SR-KNN: A High-Performance Fixed-Radius KNN Search with Spatial and Range Awareness on 2D-Mesh Accelerators

Yinsicheng Jiang

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

This thesis introduces the Spatial and Range Aware K Nearest Neighbor (SR-KNN), a fixed-radius exact K Nearest Neighbor Search (KNN) algorithm optimized for 2D-Mesh spatial accelerators, addressing the growing demands for computing power, model complexity, and data volume in machine learning applications. SR-KNN utilizes the distinct advantages of 2D-Mesh accelerators, including a fully distributed system between cores and asynchronous data exchange. The algorithm makes three significant contributions to the field: (i) the utilization of 2D-Mesh accelerators' structure for data partitioning and radius-based search optimization, enabling more efficient processing of sparse data; (ii) efficient SIMD on-chip computation for rapid distance calculations, reducing the computational load and enhancing performance; and (iii) a merge-sort algorithm for pairwise collective K selection, significantly improving the speed of the K Selection and reducing the memory consumption. By addressing key challenges such as the lack of spatial awareness in traditional CPUs and GPUs and the inefficiencies in handling sparse point datasets, SR-KNN demonstrates a potential improvement in exact KNN search throughput by 2 to 3 times compared to existing methods. We are in the process of adapting SR-KNN to the Tenstorrent, a real 2D-Mesh accelerator.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Yinsicheng Jiang*)

Acknowledgements

I extend my heartfelt gratitude to Prof. and Dr. Luo Mai, along with his PhD students Congjie He and Yeqi Huang, for their invaluable assistance with this work. Without their support, this project would not have been possible.

Table of Contents

1 Introduction **1**
- 1.1 Fixed-Radius K Nearest Neighbor Search . . . 1
- 1.2 KNN with AI Accelerators . . . 1
  - 1.2.1 Problems with Fixed-Radius KNN . . . 2
  - 1.2.2 Spatial Accelerators Solution . . . 2
- 1.3 Open Questions and Main Contributions . . . 3

2 Background **4**
- 2.1 Current Exact KNN Strategies . . . 4
  - 2.1.1 KD-Tree . . . 4
  - 2.1.2 Ball-Tree . . . 7
- 2.2 Current KNN on CPU . . . 8
- 2.3 Current KNN on GPU . . . 9
  - 2.3.1 Faiss Flat . . . 9
- 2.4 Problems with Current CPU and GPU KNN . . . 10
- 2.5 KNN on 2D-Mesh Accelerator . . . 11
  - 2.5.1 2D-Mesh Accelerators . . . 11
  - 2.5.2 Tenstorrent . . . 12
  - 2.5.3 Benefits of 2D-Mesh KNN . . . 13

3 Methodology **14**
- 3.1 Workflow . . . 15
- 3.2 Space-Conscious Data Partition . . . 16
- 3.3 Range-Aware Route Planning . . . 17
- 3.4 Broadcasting and Communication . . . 19
  - 3.4.1 Broadcasting . . . 19
  - 3.4.2 Core Communication . . . 20
- 3.5 On-Chip Computation . . . 20
- 3.6 Back Transferring and Reduce . . . 21
- 3.7 Gather and Final K Selection . . . 22
- 3.8 Simulator Implementation Details . . . 23
  - 3.8.1 Data Partition Concerns . . . 23
  - 3.8.2 Core Simulation Concerns . . . 24
  - 3.8.3 Merge-Sort Details . . . 24

4 Experiment **25**

4.1 Dataset 25
4.1.1 Balanced Random 1M Dataset 25
4.1.2 Imbalanced GNS 1.1M Dataset 26
4.2 Computational FLOPs Benchmark 26
4.3 Performance Estimation 27
4.4 Scalability 29
4.5 Performance Benchmark 30
4.6 Discussion 31
4.7 Future Work 31

5 Conclusion **33**

Bibliography **34**

A Baseline Implementation **38**
A.1 KD-Tree Implementation 38
A.2 Ball-Tree Implementation 39
A.3 Space-Conscious Data Partition Detail 41
A.4 Single Core Operations 41

Chapter 1

Introduction

The K nearest neighbor (KNN) search is widely used in various domains such as machine learning and information retrieval, with applications that include graph network simulators [Kumar and Vantassel, 2023], textual semantic retrieval [Cer et al., 2018], anomaly detection [Gu et al., 2019], recommendation systems [Zhao et al., 2019], image search across borders [Jia et al., 2021], and financial fraud detection [Alammar et al., 2022]. The goal of KNN is to locate the K most similar data points to a given query within a database containing a finite set of datapoints in a vector space. The primary challenge in developing an effective KNN algorithm lies in achieving accurate KNN results while maintaining computational efficiency.

1.1 Fixed-Radius K Nearest Neighbor Search

The core principle of KNN algorithm is to identify the neighbors surrounding the query point, meaning that only those points in close physical proximity are worthwhile of being calculated distances. Many other calculations beyond this are proven to be unnecessary, as points that are physically distant from the query are irrelevant. This highlights the importance of a fixed-radius K nearest-neighbor search, also known as a radius query, focusing on locating all data points situated within a specified distance from a query point. This method is detailed in Bentley's historical overview [Bentley, 1975]. For fixed-radius KNN search, within each specified range, the simplest approach to identifying nearest neighbors is still conducting a linear search across the entire database, a method often stated as exhaustive or Brute-Force search. Although considered unwise, it is still widely used; in addition, with the help of AI accelerators, the calculation and search speed can be greatly enhanced.

1.2 KNN with AI Accelerators

AI accelerator chips have revolutionized the application of machine learning in various industries. As the demand for computing power, model complexity, and data volume accelerates at an extraordinary rate, there is an increasing requirement for solutions that

offer both high performance and enhanced efficiency. Today, the landscape of high-performance computing hardware has developed significantly, encompassing advanced accelerators such as NVIDIA Graphics Processing Units (GPUs) and Google Tensor Processing Units (TPUs). These groundbreaking computing platforms have emerged to meet the growing demand for accelerated computational capabilities, catering to both industrial-scale projects and individual demands. K nearest neighbor search, an important tool widely used in machine learning tasks, is also further accelerated by these platforms. Recent researches, including studies on GPU-based KNN [Garcia et al., 2008, Johnson et al., 2021], TPU-based KNN [Chern et al., 2022], CPU-optimized KNN [Chen and Güttel, 2024, Pedregosa et al., 2011], and CPU-GPU hybrid KNN [Muhr and Affenzeller, 2022] have emerged to enhance KNN performance through considerable reductions in latency.

1.2.1 Problems with Fixed-Radius KNN

However, research has not been carried out extensively on exact fixed-radius KNN. We find the following challenges:

(1) GPUs and CPUs lack spatial awareness as their memories are centrally managed, making it challenging to define a specific range or radius for searching.

(2) Current sparse point datasets provide merely coordinates of each point without including information about the closeness between points so that it becomes difficult to determine which points fall within the range, although the search radius is given.

(3) Fixed-Radius KNN is also based on massive distance computations with GEMM and GEMV operations like common KNN, which provides a heavy workload.

1.2.2 Spatial Accelerators Solution

To overcome these challenges, the spatial accelerators are developed. These accelerators have numerous cores, also called Processing Elements (PEs), designed to boost MM and MV operations, and the PEs are connected through 2D-Mesh-like network-on-chips, providing massive on-chip memory bandwidth. Notable examples include Tenstorrent, Tesla Dojo, TPUv5, and Cerebras. Due to the structure of 2D-Mesh accelerators, sparse data can be partitioned among specific cores for independent processing. Moreover, the radius also becomes realistic in these accelerators because cores are placed like a 2D plane, and only the cores located within the radius are required to execute computations.

In this thesis, we will introduce SR-KNN, a novel exact KNN specifically designed to capitalize on the capabilities of 2D-Mesh accelerators. At its core, SR-KNN employs an efficient SIMD on-chip computation and a fast merge-sort algorithm for distance calculation and K selection, which are proven to have processing times close to $O(MN)$ and $O(K)$ respectively. We are in the process of adapting it for Tenstorrent. Early theoretical experiments have shown that SR-KNN can improve exact KNN search throughput by 2 to 3 times compared to existing exact KNN methods on CPU and GPU.

1.3 Open Questions and Main Contributions

To enable and optimize fixed-radius KNN on a 2D-Mesh accelerator, several key challenges must be addressed:

- How to determine an effective strategy for partitioning data across the cores of the 2D-Mesh accelerator without prior knowledge of data closeness.
- How to identify which cores are within the radius of each query.
- How to dispatch the query efficiently from the starting core to the target cores.
- How to perform efficient computation on each core.
- How to select the K values efficiently with minimum memory consumption.

In order to address above, we make the following contributions to SR-KNN:

Space-Conscious Data Partition: It is a strategy for data partitioning that allocates data points to their respective cores based on their coordinates, allowing for autonomous processing by each core in a distributed manner.

Range-Aware Route Planning: The routing method is introduced to enable communication between cores within a specific range defined by the radius setting.

Broadcasting and Core Communication: SR-KNN utilizes the asynchronous routers of the 2D-Mesh chip to broadcast the query from the starting core to the target cores.

SIMD On-Chip Computation: It is a fast distance calculation and K selection algorithm, utilizing the SIMD capabilities which allow for more than one operation per instruction cycle.

Back Transferring and Reduce: They are techniques designed by an efficient merge-sort algorithm for cores' pairwise collective K selection and aggregating for determining the final K values from the contributions of all associated cores.

Gather and Final K Selection: SR-KNN guarantees the correctness of the final K nearest neighbors by implementing atomic K selection at the starting core, with the results received from neighbor cores.

Chapter 2

Background

2.1 Current Exact KNN Strategies

Currently, exact KNN search methods such as Brute-Force, KD-Tree [Bentley, 1975], and Ball-Tree [Omohundro, 2009] are commonly employed to fulfil search requirements. Libraries like scikit-learn have effectively implemented these three KNN search methods for CPUs. Additionally, Faiss offers a Flat algorithm that achieves efficient Brute-Force search on GPUs. When compared to Brute-Force KNN searches, both KD-Tree and Ball-Tree show superior performance due to their efficient indexing mechanisms, which employ unique indices for each point to minimize unnecessary searches and computations. We will detailedly discuss them in the following sections.

2.1.1 KD-Tree

A KD-Tree is a space-partitioning data structure used for organizing points in a k-dimensional space. It is constructed by recursively partitioning the k-dimensional space into two halves while one dimension at a time is chosen. Each node in the tree is a hyperrectangle in the space, with the root node covering the entire space. When the number of points becomes less than a particular threshold or when the best depth is reached, the process of partitioning has to stop.

KD-Tree Construction: The KD-Tree search consists of two main phases: construction and searching. A KD-Tree is a hierarchal structure built by partitioning the data recursively along the dimension of maximum variance [VLFea, 2007]. To illustrate, the process of selecting the root node in this algorithm involves calculating the variances of each dimension and choosing the dimension with the highest variance. The point along that dimension's axis that equals the median value of the dimension's values is then selected as the root node. In Figure 2.1, the x dimension shows a larger variance. Consequently, the point with an x-coordinate equal to the median of the values on the x-axis (either 5 or 7) is chosen as the root node. In this case, (7, 2) is selected as the root node. Subsequently, the x-axis is divided into two halves. For each half, the points with the median value of y coordinates are selected as child nodes. In the provided figure, to the left of the root node (7, 2), the point (5, 4) has the median y-coordinate. Therefore,

(5, 4) is chosen as the left child of the root node. Similarly, on the right side of the root node, (9, 6) with the median y-coordinate is selected as the right child. This process continues with switching dimensions until the termination condition is met.

