Parallelizing the Nearest Neighbor Search

Stephen Connelly, Kyle Chen, Sophie Liu

Abstract— **This study delves into the parallelization of Nearest Neighbor Search (NNS) algorithms, particularly for applications within the robotics field, emphasizing their utility in scenarios involving rapidly-exploring random trees (RRTs). The primary objective was to assess whether employing GPU-based parallel computing frameworks could offer a reduction in computational costs when contrasted with traditional CPU-based algorithms. Our extensive experimentation and analysis reveal that GPUs do indeed offer a substantial performance uplift when handling large datasets, as they can execute multiple operations concurrently. However, it is important to note that the efficiency gains provided by GPUs become less pronounced with smaller datasets, where I/O overheads tend to mitigate the potential benefits of parallel execution. The implications of these findings suggest that while GPUs are highly effective under certain conditions, their application should be tailored to the specific requirements of the dataset size to optimize performance outcomes.**

I. Introduction

While CPUs have stagnated in performance over the last 10 years, GPUs on the other hand have exploded in performance and popularity through their ability to parallelize applications. While CPUs have remained more broad in their use-cases, GPUs, with many more cores than CPUs, can parallelize repeated and similar functions across threads allowing for applications in video rendering, cryptocurrency mining, AI/ML and models. GPUs parallelizing ability is still being explored with new applications being discovered to increase performance of computationally heavy software and algorithms. The Nearest Neighbor Search (NNS) algorithm has been a staple algorithm in statistics and has had countless applications: most notably is the traveling salesman problem and also in the k-Nearest Neighbor optimization algorithm used in machine learning.

We introduce in this research project an exploration of parallelizing the NNS by first replicating a previously researched parallelization of the NNS distance searching of nodes compared to the serial algorithm. To extend the capabilities of the NNS algorithms, implementations for finding the k-nearest neighbors (kNN) for a given query point will be developed. These kNN algorithms will incorporate a parallel merge sort algorithm to efficiently sort the distances and identify the k-nearest neighbors. A serial merge sort implementation will also be created as a baseline for performance comparison. The parallel merge sort algorithm will be implemented following the design detailed in the provided information, including the GPU kernel design, thread/block calculations, and efficient data handling techniques.

To benchmark and test the algorithms, representative datasets will be generated or obtained, and a motion planning scenario will be implemented, which can utilize the NNS algorithms. Performance testing will be conducted by measuring the execution time of the serial and parallel NNS algorithms for finding a single nearest neighbor, as well as the serial and parallel kNN algorithms for varying values of k. The performance of the serial and parallel merge sort algorithms will also be evaluated.

Accuracy testing will be carried out by integrating the NNS algorithms into the motion planning scenario and evaluating the accuracy of the motion planning results using the serial and parallel NNS algorithms. The impact of using approximate kNN on the motion planning accuracy will also be assessed.

To test scalability, the size of the input datasets will be varied, and the performance and accuracy of the serial and parallel algorithms will be measured as the dataset size increases.

II. Related Work

There has been extensive research on parallelizing the nearest neighbor search algorithm. Below are subsets of scholarship related to our research question.

A. *Exact Nearest Neighbor Methods*

Papadopoulos and Kalnis [4] proposed a declustering method parallel processing of nearest neighbor queries on spatial data which they found outperforms the traditional method, branch-and-bound. For general high-dimensional data, Xiao et al. [6] present scalable algorithms and a library for exact and approximate nearest neighbor searches, built on MPI and OpenMP, enabling efficient execution on thousands of cores for arbitrary-dimensional datasets.

B. *Approximate Nearest Neighbor Methods*

To overcome the curse of dimensionality, many approximate algorithms have been considered. Emiris et al. [1] develops a data structure for answering approximate nearest neighbor queries over a set of parallel segments in three dimensions, connecting the problem to approximate nearest neighbor searching under weight constraints. Indyk et al. [3] aims to find a collection of close points compatible with each other with applications in computer vision and databases. They propose a two step method for designing efficient data structures for Shared Nearest Neighbor, involving approximate nearest neighbor search and offline optimization.

