correctly identify gradient paths which span across multiple partitions of data, in an external memory computation task.

The goal of this project is to directly compute extremum graphs via a path tracing methodology. This is an alternate method to the flood-fill approach. In this method, we first extract the locations of extrema and $(n-1)$-saddles directly by testing the upper link topology for each data point. Then we find the gradient paths from the $(n-1)$-saddles by traversing the appropriate gradient vectors along adjacent regular points. This method has the advantage of utilizing low memory resources as we never track all data points to obtain the critical points of interest.

Chapter 2

Related Work

This chapter discusses about the paper which introduces extremum graphs, existing works for computing Morse-Smale complexes and applications of extremum graphs.

2.1 Extremum graphs

The notion of extremum graphs was first introduced in the work of Correa et. al. [2]. It was introduced as a means of visualizing high dimensional data in two dimensions, while not losing out on the topology and geometric properties of the original data. They also introduce a serial algorithm to simplify extremum graphs. It works by computing the gradient flow graph for the scalar field under study, then simultaneously performs persistence simplification and identification of gradient paths from saddles to extrema. These extremum graphs are augmented with geometric information such as volume under the descending manifold of its adjacent extrema. This augmented extremum graphs is termed as topological spines.

2.2 Morse-Smale complex

Work by Shivashankar et al. [5, 6] discuss about parallel computation for MS complex. This algorithm also works on a similar idea as in [2] i.e. to compute the gradient flow graph to identify critical points and gradient paths, but shows how to parallelize the inherently serial problem of identifying gradient paths for a scalar field. They also show how to perform external memory computations i.e. computing MS complex for large dataset which do not completely fit in memory. This is very much required as the sizes of scientific datasets keeps getting larger with the advent of increasingly sophisticated imaging devices.

2.3 Applications of extremum graphs

Various applications of extremum graphs have been described in literature since their inception. We briefly describe about two works, [7, 4].

Firstly, the work by Thomas el. al. [7] involves using augmented extremum graphs to detect symmetry in scalar fields. It utilizes the Morse decomposition of a scalar field into descending manifolds of extrema, which partitions the scalar field into Morse cells. Then a subset of these cells are selected based on some parameters. These seeds are joined with their adjacent qualifying seeds to form super-seeds. These super-seeds are again joined with their adjacent seeds until no cells are left out. To extend existing seeds, all the adjacent cells of all seeds are compared against each other using a similarity measure. The best matching cells are accepted as the extension of their adjacent seeds. These disjoint collections of cells are the symmetric regions that one would aim to find in the scalar field.

Another work utilizing extremum graphs is by Narayanan et. al. [4]. It shows how extremum graphs can be used to study time-varying data. It first describes a distance metric between two extremum graphs. This metric is then used to compute the distances between extremum graphs computed for each time step. As extremum graphs capture extrema-related features, the distance metric thus captures the evolution of various extrema-related events in the time varying data. Therefore, one can utilize this information to identify key events and features in the data.

Chapter 3

Problem Description

This chapter states the problem definition and an overview regarding ways to solve the problem. It also presents a summary of the major contributions.

3.1 Problem Statement

To efficiently compute the extremum graph for piece-wise linear scalar functions, utilizing GPU parallelism and also allow for external memory computations for scalar functions with large domain sizes.

The key challenge occurs in compiling partial gradient paths across multiple partitions of the scalar field for an external memory computation case. Partial gradient paths must be correctly identified and concatenated accordingly to finally report the set of gradient paths as if there was no partitioning of the scalar field domain during the computation.

3.2 Solution Overview

Computation of extremum graphs is done for piece-wise linear scalar functions i.e. on datasets where the underlying scalar field is sampled across a uniform grid of points. The computation pipeline is divided into two stages,

1. Point Classification

2. Gradient Path Tracing

In the first step of point classification, we perform two tasks for each grid point in the dataset. The first task is classifying a grid point as regular/critical. It is done by testing the topology of the point's neighborhood in the grid. If a point is classified as regular, the second

task is to compute the gradient vector for it. If a point is classified as critical, the second task is to identify its Morse-index. For a scalar field $f : \mathbb{R}^n \rightarrow \mathbb{R}$, we are interested in identifying critical points of Morse-index n and $(n-1)$ which are also called maxima and $(n-1)$-saddles respectively. Here we utilize GPU parallelism where each grid point is processed by a separate logical GPU thread. Specifically speaking, we perform this on CUDA-enabled NVidia GPUs where a single kernel invocation (In simple terms, a kernel is a GPU task sent from the CPU) is used to spawn enough logical threads on the GPU so that there is a one-to-one mapping between grid points and the GPU threads. The GPU is responsible for the scheduling of threads and we obtain computation results of step 1 after the kernel terminates.

The second step involves tracing gradient paths from $(n-1)$-saddles towards maxima. We first identify representative points among each of the upper link connected components of $(n-1)$-saddles. These representative points are simply the highest function valued point among all the points of the connected component it belongs to. For each such representative point of a $(n-1)$-saddle, we iteratively traverse gradient vectors until we encounter a maximum. The sequence of points discovered by this iterative gradient vector traversal form the gradient paths connecting a $(n-1)$-saddle to one-or-more maxima.

For external memory computations where the dataset does not fit into GPU memory. datasets are divided into contiguous blocks. These blocks are created by slicing the dataset across the last dimension in its domain. This results in a linear arrangement of blocks where each block is adjacent to two other blocks, except for the first and last which are adjacent to exactly one block. We perform steps 1 and 2 as described above for each block independently. The blocks are processed in increasing order of last dimension value, the same dimension across which the dataset was sliced on. This greatly simplifies gradient path tracing as the number of boundaries between blocks is linear to the number of blocks themselves. Thus the number of boundary crossing required for path tracing is expected to be less when compared to a scenario where the dataset is sliced across all n dimensions of its domain.