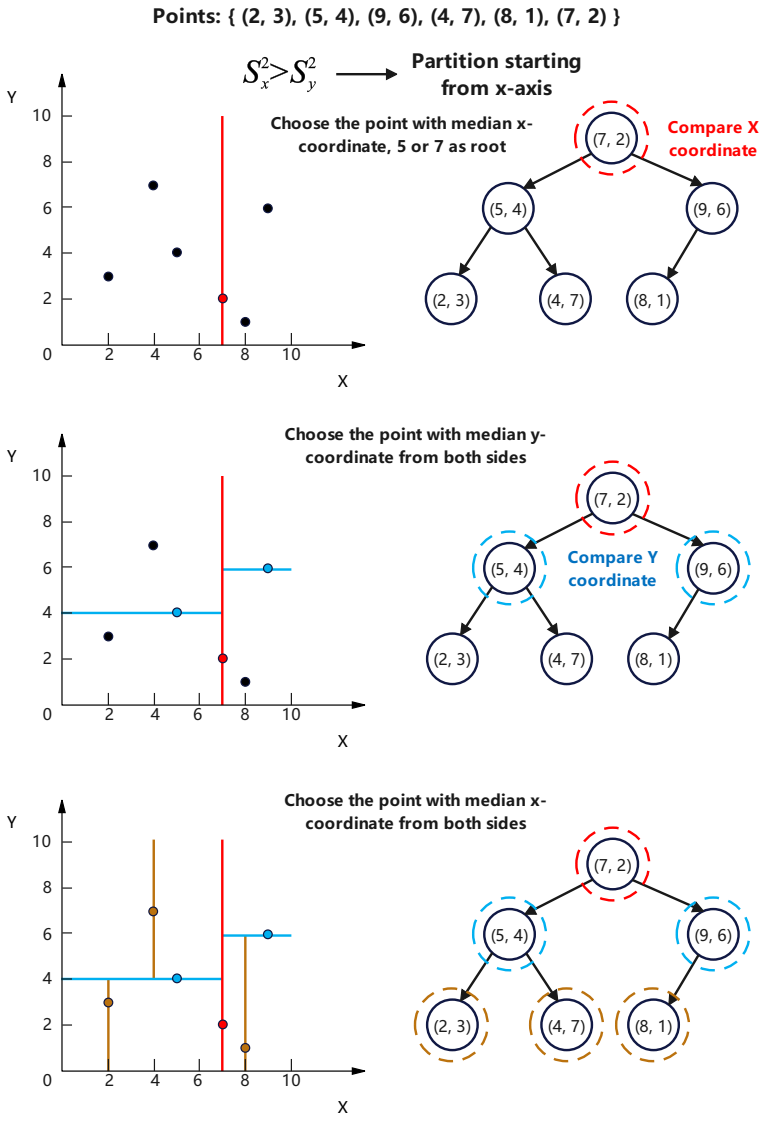

Figure 2.1: Construction of the KD-Tree based on the maximum variance

KD-Tree Search: After constructing the KD-Tree, the search is involved. There are several steps to show how the search works:

(1) Starting from the root, the algorithm moves down the tree to find the leaf node that the query point would belong to. This step is efficient because it involves comparing the query point to the values at each node, determining whether to move left or right in the tree without any distance calculation.

(2) Once the leaf node is reached, the algorithm backtracks up the tree, checking the sibling nodes and potentially further up to ensure that no closer neighbors are missed. This step is necessary because the nearest neighbors of a query point are not always in the same leaf node.

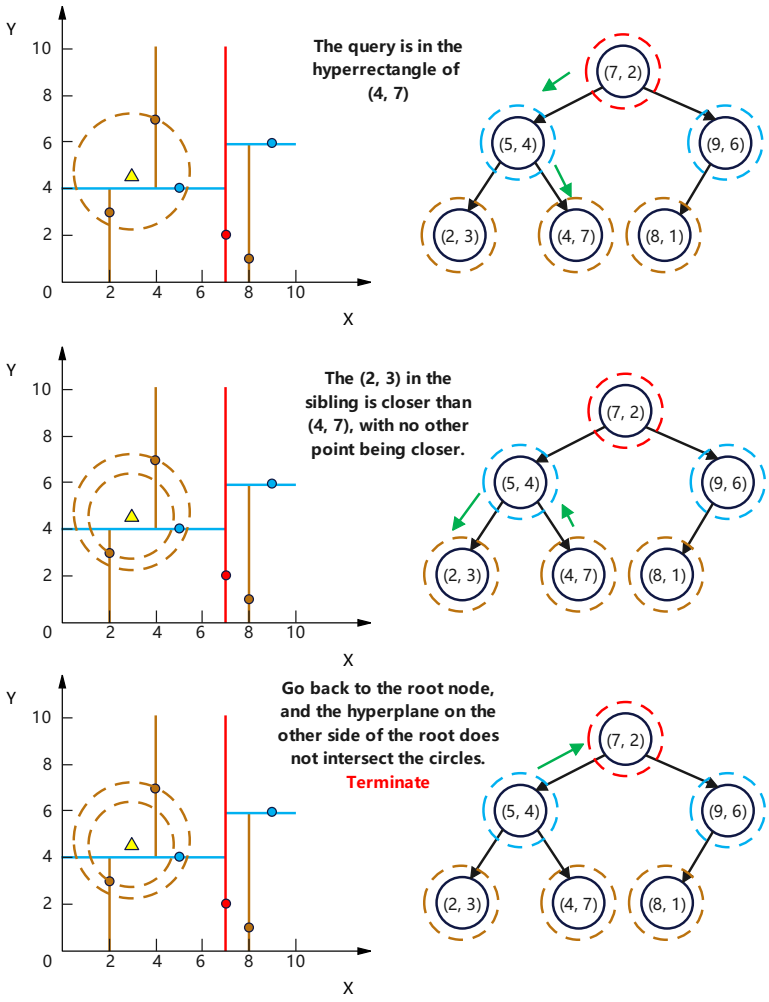

Figure 2.2: Nearest Neighbor Search with KD-Tree

(3) On the way, the distances from the query point to the points in the nodes visited are calculated. The K closest points during the search are kept.

(4) One of the key optimizations throughout the search is branch pruning. If the distance between the query point and the boundary of the hyperrectangle is greater than the distance from the query point to the Kth nearest neighbor found up to the present time, searching in this node of the tree and its subtrees can be safely stopped. Because of this pruning strategy, KD-Tree can largely reduce the number of distance calculations. It both maintains the accuracy of the search algorithm (still exact KNN search) and descends the searching latency.

Figure 2.2 shows an example of the Nearest Neighbour Search with KD-Tree. Starting from the root node, since the x-coordinate of the query point S is less than the x-coordinate of the root node, access the left child node. Since the y-coordinate of S is greater than the y-coordinate of the current node, access the right child node. Upon reaching a leaf node, stop and draw a circle centered at the current leaf node with the distance to S as the radius. Then, start to backtrack to its parent node, checking if the circle intersects with the space of the other branch. The method to check is to determine

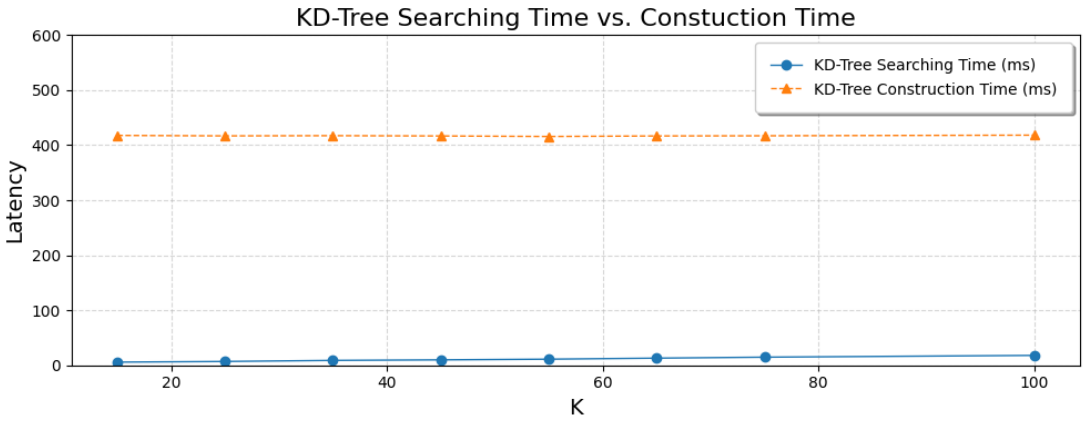

Figure 2.3: KD-Tree Searching Time vs. Constuction Time

if the difference between the y-coordinate of S and the y-coordinate of the parent node is less than the radius of the circle. If it is less than the radius, there is an intersection. Since there is an intersection, check the nodes in the other child space of the parent node to see if any are closer to S than the current nearest neighbor. As the point (2, 3) is closer to S, update the nearest neighbor to (2, 3). Afterwards, backtrack to the parent node (5, 4), and calculate the distance between the parent node and S. Since the distance is greater, the nearest neighbor remains unchanged. Finally, backtrack to the previous level's parent node, which is the root node (7, 2). Determine if the root node's other child hyperplane intersects with the circle. Since there is no intersection, do not visit the child nodes of the other branch and the search terminates with the nearest neighbour (2, 3).

For the KNN search, a max-heap is utilized to store and order the current K points based on their distance from the query point. If the heap contains fewer than K points, continue visiting nodes to find potential neighbors until the heap contains K points.

KD-Tree Challenges: However, the time to construct a KD tree is much longer than the search time. As shown in Figure 2.3, the building process is almost 400 times slower than the search. Thus, when processing moving data, this tree must be rebuilt every time new data come in, leading to a rapid degradation of performance; in addition, constructing indices for points that are significantly distant from the query is unnecessary, since their chances of being the nearest neighbors are negligible. Moreover, KD-Tree is inefficient on high-dimension data due to the curse of high dimensionality [Vishwakarma, 2023].

2.1.2 Ball-Tree

Ball-Tree is a variation of KD-Tree that performs well on high-dimension data where KD-Tree is not efficient. The structure of a Ball-Tree is similar to that of a KD-Tree, with both being binary trees, but they differ in some details.

Ball-Tree Construction: The root node is selected as the central point that minimizes the distance to all points, and then, with the distance from this point to the farthest point,

draw a hypersphere. To select the children, the approach is as follows: **(1)** Select the point farthest from the center within the current hypersphere as the left child node. **(2)** Choose the point that is farthest from the left child node as the right child node. **(3)** For the remaining points, calculate the L2 distance to the points corresponding to the left and right child nodes, and assign them to the closer one's hypersphere. **(4)** Apply the same operation to all child nodes.

Ball-Tree Search: The search in Ball-Trees is similar to that in KD-Tree, but Ball-Tree includes an additional search radius R. This means that the nearest neighbor must be within a hypersphere of radius R generated around the query point. In a KD-Tree, the search process includes determining the intersection between a circle, representing the search radius around the query point, and the hyperrectangle that defines the boundaries of the current node. However, in a Ball-Tree, the search is based on the intersection between two hyperspheres: one that encloses the points within the current node of the Ball-Tree and another formed by the query point. The other searching processes are the same. Ball-Tree search can fit more closely around a cluster of data points because it bounds data in all directions equally using the hypersphere. This fitting tends to enclose points closely, reducing the chance that regions assigned to different nodes overlap each other. Unlike KD-Tree, which partitions the space at median points along axes, Ball-Tree does not partition the space evenly, but rather based on the distribution of data within the space. This can lead to more balanced partitions, especially in unevenly distributed datasets.

Ball-Tree Challenges: The Ball-Tree algorithm requires a significant number of distance computations for both constructing the tree and conducting searches, which proves to be inefficient for data with low dimensions. Additionally, similar to the KD-Tree, it suffers from too long index construction time, which is not suitable for dynamic data tasks.

2.2 Current KNN on CPU

The advantage of CPU-based KNN is its flexibility and efficiency for datasets of small or moderate size and dimensionality. CPU-based implementations utilize complex data structures like KD-Tree or Ball-Tree to facilitate efficient nearest neighbor searches by partitioning the dataset into subsets and reducing the search space. This approach decreases the computation requirement compared to the Brute-Force method where the query point needs to calculate distances with all data points. Moreover, CPUs are well-suited for algorithms that require sequential processing and can benefit from the accurate branch prediction and out-of-order execution capabilities of modern CPUs [Majkowski, 2021]. Libraries such as scikit-learn [Pedregosa et al., 2011] provide highly optimized, user-friendly interfaces for KNN on CPUs with both Brute-Force and tree-based algorithms. These implementations also utilize multi-core processors to make computations in parallel.

2.3 Current KNN on GPU

Implementing KNN on GPUs makes massive parallelism realistic, largely speeding up computations for large datasets and high-dimensional data. GPUs are experts in handling data-parallel tasks, allowing for the simultaneous calculation of distances between a query point and multiple points in the dataset. It is particularly beneficial for the Brute-Force approach, where each GPU thread can compute the distance to a data point in parallel, which reduces the overall latency. Furthermore, GPUs can employ Approximate Nearest Neighbor (ANN) Search algorithms, which offer a trade-off between accuracy and speed, returning most of the correct nearest neighbours, enabling real-time search capabilities in extremely large datasets. Libraries like Faiss [Johnson et al., 2021] specialize in GPU-accelerated similarity search, providing optimized implementations that take full advantage of GPU architecture, including efficient memory usage and parallel sorting algorithms to identify the K nearest neighbors. The main advantage of GPU-based KNN lies in its ability to handle very large datasets and perform computations faster than CPUs. Faiss introduces `Flat` algorithm for exact KNN search, we will introduce it in the next subsection.

2.3.1 Faiss Flat

Faiss (Facebook AI Similarity Search) [Johnson et al., 2021] is a library developed specifically to search for similarities in large datasets of vectors. Examples of areas where Faiss would be beneficial include artificial intelligence and machine learning tasks, as this type of operation is a cornerstone of many tasks, often in the context of a nearest neighbor search in very high-dimensional spaces. Facebook AI Research (now Meta AI Research) created Faiss to be highly efficient in the first place: **(1)** It scales to handle an extensive number of vectors. **(2)** It maintains high search speed and acceptable accuracy for billions of vectors and more. The core features of Faiss include:

Speed & Efficiency: Faiss is highly optimized for any hardware, including CPUs and GPUs, and fully utilizes the potential of hardware to query vectors. For even large sets of data, Faiss is tested faster compared to the naive implementation of KNN searching.

Versatility: Faiss supports several types of the selected metric, such as L1, INNER_PRODUCT and L2, and is applicable in various problems specific to similarity selection or clustering.

Resilience: Faiss consistently works at peak performance levels regardless of the volume of the dataset used due to its efficient K-selection strategies.

