GPU and CPU Implementations of Motion Planning Algorithms: A comparison of algorithms and efficiency

Thais I. Velazquez, Kristine Pham

December 11, 2024

***Abstract*—Smooth and efficient motion planning is essential for robotic systems operating in dynamic environments. Robots must adapt rapidly to obstacles and recover from unexpected events, such as falls, to ensure reliable performance. However, many existing motion planning algorithms struggle to maintain real-time adaptability in rapidly changing scenarios, especially when executed exclusively on CPUs. While CPUs can handle some planned motion, their processing speed and efficiency are often insufficient for the demands of real-time motion planning across diverse and complex terrains. This paper analyzes the algorithm of two sample-based Tree Planner algorithms, the Kinodynamic Motion Planning by Interior–Exterior Cell Exploration (KPIECE) and the K-Nearest Neighbors(KNN). The KPIECE is specifically designed for systems with complex physical constraints, where integration backward in time is not possible; however, despite the restraint in backward movement, it is supposed to make up for restricted movement in efficient and fast planning forward motion. On the other hand, the KNN search algorithm (KNNS) is used to generate all possible paths from the starting point to the end destination, given a sample set of the configuration space. It does this by looking at the k nearest neighbors of each point in the sample set. This paper explores the implementation of sample-based motion planning algorithms on GPUs to harness their inherent parallelism. By dividing motion planning tasks to the GPU, we aim to achieve faster and more responsive planning, enhancing robotic adaptability in complex environments. In this paper, we will implement the two selected sample-based algorithms (KPIECE and KNN search algorithms) on the CPU, benchmarked their performance, and compared their efficiency with GPU-based implementations to evaluate the benefits of parallelism.**

I. INTRODUCTION

The development of a functional robotic system inherently involves the complex task of navigation. As the configuration space(or environment) becomes progressively dynamic by incorporating obstacles, physics constraints, and time-dependent variables—the complexity of motion planning increases significantly. This complexity requires the use of faster and more efficient planning algorithms, as well as enhanced computational capabilities, to ensure real-time adaptability, accurate path calculation, and optimal performance. Over the years, various algorithms have been proposed to address these challenges, with each algorithm emphasizing different aspects of the constraints present within a configuration space. Among these algorithms are sampling-based tree algorithms, such as K-Nearest Neighbors (KNN) and KPIECE. The KPIECE is specifically designed for systems with complex physical constraints, where integration backward in time is not possible; however, despite the restraint in backward movement, it is supposed to make up for restricted movement in efficient and fast planning forward motion. On the other hand, the KNN search algorithm (KNNS) is used to generate all possible paths from the starting point to the end destination, given a sample set of the configuration space. It does this by looking at the k nearest neighbors of each point in the sample set. Both of these planners explore the configuration space by sampling potential motion paths that the robot may take to achieve travel to its goal state. While these algorithms have proven effective in static environments, they must also handle the uncertainty and dynamic nature of real-world scenarios, where environmental changes can occur unpredictably. This paper focuses on specifically investigates two such motion planning algorithms—KPIECE and KNN—focusing on their performance and effectiveness in a given environment. By analyzing how each algorithm adapts to changing conditions and performs under varying constraints, this study aims to contribute to the ongoing development of more robust and efficient motion planning solutions for robotic systems.

Contributions. The primary contributions of this paper are as follows: 1) to analyze the implementation of the K-Nearest Neighbors (KNN) and KPIECE algorithms. 2) to evaluate which algorithm is most suitable for a given configuration space setup, based on runtime efficiency and the number of motions tracked. 3) to provide insights into the potential parallel implementation of these algorithms, considering its impact on overall algorithmic performance.

