Optimization in Extremum Graph Computation

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

BY

Dhurjati Prasad Das

INDIAN INSTITUTE OF SCIENCE
भारतीय विज्ञान संस्थान

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

July, 2023

Declaration of Originality

I, **Dhurjati Prasad Das**, with SR No. **04-04-00-10-51-21-1-19572** hereby declare that the material presented in the thesis titled

Optimization In Extremum Graph Computation

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2021-2023**.
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

DEDICATED TO

The Aspiring Researcher

who might enjoy visualizing extremum graphs

Acknowledgements

I would like to thank Prof.Vijay Natarajan of the Visualization and Graphics Lab (VGL), under the Computer Science and Automation (CSA) Department, IISc Bangalore, without whom I would have struggled a lot. His systematic approach helped me to grasp the concepts quickly, and his attention-to-detail approach created curiosity within me, which helped me to view the problem from numerous perspectives.

The Predecessor, Abhijath Ande, who implemented an efficient parallel implementation of the Extremum graph known as tachyon, also helped me understand the nook and corner of the tachyon. He has been patiently hearing all the doubts and clearing them up in an elaborate way so that such concepts remain etched into my mind.

All the people in the VGL lab had been a great help and their constant support made my stay a really memorable and enjoyable one.

Abstract

Extremum Graphs are a subset of the Morse Smale (MS) Complex. It is a simpler representation possible of Morse decomposition of scalar fields. One of the benefits it provides is the preservation of topological and geometric properties. *Tachyon* [4] is a software library that provides an efficient implementation of a parallel algorithm for computing the extremum graph. Its implementation utilizes Graphic Processing Unit (GPU) as well as Central Processing Unit (CPU) in tandem to reduce time costs as much as possible. Detailed observations on the execution of the tachyon show that as we move from one dimension to another (i.e., 3D to 4D to ND), the computation time increases exponentially. To reduce the computation time, we employed optimizations in the tachyon and found significant improvement. A few methods worth mentioning are the conversion of adjacency matrix ($O(4^D)$) to edge list ($O(3^D)$); Reducing the number of calls to a function from $2^D + 3^D$ calls to 2^D calls, with the help of a small local lookup table; Utilizing shared memory (default: 48KB) of GPU, which is an on-chip memory, gave a tremendous improvement in run time in higher dimensions, because of earlier thrashing issues persisted in GPU operations.

Contents

List of Figures

Chapter 1

Introduction

The section shows the importance of GPU in the modern world (which in turn also reflects its importance in the topological world), due to which various optimization is performed on the earlier known algorithms. The motivation section talks about it followed by the project goal.

1.1 Motivation

With the advent of GPU, most algorithms that were a few decades ago computationally infeasible become feasible and are solved within microseconds, milliseconds, or seconds. To utilize maximum efficiency from GPU, we should schedule work in such a fashion that makes them independent of each other.

Even the trending topic in modern-day visualization like the topological analysis of scalar fields (or vector fields) uses GPU to achieve maximum efficiency and faster computation time. Numerous abstractions are being developed in the visualization field to study the topology of any given structure in a more meaningful way. Some of them which are already in use are Contour Tree, Reeb Graph, cancellation Tree, and MS Complex. Such structures help us to interpret the topology of a given structure that has a scalar value at specified points. Every structure has some pros and cons associated with it. And as the world is moving towards visualization and interpretation of high-dimensional data, we want to make a visualization technique that is able to keep up with such high-dimensional data.

Most of the mentioned structure fails to make the visuals understandable for the human eye. For instance, MS Complex [9], in a higher dimension (i.e., greater than 3) suffers occlusion problems as well as storage problems due to an exponential increase in the number of connecting arcs between critical points (maxima, minima, saddles) as well as between regular points. Structures such as contour trees hardly preserve the geometric information of the structure.

A noble structure that is simple and preserves most information was required, which gave rise to Extremum Graph. It only considers critical points (i.e., either maxima or minima) and a particular saddle type (i.e., either n-1 saddle or 1 saddle, where 'n' is the number of dimensions). Since most interesting phenomenon happens around extrema (either minima or maxima), it will be useful to visualize only those points. To aid their computation comes the GPU, without which computation time would have a shoot day, months, or even year. Earlier papers [5] have shown an increase in GPU cores will further enhance the algorithm and reduce the computation. Now, if we improve performance in simple low-end GPU systems, then it can be used by normal users (i.e., students, doctors) without the help of workstations and servers.

1.2 Project Goal

Tachyon [4] is a software library that provides an efficient implementation of a parallel algorithm for computing the extremum graph. We will try to improve the existing tachyon.

The project goal is structured in three phases:

Understanding the flow of the tachyon.
It will involve finding any structural change in the data structure or flow change (i.e., refactoring) in the existing tachyon.

Understanding GPU-related architectural bottlenecks that might be occurring.
Due to the limitation of resources provided by GPU, we might need to reconfigure storage areas of certain variables.

Modifications of existing tachyon that might help improve performance and testing it thoroughly on numerous datasets.

Chapter 2

Related Work

The section shows relevant papers on extremum graphs and the earlier method of computing extremum graphs.

To represent the gradient flow behavior of Morse functions, a new set of topological structures such as Morse Complex[6] and MS complex[9] were introduced. The extremum graph is a subset of the MS complex or, in other words, it can be considered as a 1-skeleton of the Morse complex. The term 'Extremum Graph' was first coined in the paper, Topological Spines [7]. The importance of the Extremum Graph lies in the fact that it preserves the topological as well as the geometric structure of a scalar field, while the topological spine is a covering of the extremum graph to make it more presentable. It essentially preserves the relative location of extrema and knowledge of their neighborhoods with respect to the gradient path connecting them. Numerous applications of extremum graphs can be seen, a few worth mentioning are segmentation in the 2D-3D scalar field, feature tracking in time-varying data, and clustering in high-dimensional data.

Earlier, only up to 3D MS complex computation was studied extensively, and even numerous methods [10] and techniques [9] have been involved to improve the performance of MS Complex computation. But due to an increase in scientific data, we need to make our algorithm scalable and as robust as we can. MS Complex in higher dimensions (> 3), requires tremendous memory space as well we need to keep track of a lot of arcs, and still it might suffer from occlusion problems. To eliminate such problems, Extremum Graph comes in handy.

We can first construct the MS Complex, then extract the Extremum Graph, since Extremum Graph is a subset of the MS Complex. But the construction of MS Complex in higher dimensions is daunting to deal with, so we need to construct Extremum Graph directly.

We are aware of the single library 'tachyon' [4], which has been released for public use. It computes the extremum graph using a couple of stages. The first stage is point classification, which helps to assign a label to the point, whether it is regular or critical (i.e., maxima or saddle). It involves efficient usage of GPU parallelism. The second stage is constructing arcs from saddles to the maxima and due to varied path lengths, it is sensible to use the CPU which can afford task-level parallelism. Hence, we could leverage the library and try to find any improvements that might aid in the performance of the existing tachyon.

Chapter 3

Problem Statement

The section elaborates on the problem statement, and how we have tackled the problem.

3.1 Problem Statement

Tachyon [4] is released for public use. It contains an efficient implementation of a parallel algorithm for computing Extremum Graph. We tend to enhance its performance by studying the various stages it goes through and making any modifications necessary to bring down the overall computation time.

Conceptually, the tachyon follows a three-stage approach:

- Point Classification.

  Tachyon implementation classifies a point into a critical point using algorithms that make the operation independent of other points and hence GPU can be utilized for faster computation. Analysis of the current stage shows there is room for a lot of improvement.