3.3 Major Contributions

The novel contributions of our work can can be summarized as:

- Identifying critical points by testing upper link topology on GPU, thus leveraging strong scaling provided by them.
- External memory computation of extremum graphs.

Chapter 4

Background and Definitions

Given a scalar field $f : \mathbb{R}^n \rightarrow \mathbb{R}$, we say a point $x \in \mathbb{R}^n$ is *critical* iff $\nabla f(x) = 0$ and *regular* otherwise.

$$\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n})$$

The set of critical points of f, can be further partitioned based on their *Morse-index.* The *Morse-index* of a critical point x, is the number of negative eigenvalues present in the Hessian matrix H_f evaluated at x, where

$$H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}_{n \times n}$$

Thus, critical points can have an index from $\{0, 1, \ldots, n\}$. Critical points with index 0 are the minima of f and critical points with index n are the maxima of f. Other critical points with index k are called k-saddles. Hence, f has $(n-1)$ types of saddles.

Another class of scalar functions exist: Piece-wise Linear functions. The domain for such functions is restricted to vertices on a uniform n-dimensional grid. Thus, we do not have the complete definition of the scalar field and must perform interpolation across a tessellated mesh to retrieve the function value at any point not on the grid. Scientific datasets are examples of PL functions and we will shortly discuss how to perform tessellation.

4.1 Morse-Smale Complex

The *Morse-Smale Complex* for a scalar field f is a directed graph where the vertex set is the set of all its critical points and the edge set being gradient paths connecting critical points differing in index by 1.

At each regular point, we can define a gradient vector as the direction along which increase in function value is the greatest. A *gradient path* between two critical points p, q differing in index by 1, is defined as a sequence of points where for each adjacent pair of points (r, s), s is the terminal of the gradient vector at v.

As an example for a 2D scalar field, its Morse-Smale complex would consist of minima, 1-saddles and maxima, with gradient paths connecting maxima and 1-saddles, along with gradient paths connecting 1-saddles and minima.

Computing Morse-Smale complex for scalar fields with high dimensionality becomes computationally difficult as the number of critical points and gradient paths grow exponentially. Additionally, these inter-saddle gradient paths cause severe occlusion problems when the goal is to visualize extrema-related features.

4.2 Extremum Graphs

Extremum graphs are a subset of the Morse-Smale complex. Any scalar field $f : \mathbb{R}^n \to \mathbb{R}$ has two extremum graphs,

1. Maximal extremum graph: Collection of maxima and $(n-1)$-saddles, along with their connecting gradient paths.

2. Minimal extremum graph: Collection of minima and 1-saddles, along with their connecting gradient paths.

By definition of critical points, maxima of scalar field $-f$ would be the minima of f. Similarly $(n-1)$-saddles of $-f$ would be the 1-saddles of f, and thus an algorithm to compute Maxima extremum graphs can be used to compute the Minimal extremum graph. Due to their characteristics, the collection of both maxima and minima is referred to as *extrema* in literature. Thus the name of **extremum graphs** which refers to both Maximal/Minimal extremum graphs. For brevity of naming, we shall refer to maxima as extrema, $(n-1)$-saddles as saddles and Maximal extremum graphs as extremum graphs throughout our work.

By definition (discussed shortly), the number of extrema a saddle is connected to is at least two in any extremum graph, but the number of saddles shared by a pair of adjacent extrema

is arbitrary. This could lead to a pair of extrema sharing a large number of saddles, which could lead to occlusion problems in visualization tasks. To rectify this, we define the notion of a *edge-bundled extremum graph*. This is a subset of extremum graphs where, for each non-empty intersection of adjacent saddles for a pair of extrema, exactly one saddle is chosen as a representative for the entire set and the rest discarded. Our implementation selects the saddle with highest function value.

Chapter 5

Computation of Extremum Graphs

In this section, we describe how we compute extremum graphs. All datasets are the result of sampling a scalar field at points on a uniform grid i.e. piece-wise linear functions. Thus we must work with the function values at the grid points. The computation can be broken down into two major steps:

1. Critical Point Identification: In this step, all grid points are classified as *critical/regular*. If a point is classified as *critical*, we also note if it is a maximum or saddle.

2. Gradient Path Tracing: All identified saddles are collected and path tracing begins at these points. The terminal of each path is a maximum.

5.1 Point Classification

As a consequence of working with scalar fields sampled on a uniform grid, we cannot use the Hessian Matrix to classify a grid point as critical/regular. Thus we resort to infer this classification by processing the neighborhood of the point. The neighborhood is also referred to as *Link* in literature, as it represents the set of points a given grid point is linked/adjacent to.

Let $\mathcal{D}_n$ be the graph representation of the uniform n-dimensional grid, where the vertices are the grid points and edges are from the point adjacency in the grid. The edge set of $\mathcal{D}_n$, $E(\mathcal{D}_n)$ will be formally defined in section 5.2, and

$$V(\mathcal{D}_n) = \{ \ v \mid v \in \underbrace{\mathbb{Z} \times \mathbb{Z} \times \ldots \times \mathbb{Z}}_{n} \ \}$$

.

We define the link of a point p as:

$$Lk(p) = \{ \ v \mid v \in V(\mathcal{D}_n) \ \wedge \ (p, v) \in E(\mathcal{D}_n) \ \}$$

Further we define two notions based on the Link, *upper link*(Lk^+) and *lower link*(Lk_-) of a point p:

$$Lk^+(p) = \{ \ v \mid v \in Lk(p) \ \wedge \ f(v) \geq f(p) \ \}$$

$$Lk_-(p) = \{ \ v \mid v \in Lk(p) \ \wedge \ f(v) < f(p) \ \}$$

We also define $C(S)$ as the number of connected components in the sub-graph induced on the set of grid points $S \subseteq V(\mathcal{D}_n)$.