C. *Parallel Algorithm Design*

Sismanis et al. [5] introduces a number of truncated sort-algorithms for parallel k-Nearest-Neighbor search. In doing so they showed that a truncated bitonic sort, or TBiS, is efficient for parallel search on many-core processors, 'exploiting the relationships between select and sort operations.' Moreover Gao et al. [2] proposes best-first based parallel algorithms that are effective and scalable in parallel processing of nearest neighbor retrieval in multi-disk settings, aiming to reduce query cost and I/O overhead.

III. Design and Implementation

A. Serial and Parallel Nearest Neighbor Search

Our serial Nearest Neighbor Search algorithm is a conventional one. It works by finding the data points closest to a given query point in a certain multi-dimensional space. It works by calculating the distance between the query point and all other points in the dataset, and then returning the point with the minimum distance. This requires the algorithm to find each distance from the point at hand, to each other point in the plane. This can be costly and time consuming when done serially. Moreover these distances need then to be sorted and the most minimal points are chosen to be considered. The time complexity is linear, and can be computationally expensive for larger datasets.

In order to make the algorithm theoretically more efficient, we decided to explore a parallelization of it. For each distance needing to be calculated we allocated a thread to the computation as such, making the serial distance calculation parallel.

B. KNN and Parallel Merge Sort

After parallelizing the linear search of nearest neighbor search, we tried to implement a sorting algorithm to sort the k-distances needed for the K-Nearest-Neighbors algorithm. The following combinations of designs will be tested: A parallel distance calculation from the query point to all other points in the plane, and a non-parallel merge-sort. A non-parallel distance calculation from the query point to all other points in the plane, and a parallel merge-sort. A parallel distance calculation from the query point to all other points in the plane, and a parallel merge-sort.

C. The Merge Sort Algorithm

Merge sort is a divide-and-conquer algorithm that is used for sorting a list or array of elements in ascending or descending order. The algorithm works by dividing the input array into two halves, recursively sorting each half, and then merging the sorted halves back together. Pseudo code for the algorithm would look like the following:

Algorithm 1 Merge Sort

```
1:  procedure MERGESORT($A, start, end$)
2:      if $start < end$ then
3:          $mid \leftarrow \frac{(start+end)}{2}$
4:          MERGESORT($A, start, mid$)
5:          MERGESORT($A, mid + 1, end$)
6:          MERGE($A, start, mid, end$)

7:  procedure MERGE($A, start, mid, end$)
8:      Create array $L$ from $A[start$ to $mid]$
9:      Create array $R$ from $A[mid + 1$ to $end]$
10:     Merge arrays $L$ and $R$ back into $A[start$ to $end]$
```

D. The Parallelized Merge Sort Algorithm

In order to parallelize the merge sort algorithm we designed the algorithm as such using the CUDA software. The algorithm has 'two parts', the CPU code and the GPU code. The CPU function mergesort allocates memory on the GPU and copies the input data and dimension parameters. It enters a loop where the width (size of segments) is doubled in each iteration. In each iteration the number of slices is calculated based on the current width and the total number of threads. The gpu mergesort kernel is launched with the appropriate grid and block dimensions. Each GPU thread processes a specific slice of the input array, merging the two sorted halves into the destination array. After the merge operation is complete for the current width, the input and output arrays are swapped. The loop continues until the width becomes larger than or equal to the array size, indicating that the entire array has been merged. Finally, the sorted data is copied back from the GPU to the CPU.