For the exact KNN search, Faiss offers a method called `Flat` that provides a Brute-Force index search on all vectors. The vectors are saved in their original format, and no compression or transformation is maintained. At search time, the flat index measures the distance between the query vector and all vectors in the database to determine the K-nearest neighbors.

Acceleration of Faiss Flat: Although `Flat` operates as a Brute-Force approach, it still gains acceleration from the easy implementation of parallelism and the high SIMD capabilities of GPUs, which can exceed the corresponding CPU performance by 10X to

100X. Furthermore, Faiss implemented *In-register sorting*, a variation of the *Bitonic sorting network* [Batcher, 1968], which is in a position of speeding up sorting arrays and K-selection in a parallel manner; in addition, *GPU WrapSelect*, a method of K-selection designed to accelerate the K-selection algorithm by maintaining state entirely in registers and avoiding cross-warp synchronization on GPUs. Each GPU warp (a group of threads that execute in lockstep) is dedicated to K-selection for one of the n arrays, and large arrays per warp are handled through recursive decomposition, assuming that the array lengths are known in advance. These two techniques enable significantly faster performance in KNN search tasks by utilizing the parallel computing capabilities of GPUs more effectively than previous methods.

Faiss Flat Challenges: Similar to other Brute-Force searching methods, Faiss Flat performance degrades considerably as the dataset size and data point dimensionality ascend. As a result, it is not suited for extensive datasets in which the exact K nearest neighbor search is required.

2.4 Problems with Current CPU and GPU KNN

For exact KNN algorithms, both CPU and GPU are still facing several challenges:

Intensive Computation: For Brute-Force, obtaining an exact KNN search requires calculating the distances between the query point to the entire database, which can be compute-intensive and infeasible on many platforms, especially with large databases and complex search domains. For KD-Tree and Ball-Tree methods, constructing the trees requires considerable time with the tree-building phase potentially taking up to 400X longer than the search phase. The intensive computation also causes large energy consumption by CPU or GPU, as a result of the high utilization rates of these chips.

Redundant Computation: Points far away from the query physically are meaningless in KNN and calculating distances with them or building tree indices for them are redundant. However, because point datasets are usually sparse and do not provide closeness information among points, figuring out which points are near each other is difficult. Several studies have attempted to conduct fixed-radius KNN searches on individual points [Chen and Güttel, 2024]. However, these approaches quickly lose efficiency as the volume of queries grows. Due to the centralized management of memory in CPUs and GPUs, implementing concepts like a search radius or conducting calculations in a distributed manner, where each point operates within its unique range of the GPU memory, poses a challenge.

ANN Cannot Replace KNN: Numerous studies advocate the use of highly optimized Approximate Nearest Neighbor (ANN) Search [Groh et al., 2023, Chern et al., 2022, Guo et al., 2020, Malkov and Yashunin, 2018], over exact KNN search, arguing that identifying the majority of nearest neighbors is enough for large databases. Nevertheless, in domains such as AI-driven Particle Dynamics Simulation, where the attributes of neighboring particles are crucial, an wrongly identified neighbor could result in unsuccessful simulations. Similarly, in the banking and finance industry, high-precision detection of fraudulent transactions is essential to avoid financial losses. Utilizing exact KNN search is critical for accurately identifying transactions that derail from

a customer's normal behavior, as even slight errors could lead to significant financial losses. Therefore, an efficient exact KNN search is still of importance.

2.5 KNN on 2D-Mesh Accelerator

Implementing KNN on 2D-Mesh accelerators holds the potential to overcome these challenges. These accelerators are equipped with plenty of cores that are designed to enhance matrix operations. These cores are interconnected through the networks-on-chip which forms a 2D-Mesh layout, offering massive on-chip memory bandwidth. We will introduce the 2D-Mesh accelerators in the next session.

2.5.1 2D-Mesh Accelerators

Spatial accelerators are an integrated circuit architecture that provides a well-defined structured and scalable framework to integrate communication and computation. They are widely employed in Machine Learning tasks like Autonomous Driving [Mohanarangam and Shetty, 2022], Large Language Model Training [Thangarasa et al., 2024, Kosson et al., 2021, Thangarasa et al., 2023], Brain Tumor Segmentation [Pendse et al., 2021] and AI-driven Molecular Simulation [Brace et al., 2021]. A 2D-Mesh accelerator contains cores, also called processing elements, which are arranged like a grid to create a two-dimensional mesh network. These cores can autonomously perform tasks and are linked by a communication network called Network-on-Chip (NoC) [Serpanos and Wolf, 2011]. Each core has its own SRAM for instant data storage and fast execution.

Distributed & Scalable: A key advantage of the 2D-Mesh accelerator is its regular and predictable communication. Each core is attached directly to its four neighboring cores by NoC. Data can flow directly and in position between these cores, and data can be sent to different directions simultaneously called `multi-cast` [Lan and Chen, 1994], which is essential for parallel algorithms and applications that require parallel processing and data movement. Each core operates independently, equipped with its own resources, ensuring a fully distributed execution without synchronization overhead. Users have the flexibility to choose a core and define customized routes to configure communications with other cores. The 2D-Mesh accelerators also allow for scalability [Jain et al., 2019]. Assuming that the accelerator is planned to increase, the mesh network can be expanded. Hence, more cores can be added to the enlarged mesh. As a result, the computational power and capacity of the system will increase. 2D-Mesh accelerators are thus ideal for various applications, ranging from small-scale embedded systems to large-scale supercomputing systems.

Figure 2.4 shows the structure of a 2D-Mesh accelerator. Initially, users distribute data to each core according to their configuration, and subsequently, data movement occurs as needed. Data is transferred through a Network-on-Chip that is connected to the router (R) of each core. The network interface of the core communicates with the router. If data is required for this core, the data will be passed into the core for task execution. Otherwise, the data will be transferred to the next core determined by the router and through the NoC.

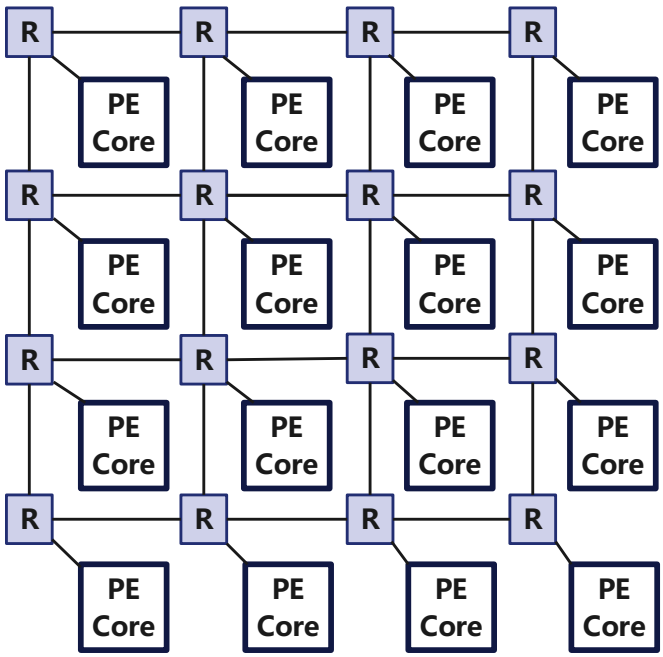

Figure 2.4: 2D-Mesh Accelerator Architecture

Fast Core Communication & Flexible Message Passing: The communication design of the 2D-Mesh accelerator makes the data can almost instantly be made available for neighboring cores due to direct transferring through NoC without any complex strategy, which is beneficial for the applications with frequent data exchange between cores. For message passing, the 2D-Mesh structure of the accelerator can support both synchronous and asynchronous models because the router is designed to be asynchronous [Guan et al., 2010], which means that the cores could coordinate operations at specific synchronization points or exchange data independently. This ensures maximum flexibility and performance of parallel algorithms due to lower synchronization overhead and greater efficiency. A core communication example is described in Section 3.4.2.

2.5.2 Tenstorrent

There are plenty of 2D-Mesh structured accelerators. In this thesis, we refer to the datasheet of *Tenstorrent Grayskull*^{TM} e150 [Tenstorrent, 2024]. The cores in this accelerator are known as *Tensix*, and they are placed and connected using two torus-shaped NoC [Adiga et al., 2005], which enables bidirectional communication between neighboring cores. Additionally, the torus-shaped NoC allows cores situated on the head and tail of the same row or column to communicate with each other, further enhancing the connectivity and communication capabilities of the system. Each Tensix core contains a high-density tensor math unit (FPU) which performs most of the heavy lifting, a SIMD engine (SFPU), five Risc-V CPU cores, and a large local memory storage (SRAM). There is also a package manager on each Tensix core for handling data transfers, storage, and manipulation.

Figure 2.5 shows the construction of the Tensix core package manager. The Data

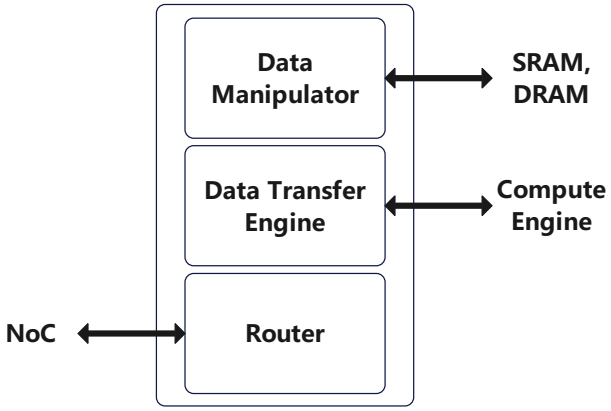

Figure 2.5: Tensix core's Package Manager

Manipulator is positioned in line between the memory and the compute engine, allowing it to preprocess data and communicate with SRAM and DRAM. The Data Transfer Engine works with the run-time software, triggering computation when data is ready. The router in the Package Manager is deadlock-free, ordering-guaranteed, and able to handle multiple data in parallel.

2.5.3 Benefits of 2D-Mesh KNN

The 2D-Mesh design of these accelerators allows for sparse data to be distributed among specific cores for independent processing which reduces the computational workload on individual cores, ensures parallel processing, and allows each core to operate at its maximum capacity. Furthermore, the concept of a radius is applicable in this setup, as the cores are arranged in a 2D plane. When utilizing a 2D-Mesh accelerator for KNN, it's only necessary to send the query to cores lying within a specified radius, enabling each core to independently execute KNN calculations in a distributed manner. Additionally, cores not involved in computation remain inactive, thereby not contributing to energy consumption. This not only ensures efficient processing but also reduces energy wastage in performing KNN tasks.

Chapter 3

Methodology

SR-KNN is driven by several design goals aimed at improving its performance and efficiency. These goals include:

- Integration of spatial closeness properties: We incorporate insights into the spatial closeness properties of the data into the KNN algorithm and partition them accordingly.
- Communication strategy and data reduction: To avoid heavy traffic and bandwidth bottlenecks on a single channel, we design a communication strategy that optimizes data transfer in the system. Additionally, we incorporate a Reduce technique that reduces the amount of data transmitted back to the start, thereby enhancing the overall throughput of the system.
- Efficient on-chip computation: We design an on-chip computation method that can minimize the time required for processing distance computation and improve the algorithm's overall efficiency.
- Fast and correct K-selection: We employ the Reduce function and perform it atomically at the starting core to aggregate and select the K nearest neighbors in order to maintain corretness.
- Feasibility and effectiveness: To facilitate the evaluation and theoretical performance analysis of the SR-KNN algorithm, we implement a simulator. This simulator emulates the operations of a real 2D-Mesh device. It allows us to assess the feasibility and effectiveness of SR-KNN to be implemented on a real 2D-Mesh accelerator.

To achieve these objectives, we have structured our algorithm to operate under the following workflow.