II. Kinodynamic Planning by Interior–Exterior Cell Exploration(KPIECE) Algorithm

As stated before, the KPIECE is a sampling-based algorithm that explores the configuration space that the robotic system is in. Kinodynamic Planning by Interior-Exterior Cell Exploration(KPIECE) is a sampling-based motion-planning algorithm designed for systems with complex dynamics. It explores the state space(the environment) by growing a tree of motions through forward propagation, while using grid-based discretization to estimate coverage in a lower-dimensional projection of the state space. KPIECE prioritizes exploration of exterior cells—those on the boundary of the explored region—by assigning importance metrics based on coverage, selection frequency, and exploration progress. This approach minimizes computational overhead by reducing unnecessary forward simulations, which typically dominates runtime. The algorithm is further enhanced by optional features, such as hierarchical discretization for handling high-dimensional state spaces and goal biasing to direct exploration toward target regions. My implementation captures several key principles of the (KPIECE) algorithm, including forward-only exploration, cell-based state-space discretization, and motion expansion based on heuristic cell importance. While my implementation of the KPIECE has some of the key components that makes the KPIECE the method it is, it excludes most of the advanced features of the algorithm, such as the explicit differentiation between interior and exterior cells, which is crucial for prioritizing boundary exploration and enhancing efficiency. Additionally, my implementation employs a single-level grid discretization, whereas the originally KPIECE supports hierarchical discretization for improved scalability in high-dimensional state spaces. The heuristic for cell importance in your implementation could also be refined to include factors like the number of neighbors or progress toward the goal, as in KPIECE. Furthermore, I also left out the addition of the physical constraints of the motion planning simulations to model complex system dynamics accurately. My implementation of KPIECE just focuses on simple grid movements without accounting for dynamic controls or optimized motion trajectories.

Algorithm 1 kpiece(initial_motion, iterations)

$T \leftarrow INITIALIZETREE(initial\_motion)$
for $i \leftarrow 1$ to $iterations$ **do**
  $cell \leftarrow SELECTCELL(G, bias)$
  if $cell$ is **None then**
    continue
  end if
  $motion \leftarrow SELECTMOTION(G, cell)$
  if $motion$ is **None then**
    continue
  end if
  $new\_motion \leftarrow EXPANDMOTION(motion, G, visited)$
  if new_motion is valid **then**
    $node \leftarrow ADDCHILD(T, new\_motion)$
    ADDMOTION(G, new_motion)
    if ISGOALREACHED(new_motion) **then**
      PRINT("Goal reached at: ", new_motion)
      return T
    end if
  end if
  $P \leftarrow COMPUTEPROBABILITY()$
  UPDATECELLIMPORTANCE($G, cell, P$)
end for
PRINT("No solution found within the iteration limit.")
return T

note: Completes the maim functionality of the KPIECE and returns the tree of all visited nodes.

figure

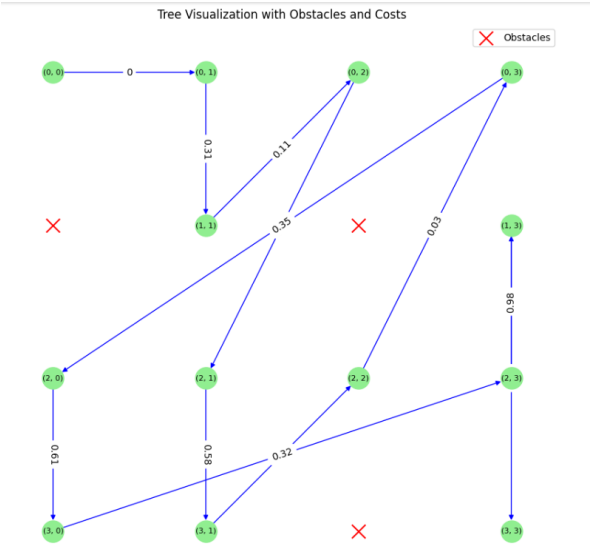

Figure 1:Visual the paths visited given a starting point and goal

```
Configuration Space (1 = obstacle, 0 = free):
[0, 0, 0, 0]
[1, 0, 1, 0]
[0, 0, 0, 0]
[0, 0, 1, 0]
Goal reached at: (3, 3)

All visited states in the tree (including repeated positions):
((0, 0), 0, 0)
((0, 1), 0.9910926864151633, 0.3661151566362363)
((1, 1), 0.25659672691354796, 0.04541731299537122)
((2, 1), 0.12625875402753195, 0.5789197222550448)
((2, 2), 0.7460070628475027, 1.0315469779277557)
((3, 1), 0.9709459846024485, 0.9476201232510661)
((2, 0), 0.3579137223602119, 0.16312547784583897)
((3, 0), 0.5915258373193918, 1.505101131116411)
((2, 3), 0.8636671842527567, 1.4736895264644345)
((0, 2), 0.37356582657315596, 0.7202569821999829)
((3, 3), 0.6242486829920025, 1.6273940251681545)
```