Algorithm 2 MergeSort

```
1:  procedure MERGESORT(data, size, threadsPerBlock, blocksPerGrid)
2:      Set up pointers for data and swap space on GPU
3:      Calculate total number of threads
4:      for width from 2 to size of array, doubling each iteration do
5:          Calculate slices as size / (total threads * width) +1
6:          Call GPU_MERGESORT to sort slices

7:  procedure GPU_MERGESORT(source, dest, size, width, slices, threads, blocks)
8:      Calculate thread index based on block and thread identifiers
9:      for each slice assigned to this thread do
10:         Calculate starting, middle, and ending indices for the current slice
11:         Merge from source to dest for the current slice
12:         if left item is smaller or right half is exhausted then
13:             Take from left
14:         else
15:             Take from right
16:         Adjust start for next slice
```

E. The Rapidly Exploring Random Tree (RRT) Algorithm

The Rapidly-Exploring Random Tree (RRT) is a single query planner that, given a start and goal configuration, seeks to find a path that a robot can navigate from the start to the goal configuration. It is a slight variation upon Expansive-Space Trees (EST) that guarantees a dense covering of the search space and simplifies the sampling process, which can be randomly generated.

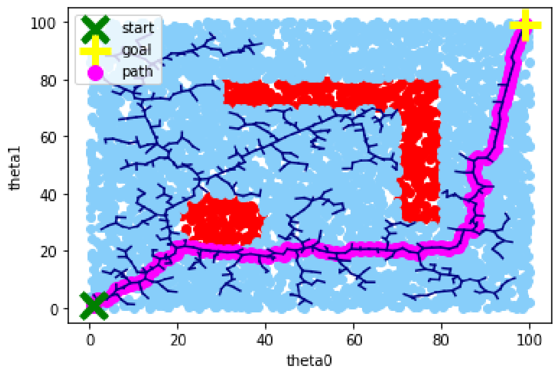

Fig. 1. Example RRT Visualization

In figure 11, an example of the built RRT tree leading to the discovery of a path from start, $(1, 1)$ in the bottom left, to goal, $(99, 99)$ in the top right, is shown. The RRT utilizes an ϵ-greedy approach to generating the next sample direction, exploiting the goal with probability ϵ and exploring randomly with probability $(1 - \epsilon)$.

Algorithm 3 Build RRT Algorithm

```
1: Initialize $T$ with initial node $x_{init}$
2: **while** not at maximum iterations **do**
3:     $x_{rand} \leftarrow$ a randomly chosen free configuration
4:     $x_{new} \leftarrow$ extend RRT $(T, x_{rand})$
5:     **if** Is_Goal($x_{new}$) **then**
6:         **return** $T$
7: **return** $T$
```

Algorithm 4 Extend RRT Algorithm

```
1: **procedure** EXTEND($T$, $x_{rand}$)
2:     $x_{near} \leftarrow$ closest neighbor of $x_{rand}$ in $T$
3:     $x_{new} \leftarrow$ progress $x_{near}$ by step_size along the
           straight line between $x_{near}$ and $x_{rand}$
4:     **if** valid_edge($x_{near}$, $x_{new}$) **then**
5:         Add $x_{new}$ to $T$
6:         Add edge from $x_{near}$ to $x_{new}$ in $T$
7:         **return** $x_{new}$
```

F. The Parallelized RRT Algorithm

The RRT was solely parallelized in finding the nearest neighbor. The adaptation of the parallelized KNN algorithm to find the single nearest neighbor was an effort to specifically test the gains of parallelizing this computationally intensive step in the RRT algorithm. By targeting this segment for parallelization, we aimed to explore our parallelized KNN algorithm in a practical application and potentially alleviate a computational bottleneck in the RRT algorithm.

The task of locating the nearest neighbor within the RRT algorithm is pivotal and a mandatory step with each new point that is sampled; each prospective node must identify a candidate node in the existing RRT tree to attempt to connect to. As the tree expands, the complexity of finding the nearest node increases exponentially in higher-dimensional spaces.