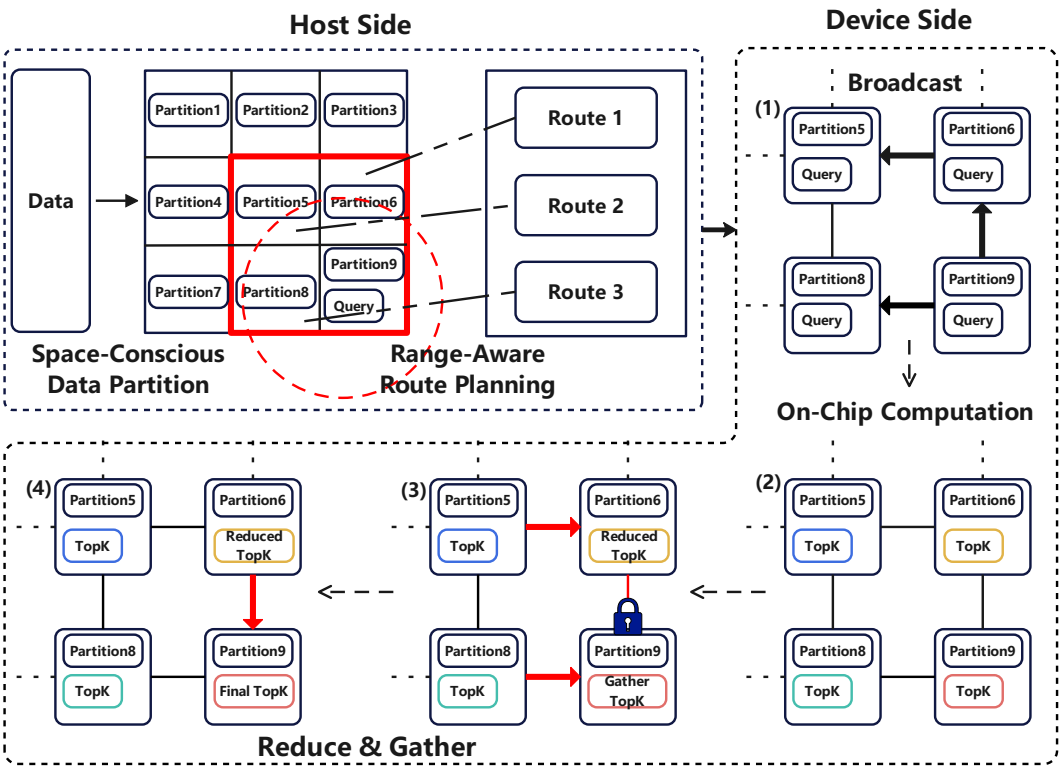

Figure 3.1: SR-KNN Design

3.1 Workflow

Figure 3.1 illustrates the workflow of our SR-KNN. Initially, on the host side, data is organized using a Space-Conscious Data Partition, which maps the closeness of each data point. Following this, Range-Aware Route Planning is implemented, utilizing the established closeness and a predefined radius to identify specific cores on the 2D-Mesh accelerator that are required to compute this particular query. Subsequently, the data and query are allocated on the device according to the partition layout. On the device side, the starting core (the core initially containing the query) broadcasts its query to all target cores using asynchronous core communication (indicated by thick black arrows). Upon receiving the query, each target core conducts On-Chip computation to determine its local K nearest neighbors. These local neighbors are then sent back to the starting core. During this return journey, a merge-sort operation occurs at each passing core to combine the incoming local neighbors with its own, thereby updating the K nearest neighbors, called Reduce. Upon reaching the starting core, it merges with the starting core's local TopK in a secure, atomic (locked) manner, called Gather, requiring others to wait until its completion. Finally, the process concludes with the last aggregation of neighbors from one of the neighboring cores, resulting in the final K nearest neighbors. We will introduce each part of the workflow detailedly in the following sections.

3.2 Space-Conscious Data Partition

In recent times, point datasets are sparse and commonly presented as collections of data points. These points are organized in a format denoted as (N, D), where N indicates the total number of points, and D specifies the dimensionality of each point. *(In this thesis, we focus solely on datasets where the dimensionality, $D = 2$)* This architecture enables the depiction of every data point across D dimensions, concentrating on the features or attributes of the points instead of their physical or spatial locations. In other words, it does not provide information about the closeness of the points within the space. However, 2D-Mesh accelerators are designed with cores interconnected on a 2D plane, allowing each chip to process points that fall within its range. Therefore, we introduce a method to organize the dataset into blocks to form a shape (R, C), where R indicates the number of rows in the 2D-Mesh layout and C indicates the number of columns of the 2D-Mesh layout. Each point will be placed to the corresponding block based on its transformed coordinate. This approach enables the categorization of data points into a structured 2D plain format, facilitating the analysis and processing of spatial relationships and closeness of points.

Figure 3.2 explains the details of our data separation strategy, and its detailed code is shown in Appendix A.3. It begins by transforming all points such that the point with the minimum x and y coordinates moves to the origin (0, 0). This transform is achieved by subtracting the minimum x and y values from all points. After transforming, the maximum of the x-axis divided by the layout size R on the x-axis, and the maximum of the y-axis divided by the layout size C on the y-axis, yields $block\_size\_x$ and $block\_size\_y$. Then each point's x and y coordinates are divided by the $block\_size\_x$ and $block\_size\_y$

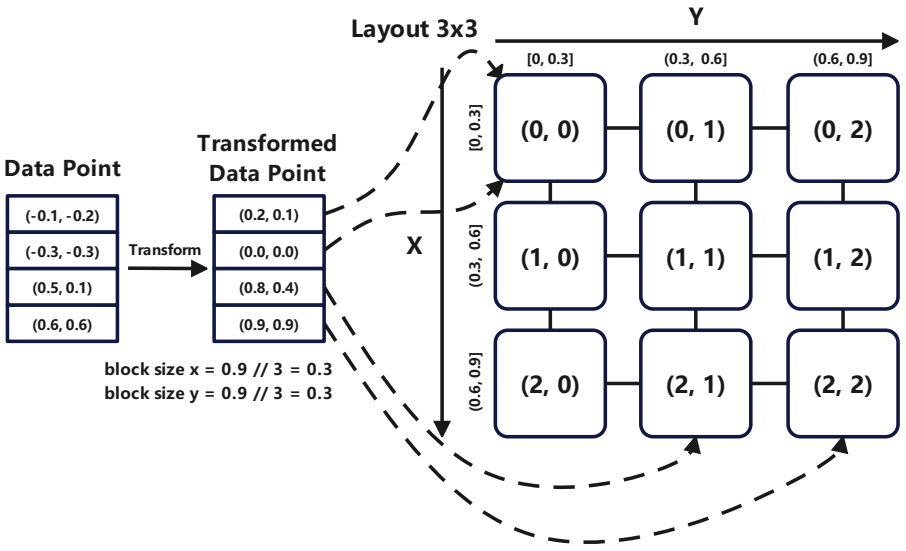

Figure 3.2: An explanation of the approach for spatial data partition. It first converts the original points to all non-negatives (starting at (0, 0)), and then uses $max\_x//layout\_size\_x$ and $max\_y//layout\_size\_y$ to determine the block_sizes on x and y, respectively. Lastly, it inserts these points into the block of which the point is inside its range based on the absolute coordinates of these points.

respectively to determine which block the point belongs to. The division here uses floor division ("//" in Python), ensuring that the result is an integer that represents the block index rather than a floating-point number. This process effectively groups the points into discrete blocks based on their location, with each block identified by a pair of indices (*block_x*, *block_y*). Finally, the function combines these block indices for all points into a single array for compatibility with array indexing or further processing that requires them.

Moreover, this method is also employed to partition point queries, i.e. those awaiting K-nearest neighbor calculations. This means that data points that are near a given query point are prioritized for calculation. This method improves the efficiency of query processing by first allocating computational resources to data points near the query point. It utilizes spatial locality in 2D-Mesh accelerators, allowing cores to process their corresponding points in a distributed manner.

3.3 Range-Aware Route Planning

For fixed-radius KNN, each query is processed by a number of cores within a specified range, determined by the dataset or user-defined settings. Setting the query point at the center of a circle with radius r, the circumference of this circle intersects various cores. The cores that lie within the circle's boundaries are the destinations the query is dispatched to for processing. As illustrated in Figure 3.3, consider two query points, $P1$ and $P2$, positioned within a grid of cores. A circle with radius r is drawn around each

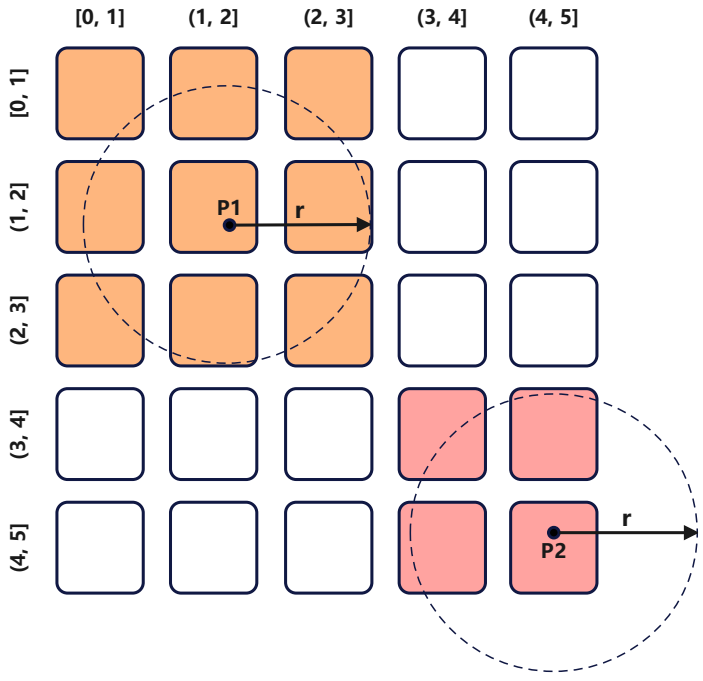

Figure 3.3: Cores that intersect with a circle drawn from the query point with a specified radius r are chosen as the target destinations for the query to be dispatched to.

query point. The circle around $P1$ encompasses the cores colored in orange; hence,

these orange cores are the ones to which $P1$ is sent for computation. Similarly, the circle around $P2$ passes through the pink-shaded cores, and therefore $P2$ is sent to and processed by all the pink cores.

Once the target cores for a query have been identified, designing an effective routing strategy becomes crucial. In a 2D-Mesh accelerator where each core is connected to its adjacent neighbors, we suggest a routing method based on the relative position of the destination cores to the initial core, which is treated as the reference point. As illustrated in Figure 3.4, data packets are sent directly along the x or y-axis from the starting core to the target cores if the target core is on the x or y-axis of the reference core. For instance, the core at (0, 1), directly above the starting point at (1, 1), would receive the data after a single upward step.

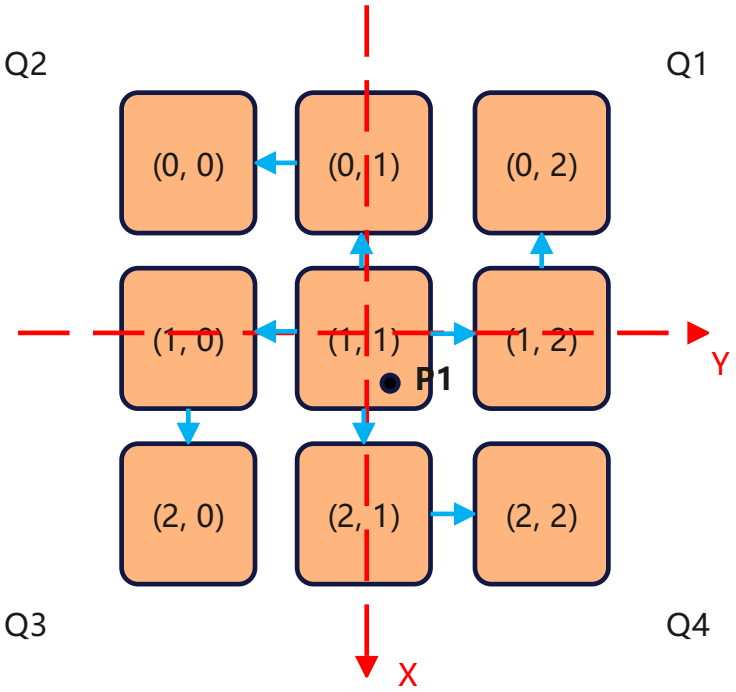

Figure 3.4: Quadrant-based routing is illustrated with blue arrows marking the path and direction. The core at (1, 1) serves as the origin for dispatching queries to designated target cores within the coordinate system from this reference point.

For destination cores that do not align with the axes of the reference core, the routing strategy varies. To reach cores in the first quadrant of the reference system, the data would first travel right to align horizontally with the target core, followed by an upward movement to the final destination. For example, reaching the core (0, 2) from (1, 1) would involve a rightward step to (1, 2), then upward to (0, 2).

The cores in the second quadrant would be approached by first moving upward to align vertically with the target and then moving leftward. For example, the route to core (0, 0) from (1, 1) would contain an upward step to (0, 1), then a leftward step to (0, 0).

Similarly, for cores in the third quadrant, the route involves moving left from the reference core initially, followed by downward steps. On the other hand, reaching the

cores in the fourth quadrant requires an initial downward movement from the reference core, then turning right to arrive at the destination.

This routing strategy benefits from a faster and wider transfer of data by avoiding congestion in a single direction, which could lead to bandwidth bottlenecks and reduced communication efficiency. This is achieved by ensuring that data transmission is evenly distributed in all directions and reducing the data transmission workload for each core.

3.4 Broadcasting and Communication

3.4.1 Broadcasting

Having successfully constructed routes and segregated and placed data points and queries to the corresponding cores based on their spatial attributes, the next step involves dispatching these queries to their cores for the computation of distances between the queries and nearby data points, which is a crucial step for subsequent K selection. Upon allocation to the appropriate core, each query is broadcasted across the predefined network of routes to all other cores that need to participate in the distance computation process, as shown by Figure 3.5.