Figure 1: expected result for both implementations

III. Further implementation of my KPIECE and flaws

As previously mentioned, while my implementation of KPIECE retains much of the original algorithm's logic, it omits certain components. Nevertheless, the algorithm demonstrates consistent runtime performance and serves as a proof of concept for sample-based motion planning. Specifically, it utilizes a grid-based data structure to track cells in the discretized state space, associates motions with their corresponding cells, and prioritizes cells that have converged less during traversal. To evaluate the implementation, I developed two versions of KPIECE: one in C++ and the other in Python. Both versions were tested under similar environmental conditions. The C++ implementation exhibited significantly faster runtimes, particularly when navigating large configuration spaces, due to its more efficient handling of computational tasks. However, despite the performance advantage in C++, both implementations failed to reach the goal when the grid was scaled to a size of 100 $\times$ 100. This failure to reach the goal, can be directly linked to the number of iterations we put into the method. The number of iterations, is a part of the method that is included in the original version of the KPIECE algorithm, that serve as the counter for how many times the algorithm will continue looking for nodes to travel to until the number of iterations is reached. Each iteration represents a single step in which the algorithm expanding the motion tree by selecting a cell, choosing a motion, and deciding to take that forward step to the goal state. This process begins by sampling a cell from the grid based on its importance metric, which helps prioritize regions of the space that are either unexplored or likely to lead toward the goal. With each iteration, the algorithm also updates the grid, motions data structure, and path to reflect newly explored regions. While iterations are crucial for incrementally progressing toward the goal without having a good estimate of what the iterations are, we run the risk of the using more iteration then needed, resulting in an increase of runtime, making the algorithm less efficient with each computation. While this over-estimation in iterations, does not pose a problem for bigger configuration space, as we work with a smaller grid size, the mass amount of iterations, can lead to making unnecessary calculations to find the goal and establish a valid path.

Algorithm 2 KPIECE Algorithm in C++

1: **Initialize grid** with obstacles
2: **Define** Grid, TreeNode, and Motion structures
3: **Define** functions for motion expansion, grid operations, etc.
4: **Function** `KPIECE(iterations, initialMotion, grid, bias)`
5: **Create** root node with initial motion
6: **Set** visited nodes to include the start position
7: **Add** initial motion to grid
8: **for** $i \leftarrow 1$ to *iterations* **do**
9: **Select** cell using bias from grid
10: **if** cell is invalid **then**
11: **Continue**
12: **end if**
13: **Select** motion from grid for selected cell
14: **if** motion is invalid **then**
15: **Continue**
16: **end if**
17: **Expand** motion to generate new motion
18: **if** new motion is valid **then**
19: **Add** new motion as child to the root node
20: **Add** new motion to grid
21: **if** goal is reached by new motion **then**
22: **Print** "Goal reached at position"
23: **Return** root node
24: **end if**
25: **end if**
26: **end for**
27: **Print** "No solution found within iteration limit."
28: **Return** root node

a. Runtime comparison

When comparing the C++ and Python implementations of KPIECE on the CPU, it is important to note that while the underlying logic remains the same, the algorithm runs significantly faster in C++. This performance improvement can be attributed to C++'s efficient memory management and lower-level access to system resources, which are crucial for computationally intensive algorithms like KPIECE. Even on smaller problem sizes, the

C++ implementation consistently outperforms its Python counterpart. While KPIECE has yet to be implemented in a parallelized, these findings suggest that further optimizations, such as GPU acceleration, could yield even greater improvements.