IV. Results

In assessing the computational efficiency and scalability of Rapidly-exploring Random Trees (RRT) and K-Nearest Neighbors (KNN) algorithms, this study performed both comparative analysis of parallel vs serial implementations of the algorithms above as well as performance analysis with respect to varying thread counts. The main objective behind our analysis was to characterize the performance of these algorithms under different computational loads and implementation strategies, thereby providing insights into their practical implications.

These tests were conducted using Google Cloud Platform (GCP)'s Compute Engine. In order to guarantee the robustness and reproducibility of our findings, each configuration was repeated in 100 trials and the average execution time was measured. The tests were also conducted in close temporal proximity ensure consistency across tests.

The results can be sectioned based on the type of algorithm (RRT or KNN) and the nature of the analysis (scalability or performance).

A. RRT - Scalability Analysis of Parallel vs Serial Implementation

The scalability of the RRT algorithm was assessed by comparing its performance in both serial and parallel implementations as measured by average execution time. The parallel implementation was run with a constant 1 block per grid and 1024 threads per block.

A methodological decision was made to assess the efficiency of the algorithm under worse-case scenarios. This is because any search algorithm has the possibility of terminating prematurely by finding the path to the goal. This would introduce random variability in the execution time that was mitigated by providing the RRT algorithm a goal that was impossible to reach within the constraints of the search space. This allowed each trial to measure the execution time for the RRT to exhaustively explore the environment until the maximum number of nodes in the tree was reached.

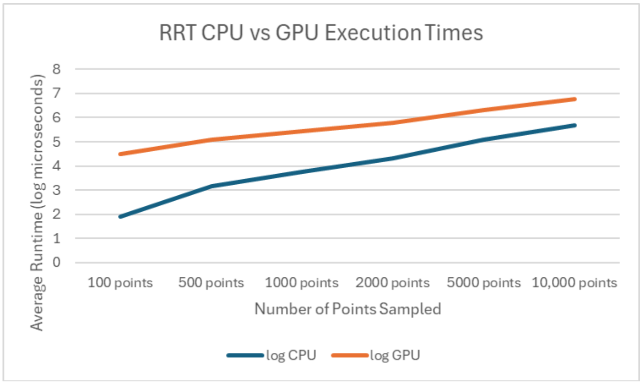

Fig. 2. RRT Average Execution Time of Parallel vs Serial Implementation

Figure 22 illustrates log average execution times for the parallel versus serial implementation of the RRT algorithm. The graph shows the CPU implementation dominates with lower average execution times across all varying numbers of points sampled. However, it is also evident that the GPU implementation is superior in terms of scalability, notably as the number of points sampled increases.

N	CPU (ms)	GPU (ms)
100	80	32,035
500	1,401	118,877
1,000	5,652	263,172
2,000	21,312	602,209
5,000	126,904	2,041,917
10,000	491,772	5,863,018

Fig. 3. RRT Execution Time Comparison of CPU and GPU Performance

As seen in figure 3IV-A, the number of points sampled by the RRT was varied from 100 points sampled to $10,000$ points sampled. At lower numbers of points sampled, the CPU's performance advantage is superior, with $80 \ll 32,035$. However, this advantage becomes much less pronounced with the scaling of the number of points sampled, pointing to the GPU's superior scalability and capability to manage increasing large computational loads effectively.

B. RRT - Performance Analysis with Varying Thread Counts

Further analysis into the gains of increasing the number of threads was also examined. Theoretically, an increased number of threads allows for greater parallelization capacity with more threads to divide the work among. However, each additional thread also incurs an overhead cost to create, manage, and terminate which may degrade performance. It was of interest to understand how average execution time was impacted by an increase in the number of threads.