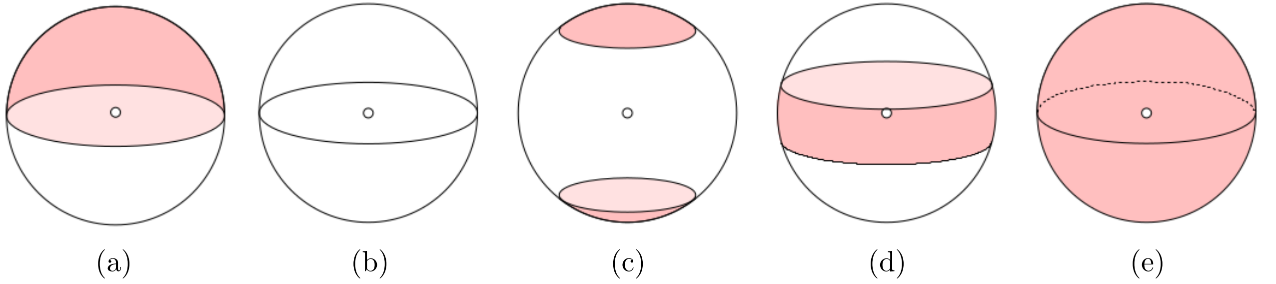


Figure 5.1: Various classification of a point in a 3D grid.
Each sphere represents the link of the point in a 3D grid at its center. Upper link is highlighted in pink, lower link in white. Link of a (a) regular point, (b) maximum, (c), 2-saddle, (d) 1-saddle and (e) minimum. Notice the empty upper link for the maximum and 2 connected upper link components for the 2-saddle. Picture credits [3].

Arriving at the problem of point classification, a point p is regular if and only if $C(Lk^+(p)) = C(Lk_-(p)) = 1$. Point p is a maximum iff $Lk^+(p) = \phi$. This definition is intuitive, as all points adjacent to a maximum must be lower in function value compared to the maximum itself. A similar classification for minima follows where the lower link must be empty.

Classification for saddles requires some insight. For this, we define the descending manifold for a maximum m as the collection of points, whose gradient paths terminate at m. The saddles can now be identified as the highest function valued point among the intersection of descending manifolds for two or more extrema. Intuitively, these are points where their upper link is not a single connected component, as it represents the intersection of two or more disjoint collections of points i.e. descending manifolds. Thus, for a grid point p classified as a saddle, $C(Lk^+(p)) \geq 2$.

Examples of the link topology for various point classifications in a 3D grid are shown in Figure 5.1. Note that in a 3D grid, orientation and position of the upper link components is

dictated by the function value of its constituent points. Thus the upper link components may not be at the top of the link's diagram.

5.2 Grid Tessellation

For an n-dimensional uniform grid, the size of the neighborhood of a point is at most $2n$ (Exactly $2n$ for points not on the boundary of the grid). Now consider the number of points contained in all the n-hypercubes in which the same point lies in. The total is $3^n - 1$ by a simple counting argument. We can make a trivial observation that, the size of the neighbourhood of a point becomes increasingly sparse for larger values of n. This makes point classification based on its neighborhood inaccurate for higher dimensional grids.

Thus we resort to a standard method of tessellating the uniform grid into irregular n-simplices. Here a simplex (simplices, plural) is the generalization of tetrahedron in 3D to higher dimensions. It represents the simplest polytope which can be created in a specified dimension. Thus, a point is a 0-simplex, a line segment is a 1-simplex, a triangle is a 2-simplex, a tetrahedron is a 3-simplex, and so on.

The tessellation gives rise to additional edges in the uniform grid. Adjacency between any two grid points in this tessellated grid can be tested using algorithm 1. The algorithm essentially dictates that two distinct points are adjacent if and only if the difference vector between the two points must entirely constitute of non-negative or non-positive values and magnitude of non-zero values must be exactly 1.

Algorithm 1: *GridPointAdjacency*

Input: p, q : Points to test adjacency for
Result: $True$ if p, q are adjacent and $False$ otherwise

$U \leftarrow \{\ (p_i - q_i) \mid i \in \{1, 2, \cdots, n\}\ \}$ *p_i represents the ith component of vector p*

if $U = \{0, 1\}$ *or* $U = \{0, -1\}$ **then**
  return $True$
end
else
  return $False$
end

Using algorithm 1, we can define $E(\mathcal{D}_n)$ for a tessellated grid as

$$E(\mathcal{D}_n) = \{ \ (u, v) \mid u, v \in V(\mathcal{D}_n) \ \wedge \ GridPointAdjacency(u, v) \ \}$$

Thus, size of the Link for a grid point is twice the number of possible difference vectors which are not zero vectors i.e. $2 \times (2^n - 1) = 2^{n+1} - 2$. Tessellation of a few 2D and 3D grids is shown in Figure 5.2.