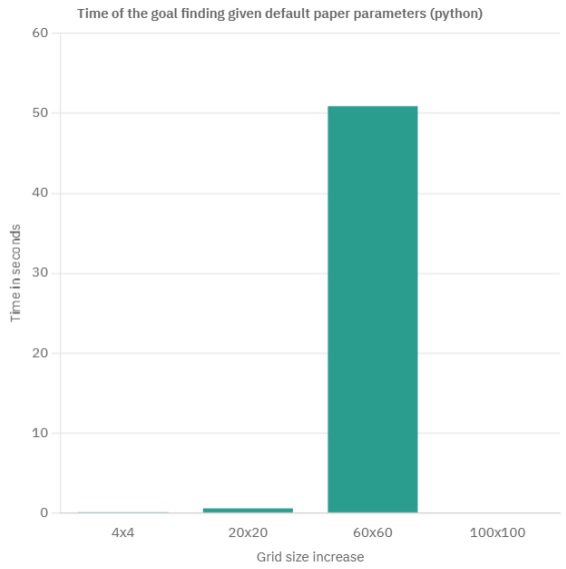

Figure 2: python runtime on CPU[KPIECE]

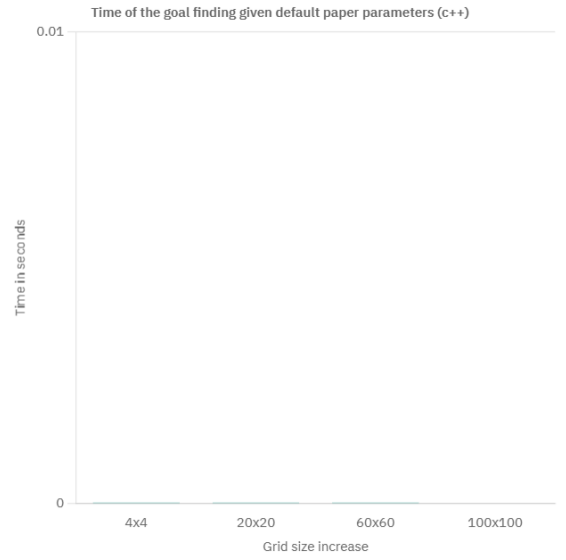

Figure 3: C++ runtime on CPU[KPIECE]

Take note of the 100x100 graph search having no runtime available do to its inability to completely its search for the goal. It is also worth noting just how little time it takes for the C++ algorithm to run on the CPU, especially for smaller configuration spaces. Another note about the these runtime comparisons, is that if there is each goal was scaled up along with the graph, so that each goal was directly in the middle of the given grid size.

VI. KPIECE in parallel

Although I was unable to successfully implement the KPIECE algorithm on the GPU in its entirety, my attempt led to valuable insights regarding the underlying causes of the failure to reach the goal within the given number of iterations. The primary issue appears to be related to the way in which multiple potential moves are evaluated for a given motion within the algorithm. In the current implementation, each movement expansion is performed sequentially, one per iteration, which can significantly limit the algorithm's efficiency. This limitation becomes particularly pronounced when working with larger grid sizes and more complex configuration spaces, where the number of possible states to explore increases exponentially. In such cases, it is highly likely that the algorithm will exhaust its available iterations before even considering the goal state, ultimately preventing the successful completion of the task.

A potential solution to this problem lies in parallelizing the movement expansion process. By distributing the computation across multiple processors, the algorithm can explore many possible configurations simultaneously, drastically reducing the time needed to evaluate all potential moves. This parallel approach would enable a much more rapid exploration of the configuration space, allowing the algorithm to cover a significantly larger number of possibilities within the same time frame. As a result, it would be much more likely to reach the goal state within the available iteration limits, improving the overall performance and effectiveness of the KPIECE algorithm, particularly in high-dimensional spaces.

While the current implementation of the KPIECE algorithm on the GPU demonstrates the foundational approach to motion expansion and grid-based exploration, there remain several opportunities for enhancement of this method is properly implemented. Such as the scalability in higher Dimensions. By Extending the algorithm to handle problems with higher-dimensional state spaces remains untested with the KPIECE. Studies of how this could be implemented on the GPU, could prove to be possible and effiecant. The motion planning of the KPIECE can be further expanding by seeing how incorporating the GPU implementation has an effect on the physical constraints the KPIECE was designed to handle. Questions such as if it would retain the same accuracy could be tested along with not this could speed up the decision making process.