- Point Collection.

  All critical points are aggregated in the stage to be utilized for further stages. Analysis of the current stage shows there is little room for improvement.

- Gradient Path Tracing

  Critical points collected from the above stage are utilized to create an extremum graph. Here task level parallelism is involved which utilizes the CPU core of the system. Analysis of the current stage shows the workload imbalance is tremendous, and we may require advanced graph algorithms to reduce the time complexity of the current stage.

In short, analyze the tachyon, search for improvement, implement it, and report the findings.

3.2 Overview of Solution

Modification is done majorly in three phases:

- Conversion of adjacency matrix to edge list.

  In tachyon, we need to find the number of connected components among neighbors, so we used to store the potential edge matrix in the *neighbors*neighbors* matrix. As we increase dimensions, the matrix becomes sparser and sparser and we incur more traversal time as well as more storage space. To annihilate such problems we use an edge list, which has a size less than the matrix, as well as incurs less traversal time.

- Use of lookup table to reduce calls to a particular function

  The function takes as input the neighbor of the reference point and returns whether this point has a function value greater than the reference point or not (whether it is part of the upper link or not). The function is significant because it almost consumes 18-30% of the execution time in the point classification stage. Earlier every time either 'i' or 'j' of the edge list changes (endpoints of edges), we called the function. So, the number of times the function was called O(neighbors + potential edges). Now if we store only the return result of the function, we call the function O(neighbors) times.

- Utilizing Shared Memory of GPU to avoid thrashing

  In Tachyon, there is an adjacency matrix that is used by the CPU as well as GPU, so we stored the matrix in UNIFIED memory (i.e., accessible to both CPU and GPU). A quite interesting fact emerge when we found out there is a huge difference between kernel execution time and function1 (i.e., the first and foremost function executed by the kernel). We hypothesize that due to the aggressive pace of page replacement, thrashing is occurring at GPU.So, we shifted the adjacent matrix of the tachyon (or edge list of modified tachyon) to the Shared Memory of GPU, and finally found huge improvements in run-time at higher dimensions.

3.3 Contributions

- Conversion of adjacency matrix to edge list
- Use of a lookup table to reduce the number of calls to a particular function.
- Use of Shared memory of GPU to avoid the issue of thrashing.
- We have successfully reduced tachyon's execution time by 65%-75% at higher dimensions.

Chapter 4

Background

The prerequisites are elaborated here.

4.1 Extremum Graph

The term ”Extremum Graph” was coined in the paper by Carlos D.Correa [7]. To ease the understanding of the Extremum Graph, we will quickly define some preliminaries.

Given a smooth function $f : \mathbb{M} \to \mathbb{R}$ over a manifold $\mathbb{M}$ of dimension n, we say that a point $x \in \mathbb{M}$ is critical if and only if $\nabla f(x) = 0$, otherwise x is called regular point. A function f is a *Morse function* if all critical points have pairwise distinct values and non of them are degenerate i.e., the Hessian evaluated at the point is non-singular. For a critical point x of f, its Morse index is defined as the number of negative eigenvalues of its Hessian matrix evaluated at x.

An *integral line* is a maximal curve in $\mathbb{M}$, whose tangent at every point is equal to the gradient of f at that point. Usually, along the integral line f increases monotonically, and its two endpoints (limit points) are critical points of f. The Morse function f determines a decomposition of $\mathbb{M}$ based on integral lines. The fusion of integral lines that converge at a critical point is called a *descending* manifold. The descending manifold of a maximum is a n-dimensional manifold. The collection of descending manifolds of critical points of f partition $\mathbb{M}$ is called the *Morse* decomposition. Similarly, ascending manifold of a minimum is the fusion of integral lines that diverge from a minimum. Again, the collection of ascending manifolds of critical points of f partition $\mathbb{M}$.The extremum graph is a representation of the Morse decomposition. A $(n-1)$

saddle s of f lies on the boundary of descending manifold of a maximum m, and the extremum graph captures such relationship between $(n-1)$ saddle and maximum, and thereby captures the combinatorial structure of Morse decomposition. Such extremum graphs are called the Maximal Extremum graph, and if we had considered 1 saddle and minima then we would have called it a Minimal Extremum graph. Most discussions will be around the Maximal Extremum graph, we would drop the term Maximal from subsequent discussions.

In brief, Extremum Graph is a balance between a complete representation (i.e., MS Complex), and an easily comprehensible representation (i.e., cancellation tree). The Maximal Extremum graph is formed by joining 'n-1' saddles with maxima in n-dimensional data set. A practical data set always contains some noise and to remove such noises, we employ simplification, where there are two thresholds: noise threshold for extrema (value above a certain point is considered noise), and, variation threshold for saddles to remove some uninteresting points. There are numerous ways to implement the Extremum Graph, one of the ways is to first construct MS Complex and then construct the Extremum Graph [10] [11], since it is a subset of MS Complex. We can construct MS Complex for 3D but for higher dimensions (i.e., > 3), it takes a toll on the computational time and storage resources. Another method that is implemented in the tachyon can be studied in section 4.2.

4.2 Computation

Input comprises data sets that are formed by sampling scalar field values at points on a uniform sample space, which further maintains the piece-wise linear function properties with the help of linear interpolation if values are missing at those sample points. We usually work with the function values at the sample points. Computation majorly involves numerous stages, some stages which are important are as follows :

- Point Classification: All grid points are classified as either critical or regular.
- Point collection: All grid points which are critical are collected and stored in the data structure.
- Gradient Path Tracing: Critical points mainly comprise saddles (with Morse index 'n-1') and maxima. Path tracing begins at saddles and terminates at maxima.

4.2.1 Point Classification

As we know point classification can be done in numerous ways, but since we are working with uniformly spaced sample points in n-dimensional space (In 2D, imagine a grid-like structure), we can't make use of the Hessian matrix, which in general is applicable in the continuous domain, and allows us to classify points into either critical or regular points. The number of negative eigenvalues in the Hessian matrix helps us to determine the Morse index of the point. In an n-dimensional structure, a point with 'n' negative eigenvalues is classified as maxima, and '0' negative eigenvalues are considered as minima, others with '1' to 'n-1' negative eigenvalues are known as saddles. Since we can't use the Hessian matrix in the discrete domain (i.e., In 2D,grid-like structure), we need to devise a noble method to classify sample points. Already developed ideas in topology will help us in it.

In topology literature, we came across a popular term, that is LINK which will guide us through the neighborhood of the sample point in an abstract way. First, we have to know in an n-dimensional field which points are the neighbor of the inspected sample point, then we can talk about the LINK of the sample point in question. In general in a normal scenario, if we consider a point in sample space, it will have $2n$ neighbors, since there are n-dimension, as each dimension will comprise two neighbors around the point (i.e., 1^{st} neighbor on one side of the point and 2^{nd} neighbor on the other side of each axis, total $2n$ neighbors). The number of points that is present around the reference point in an n-dimensional hypercube, is $3^n - 1$. We make an observation, that as we increase dimensions the size of the neighborhood of the sample point becomes sparser and sparser with respect to the number of points present around the sample point, which makes such a concept of the neighborhood not quite intuitive. We start to have doubts about whether the few neighbors really classify the reference point correctly or not. Hence, we resort to using concepts of the Freudenthal subdivision[8]. It will help to make the neighborhood of a reference point denser, otherwise, in higher dimensions, it became sparser and sparser. The restriction imposed on such subdivision is that common faces between adjacent cells are consistent so that we can tessellate throughout the whole sample space. In 3D we can assume the cube is divided into 6 tetrahedrons (irregular cells). We will tessellate such blocks over all the grid structures given to us. Tessellation will give rise to additional edges in the uniform grid. And adjacency between two grid points in the tessellated grid are checked by condition, which says, two distinct points are adjacent if and only if the difference vector between the two points must entirely constitute non-negative or non-positive values and the magnitude of non-zero values must be exactly 1. From now on, we will term this special condition as the $grid - adjacency$ condition.