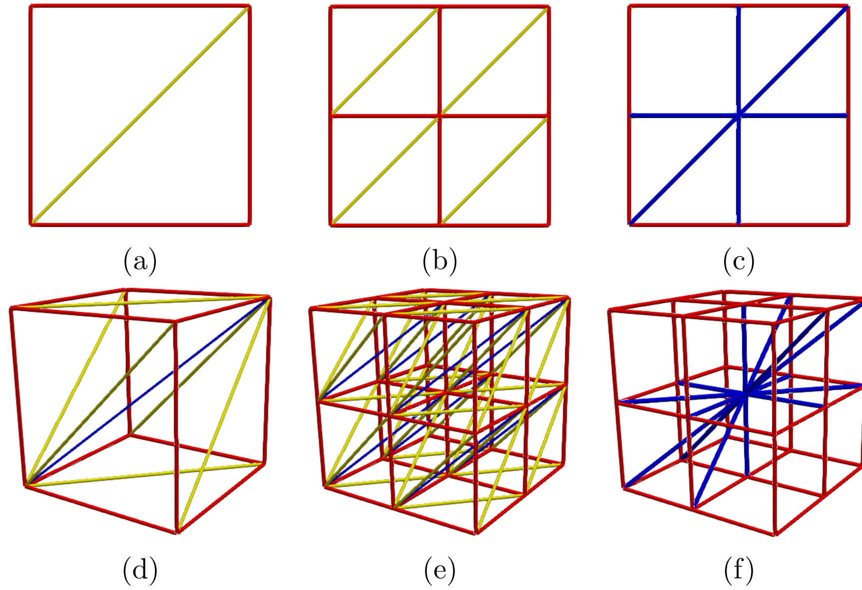


Figure 5.2: Grid tessellation in 2D and 3D grids.
Tessellation of (a) 2x2 grid, (b), 3x3 grid, (d) 2x2x2 grid and (e) 3x3x3 grid. Neighborhood graph (highlighted in blue) of the central point in a (c) 3x3 grid and (f) 3x3x3 grid. Notice there are 6 and 14 neighbors for the central points in (c) and (f) respectively.

5.3 Point classification for boundary points

In practice, we work with piece-wise linear versions g of any continuous scalar field f where the domain $\mathcal{D}'_n$ is a finite sized uniform n-dimensional grid. This is a subset of $\mathcal{D}_n$ we defined before. As we work with a finite sized sub-grid, we define a notion of boundary grid points in the sub-grid which are adjacent to points not belonging to the sub-grid $\mathcal{D}'_n$.

In a tessellated grid, the *Link* for boundary points is smaller than for non-boundary points. If upper link topology testing is used for critical point identification for these boundary points, it is possible to falsely classify an $(n-1)$-saddle as a regular point if one of its upper link could completely lie outside the sub-grid $\mathcal{D}'_n$. To remedy this, we define $g(x) = \infty, \ \forall \ x \in$

$V(\mathcal{D}_n) \backslash V(\mathcal{D}'_n)$. Now we can include these exterior grid points in the neighborhood of boundary points and correctly classify them.

5.4 Gradient Path Tracing

We begin gradient path tracing from the saddles and continue till all paths approach an extremum. This is done from saddles to extrema and not in the reverse direction of gradient, since an extremum can be connected to an arbitrary number of saddles. But a saddle has an outgoing gradient path through each of its upper link component. This follows from the fact that saddles are the first intersection point among the descending manifolds of two or more extrema. Thus, the gradient path from the saddle to all these extrema must pass through each of the upper link components. An example in a 3D grid with 2-saddle to maximum path tracing is depicted in Figure 5.3.

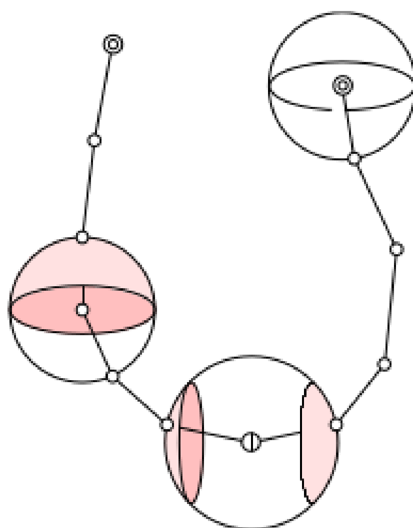


Figure 5.3: Gradient path tracing in a 3D grid
. A 2-saddle with its link is depicted at the bottom and two gradient paths emerging from each of its upper link components in pink. A regular point with its link is depicted midway on the left path and a maximum with its link on the terminal of the right path. Picture credits [3].

5.5 Persistence Simplification

Persistence simplification is the process of canceling adjacent pairs of critical points based on their persistence value. Here, canceling refers to removing the pair if the criterion is satisfied and persistence is usually the difference in function value of the point pair. The criterion is usually a threshold value, where all adjacent critical point pairs are removed if their persistence

falls below the threshold. This allows one to remove noisy features for e.g. low persistence extrema (small bumps in an overall bigger feature) and enable clearer visualization of important features without clutter.

In the simplification of extremum graphs, a lot of gradient paths are left back by the standard persistence simplification algorithm. This occurs, since many adjacent critical point pairs would usually be canceled in a Morse-Smale complex. But in the extremum graph, there are no $(n-2)$-saddles, $(n-3)$-saddles, etc. to remove many occluding and unnecessary features connected to saddles. Thus we use a modified Simplification algorithm presented in the work of [2]. It allows one to remove long and non-intuitive saddle - extrema gradient paths from the extremum graph, as many such paths are expected to be a result of non-cancellation due to absence of other non $(n-1)$-saddles.

Chapter 6

Design and Implementation

Computing extremum graphs for large datasets requires a large amount of parallelism to reduce the execution time. Thus we have leveraged the CUDA architecture of NVidia GPUs to achieve massive parallelism in our library. CUDA allows a subset of the standard C++ code to be compiled as GPU code. This allows integration into existing C++ code, while enjoying the same features and guarantees the language standards provides.

We assume the required data structures will fit in main memory along with the dataset itself. To support external memory computation i.e. for datasets which do not fit in the GPU memory, we perform division of the dataset into contiguous blocks. For each block we perform point classification and then gradient path tracing. All results are collated after the final block is processed.