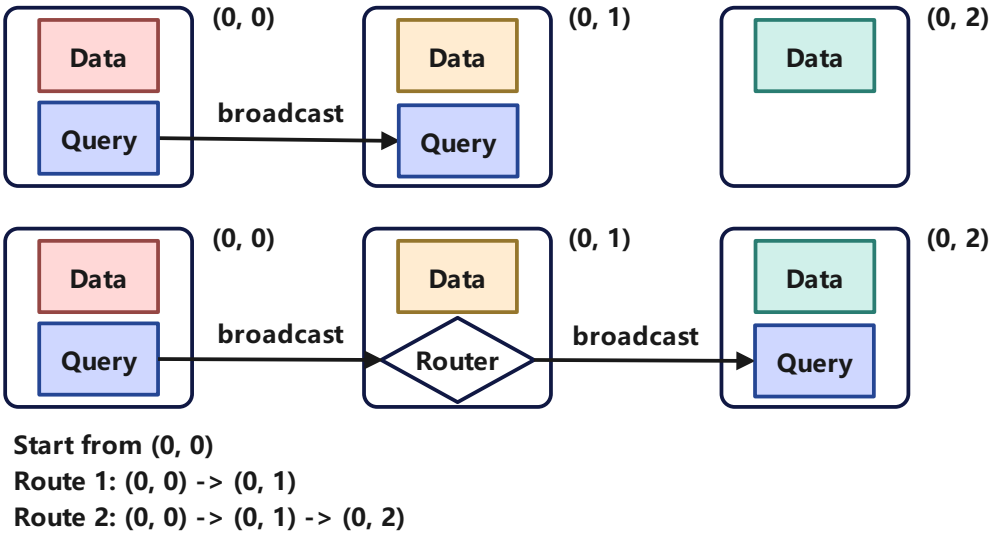

Figure 3.5: The initial core is broadcasting its query to other specified target cores along pre-defined routes. For example, in Route 2, the route involves passing through the core (0, 1). Its packet manager will engage its router to forward the query to the final destination at the core (0, 2).

This strategy maximizes the benefits of parallel processing on 2D-Mesh architecture-based accelerators, allowing simultaneous calculations across multiple cores. The 2D-Mesh accelerators offer a distinct advantage that each core is equipped with its own SRAM, ensuring operations by one core do not interfere with another. This architecture inherently prevents race conditions, as each calculation is isolated. Thus, by utilizing the unique capabilities of 2D-Mesh accelerators, we can achieve highly efficient and

concurrent computations, significantly enhancing the processing speed and reliability of the system for KNN.

3.4.2 Core Communication

By utilizing a pre-defined routing scheme, each query is transmitted across the Network-on-Chip (*NoC*). This *NoC* essentially acts as a pipe, linking two neighboring cores and facilitating the close transfer of data between them. As a query traverses this network, it is relayed from one core to the next through the *NoC*. Upon reception, the packet manager of the core will assess whether the incoming data is intended for its own processing tasks. If the data is relevant, the manager forwards the data to the computing processor for on-chip calculation. Otherwise, the manager will call the router to send the data through the *NoC* to another core based on the pre-defined route.

This iterative process of data transmission continues until the data reaches its destination core. This method allows for the data transfer and computation processes to occur asynchronously, making full use of both computational resources and data bandwidth.

3.5 On-Chip Computation

Once the core receives the query data intended for it, the packet manager instructs the packet computation processor to carry out distance calculations between the query and the data points stored in its SRAM. All operations happening in this core include:

- Calculating the L2 distances among query points and data points.
- Arranging the distances in ascending order from the smallest to the largest
- Selecting the K smallest distances and their associated indices from the original dataset.

To minimize the computational effort involved in calculating L2 distances, we skip the square root step since our goal is not to find the precise distances but rather to identify the top k smallest neighbors. Furthermore, to enhance performance, we apply SIMD operations [Wikipedia, 2024] (Single Instruction Multiple Data) to both the distance calculation and the K selection process. This approach allows the algorithm to be accelerated by performing multiple operations in parallel.

In a single core, suppose that the queries, denoted as $\mathbf{Q}$, are organized in a matrix of shape $M \times D$, and our data points, denoted as $\mathbf{P}$, in a shape of $N \times D$. Additionally, we maintain the indices $\mathbf{A}$ of the data points from the original dataset, arranged in a shape of M, to facilitate the sorting of indices.

The algorithm 1 shows the computation process executed on a single core. Without SIMD operations, the processing time would be $O(MND + MNK)$, where

- The distance computation for D dimensions is $O(MND)$.
- The K comparisons add an additional $O(MNK)$.

Algorithm 1 SR-KNN On-Chip Computation

Input: $\mathbf{Q} \in \mathbb{R}^{M \times D}$
Input: $\mathbf{P} \in \mathbb{R}^{N \times D}$
Input: $\mathbf{A} \in \mathbb{R}^{M}$
Output: $\mathbf{D} \in \mathbb{R}^{M \times K}$ **TopK smallest distances**
Output: $\mathbf{S} \in \mathbb{R}^{M \times K}$ **TopK smallest indices**
for $i \leftarrow 1$ to **M do** ▷ Vectorized
    for $j \leftarrow 1$ to **N do**
        $y_{i,j} \leftarrow L2\_Distance(\mathbf{q}_i, \mathbf{p}_j)$
        for $k \leftarrow 1$ to K **do** ▷ SIMD Comparisons
            if $y_{i,j} < d_{i,k}$ **then**
                $d_{i,k} = y_{i,j}$
                $s_{i,k} = a_j$
            end if
        end for
    end for
end for

Through vectorization and SIMD operations, this algorithm can achieve faster performance by utilizing the SIMD capabilities, which allow for more than one operation per instruction cycle. Therefore, optimally, SIMD operations can process all D dimensions of the distance calculation and K comparisons in parallel which results in a processing time close to $O(MN)$.

The outputs **S** and **D** will be retained within their own SRAM to be available for any later comparisons with data from other cores, which will be introduced in section 3.6. The detailed single core operations are stated in Appendix A.4.

3.6 Back Transferring and Reduce

After a core completes its calculations on-chip, it sends the resulting matrices, **S** and **D**, back to the starting core through the reversed route to carry out the final selection of the top K smallest values. However, if every core related to the starting core transmits its full sets of **S** and **D** back to it, this can lead to high data transfer load and create a communication bottleneck. In particular, the NoC linked to the starting cores will experience significant stress, as it will have to manage large volumes of data coming in from all neighbor cores.

To tackle this problem, each query is given a unique ID, and we introduce a *reduce* technique to be used between pairs of cores acting as the sender and the receiver. As shown in Figure 3.6, a core with computation results sends its TopK to the next core following a reverse route. The recipient core will perform a merge-sort, combining the received TopKs with its own local TopKs that have matching IDs. It then stores the merged TopKs, along with any others with non-matching IDs, in its SRAM. After this step, the recipient core forwards the newly merged TopKs to the subsequent core on the route.

The merge-sort process is highly efficient since both sets of candidates are already sorted arrays, so it is not necessary to divide each array in half and then sort them, an operation that would lead to a time complexity of $O(2K\log(2K))$, given that each array has a length of K. Therefore, this efficient merge-sort algorithm for two sorted arrays has a time complexity of $O(K)$ since it only requires comparing elements from the beginning of each array a total of K times. If the received TopKs contains L TopK, the complexity of the *reduce* process is $O(LK)$ and with parallel processing, the time can be close to $O(K)$. We will explain the details of the merge-sort implementation in Section 3.8.3.

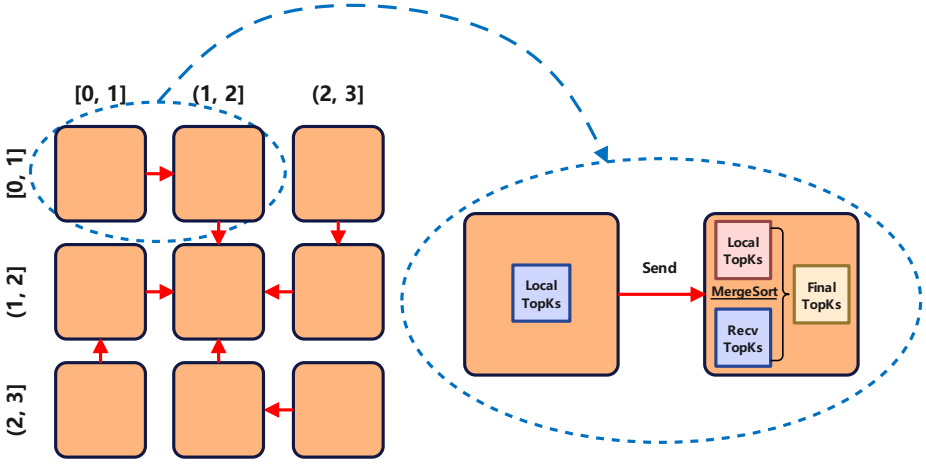

Figure 3.6: Back transferring the TopKs through the reverse route (red arrows) and applying the merge-sort to reduce the TopKs by pairwise merging them between adjacent cores along the reverse route.

3.7 Gather and Final K Selection

As the TopK elements from the four directions reach the starting core, they must be merged into a single sorted list for the final K selection. However, synchronizing the arrival of these data packets from all directions to occur simultaneously is challenging. Consequently, the TopKs will arrive in order. Upon the arrival of each TopK elements, the starting core will apply the merge-sort, as detailed in Section 3.6, with its own TopK elements and those just received.

However, while the merge-sort operation is in progress for one pair of TopK sets, additional sets may arrive from other directions, seeking to be merged with the starting core's local TopKs. This concurrent arrival can lead to a race condition, potentially resulting in wrong sorting outcomes. To manage this problem, the starting core's local TopK elements are secured with a locking mechanism as shown in Figure 3.7. With this lock in place, the local TopK elements become inaccessible for merge-sorting with newly arrived TopKs from other cores until the previous merge-sort operation is complete and the lock is released. This ensures that merge-sort operations are applied in an orderly fashion, preventing race conditions and preserving the correctness of the sorted output. The final selection of the TopK elements is determined by the outcome

of the last merge-sort operation.

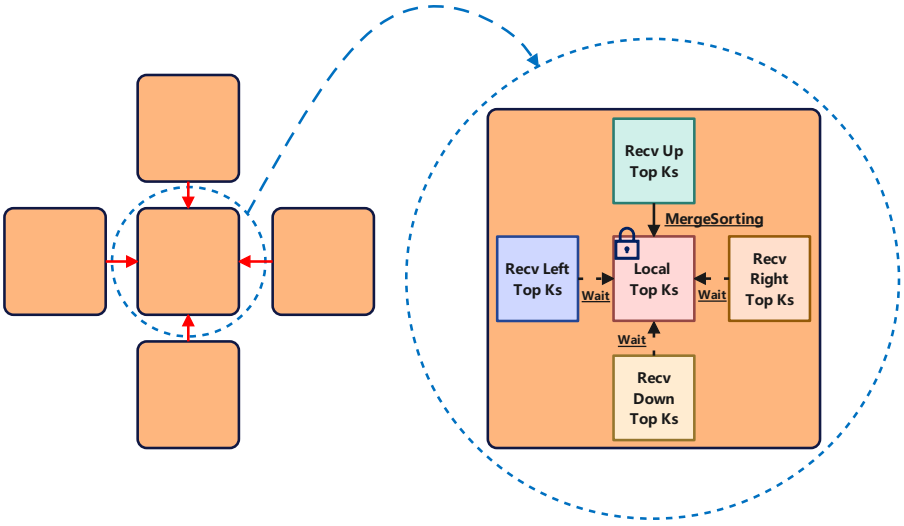

Figure 3.7: The gathering process at the starting core. While a merge-sort is being applied between the local TopKs and a set of received TopKs, the local TopKs are locked to prevent race conditions

The SR-KNN algorithm reaches completion once all the cores involved in the gathering process have finished their operations.

3.8 Simulator Implementation Details

3.8.1 Data Partition Concerns

To implement the Space-Conscious Data Partition (Section 3.2), When considering how to manage the transformation of original data from a format of (N,D) to (R,C), it's crucial to select an appropriate data structure. In Object-Oriented Programming languages such as Python, C++, and Java, utilizing a 2D Object Array is often the preferred approach. This choice allows for the definition of a custom structure with an initial state and we can allocate a 2D array to store this structured data whose indices can be regarded as the coordinate of each block (core) in the layout.

In Python, for enhanced operational speed and ease of implementation, the NumPy Array [Harris et al., 2020] is the data structure of choice. Using NumPy's vectorization capabilities and high-level data structure API allows for efficient manipulations. A single 2D NumPy Array can be defined for this purpose using the command data_blocks = np.zeros((R, C), dtype=Object). Here, "data_blocks" represents the partitioned results inside a $R \times C$ layout.

Later, we can use indices to show the data partitioned to the block (i, j) with data_blocks[i, j].

3.8.2 Core Simulation Concerns

Core Layout: We use Python's *multiprocessing* package to represent each core as a *Process*. These *Process* instances are arranged within a 2D array to create a 2D-Mesh layout, allowing each *Process* to be accessed through its row and column indices.

NoC Simulation: In our approach, the *Queue* class from the *multiprocessing* package serves as the NoC for communication between each core and its neighboring cores. Data transfer is accomplished through the *Queue*, with each core utilizing the *put* method to send data and the *get* method to receive data.