The term 'LINK' is an interconnection among the neighbors of the reference point. A neighbor of a reference point is the vertex that is connected to the reference point, or, in other words, the vertex that satisfies the grid-adjacency condition when tested with the reference point.

Imagine the neighborhood of 2D and 3D respectively in Figure 1.

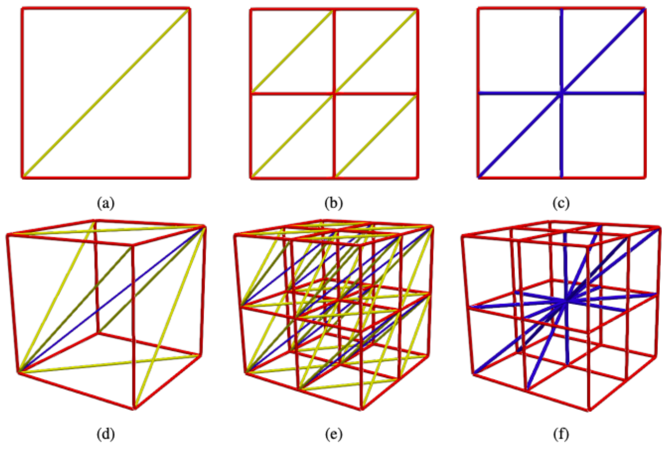


Figure 4.1: Tessellating 2D and 3D grids.Blue represents the edges of the reference point (i.e., vertex at center). 6 neighbors in 2D, and 14 neighbors in 3D. Image from paper:TACHYON [5]

If all neighbors have function values greater than the reference point, we classify the reference point as minima, another scenario happens when all the neighbors have lesser function values compared to the reference point, then we classify the reference point as maxima. The link is further classified into upper links and lower links; the Upper link comprises the vertices that have a greater function value than the reference grid point, similarly for the lower links comprise the vertices that have a lower function value than the reference grid point. We ponder over the thought of the number of connected components in the upper link (or lower link), which might help us to classify the reference grid point. It is observed that a point (i.e., critical or regular) can be distinguished from the number of connected components in its upper link (or lower link). For a point to maxima, its upper link component must be 0. For a point to be regular, its upper link and lower link component must be 1. For a point to be a saddle, its upper link component must be greater than or equal to 2. Hence we have successfully classified a sample point (or a grid point in 2D).

4.2.2 Gradient Path Tracing

Tracing usually starts from a point and follows either an increasing function value point or decreasing one. In our algorithm, ‘n-1’ Morse index saddles are used as starting points, following the neighbor which has the maximum valued function until we reach the maxima.

Chapter 5

Experimental Work

The section deals with the practical work carried out to improve the existing tachyon. It elaborates on the solution we adopted to increase the efficiency of the tachyon.

5.1 Understanding the tachyon

Most real-world data sets [1] are usually limited to 2D-3D dimensions. Synthetic data sets can be created using random assignments at sample points of higher dimensions. Assume, synthetic data sets are given as input to the tachyon. For better understanding, we can take 3D data sets (with dimensions 128x128x128). Each grid point (i.e., (1,2,3)) has a point index (i.e., 16643) as well as a function value (i.e., 12) associated with it.

The first step is point classification, where each point is launched independently and classified using kernel launch of GPU. Each thread undergoes numerous stages, among them the most important; is finding which neighbors are in the upper component link and which neighbors are in the lower component link, and the number of connected components they form. This stage consumes 98%-99% percent of the whole thread execution time. To utilize maximum efficiency, we need to make sure each thread works the same amount otherwise different workload balance gives rise to divergence problems in GPU which will increase the execution time. The main aim is to find the number of connected components in the upper link or lower link. We can use Breadth First Search (BFS) or UNION-FIND. Since divergence issues are too much in the BFS algorithm, we use the UNION-FIND algorithm. To traverse the neighbor-neighbor potential connections of the reference point (i.e., point index of launched thread), we created an adjacency matrix of neighbors in UNIFIED Memory, which can be accessed by both CPU and

GPU. The adjacency matrix is fixed that's why they were already stored in UNIFIED Memory before launching the threads. The adjacency matrix mainly stores the potential edges from each neighbor to another neighbor. For a potential edge to become a real edge, both neighbors either must lie in the upper link or lower link (in other words both neighbors' function value should either be greater or lower than the reference points). After the UNION FIND algorithm is performed, we classify the point as maxima,1-saddle or regular points based on the number of components in the upper link and lower link.

The second step is Point collection: It collects all the critical points and stores them in an array.

The third step is Gradient Path Tracing: We start tracing paths from the saddles to maxima for the Maximal Extremum Graph. We won't dive too much into this stage, because the time here is of little significance at higher dimensions. Figure.2 shows a preview of the execution time distribution of the point classification stage vs gradient path tracing stage for data sets of uniform size (Refer Appendix .2 for x-axis units)

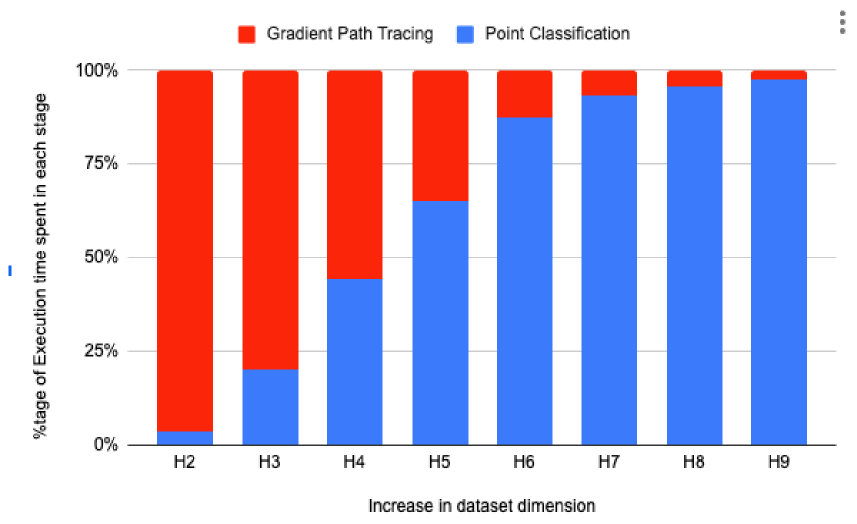


Figure 5.1: Ratio of execution time: Point Classification vs Gradient Path Tracing.