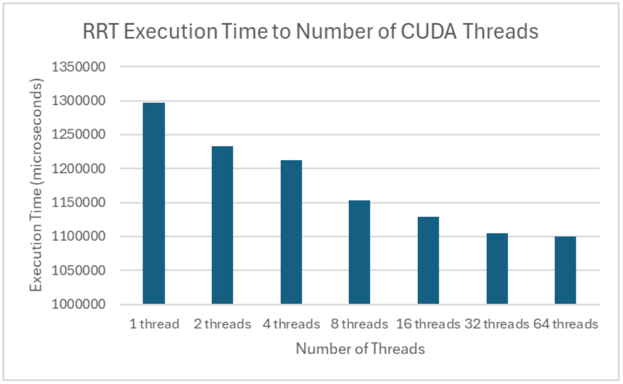

Fig. 4. RRT Average Execution Time to CUDA Kernel Thread Count

Figure 44 suggests that an increase in the number of threads per block experiences diminishing returns. This is as predicted by Amdahl's Law, which states that there is a theoretical maximum speedup for a program dependent on the proportion of a program that can be parallelized. In this RRT implementation, the only parallelized step was in finding the nearest neighbors. It can be hypothesized that further increases in thread count may lead to minimal gains in average execution time or increases if management costs become too large.

C. KNN - Scalability Analysis of Parallel vs Serial Implementation

The scalability of the KNN algorithm was similarly assessed by comparing its performance in serial and parallel implementations as measured by average execution time. The number of data points each query point was evaluated against to determine its k nearest neighbors was increased from 10 points to $10,000$ points (see Figure 5IV-C). The parallel implementation was run with a constant 1 block per grid and 1024 threads per block.

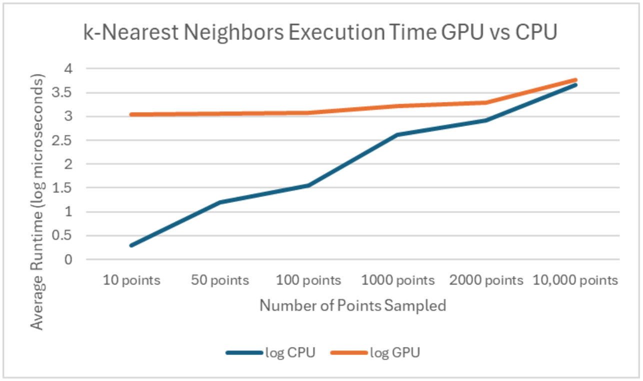

Fig. 5. KNN Average Execution Time of Parallel vs Serial Implementation

Figure 55 shows the average execution time of the CPU implementation was lower than that of the GPU implementation across all dataset sizes. However, the graph also reflects the superior scalability of the GPU implementation. This advantage of scalability in the GPU implementation is even more so with the KNN algorithm than with the RRT algorithm. This can be understood by Amdahl's Law since a greater proportion of the program for the KNN algorithm was parallelized in this parallel implementation (distance computation and mergesort) compared to the RRT (which is still largely sequential by nature). The largest improvements in the average execution time can be attributed to the parallelized merge sort, which can theoretically better the computational complexity of sorting from $O(n \log n)$ to a much more optimal $O(\log n)$.

N	CPU (ms)	GPU (ms)
10	2	1,084
50	16	1,151
100	35	1,204
1,000	409	1,619
2,000	837	1,969
10,000	4532	5,846

Fig. 6. KNN Execution Time Comparison of CPU and GPU Performance

The average microseconds spent per data point for the parallel implementation was approximately $238\mu s$ for 500

points and $586\mu s$ for $10,000$ points, while the same was $3\mu s$ and $49\mu s$ for the serial implementationIV-C. The multiple that the average microseconds spent per data point increased by was 2.466 for the GPU implementation in comparison to a much larger 17.551 for the CPU implementation, amplifying the difference in scalability.

D. KNN - Performance Analysis with Varying Thread Counts

Additional analysis of the gains of increasing the number of threads was also examined, similar to with the RRT algorithm. Although each additional thread provides more parallelization capability, but it also comes with necessary creation, management, and termination overhead that makes it valuable to understand the advantages and tradeoffs of increasing thread counts.