Packet Manager and Packet Compute Engine: We design each process to continuously poll its associated queues for incoming data. When data is placed into the queue, the process initially acts as a Router to get the data and determine if the data is intended for itself. If the data is meant for the current process, the process then functions as a Data Transfer Engine, forwarding the data to subsequent tasks. Before initiating the computation process, each process preloads its corresponding data points, which match the coordinates specified in the data_blocks, into its allocated memory space, similar to SRAM, and then the process operates as a Tensor Manipulation unit, retrieving data from this memory space. It then engages the Data Transfer Engine to send the data to the task for completing the computation. Once the computation is finished, the results are sent back to the Manager. At this stage, the process assumes the role of a Router again, responsible for forwarding the computed results back to the starting process. If there are multiple data got from the queue, the process can execute them concurrently using multithreads.

3.8.3 Merge-Sort Details

Our merge-sort algorithm is based on two sorted lists by comparing their elements, tracked by indices i for the first list and j for the second. It iterates through both lists, selecting the smaller of the two current elements to add to a merged list, thus maintaining sorted order. The process continues until either the merged list reaches the length of K or both input lists have been fully traversed. If any elements remain in either list after the main loop and there are fewer than K elements in the merged list, these elements are appended to the merged list until it reaches the size of K. The implementation code is shown in Appendix A.4.

The merge-sort algorithm can be adapted for vectorization, enabling it to execute SIMD operations. This enhancement is particularly useful when sorting the results of multiple queries simultaneously. By utilizing vectorized instructions, the algorithm can process multiple elements across different queries in parallel, increasing the efficiency and speed of the sorting process.

Chapter 4

Experiment

This chapter offers a comprehensive evaluation of SR-KNN, covering three key aspects: **(1)** assessing the computational FLOPs of our SR-KNN and comparing it with other baseline methods, **(2)** examining the scalability of SR-KNN by increasing the number of cores. **(3)** evaluating SR-KNN performance against other KNN methods implemented on multiple platforms like GPU and CPU.

4.1 Dataset

4.1.1 Balanced Random 1M Dataset

We created a random 1M Dataset using functions in NumPy Random [Harris et al., 2020]. These functions can be used to generate points, uniformly spread in the 2D space. This method guarantees that each block (core) within the space receives an equivalent number of points.

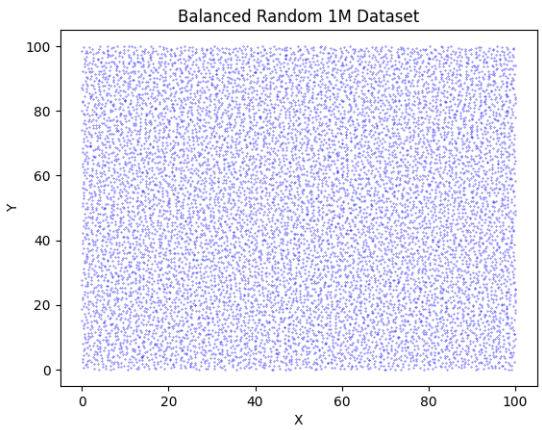

Figure 4.1: Visualization of Balanced Random 1M Dataset. The points inside this dataset are evenly distributed across the entire space. This makes sure that the data points partitioned inside each block (core) is equivalent.

The distribution of these points, as visualized in Figure 4.1, shows an even spread across

the whole space. We adjust the search radius to the minimum size that yields the same outcomes as a brute force search.

4.1.2 Imbalanced GNS 1.1M Dataset

We utilize the largest dataset, *SandRamp*, from the GNS dataset [Sanchez-Gonzalez et al., 2020]. The *SandRamp* dataset contains 400 frames, recording the movements of 2,800 sand grains within a fixed area. As illustrated in Figure 4.2, the positions of the sand grains vary between two selected frames, showing a biased and uneven distribution within the space, a condition called imbalanced. To form the extensive 1.1M dataset, we concatenate all frames, resulting in a large and imbalanced dataset where the distribution of points across different blocks varies, and some blocks even have no point, as no sand grain is within their areas. The search radius is given by the GNS dataset.

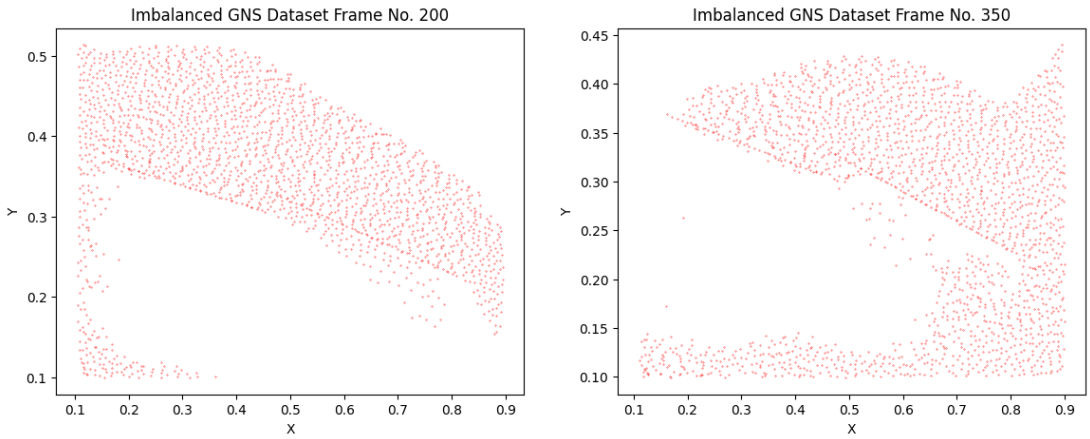

Figure 4.2: In the GNS *SandRamps* Dataset, frame 200 and frame 350 exhibit an imbalanced distribution of sand grains (points) throughout the space, with some areas remaining empty.

Our choice of the imbalanced dataset for evaluation aims to examine how an uneven workload distribution among cores affects the efficiency of our SR-KNN. This strategy helps identify whether workload disparity, where some cores are burdened with heavy computational tasks while others have minimal workloads, affects the overall effectiveness of the algorithm.

4.2 Computational FLOPs Benchmark

We evaluate the total computational FLOPs consumed for each well-known KNN algorithm and SR-KNN on both datasets. We choose Brute-Force, KD-Tree [Bentley, 1975], and Ball-Tree [Omohundro, 2009] KNN algorithms as our baseline. The implementations of KD-Tree and Ball-Tree are presented in Appendix A.1 & A.2. The computational FLOPs are counted when calculations and comparisons on floating points happen. Most FLOPs lie in the calculation of L2 Distance without the square root operation, also known as the Squared Euclidean Distance. Suppose we have a $\mathbf{Q} \in \mathbb{R}^D$

would like to calculate the L2 distance with $\mathbf{P} \in \mathbb{R}^D$. The formula of the L2 Distance without square root is:

$$d^2(Q,P) = \sum_{i=1}^{D} (q_i - p_i)^2$$

The equation illustrates that the squared L2 distance needs D subtractions, D squaring, and $D-1$ additions, which results in a total $3D-1$ FLOPs. If we have M queries calculating L2 distances with N data points with the dimension D, the overall computational FLOPs for distance calculation are $M \times N \times (3D-1)$.

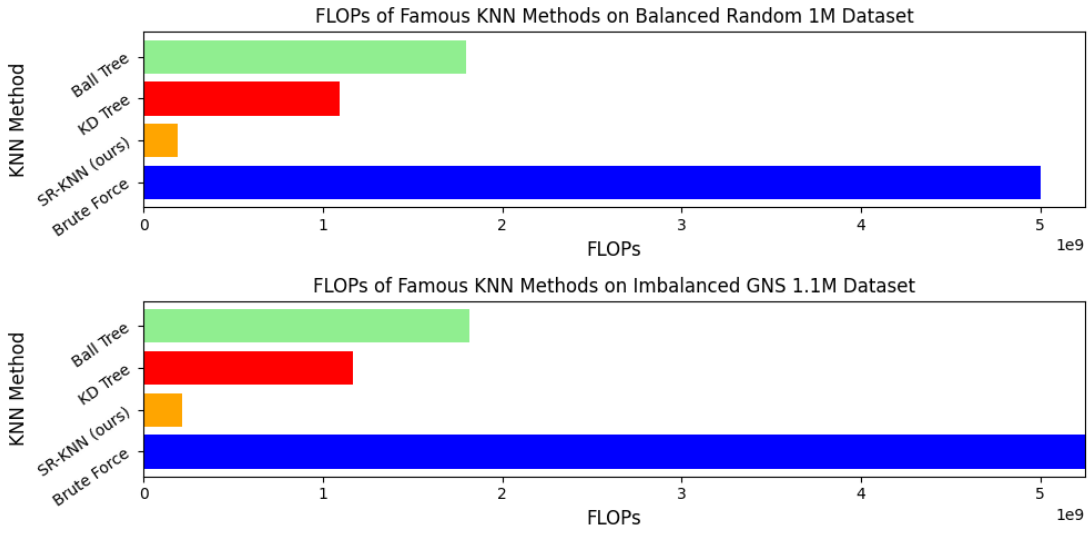

Figure 4.3: Computational FLOPs of our SR-KNN and baseline KNN (k=15) methodologies, the lower the better. We use 100 cores for SR-KNN evaluation.

Figure 4.3 shows that our SR-KNN can reduce a massive portion of computational FLOPs around 96% compared to the Brute-Force KNN, 89 % compared to the Ball-Tree KNN, and 83% compared to the KD-Tree. This tremendous decrease is brought by the fact that SR-KNN only needs to calculate the distances with points within the circle range of each query.

4.3 Performance Estimation

To estimate the performance of SR-KNN through our simulator, we need to consider and estimate the following terms:

- Latency for On-chip computation
- Latency for Data transfer
- Latency for Reduce
- Latency for Gather

These terms constitute the overall latency of SR-KNN. However, it's important to note that there are additional overheads introduced by the hardware, which are challenging

to capture accurately within the simulator. Moreover, our datasets are able to fit within the SRAM of all cores after partitioning, ensuring that during evaluation, there is no need for memory swapping between each core's SRAM and the accelerator's DRAM so that there is no latency due to SRAM cache misses. Therefore, our performance estimation is carried out under an ideal scenario, assuming optimal hardware conditions where all processes run smoothly without these hardware delays.

We use the hardware specification of *Tenstorrent Grayskull*^{TM} e150 [Tenstorrent, 2024] as our example 2D-Mesh accelerator. The statistics can be easily found on [Vasiljevic et al., 2021] and the official website of *Tenstorrent*.

Estimating On-Chip Computation Latency: Referring to Algorithm 1, the latency of the algorithm is primarily affected by reading data points from the SRAM, calculating distances, comparing floating-point numbers, and assigning values. As we utilize SIMD operations, the computation can be conducted under each core's SIMD performance for floating-point numbers which is quantified as FLOPS (Floating-Point Operations Per Second). Thus, the latency of the On-Chip Computation is:

$$T_{on\_chip} = \frac{Data\ Size}{Memory\ I/O\ Bandwidth} + \frac{FLOPs + Comparisons + Value\ Assignments}{FP\ SIMD\ Performance}$$

Estimating Data Transfer Latency: The latency of data transfer depends on the size of the transmitted data recorded in Bytes and the bandwidth of the NoC recorded in Bytes/s. So the latency of data transfer is:

$$T_{transfer} = \frac{Transferred\ Data\ Size}{NoC\ Bandwidth}$$

Estimating Reduce Latency: The latency of the reduce procedure is determined by reading current TopKs from the SRAM, the comparisons for floating-point numbers, and the number of value assignments in the merge-sort algorithm. Due to the usage of SIMD operations, the latency of the reduce is:

$$T_{reduce} = \frac{Data\ Size}{Memory\ I/O\ Bandwidth} + \frac{Comparisons + Value\ Assignments}{FP\ SIMD\ Performance}$$

Estimating Gather Latency: T_{gather} is similar to T_{reduce} as Gather can be seen as multiple Reduce with locks. However, due to its locking mechanism, Gather of the incoming query must wait for the Gather operations of preceding queries in the queue to complete. Therefore, T_{gather} is equal to the T_{gather} of the immediately preceding query plus its own Reduce time, which can be regarded as a Dynamic Programming problem [Bellman, 2021]. If there are currently M queries waiting when a new query arrives, and because the Reduce time for each query is measurable, we can have the state transition equation:

$$i \rightarrow 2\ to\ M$$

$$T_{gather_1} = T_{reduce_1}$$

$$T_{gather_i} = T_{gather_{i-1}} + T_{reduce_i}$$

In our simulator, it has been observed that the queue typically holds a maximum of three queries at any given time. This is because the Reduce operation consistently processes two sorted arrays of equal length K, which ideally should take the same amount of time to complete for every Reduce. Consequently, as a new query enters the system just after a Reduce operation with the preceding core, the Gather for one query on the starting core is expected to be completed at the same time.

Estimating overall latency: As queries are travelled by routes, the total runtime of each route is the combination of T_{on_chip}, $T_{transfer}$, T_{reduce} and T_{gather}. For example, the running latency of the route $(0,0) \rightarrow (0,1) \rightarrow (0,2)$ is equal to $T_{on\_chip} + T_{transfer} + T_{transfer} + T_{on\_chip} + T_{transfer} + T_{reduce} + T_{transfer} + T_{gather}$. Since 2D-Mesh accelerators are fully distributed and enable asynchronous and multithreaded processing in every core, allowing them to execute various tasks simultaneously, the overall latency of SR-KNN is determined by the latency of the slowest route, as everyone needs to wait for it to finish before terminating.