5.2 Idea 1: Conversion of adjacency matrix to edge list

The use of irregular grid cells helps to classify the reference point more accurately than without it. Without the irregular grid cells, the number of neighbors of a reference point is determined by '$2n$' whereas, with the use of irregular grid cells, it is determined by '$2 * (2^n - 1)$'.The hypercube of a reference point has '3^n' points, where 'n' is the number of dimensions. For example: for 3D, without the use of irregular grid cells, we have 6 neighbors, 2 about the x-axis,2 about the y-axis, and 2 about the z-axis. But with irregular cells, we have 14 neighbors of the reference points (The points which are neighbor satisfies the condition: difference vector between the points constitutes either all non-negative or non-positive values, along with the condition that the magnitude of non-zero values must be exactly one). So adjacency matrix size is 14*14=196. As we increase dimensions, the adjacency matrix becomes sparser and sparser, so that's why we got the idea of using an edge list. The number of edges present also has a formula associated with it. The formula:- $3^{n+1} - 3 * 2^{n+1} + 3$, where 'n' is the number of dimensions. To get an idea of how sparse it becomes, let us first look at each dimension; the number of neighbors; the number of edges between neighbor and neighbor; the size of adjacency matrix (=neighbor*neighbor); the size of edge list (=2*edges, because we need to store the endpoints of the edges). Each component is shown in Figure 5.2

Dim	Neighbor	Edge	Adj.matrix	Edge list
2	6	6	36	12
3	14	36	196	72
4	30	150	900	300
5	62	540	3844	1080
6	126	1806	15876	3612
7	254	5796	64516	11592
8	510	18150	260100	36300
9	1022	55980	1044484	111960

Figure 5.2: Increase in size of Matrix vs List

The chart of how sparser the adjacent matrix becomes is shown in Figure 5.3. With the increase in dimensions, the cell entries with a value of zero increase drastically.

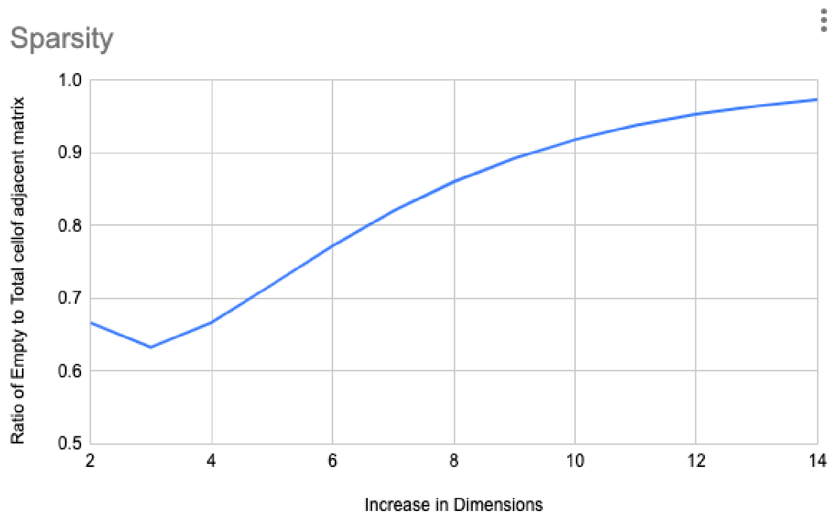


Figure 5.3: Sparsity

In the tachyon, we were traversing every cell of the adjacent matrix, which was mainly filled with zeroes (and were of no use), now we only need to traverse the potential edges which eliminates the time spent traversing through zeroes of the adjacent matrix. After applying the structure of the edge list to the tachyon (we name it as 'Modified-1'), there is an improvement in point classification time, and it is only visible at higher dimensions; because at lower dimensions, due to the high computation power of GPU, unless there is a difference of 10000 operations, no significant improvement can be seen.

Specification:
Software: Tachyon vs Modified-1
Hardware, and Data sets: Refer to Appendix .2

The table for total point classification stage time (in seconds) is represented in Figure 5.4.

Data set	Tachyon	Modified-1
H2	0.226	0.224
H3	1.215	1.225
H4	5.089	5.092
H5	20.607	20.905
H6	158.650	156.992
H7	560.959	560.298
H8	1638.434	1628.116
H9	5316.526	5043.987

Figure 5.4: Time elapsed in Point classification stage: Tachyon vs Modified-1

To get a graphical idea of how much the execution time (of point classification stage) decreases as we increase the dimension in the data set is shown in Figure 5.5.

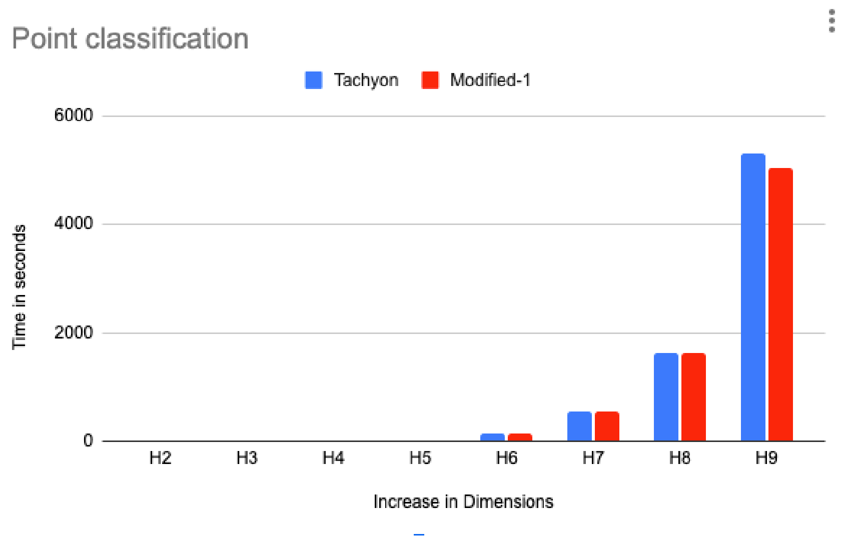
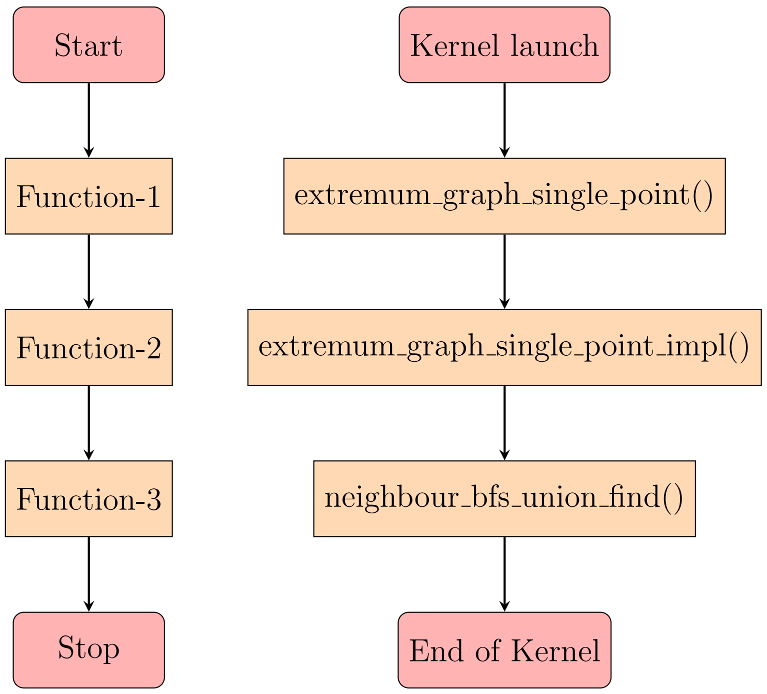


Figure 5.5: Time elapsed in Point classification stage: Tachyon vs Modified-1

5.3 Idea 2:Use of Local LOOKUP table to reduce calls to a particular function

After kernel launch, each thread of a GPU goes through numerous function calls, we will describe them briefly using a flow chart which is represented in Figure 5.6.