6.1 Simulation of Simplicity

Processing flat regions in a dataset, i.e. a subset of grid points forming a single connected component whose function value is the same, must be done carefully. Consider the following scenario where these points are classified.

By definition of $Lk^{+}(\cdot)$, all points in this flat region will be marked as an extrema. This should never happen as two extrema can never be immediately adjacent to each other. There must always exist a saddle in-between the intersection of adjacent extrema's descending manifolds. To resolve this dilemma, we resort to using an overloaded point comparison operator as shown in algorithm 2. Essentially, the tie breaking on comparison of a pair of points with equal function value is done by comparison on the points indices within the dataset grid. With the

use of the overloaded point comparison, exactly one point in a flat region will be identified as an extrema.

Algorithm 2: *SoSPointCompare*

Input: f : Dataset, p, q : Points to compare

$f : V(\mathcal{D}_n) \rightarrow \mathbb{R}$, $p, q \in V(\mathcal{D}_n)$

Result: Overloaded result for $p < q$

```
if f(p) = f(q) then
  | return index(p) < index(q)
end
else
  | return f(p) < f(q)
end
```

$index(\cdot)$ is a *grid point* to *numeric index* bijective map i.e. *index*: $V(\mathcal{D}_n) \rightarrow [0, N) \cap \mathbb{Z}$, where $N = |V(\mathcal{D}_n)|$ is the number of points in the dataset.

6.2 Point Classification

For the first step, a CUDA thread is spawned for each point of the dataset. Due to the underlying hardware design of CUDA, threads do not execute asynchronously as conventional threads do on a general purpose CPU. Threads execute in groups of 32, in a lockstep manner and hence if the execution paths across all threads is nearly identical, execution time for the group of 32 threads will be same as a single thread's execution time. Essentially granting a speedup of nearly 32 for the work done by the thread group. On the contrary, a delay/hazard within a single thread of a group will halt execution of the entire group.

Thus to find the connected components of the upper link, a conventional Breadth First Search traversal algorithm is inefficient in practice as it creates significant divergence in the execution paths of threads within a same group. We resort to using a Union-Find based approach to find connected components.

Additionally as an optimization step, each point also computes its highest function valued neighbor. This step creates negligible execution divergence, as a majority of points have the same neighborhood size of $2^{n+1} - 2$ (boundary points will have a smaller neighborhood size). This simplifies gradient path tracing as described in the next section. Overview of the algorithm

executed in CUDA is shown in algorithm 3.

As the code is written in C++, our library also allows machines without GPUs supporting CUDA architecture to perform the point classification step on the CPU. This allows extremum graph computation on any machine regardless of it being fitted with a GPU.

Algorithm 3: CUDA Kernel Pseudocode

Input: f : Dataset block, p : Point

Result: $pclass$: Point index by classification, p^* : Highest function valued neighbour point

$uf_data \leftarrow$ ***ConnectedComponentIdentification***(f, p) *Ref algorithm 4*

return ***PointClassification***(f, p, uf_data) *Ref algorithm 5*

.

Algorithm 4: *ConnectedComponentIdentification*

Input: f : Dataset block, p : Point

Result: uf_data : Union find data strcuture, containing connected component information for the *link* of p

$uf_data \leftarrow$ *Init Union-Find structure.*

Part 1: Union-Find Component Identification

for $i = 0$ *to* $2^{n+1} - 2$ **do**
- $p_i \leftarrow i$'th neighbour of p.
- $p_i lower \leftarrow p_i < p$
- **for** $j = i + 1$ *to* $2^{n+1} - 2$ **do**
  - $p_j \leftarrow j$'th neighbour of p.
  - $p_j lower \leftarrow p_j < p$
  - **if** $GridPointAdjacency(p_i, p_j) \wedge (p_i lower = p_j lower)$ **then**
    - *p_i, p_j are adjacent and both belong to the same upper or lower link component*
    - Union(uf_data, i, j)
  - **end**
- **end**

end

Algorithm 5: *PointClassification*

Input: f : Dataset block, p : Point, uf_data : Union find data structure containing information on connected components of the *link* of p.

Result: $pclass$: Point index by classification, p^* : Highest function valued neighbour point

$ucount \leftarrow 0$ — *Number of upper link connected components*

$p^* \leftarrow p$ — *Highest function valued neighbor i.e. gradient vector terminal*

$pclass \leftarrow -1$ — *Assume regular pt index is (-1)*

Part 2: Count components by checking set representatives

```
for i = 0 to 2^(n+1) - 2 do
    if i = Find(uf_data, i) then
        if (p < p_i) then
            ucount += 1
            if p* < p_i then
                p* <- p_i                    // Update gradient vector terminal
            end
        end
    end
end
```

Part 3: Point classification

```
if ucount = 0 then
    pclass <- n                              // Extrema
end
else if ucount >= 2 then
    pclass <- (n - 1)                        // Saddle
end

return (pclass, p*)
```

6.3 Gradient Path Tracing

This step is completed on the CPU and not on the GPU. Again, the reasoning is divergence in execution of a thread group. Each path traced can be of arbitrary length, hence each thread

will take varying amounts of time to execute. Path tracing on the CPU was found to be faster and the reasoning is compounded by the trend of increasing core counts in CPUs.

For each saddle, BFS is performed again on the CPU, while noting highest function valued points among each upper link components (not across entire neighborhood). For each such point among the upper links, algorithm 6 is called with itself as the starting point for the gradient path. *phi_arr* is constructed from the output of point classification step.

Parallelism is achieved using the C++11 thread library. The collection of saddles are equally divided among all threads. The same is done with the collection of pending path tracings in external memory computation runs. As a majority of $(n-1)$-saddle are not multi-saddles, their upper link has exactly two connected components. Thus the number of paths traced by all the threads is roughly equal to one another.