4.4 Scalability

Figure 4.4 illustrates the potential scalability trends of SR-KNN (k=10) when applied to the Balanced Random 1M **(Top)** and Imbalanced GNS 1.1M **(Bottom)** datasets on the Tenstorrent Grayskull accelerator. We investigated how scaling up the number of cores of 2D-Mesh accelerators affects the SR-KNN's performance.

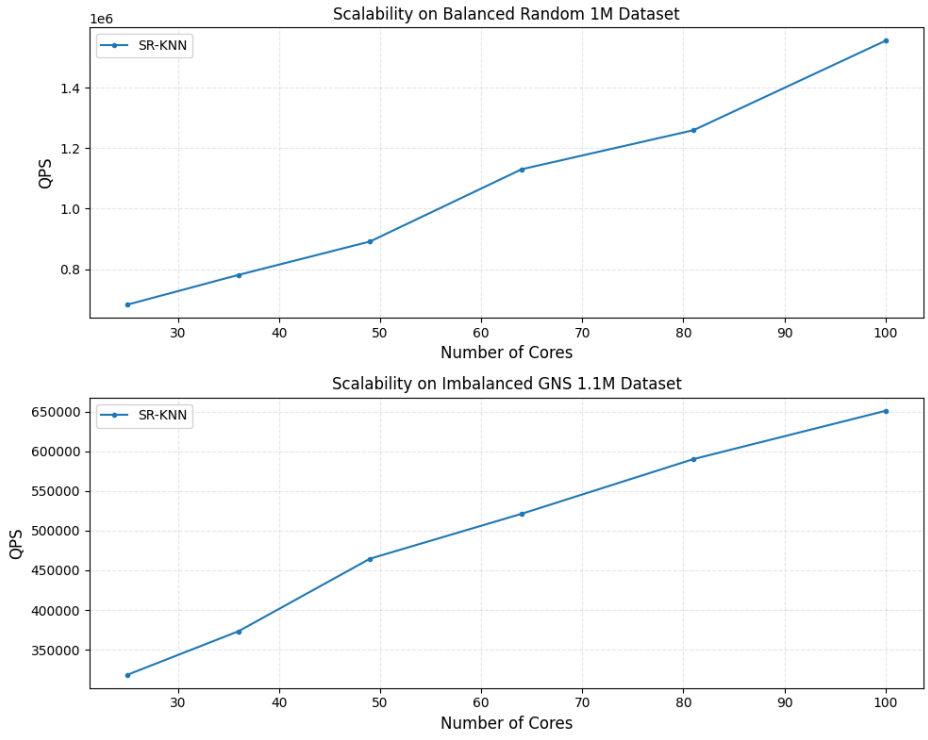

Figure 4.4: Scalability trends of SR-KNN (k=10)

We assessed the performance of SR-KNN by changing the number of cores from 25

to 100, using QPS (Queries Per Second) as the metric. The results indicate that as the number of cores increases, there is an obvious enhancement in SR-KNN performance as both datasets demonstrated a 3X improvement in performance. This outcome is in line with expectations because scaling up the core usage implies fewer points per core and more tasks to be executed in a distributed and parallel manner. The processing speed is further amplified by the powerful performance of each Tensix core (Tenstorrent Grayskull core).

4.5 Performance Benchmark

To evaluate the effectiveness of our SR-KNN algorithm, we applied the method to both datasets and compared the end-to-end performance against other methods. Our SR-KNN is evaluated based on the setting of Tenstorrent GraySkull accelerator. The competing GPU-based algorithms are tested on a single NVIDIA RTX A5000 GPU. Other CPU-based methods are tested on an AMD EPYC 7453 - 28 cores @ 2.75GHz CPU.

We select Faiss-GPU [Johnson et al., 2021] and Scikit-learn [Pedregosa et al., 2011] as our baselines due to their high-level optimizations and SOTA performance. Faiss provides the Flat algorithm. It performs a Brute-Force and accurate search with GPU, which aligns our requirement for exact nearest neighbors. Scikit-learn provides Brute-Force, KD-Tree and Ball-Tree KNN algorithms, which are accurate and well-optimized on the CPU. We also provide the Brute-Force search on the Tenstorrent as a baseline with the same accelerator as SR-KNN.

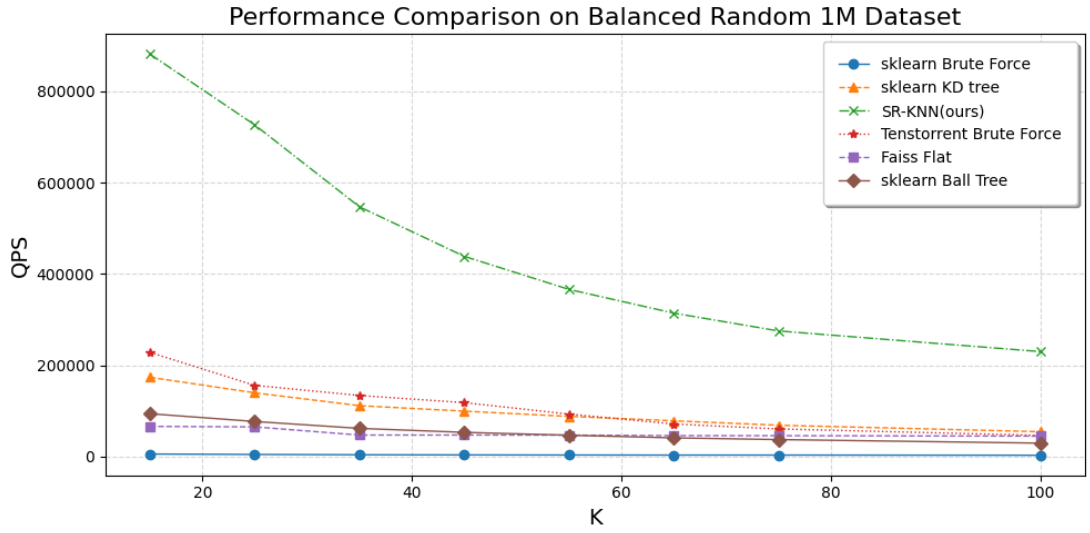

Figure 4.5: Speed-K trade-off on Balanced Random 1M Dataset

Figure 4.5 shows that our performance significantly outperforms competing methods, especially for small K values that can have around 4X speedup. With larger K values, the difference narrows but maintains a roughly 2X increase in speed.

Figure 4.6 demonstrates that, despite the dataset being imbalanced, our algorithm continues to outperform all other methods in terms of efficiency. In this evaluation, SR-KNN consistently achieves a speedup of about 2-3X among all K values compared to other competitors.

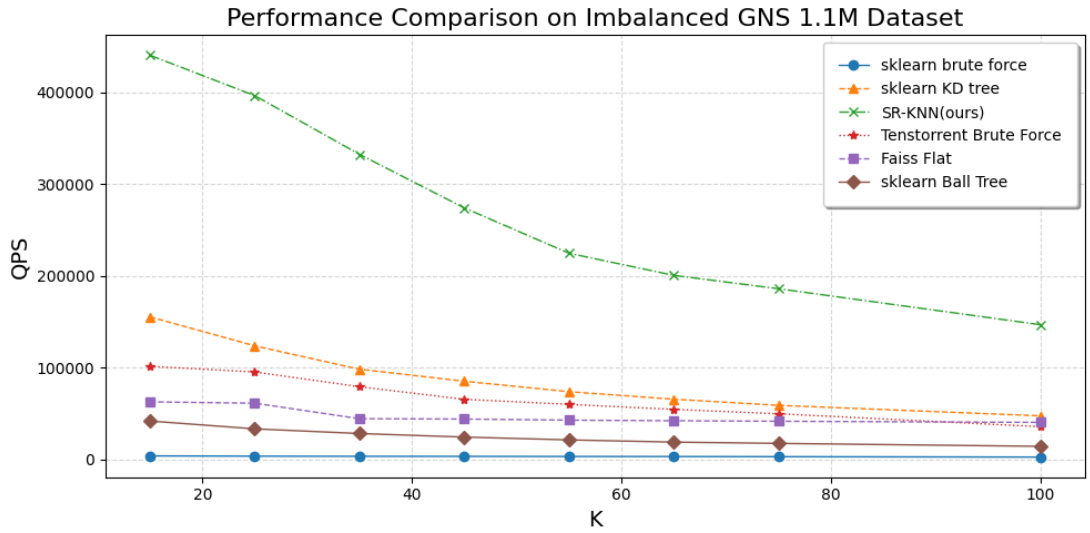

Figure 4.6: Speed-K trade-off on Imbalanced GNS 1.1M Dataset

4.6 Discussion

We limit our experiments and discussion to single-chip accelerator KNN performance of dense vectors. Due to the scalability of the 2D-Mesh accelerators, our algorithm can also be extended to multiple devices as we can link more devices to establish an expanded layout. The expanded layout not only enhances the use of distributed and parallel processing, but also allows for faster KNN search with much larger datasets.

Nevertheless, the performance of accelerators on sparse vectors (with a lot of zero values) operates under a different paradigm because of the random memory access. Moreover, we only estimate the performance on 2D data, which is not enough as currently KNN is also applied to high-dimension data.

Finally, in our evaluation, we have compared our algorithm to CPU-based well-optimized KD-Tree and Ball-Tree KNN algorithms. We still need to compare the GPU-based performance of these two algorithms but so far we have not found reliable Python integrated libraries that completely implement correct GPU-based KD-Tree and Ball-Tree. We are in the process of implementing them using CUDA.

4.7 Future Work

In order to handle high-dimensional data, we are going to try dimension reduction techniques to project the data onto a 2D plane. This enables us to effectively partition the data in a space-conscious manner within the 2D space.

While in our evaluation, SR-KNN has demonstrated adequate performance on imbalanced datasets, this does not necessarily confirm its effectiveness on datasets that are even more imbalanced, where points are densely clustered in a specific region. Such extreme imbalance can lead to high computational demands on the core responsible for processing the heavily concentrated area. Therefore, we plan to partition the data depending on the density of points which is in a position of partitioning concentrated points to various cores.

Moreover, we think the GPU-based KD-Tree method will have a much better performance by utilizing the coalescing memory of GPU for tree indexing. Therefore, we plan to discover a strategy in which SR-KNN can solve KNN requests on dynamic datasets. Like the GNS dataset, we do not concatenate them into one large and static dataset, instead, we use the original data and make SR-KNN do its job frame by frame without transferring data from the host to the device except for the first frame. Index-based KNN search methods such as KD-Tree and Ball-Tree are not suitable for this task due to their requirement to construct indices for each frame. Building these indices is time-consuming and even slower than the searching time, thereby inefficient for the dynamic KNN searching.

Chapter 5

Conclusion

This thesis describes SR-KNN, a fixed-radius exact K Nearest Neighbor operated on 2D-Mesh accelerators. We demonstrated that SR-KNN outperforms brute force search implemented on various platforms, optimized CPU-based KD-Tree search and Ball-Tree search in an ideal manner. The design of SR-KNN uncovers significant opportunities for efficient algorithms and systems research on 2D-Mesh accelerators including designing other efficient search algorithms, fast matrix calculation strategies and tensor parallelism for large model training and inference. We believe our work can extend to a dynamic situation where data points are moving, such as GNN-based particle dynamics simulation [Kumar and Vantassel, 2023], by utilizing the layout of 2D-Mesh accelerators. We are in the process of implementing the SR-KNN on a real 2D-Mesh accelerator - *Tenstorrent*. We look forward to addressing these issues in the future.

Bibliography

N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. Blue gene/l torus interconnection network. *IBM Journal of Research and Development*, 49 (2.3):265–276, 2005. doi: 10.1147/rd.492.0265.

Ahmad Alammar, Yazeed Al Moayed, and Nasir Ahmed Algeelani. TRANSACTION FRAUD DETECTOR USING KNN IN DEEP LEARNING. *International Journal of Novel Research in Computer Science and Software Engineering*, 9(3), December 2022. doi: 10.5281/zenodo.7365019. URL https://doi.org/10.5281/zenodo.7365019.

K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery. ISBN 9781450378970. doi: 10.1145/1468075.1468121. URL https://doi.org/10.1145/1468075.1468121.

Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Aug 2021. URL http://books.google.ie/books?id=RpkvEAAAQBAJ&printsec=frontcover&dq=9780486428093&hl=&cd=5&source=gbs_api.

Jon L Bentley. A survey of techniques for fixed radius near neighbor searching. Technical report, Stanford, CA, USA, 1975.

Alexander Brace, Michael Salim, Vishal Subbiah, Heng Ma, Murali Emani, Anda Trifa, Austin R. Clyde, Corey Adams, Thomas Uram, Hyunseung Yoo, Andew Hock, Jessica Liu, Venkatram Vishwanath, and Arvind Ramanathan. Stream-ai-md: streaming ai-driven adaptive molecular simulations for heterogeneous computing platforms. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385633. doi: 10.1145/3468267.3470578. URL https://doi.org/10.1145/3468267.3470578.

Daniel Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder, 2018.