Start
Function-1
Function-2
Function-3
Stop
Kernel launch
extremum_graph_single_point()
extremum_graph_single_point_impl()
neighbour_bfs_union_find()
End of Kernel

Figure 5.6: Function calls after the kernel launch

The left side of the flowchart shows the function name in abstract form, and the right side of the flowchart shows the real function name. We will use only abstract function names from now on.

Within the kernel, measuring time consumed by each function (i.e., the function called within the kernel) is not being implemented by any NVIDIA's Pro-filer, be it nsight-system or nsight-compute. We need to create an array to measure time, which won't be accurate, but it will be approximately accurate. There is a huge difference between the whole kernel function execution time vs Function-1 execution time, which will be studied in the next subsection. In this subsection, our focus will be on Function-3.

Within Function-3, two more functions (i.e., let's say Function-31 and Function-32) are called, and we track the time of the two functions, and the result are tabulated and shown in Figure 5.7.

Specification:
Software: Tachyon
Hardware, and Data sets: Refer to Appendix .1

Data set	Function-3	Function-31	Function-32
L2	1.215	0.303	0.417
L3	5.794	1.731	3.667
L4	25.51	7.459	17.123
L5	80.831	26.887	52.710
L6	347.830	67.209	148.990
L7	707.536	149.161	319.761
L8	1952.709	365.276	827.611

Figure 5.7: Function-3 and its components execution time in point classification stage

Note: Time is measured and recorded in seconds.

Almost 18%-30% of Function-3 execution time is consumed by Function-31.Function-31 can be optimized with the help of a lookup table.

Working of Function-31 (i.e., Point Updating) in the Point Classification stage of GPU: It takes the $i'th$ neighbor of the reference point and tells us whether it is in the upper link or lower link components (in other words, whether its function value is greater than the reference point or not).In the tachyon, it is called whenever either 'i' changes or 'j' changes (where (i,j) represents the coordinates in the adjacency matrix).In short, it is called *O(neighbours+potential_edges)* number of times, though a lot of times the same neighbor is passed, and the same value is returned. To reduce such calls we create a Boolean array of size neighbors to store whether the neighbors' function value is greater than or less than the reference point. So now, we only traverse *O(neighbors)* number of times, using extra space of *O(neighbors)*.

Let's see the reduction in the number of calls with the help of a reduction % which is shown in Figure 5.8.

Dim	Neighbor	Edge+Neighbor	Reduction%
2	6	6	0.5
3	14	50	0.72
4	30	180	0.833333
5	62	602	0.89701
6	126	1932	0.934783
7	254	6050	0.958017
8	510	18660	0.972669
9	1022	57002	0.982071

Figure 5.8: Reduction % age in the number of calls

Implementing Idea 1 and Idea 2 together in the tachyon, we get another modified code base (let's say Modified-2), and we compare the execution time of the point classification stage of both the tachyon vs Modified-2.

Specification:
Software: Tachyon vs Modified-2
Hardware, and Data sets: Refer to Appendix .2

Figure 5.9 shows only the total point classification stage time.

Data set	Tachyon	Modified-2
H2	0.226	0.204
H3	1.215	0.974
H4	5.089	3.129
H5	20.607	12.654
H6	158.650	139.965
H7	560.959	488.774
H8	1638.434	1379.507
H9	5316.526	4027.727

Figure 5.9: Tachyon vs Modified-2

Note: Time is measured and recorded in seconds.

The graphical format is shown in Figure 5.10

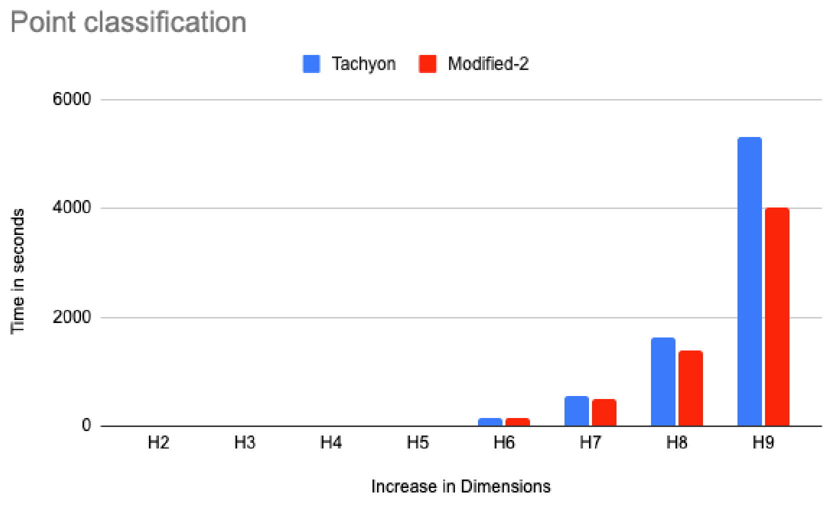


Figure 5.10: Tachyon vs Modified-2 execution time

5.4 Idea 3: Utilising the SHARED MEMORY Of GPU to avoid thrashing

An astonishing discovery was made when we measured the total kernel execution time vs the Function-1 execution time. Function-1 is the first and foremost function that is called by each thread launched by the kernel, it should be at least equal to the total execution time. But to our surprise, it was not. Let's revisit the flowchart (Figure 5.6) in subsection 4.3 and observe the time elapsed at each function. To again stress the fact that functions within the kernel can't be accurately recorded, but approximately accurately recorded using the clock function, so tolerance of 5-10 seconds is ignored. Figure 5.11 shows the time recorded for each function.

Specification:
Software: Tachyon
Hardware, and Data sets: Refer to Appendix .1

Data set	Kernel Exec. time	Function-1 Exec.time
L2	0.952	1.228
L3	4.463	5.840
L4	19.329	25.445
L5	73.426	80.556
L6	740.973	363.439
L7	3012.142	713.747
L8	9680.262	1899.302

Figure 5.11: Kernel vs Function-1 execution time

Note: Time is measured and recorded in seconds.

The graphical format is shown in Figure 5.12.

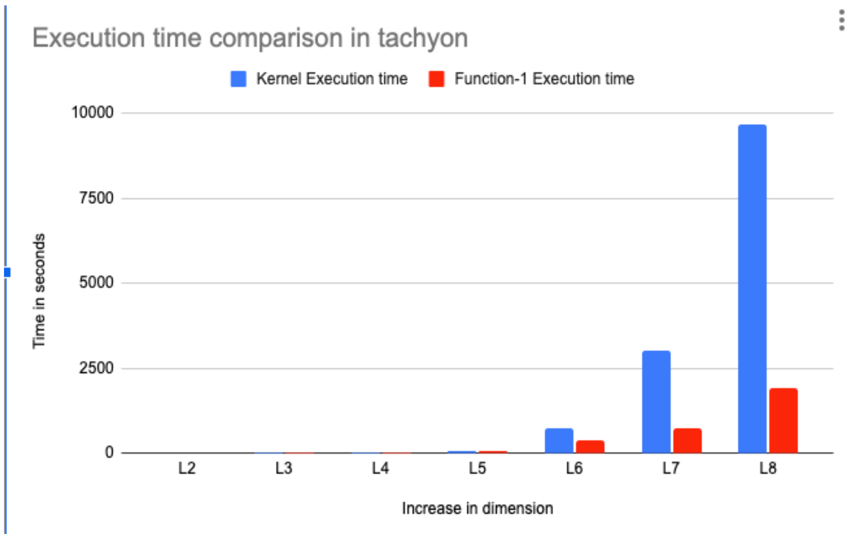


Figure 5.12: Kernel vs Function-1 execution time

The reason for the time lapsed which isn't recorded is due to thrashing occurring at an aggressive level. The space (i.e., page) required for storing the edges keep on increasing as the dimensions increase. Memory space is limited in each SM (streaming multiprocessor) of the GPU. Each SM can handle only a few thread blocks. Different threads in each thread block require access to different edge points (for example thread 1 requires access to the 12^{th} edge point and thread 35 require access to the 105^{th} edge point which is on a different page, so page replacement will happen). The frequency of page replacement increases as the number of edges keeps on increasing. That's why we can see as we move to higher dimensions, kernel execution time is much much greater than Function-1 execution time. Originally to improve memory access time, the edge list was stored in Unified Memory. Whenever a thread requires an edge from the edge list, the page containing the edge and nearby edges is transferred to the LOCAL memory of SM (which is also known as L1 cache). The local memory is replaced frequently and hence the time elapsed in replacement is not recorded by our clock functions.