Algorithm 6: *SingleGradientPathTracing*

Input: p : Starting point,
  phi_arr : Array of gradient vectors p^* from algorithm 3,
  $\mathcal{D}'_n$: Sub-grid of the block being processed

Result: $path_vector$: Gradient path points

$path_vector \leftarrow \phi$
$partial_termination \leftarrow False$
$q \leftarrow p$

while *true* **do**
  if $q \notin V(\mathcal{D}'_n)$ **then**
    $partial_termination \leftarrow True$
    break — *Path tracing leaves current block. Save this for future processing*
  end
  $path_vector \leftarrow path_vector \parallel q$ — *Append*
  if $phi_arr[index(q)] = q$ **then**
    break — *Gradient vectors at extrema point to themselves.*
  end
  $q \leftarrow phi_arr[index(q)]$
end
return $(path_vector,\ partial_termination)$

For internal memory computations, $\mathcal{D}'_n = \mathcal{D}_n$ as computation happens over the entire dataset as a single block.

6.4 External Memory Algorithm

To enable division of datasets into sizable blocks which fit in GPU memory, we resort to slicing the dataset across its last dimension. For eg, a 3D dataset would be sliced along its Z-axis. A visual example of this for a 3D volume is shown in Figure 6.1. This enables two benefits,

- Individual blocks are contiguous in the dataset.
- Trivial indexing of points within a block w.r.t. to the entire grid.

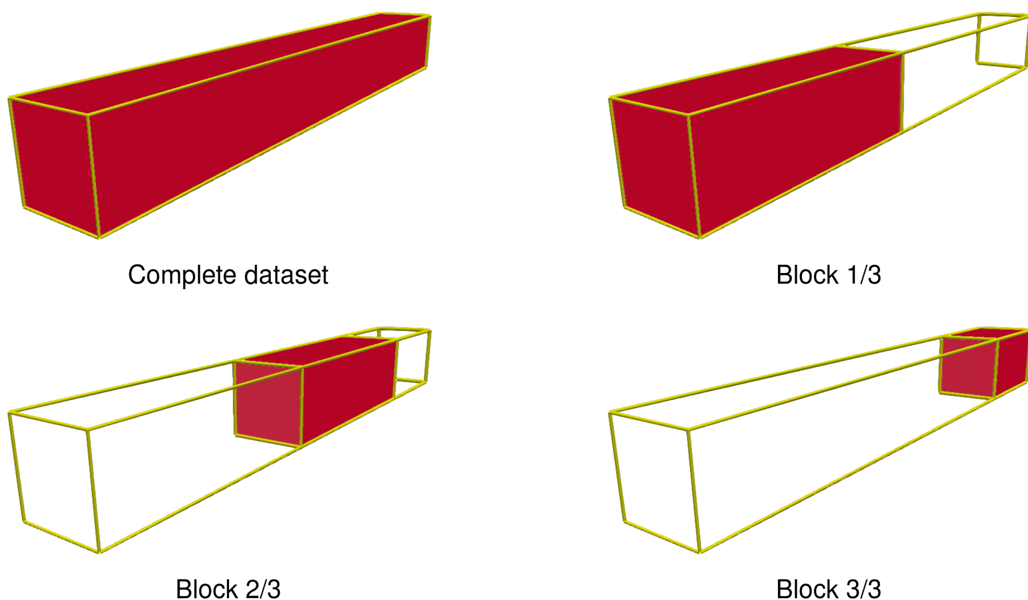


Figure 6.1: External memory computation: Block division of a 3D dataset. Example of block division of a cuboid shaped 3D dataset into 3 blocks. Notice block 1 and 2 are of same size, while block 3 is smaller.

The expected size of each block is based on a linear regression model derived from statistics of experiments on datasets which fit in GPU memory. The model describes a linear relationship between the number of points in the dataset vs the amount of GPU memory used.

But for points on the boundary of a block but not on the boundary within the entire dataset, point classification maybe wrong due to their missing neighborhood. To solve this issue, point classification occurs for points not on a block's boundary, unless they lie on the dataset's boundary. These two different n-rectangles for a block are referred to as bounding and

compute rectangles respectively in the library implementation i.e. bounding rectangle simply represents the collection of points in the block and compute rectangle represents the subset of points for which point classification must happen. For internal memory computations, both bounding and compute rectangles are the same.

Chapter 7

Results

Experiments were run on a high end workstation on datasets of varying sizes from 64 x 64 x 64 to 2048 x 2048 x 2612. These datasets were obtained from the Klacansky scivis repository [1]. Experiments were run on a workstation with a Intel(R) Xeon(R) Gold 6230 @ 2.10GHz, 20 Core/40 Thread CPU, 384GB main memory and a GeForce RTX 2080 Ti GPU with 11GB GDDR6 ECC memory. Runtime averages were taken on 5 samples after dropping the best and worst timings out of 7 runs. GPU temperatures were also kept within optimal operating ranges of $< 85^o$C.

7.1 Internal Memory Computations

All datasets which completely fit in memory were tested on for this experiment. These datasets did not require division into contiguous blocks, thus the point classification was complete in a single GPU run. 28 datasets were tested and results for the 7 largest ones are visualized in Figure 7.1. Details of the plot are show in Table 7.1.

Time taken for the gradient path tracing step is proportional to the number of 2-saddles identified in the datasets, whereas there is a linear relationship between the time for point classification to the number of points in the datasets. This is expected as classification for a point is independent of the process for other points and the number of points as well. To summarize, all datasets which did fit in GPU memory completed execution under a minute.