Xinye Chen and Stefan Güttel. Fast and exact fixed-radius neighbor search based on sorting, 2024.

Felix Chern, Blake Hechtman, Andy Davis, Ruiqi Guo, David Majnemer, and Sanjiv Kumar. Tpu-knn: K nearest neighbor search at peak flop/s, 2022.

Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu, 2008.

Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. Ggnn: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data*, 9(1): 267–279, February 2023. ISSN 2372-2096. doi: 10.1109/tbdata.2022.3161156. URL http://dx.doi.org/10.1109/TBDATA.2022.3161156.

Xiaoyi Gu, Leman Akoglu, and Alessandro Rinaldo. Statistical analysis of nearest neighbor methods for anomaly detection. *ArXiv*, abs/1907.03813, 2019. URL https://api.semanticscholar.org/CorpusID:195848253.

Xu-Guang Guan, Xingyuan Tong, and Yintang Yang. Quasi delay-insensitive high speed two-phase protocol asynchronous wrapper for network on chips. *J. Comput. Sci. Technol.*, 25:1092–1100, 09 2010. doi: 10.1007/s11390-010-9390-5.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization, 2020.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

Anugrah Jain, Vijay Laxmi, Meenakshi Tripathi, Manoj Singh Gaur, and Rimpy Bishnoi. S2dio: an extended scalable 2d mesh network-on-chip routing reconfiguration for efficient bypass of link failures. *The Journal of Supercomputing*, 75(10):6855–6881, Jun 2019. doi: 10.1007/s11227-019-02915-5. URL http://dx.doi.org/10.1007/s11227-019-02915-5.

Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc V. Le, Yunhsuan Sung, Zhen Li, and Tom Duerig. Scaling up visual and vision-language representation learning with noisy text supervision, 2021.

J. Johnson, M. Douze, and H. Jegou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(03):535–547, jul 2021. ISSN 2332-7790. doi: 10.1109/TBDATA.2019.2921572.

Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Köster. Pipelined backpropagation at scale: Training large models without batches, 2021.

Krishna Kumar and Joseph Vantassel. Gns: A generalizable graph neural network-based simulator for particulate and fluid modeling. *Journal of Open Source Software*, 8(88):

5025, 2023. doi: 10.21105/joss.05025. URL https://doi.org/10.21105/joss.05025.

Youran Lan and Ling-Fen Chen. Multicast communication in 2-d mesh networks. In *Proceedings of 1994 International Conference on Parallel and Distributed Systems*, pages 63–68, 1994. doi: 10.1109/ICPADS.1994.589897.

Marek Majkowski. Branch predictor: How many ”if”s are too many? including x86 and m1 benchmarks!, May 2021. URL https://blog.cloudflare.com/branch-predictor.

Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, 2018.

Kamalesh Mohanarangam and Amrita Shetty. Tesla's dojo supercomputer: A game-changer in the quest for fully autonomous vehicles, Sep 2022. URL https://www.frost.com/frost-perspectives/teslas-dojo-supercomputer-a-game-changer-in-the-quest-for-fully-autonomous-v

David Muhr and Michael Affenzeller. *Hybrid (CPU/GPU) Exact Nearest Neighbors Search in High-Dimensional Spaces*, pages 112–123. 06 2022. ISBN 978-3-031-08336-5. doi: 10.1007/978-3-031-08337-2_10.

Stephen M. Omohundro. Five balltree construction algorithms. 2009. URL https://api.semanticscholar.org/CorpusID:61067117.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Mihir Pendse, Vithursan Thangarasa, Vitaliy Chiley, Ryan Holmdahl, Joel Hestness, and Dennis DeCoste. *Memory Efficient 3D U-Net with Reversible Mobile Inverted Bottlenecks for Brain Tumor Segmentation*, page 388–397. Springer International Publishing, 2021. ISBN 9783030720872. doi: 10.1007/978-3-030-72087-2_34. URL http://dx.doi.org/10.1007/978-3-030-72087-2_34.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. Learning to simulate complex physics with graph networks, 2020.

Dimitrios Serpanos and Tilman Wolf. Chapter 13 - networks on chips. In Dimitrios Serpanos and Tilman Wolf, editors, *Architecture of Network Systems*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 239–248. Morgan Kaufmann, Boston, 2011. doi: https://doi.org/10.1016/B978-0-12-374494-4.00013-X. URL https://www.sciencedirect.com/science/article/pii/B978012374494400013X.

Tenstorrent, 2024. URL https://tenstorrent.com/cards/.

Vithursan Thangarasa, Abhay Gupta, William Marshall, Tianda Li, Kevin Leong,

Dennis DeCoste, Sean Lie, and Shreyas Saxena. Spdf: Sparse pre-training and dense fine-tuning for large language models, 2023.

Vithursan Thangarasa, Shreyas Saxena, Abhay Gupta, and Sean Lie. Sparse-ift: Sparse iso-flop transformations for maximizing training efficiency, 2024.

Jasmina Vasiljevic, Ljubisa Bajic, Davor Capalija, Stanislav Sokorac, Dragoljub Ignjatovic, Lejla Bajic, Milos Trajkovic, Ivan Hamer, Ivan Matosevic, Aleksandar Cejkov, Syed Gilani, Utku Aydonat, Tony Zhou, Joseph Chu, Djordje Maksimovic, Zahi Moudallal, Akhmed Rakhmati, Stephen Alexander, and Boris Drazic. Compute substrate for software 2.0. *IEEE Micro*, PP:1–1, 03 2021. doi: 10.1109/MM.2021.3061912.

Swapnil Vishwakarma. The curse of dimensionality in machine learning!, Nov 2023. URL https://www.analyticsvidhya.com/blog/2021/04/the-curse-of-dimensionality-in-machine-learning/.

VLFea. Kd-trees and forests, 2007. URL https://www.vlfeat.org/overview/kdtree.html.

Wikipedia, Mar 2024. URL https://en.wikipedia.org/wiki/Single_instruction,_multiple_data#.

Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems*, pages 43–51, 2019.

Appendix A

Baseline Implementation

A.1 KD-Tree Implementation

```
class KDTree:
    def __init__(self, points, depth=0):
        if not points:
            self.root = None
            return

        num_points = len(points)
        axis = depth % len(points[0])
        points.sort(key=lambda point: point[axis])
        median_index = num_points // 2

        self.root = points[median_index]
        self.axis = axis
        self.left = KDTree(points[:median_index], depth + 1) if
            median_index > 0 else None
        self.right = KDTree(points[median_index + 1:], depth + 1) if
            num_points - median_index - 1 > 0 else None

    def knn(self, target, k=1):
        nearest_neighbors = [(-1, None) for _ in range(k)]

        def distance_squared(point1, point2):
            return sum((p1 - p2) ** 2 for p1, p2 in zip(point1, point2))

        def insert_sorted(neighbors, dist, node):
            for i, (d, _) in enumerate(neighbors):
                if d == -1 or d > dist:
                    neighbors.insert(i, (dist, node))
                    neighbors.pop()
                    return

```

```
        def search(node, depth=0):
            if node is None:
                return
            axis = depth % len(target)

            diff = node.root[axis] - target[axis]
            next_branch, opposite_branch = (node.left, node.right) if diff
                > 0 else (node.right, node.left)
            search(next_branch, depth + 1)

            dist = distance_squared(node.root, target)
            insert_sorted(nearest_neighbors, dist, node.root)

            if nearest_neighbors[-1][0] == -1 or abs(diff) <
                nearest_neighbors[-1][0]:
                search(opposite_branch, depth + 1)

        search(self)
        return [(dist, point) for dist, point in nearest_neighbors if
            point is not None]
```

A.2 Ball-Tree Implementation

```
import numpy as np

def distance(X, Y, p=2):
    return np.sum(np.abs(X - Y) ** p, axis=1)

class BallNode:
    def __init__(self, value, index, radius, left=None, right=None):
        self.value = value # Feature values for the node; feature vectors
            for leaf nodes
        self.index = index # Indexes of training set; index vectors for
            leaf nodes
        self.radius = radius # Radius of the hypersphere
        self.left = left # Left subtree
        self.right = right # Right subtree

class BallTree:
    def __init__(self, X, leaf_size=1000, p=2):
        def build_node(X, X_indexes, leaf_size):
            if X.shape[0] <= leaf_size:
                return BallNode(X, X_indexes, None)
            feature = np.argmax(np.std(X, axis=0))
            X_feature_max = X[np.argmin(X[:, feature])]
            X_feature_min = X[np.argmax(X[:, feature])]
            X_feature_median = (X_feature_max + X_feature_min) / 2
```

```
23            radius = np.max(distance(X, np.array([X_feature_median]), p))
24            left_index = (distance(X, np.array([X_feature_max]), p) -
                  distance(X, np.array([X_feature_min]), p)) < 0
25            left = right = None
26            if left_index.any():
27                left = build_node(X[left_index, :], X_indexes[left_index],
                      leaf_size)
28            right_index = ~left_index
29            if right_index.any():
30                right = build_node(X[right_index, :],
                      X_indexes[right_index], leaf_size)
31            return BallNode(X_feature_median, None, radius, left, right)
32        self.root = build_node(X, np.arange(X.shape[0]), leaf_size)
33
34    def query(self, X, k=1, p=2):
35        nearests = -np.ones((X.shape[0], k), dtype=np.int)
36        distances = np.full((X.shape[0], k), np.inf)
37        return self.search(X, self.root, nearests, distances, p)
38
39    def search(self, X, node, nearests, distances, p=2):
40        if node.left is None and node.right is None:
41            for i, value in enumerate(node.value):
42                dist = distance(X, np.array([value]), p)
43                for x_index, x_dist in enumerate(dist):
44                    for k_index, k_dist in enumerate(distances[x_index]):
45                        if k_dist > x_dist:
46                            distances[x_index, k_index+1:] =
                                  distances[x_index, k_index:-1]
47                            distances[x_index, k_index] = x_dist
48                            nearests[x_index, k_index+1:] =
                                  nearests[x_index, k_index:-1]
49                            nearests[x_index, k_index] = node.index[i]
50                            break
51            return nearests, distances
52
53        max_distance = np.max(distances, axis=1)
54        pivot_distance = distance(X, np.array([node.value]), p)
55        in_radius = pivot_distance - node.radius <= max_distance
56
57        if node.left and in_radius.any():
58            nearests[in_radius], distances[in_radius] =
                  self.search(X[in_radius], node.left, nearests[in_radius],
                  distances[in_radius], p)
59        if node.right and in_radius.any():
60            nearests[in_radius], distances[in_radius] =
                  self.search(X[in_radius], node.right, nearests[in_radius],
                  distances[in_radius], p)
61
62        return nearests, distances
```

A.3 Space-Conscious Data Partition Detail

Algorithm 2 Space-Conscious Data Partition 2D

```
Input: P ∈ ℝ^{N×2}
Input: R, C
min_x, min_y = min(P)
max_x, max_y = max(P)
block_size_x = (max_x - min_x) // R
block_size_y = (max_y - min_y) // C
mapped_core_coords = []
for point in P do                                          ▷ in parallel
    block_x = (point.x - min_x) // block_size_x
    block_y = (point.y - min_y) // block_size_y
    mapped_core_coords.append([block_x, block_y])
end for
return mapped_core_coords
```

A.4 Single Core Operations

```
From host: Q ∈ ℝ^{M×D}
From host: P ∈ ℝ^{N×D}
From host: Routes

procedure BROADCAST(Q, Routes, is_backward)
    for r in Routes do                                     ▷ in parallel
        step = r.pop()
        forward_to(r, Q, is_backward=False)
    end for
end procedure

procedure ON-CHIP-CALCULATION(Q, P, A, D, S)
    for i ← 1 to M do                                      ▷ Vectorized
        for j ← 1 to N do
            y_{i,j} ← L2_Distance(q_i, p_j)
            for k ← 1 to K do                              ▷ SIMD Comparisons
                if y_{i,j} < d_{i,k} then
                    d_{i,k} = y_{i,j}
                    s_{i,k} = a_j
                end if
            end for
        end for
    end for
return D, S
end procedure
```

```
27: procedure REDUCE(Curr, Recv)
28:     for d in Recv do                                          ▷ in parallel
29:         curr = Curr[curr.id == d.id]
30:         curr = MERGE_SORT(d, curr, k)
31:     end for
32:     forward_to(next_step, Curr, is_backward=True)
33: end procedure
34:
35: procedure MERGE_SORT(sorted_left, sorted_right, k)
36:     i = 0
37:     j = 0
38:     merge_count = 0
39:     result = []
40:     while i < sorted_left.length && j < sorted_right.length && merge_count < k
    do
41:         if sorted_left[i] < sorted_right[j] then
42:             result.append(sorted_left[i])
43:             i += 1
44:             merge_count += 1
45:         else
46:             result.append(sorted_right[j])
47:             j += 1
48:             merge_count += 1
49:         end if
50:     end while
51:     while i < sorted_left.length && merge_count < k do
52:         result.append(sorted_left[i])
53:         i += 1
54:         merge_count += 1
55:     end while
56:     while j < sorted_right.length && merge_count < k do
57:         result.append(sorted_right[j])
58:         j += 1
59:         merge_count += 1
60:     end while
61: end procedure
```