To improve memory management of the whole process of the point classification stage, we use the SHARED Memory of the GPU. The most important characteristic of the SHARED Memory is that it is an ON-CHIP memory, its access time is almost equal to or a little worse than the local registers of each thread. Each thread block has a separate shared memory assigned to it. Now the problem of page replacement is reduced by the fact that at the start of the function we store the edge list in the shared memory and then execute all the remaining operations, here we minimize the number of page replacements that will occur. As the size of shared memory is limited (default:48 KB), we have to use it judiciously. The number of edges that are stored in shared memory is 5796 at one pass, the limit is set explicitly by the programmer. The reason to store only 5796 edges is that dimension 7 has 5796 edges, and we could dynamically allocate that many edges quite peacefully, but at dimension 8 there are 18150 edges, as soon as we compile the code, we get an error of allocating too much-shared memory, which isn't available in our GPU.So as a workaround for dimensions greater than 7, we reinitialize the shared memory with the next set of 5796 edges and used the function syncthreads (), which synchronizes all the threads in a thread block. If we don't use synchronization, some threads within a thread block will execute faster and change the shared memory without other threads realizing it. So we need to avoid that, as other operations are dependent on those edges. Finally, we reached the greatest reduction in time we have ever seen.

Specification:
Software: Tachyon vs Modified-3
Hardware, and Data sets: Refer to Appendix .2

We are using the default GPU, with device ID:0, though it has two GPUs available. The data set size is kept constant as we increase dimensions (i.e., as we move from 'n' to 'n+1' dimensional data set, where 'n' represents dimension).

The Point Classification stage execution time for both the tachyon and say Modified-3 code (which includes Idea 1 + Idea 2 + Idea 3) [2] are as shown in Figure 5.13.

Data set	Tachyon	Modified-3
H2	0.226	0.188
H3	1.215	1.015
H4	5.089	3.328
H5	20.607	14.725
H6	158.650	139.358
H7	560.959	210.109
H8	1638.434	571.683
H9	5316.526	1724.915

Figure 5.13: Tachyon vs Modified-3 execution time

Note: Time is measured and recorded in seconds.

The graphical representation is shown in Figure 5.14.

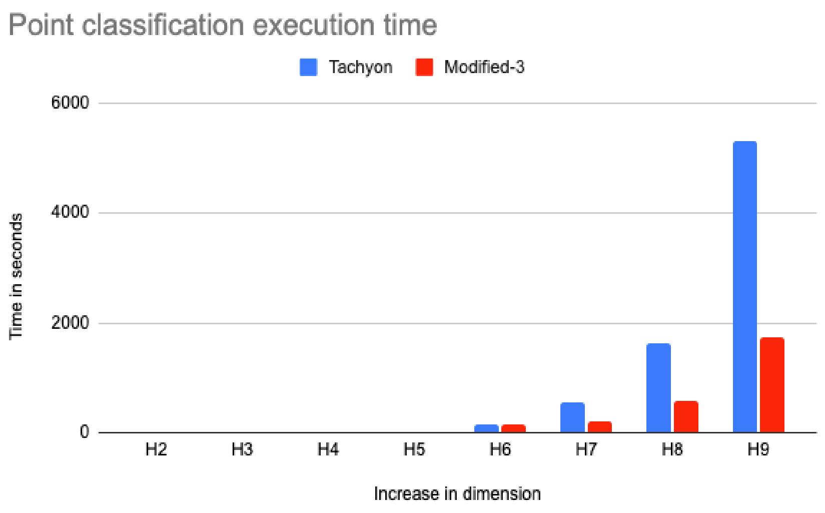


Figure 5.14: Tachyon vs Modified-3 execution time graphical view

To probe further into code-level changes, programmers and future developers are advised to go through the file available at site [3].

5.5 Total Execution Time for Computation of Extremum Graph

Abbreviations which might help:
Point Classification (PC)
Gradient Path Tracing (GPT)
Extremum Graph Computation (EGC)

5.5.1 HIGH END GPU

Specification:
Software: Tachyon vs Modified-3
Hardware, and Data sets: Refer to Appendix .2

Tachyon's Performance is tabulated in Figure 5.15.

Data set	PC	GPT	EGC
H2	0.226	5.866	10.694
H3	1.215	4.866	9.282
H4	5.089	6.374	14.434
H5	20.607	11.061	33.598
H6	158.650	22.431	182.751
H7	560.959	41.411	603.618
H8	1638.434	72.336	1711.764
H9	5316.526	124.124	5441.502

Figure 5.15: Tachyon's Execution time on High-End GPU

Note: Time is measured and recorded in seconds.

MODIFIED-3's performance is shown in Figure 5.16.

Data set	PC	GPT	EGC
H2	0.188	5.426	10.789
H3	1.015	4.636	9.172
H4	3.328	5.607	11.596
H5	14.725	9.968	26.535
H6	139.358	18.392	159.267
H7	210.109	24.761	235.939
H8	571.683	31.262	603.774
H9	1724.915	45.379	1770.993

Figure 5.16: Modified-3's execution time on High-End GPU

Note: Time is measured and recorded in seconds.

Percentage Reduction in Extremum Graph Computation time is shown by the graph which is displayed in Figure 5.17:

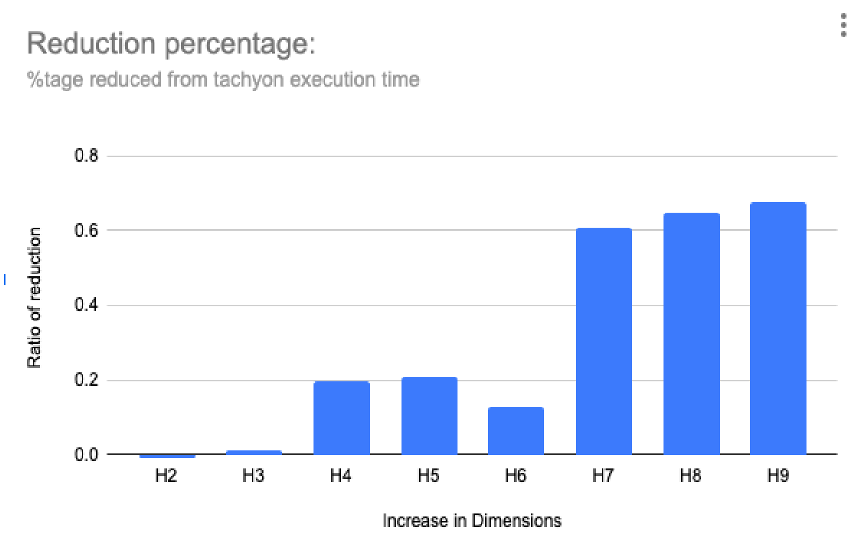


Figure 5.17: % age reduced from tachyon's execution time

5.5.2 LOW END GPU

Specification:

Software: Tachyon vs Modified-3

Hardware, and Data sets: Refer to Appendix .1

Tachyon's Performance is tabulated in Figure 5.18.