Dataset	Size	Computation Time		
		Point Classification	Gradient Path Tracing	Total
Magnetic Reconnection	512x512x512	2.12	4.46	9.44
Marmoset Neurons	1024x1024x314	5.37	7.85	18.88
Pawpawsaurus	958x646x1088	11.07	12.21	31.14
Spathorhynchus	1024x1024x750	10.99	6.48	23.58
Kingsnake	1024x1024x795	11.99	8.21	27.10
Chameleon	1024x1024x1080	16.23	7.24	31.11
Beechnut	1024x1024x1546	18.40	23.30	57.92

Table 7.1: Timings for internal memory computation
(All times in seconds)

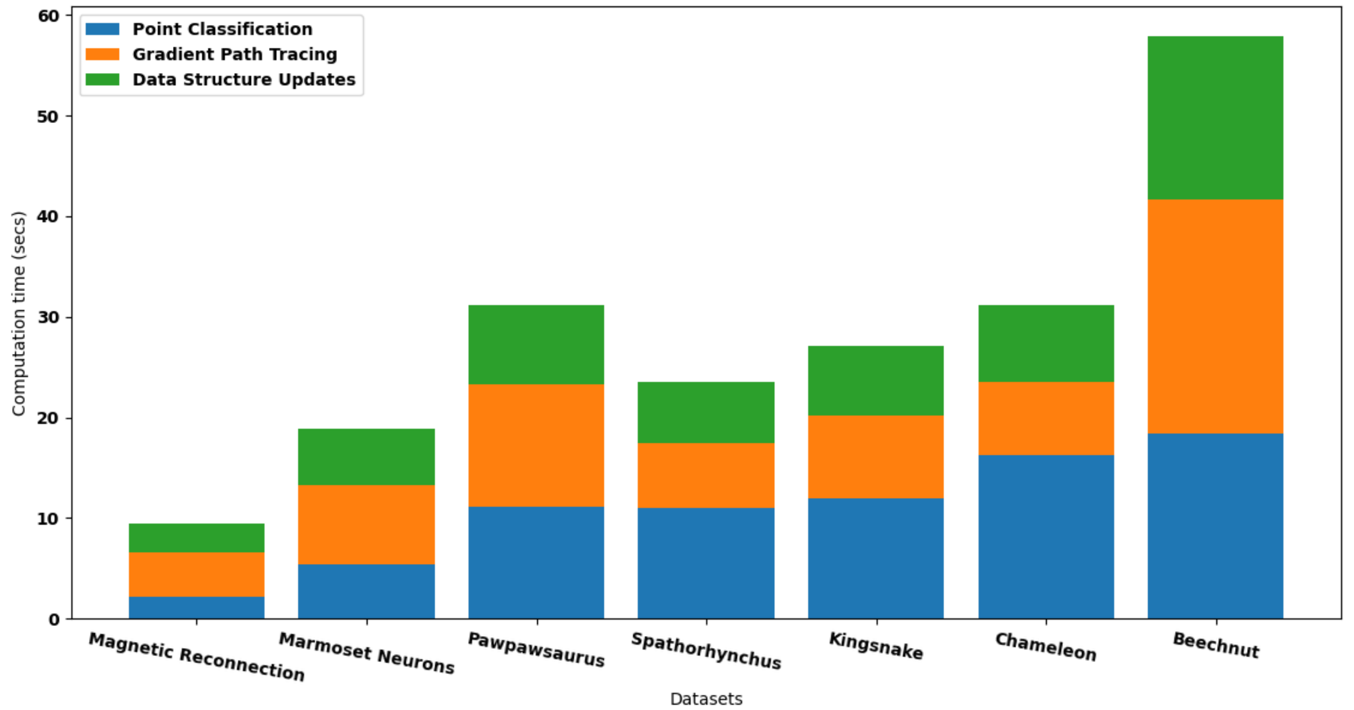


Figure 7.1: Internal memory computation experiment results

7.2 External Memory Computations

Large datasets which do not fit in GPU memory but fit in main memory are scarce to find, as the sizes of datasets increases exponentially. Thus the number of datasets available which fit this criterion are exactly 3. The computation times are depicted in Figure 7.2 and details of

Dataset	Size	Computation Time		
		Point Classification	Gradient Path Tracing	Total
Woodbranch	2048x2048x2048	141.78	380.57	837.63
3D Neurons	2048x2048x2384	130.83	650.78	1233.21
Pig Heart	2048x2048x2612	142.85	237.89	574.41

Table 7.2: Timings for external memory computation
(All times in seconds)

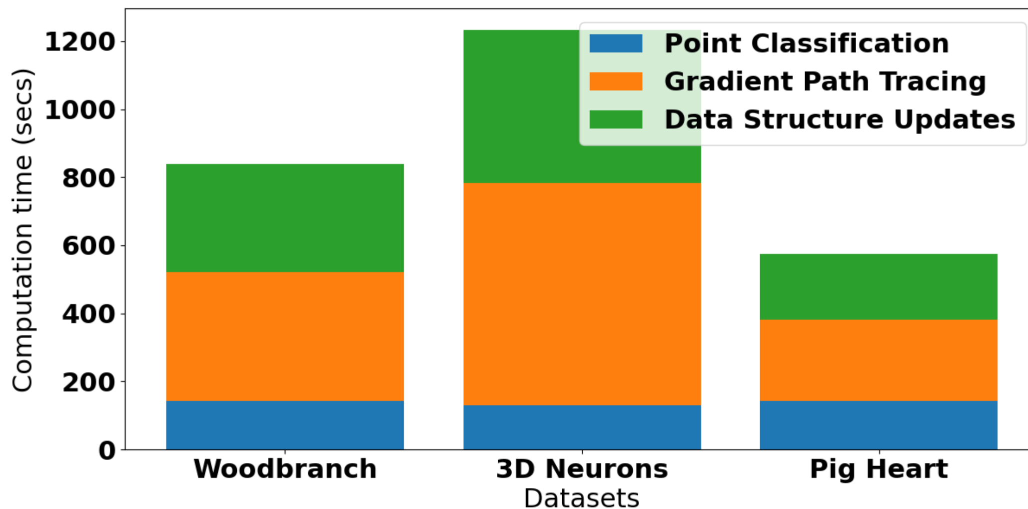


Figure 7.2: External memory computation experiment results

the plot are show in Table 7.2.