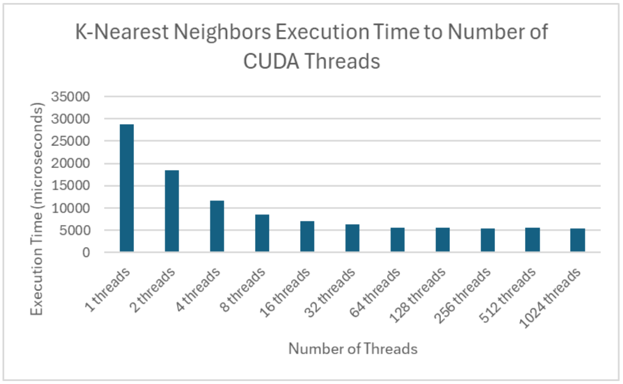

Fig. 7. KNN Average Execution Time to CUDA Kernel Thread Count

Figure 77 suggests that an increase in the number of threads per block experiences diminishing returns, with very little gains after 64 threads per block. However, there were few penalties observed due to increasing the number of threads per block, up to the maximum tested of $1,024$ threads per block. As shown by the figure7, the initial gains from KNN parallelization are particularly large, resembling an exponential decay.

V. Conclusion and Future Work

This research aimed to explore the parallelization of the Nearest Neighbor Search (NNS) algorithm, specifically in the context of rapidly-exploring random trees (RRTs). The objective was to see whether or not GPU-based parallel computation could significantly decrease the time required for these computations compared to CPU-based methods. Our results demonstrate that GPUs do enhance performance for large (10,000) datasets by leveraging parallel computation capabilities. However, we encountered significant I/O overheads, which negatively affected the performance gains in smaller datasets. This overhead is a critical factor in the practical application of GPU-accelerated NNS algorithms and suggests that the move to parallel computing platforms like the NVIDIA Jetson might mitigate some of these issues by optimizing data transfer processes. Our findings showed improvements in performance as we adjusted the number of threads and managed computational loads, which confirms the scalability of our approach.

In summation, our study confirms that while GPUs offer considerable advantages in processing speed for large datasets, the associated overhead costs can diminish these benefits in scenarios that do not require lots of computation.

Future work in the parallel K-Nearest Neighbors space would look into other, potentially probabilistic methods to determine a satisfactory nearest neighbor node, as well as alternative data structures such as K-D trees. Application specific nearest neighbor algorithms could also be designed to specific use cases.

Further algorithm-level parallelism with the RRT is another direction of future research. Many other steps in the RRT algorithm can be parallelized, leading to further gains in computational efficiency despite a largely serial algorithm. Ideas include building multiple trees in parallel within the search space; validating multiple nonlinear, curved edges in parallel to bypass obstacles; testing the RRT algorithm in higher dimensions to further assess the scalability of parallelism, and more.

References

[1] Emiris, Ioannis Z. et al. “Approximate Nearest Neighbor Queries among Parallel Segments.” (2010).

[2] Gao, Yunjun, Gencai Chen, Ling Chen and Chun Chen. “Best-First Based Parallel Nearest Neighbor Queries.” International Conference on Management of Data (2005).

[3] Indyk, Piotr, Robert D. Kleinberg, Sepideh Mahabadi and Yang Yuan. “Simultaneous Nearest Neighbor Search.” International Symposium on Computational Geometry (2016).

[4] Papadopoulos, Apostolos N. and Yannis Manolopoulos. “Parallel processing of nearest neighbor queries in declustered spatial data.” ACM SIGSPATIAL International Workshop on Advances in Geographic Information Systems (1996).

[5] Sismanis, Nikos, Nikos Pitsianis and Xiaobai Sun. “Parallel search of k-nearest neighbors with synchronous operations.” 2012 IEEE Conference on High Performance Extreme Computing (2012): 1-6.

[6] Xiao, Bo and George Biros. “Parallel Algorithms for Nearest Neighbor Search Problems in High Dimensions.” SIAM J. Sci. Comput. 38 (2016): n. pag.