Data set	PC	GPT	EGC
L2	0.831	6.544	11.509
L3	4.278	11.829	18.743
L4	19.116	28.979	50.386
L5	73.545	65.660	140.803
L6	741.311	129.766	872.340
L7	2974.154	260.013	3235.094
L8	9970.500	416.006	10387.190
L9	34019.706	620.995	34641.324
L10	109936.446	678.582	110615.565

Figure 5.18: Tachyon's Execution Time on Low-End GPU

Note: Time is measured and recorded in seconds.

MODIFIED-3 performance is shown in Figure 5.19:

Data set	PC	GPT	EGC
L2	0.605	6.365	11.244
L3	2.924	11.069	16.938
L4	10.771	21.102	34.269
L5	42.154	59.575	103.360
L6	463.255	105.124	569.683
L7	1179.293	187.164	1367.402
L8	3743.255	185.955	3929.919
L9	11543.160	337.020	11880.751
L10	34683.603	284.878	34969.013

Figure 5.19: Modified-3's execution time on Low-End GPU

Note: Time is measured and recorded in seconds.

We observe that in data set ‘L10’ which is 10 dimension data set, the elapsed time in the point classification stage is 109936 seconds which is approximately equal to 30 hours (or 1 day 6 hours), whose execution time is reduced tremendously by our Modified-3 implementation. The elapsed time for the point classification stage for Modified-3 is 34683 seconds (or 9 hours) showing a huge success of Modified-3 implementation and a great reduction in execution time as well.

Percentage Reduction in Extremum Graph Computation time is shown by the graph which is displayed in Figure 5.20:

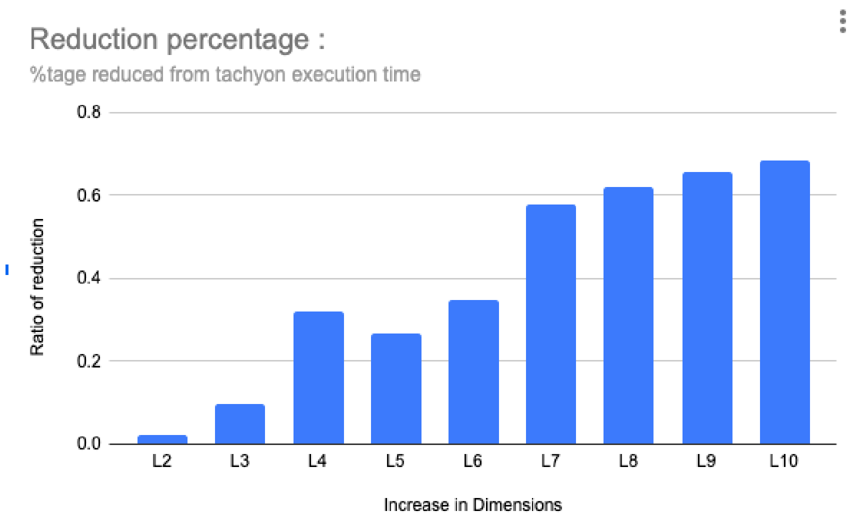
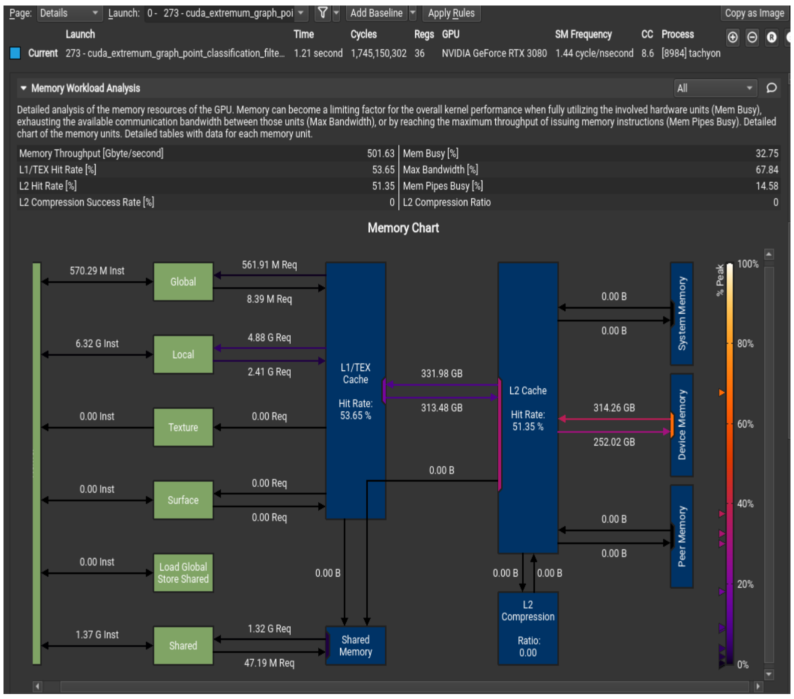


Figure 5.20: % age reduced from tachyon’s execution time

5.6 Memory workload analysis by Nsight compute

Memory Workload Analysis of tachyon and Modified-3 are shown in Figure 5.21 and in Figure 5.22 respectively.



Page: Details Launch: 0 - 273 - cuda_extremum_graph_poi Add Baseline Apply Rules Copy as Image

	Launch	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	273 - cuda_extremum_graph_point_classification_filte...	1.21 second	1,745,150,302	36	NVIDIA GeForce RTX 3080	1.44 cycle/nsecond	8.6	[8984] tachyon

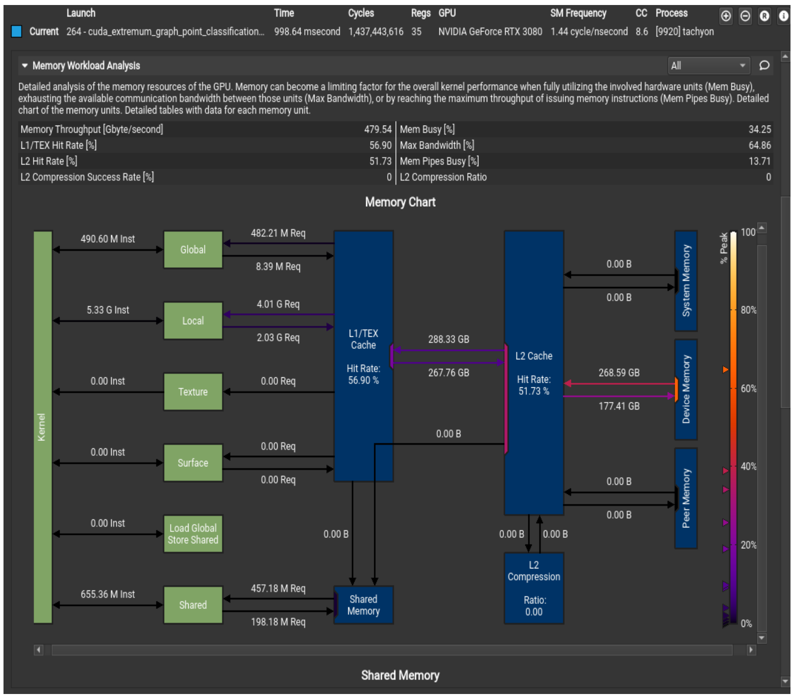
Memory Workload Analysis All

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	501.63	Mem Busy [%]	32.75
L1/TEX Hit Rate [%]	53.65	Max Bandwidth [%]	67.84
L2 Hit Rate [%]	51.35	Mem Pipes Busy [%]	14.58
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