Time taken by the Point classification step is nearly identical across all datasets as the number of points in each of them is very close to one another. This also shows that, our CUDA code is well optimized and does not suffer from divergent execution among thread groups. There is a significant increase in the gradient path tracing time for 3D Neurons. This is due to an increased number of path tracings being partially terminated and waiting for the point classification of the next contiguous block.

Number of maxima and $(n-1)$-saddles identified for all datasets from both internal and external memory algorithms is shown in Figure 7.3.

Dataset	Size	Num Extrema + Saddles
Magnetic Reconnection	512x512x512	31195066
Marmoset Neurons	1024x1024x314	56101022
Pawpawsaurus	958x646x1088	43713113
Spathorhynchus	1024x1024x750	47569170
Kingsnake	1024x1024x795	32483997
Chameleon	1024x1024x1080	49325513
Beechnut	1024x1024x1546	159472969
Woodbranch	2048x2048x2048	1520839571
3D Neurons	2048x2048x2384	1913765935
Pig Heart	2048x2048x2612	906730898

Table 7.3: Number of critical points identified for various datasets

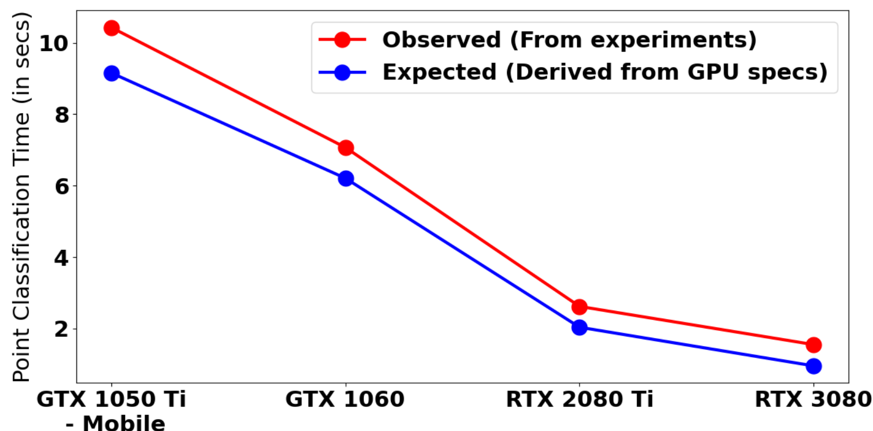


Figure 7.3: Strong Scaling results for point classification

7.3 Strong Scaling

Point classification for the same dataset (Zeiss, Size: 680 x 680 x 680) was performed on different workstations fitted with distinct NVidia GPUs. The average point classification time was computed for runs on each workstation.

The ideal performance for each GPU is computed as the number of available CUDA cores multiplied by its base clock frequency. This gives us a rough estimate to the total number of instructions executed by the GPU per second. Scaling calculated from experimental observations nearly matches the ideal performance across all GPUs, but the general downward trend is observed. This result is depicted in Figure 7.3.

Chapter 8

Conclusions & Future Work

8.1 Conclusions

A library to directly compute extremum graphs was successfully created. Its main features are summarized below:

- The library scales well with increasing number of CUDA cores i.e. strong scaling holds.
- Supports external memory computation of datasets too large to fit in GPU memory.
- Dimension generic and supports datasets with dimensionality below 32.
- Exports a fast algorithm which allows for user interactive simplification of extremum graphs.

8.2 Future Work

Currently, our library supports utilization of exactly one GPU, but our goal is to leverage multi-GPU setups for point classification. Another possible extension is to support distributed computation, where multiple workstations work on blocks of the same dataset.

Bibliography

[1] Open scientific visualization datasets. *https://klacansky.com/open-scivis-datasets/*, (accessed: 12-June.2021). 23

[2] Carlos Correa, Peter Lindstrom, and Peer-Timo Bremer. Topological spines: A structure-preserving visual representation of scalar fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1842–1851, 2011. 3, 15

[3] Herbert Edelsbrunner, John Harer, Vijay Natarajan, and Valerio Pascucci. Morse-smale complexes for piecewise linear 3-manifolds. In *Proceedings of the nineteenth annual symposium on Computational geometry*, pages 361–370, 2003. 11, 14

[4] Vidya Narayanan, Dilip Mathew Thomas, and Vijay Natarajan. Distance between extremum graphs. In *2015 IEEE Pacific Visualization Symposium (PacificVis)*, pages 263–270. IEEE, 2015. 4

[5] Nithin Shivashankar, M Senthilnathan, and Vijay Natarajan. Parallel computation of 2d morse-smale complexes. *IEEE Transactions on Visualization and Computer Graphics*, 18 (10):1757–1770, 2011. 3

[6] Nithin Shivashankar, M Senthilnathan, and Vijay Natarajan. Parallel computation of 3d morse-smale complexes. *Computer Graphics Forum, 31(3pt1):965-974*, 2012. 3

[7] Dilip Mathew Thomas and Vijay Natarajan. Detecting symmetry in scalar fields using augmented extremum graphs. *IEEE transactions on visualization and computer graphics*, 19(12):2663–2672, 2013. 2, 4