Memory Chart

Figure 5.21: Tachyon's Memory Workload Analysis



	Launch	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	264 - cuda_extremum_graph_point_classification...	998.64 msecond	1,437,443,616	35	NVIDIA GeForce RTX 3080	1.44 cycle/nsecond	8.6	[9920] tachyon

Memory Workload Analysis All

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	479.54	Mem Busy [%]	34.25
L1/TEX Hit Rate [%]	56.90	Max Bandwidth [%]	64.86
L2 Hit Rate [%]	51.73	Mem Pipes Busy [%]	13.71
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

Memory Chart

Shared Memory

Figure 5.22: Modified-3's Memory Workload Analysis

Dataset:

LThrees_512x512x512_uint8.raw

Profiler:

Nsight Compute 2021.4

Profiling Application:

tachyon vs Modified-3

GPU specs:

2 x NVIDIA GeForce GTX 3080 card with 8704 CUDA cores, 11 GB RAM.

We can't profile data sets in NVIDIA-provided Nsight compute; whose GPU time period is $>$ 5 seconds. We need to content ourselves with a 3D data set or 2D data set, whose GPU time period is $<$ 5 seconds.

Let's look below two diagrams extracted from Nsight compute pro-filer.
Memory Workload analysis is available to us. We observe carefully and see the request for data is reduced, and the heavy performance arrows (i.e., the red arrows) have less load to deal with.

The L1 Hit ratio is improved as well as the L2 Hit ratio, which is a good sign for our Modified-3 algorithm.

Chapter 6

Discussion and Future Work

The section describes some intricacies of the current work and also discusses the future scope of the project.

6.1 Discussion

One of the stages of Extremum Graph Computation is Gradient Path Tracing. As we increase the dimension, the execution time of this stage decreases due to the fact of less number of critical points are being discovered at higher dimensions. But, this stage plays an important role in lower dimensions, where its execution time contributes to almost always > 50% of total execution time, to support my statement we can refer to Figure 2. Gradient Path Tracing computation time (at higher dimensions>6) is too less compared to point classification to be considered for optimization. But apart from that in the lower dimensions, it is hard to parallelize because the path length can vary too much. If one thread constructs 10 arcs along a path and the other 1 thread constructs 1 arc along a path, then their workload imbalance is too high to be considered for parallelism. In other words, due to divergence problems, we cannot extract maximum performance from the GPU in a simpler fashion. But there are advanced graph algorithms that are being developed to handle such issues. To reduce the execution time of the gradient path tracing stage we need further literature study on advanced GPU-CPU algorithms that are being developed at a rapid pace. Hence, it's daunting to even think to parallelize the path-tracing stage of the algorithm with simpler techniques or without appropriate knowledge of present-day advanced methods.

Extremum Graph is a significant structure that helps in numerous applications. There are applications where critical points are tracked at every time step using an extremum graph, and we can determine the origin of the critical point (For example Cyclone, thunderstorm, in a real-life application, etc).

Memory is limited in GPU, and managing it in an efficient way becomes crucial. Extremum graph computation at lower dimensions (<3D) was performing well enough, but at higher dimensions (>5D), the performance took minutes, hours, and sometimes days to complete execution. With the help of numerous ideas: creating a local lookup table; conversion of the adjacency matrix to an edge list; the utilization of the shared memory of GPU; we have successfully reduced the tachyon's time by at least 65%-75%, which is a significant improvement from the tachyon at higher dimensions.

6.2 Future Work

Currently, the tachyon is working on a single GPU. We can distribute the workload among multiple GPUs. The distributed system is one of the future ideas, where we can distribute the workload of all the threads launched by the kernel at different systems and utilize the efficiency of all the systems available to us.

Appendices

.1 Appendix A: Datasets for Low End GPU

GPU specs:

NVIDIA Corporation GP106: NVIDIA GeForce GTX 1060 6GB (rev a1)

Data set along with their shape and datatype:

L2: LTwos_16384x8192_uint8.raw
L3: LThrees_512x512x512_uint8.raw
L4: LFours_128x128x128x64_uint8.raw
L5: LFives_32x32x32x32x128_uint8.raw
L6: LSixs_16x16x16x16x16x128_uint8.raw
L7: LSevens_16x16x16x16x16x16x8_uint8.raw
L8: LEights_16x16x16x8x8x8x8x8_uint8.raw
L9: LNines_8x8x8x8x8x8x8x8x8_uint8.raw
L10: LTens_8x8x8x8x8x8x8x4x4x4_uint8.raw

.2 Appendix B: Datasets of High-End GPU

GPU specs:

2 x NVIDIA GeForce GTX 3080 card with 8704 CUDA cores, 11 GB RAM.

Dataset along with their shape and datatype:

H2: HTwos_8192x16384_uint8.raw
H3: HThrees_512x512x512_uint8.raw
H4: HFours_128x128x128x64_uint8.raw
H5: HFives_64x64x32x32x32_uint8.raw
H6: HSixs_16x16x16x32x32x32_uint8.raw
H7: HSevens_16x16x16x16x16x16x8_uint8.raw
H8: HEights_16x16x16x8x8x8x8x8_uint8.raw
H9: HNines_8x8x8x8x8x8x8x8x8_uint8.raw

Bibliography

[1] Open scientific visualization datasets. URL https://klacansky.com/open-scivis-datasets/. accessed: 12-June.2021. 13

[2] Modified tachyon library. https://bitbucket.org/vgl_iisc/tachyon/src/edge-list-link/. 24

[3] Implementation details. https://bitbucket.org/vgl_iisc/tachyon/src/edge-list-link/README_v2.md. 25

[4] Tachyon library. https://bitbucket.org/vgl_iisc/tachyon. ii, 2, 4, 5

[5] Varshini Subhash Abhijath Ande and Vijay Natarajan. Tachyon: Efficient shared memory parallel computation of extremum graphs. *Computer Graphics Forum*, 1(1):1–14, 2023. v, 2, 11

[6] P.-T. Bremer, B. Hamann, H. Edelsbrunner, and V. Pascucci. A topological hierarchy for functions on triangulated surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):385–396, 2004. doi: 10.1109/TVCG.2004.3. 3

[7] Peer-Timo Bremer Carlos Correa, Peter Lindstrom. Topological spines: A structure-preserving visual representation of scalar fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1842–1851, 2011. 3, 8

[8] SNOEYINK J. CARR H., MOLLER T. Artifacts caused by simplicial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):231–242, 2006. 10

[9] Vijay Natarajan Herbert Edelsbrunner, John Harer and Valeri Pascucci. Morse-smale complexes for piecewise linear 3-manifolds. *In Proceedings of the nineteenth annual Symposium on Computational Geometry*, 1(1):361–370, 2003. 1, 3

[10] M Senthilnathan Nithin Shivashankar and Vijay Natarajan. Parallel computation of 2d morse-smale complexes. *IEEE Transactions on Visualization and Computer Graphics*, 18 (10):1757–1770, 2011. 3, 9

[11] M Senthilnathan Nithin Shivashankar and Vijay Natarajan. Parallel computation of 3d morse-smale complexes. *Computer Graphics Forum*, 31(3):965–974, 2012. 9