Algorithm 3 Expand Motion Kernel on GPU

Initialize variables: `moves`, `visited`, `controls`, `newTimes`, `newPositions`, and `validResults`
Allocate memory on GPU for `moves`, `visited`, `controls`, `newTimes`, `newPositions`, and `validResults`
Copy `moves` and `visited` from host to device memory
Launch kernel `expandMotionKernel` with thread block configuration
Inside kernel
for each thread in grid **do**
    Calculate thread index: `idx = blockIdx.x * blockDim.x + threadIdx.x`
    if `idx < numMoves` **then**
        Extract move `(dx, dy)` from `moves[idx]`
        Calculate new position `newPosition = (x + dx, y + dy)`
        Check if `newPosition` is valid and not visited
        if `newPosition` is valid **then**
            Generate random control and time values
            Store values in `controls[idx]`, `newTimes[idx]`, `newPositions[idx]`
            Set `validResults[idx] = true`
        else
            Set `validResults[idx] = false`

            Copy results from device to host: `controls`, `newTimes`, `newPositions`
            Free GPU memory

V. K-Nearest Neighbor Search Algorithm (KNNS)

To reiterate, KNNS goes through a sample set of the configuration space S. For each point q in S, it finds the closest k points from it, creating k potential paths from q.

A. KNNS on the CPU

For the serial implementation on the CPU, we first generate a sample of the configuration space that is some percentage p of it. We add the goal point g to S. For each point in q in S, we serially find the k nearest neighbors of q. We store the k nearest neighbors for each point in a global map structure. With this mapping, we can then generate all the possible paths from start to goal. The generation would like this: "From start, connect the k nearest neighbors. Connect the neighbors of the first neighbor to the first neighbor. Connect the neighbors of the second neighbor to the second neighbor...Connect the neighbors of the kth neighbor to the kth neighbor...". This tree-building-like generation continues for each node that is part of a valid path until 1) the path to node becomes invalid (there is a repeat in the path or does not end in a goal state after) or 2) the path's latest addition is a goal state. A general implementation of KNNS can be seen in Algorithm 4.

Algorithm 4 KNNS Algorithm on CPU

Input: sample set S, neighbor size k
Output: shortest path from start to goal p
$knnMap \leftarrow \{\}$
for $q \in \mathcal{S}$ **do**
    $knnMap[q] \leftarrow findNeighbors(q, k)$
end for
$p \leftarrow findShortestPath(knnMap, k, S)$
if p is empty **then**
    PRINT("No solution found for this sample set and neighbor size k.")
    return p
end if
PRINT("The shortest path found: ")
return p =0

B. KNNS on the GPU

In order to incorporate parallelism to the process, we parallized the process of finding k neighbors for each point q. Each thread works on a point p to find its neighbors and add it to the global map structure of point to its neighbors. So the number of threads is equal to the number of points in S. Everything else remains the same and is done serially.

Algorithm 5 shows this implementation each thread would go through.

Algorithm 5 KNNS Algorithm on GPU

Input: sample set S, neighbor size k
Output: shortest path from start to goal p
$knnMap \leftarrow \{\}$
$q \leftarrow S[threadIdx.x]$
$knnMap[q] \leftarrow findNeighbors(q, k)$
if threadIdx.x = 0 **then**
    $p \leftarrow findShortestPath(knnMap, k, S)$
    if p is empty **then**
        PRINT("No solution found for this sample set and neighbor size k.")
        return p
    end if
    PRINT("The shortest path found: ")
    return p
end if

C. Runtime Comparison

After running the KNNS search on the implementation, there are two significant features that we noticed with the runtime comparison.

i. CPU is faster than GPU for 4x4 and 10x10 grid size

For the 4x4 and 10x10 grids, the algorithm runs in 0.12 and 0.13 seconds on the GPU respectively while the algorithm runs in 0.00018 and 0.00099 seconds on the CPU. In the beginning of the grid search increase, the CPU runs more than 100 times faster than the GPU.

ii. GPU is faster than CPU for 100x100 grid size

For the 100x100, the algorithm runs in 0.16111 seconds on the GPU. When it comes to the CPU, the algorithm is still looking for the k nearest neighbors for a point in the sample set after 30 min and finds the shortest path in about 58 minutes. The change from a 10x10 grid size to a 100x100 grid seach shows an exponential increase in runtime on the CPU.

VI. Conclusion

The KPIECE algorithm is an algorithm that tries to find a path from start to goal by probing its surroundings and building towards the goal given a set amount of iterations. Because KPIECE is heavily constrained to the amount of iterations the user decides to pick, KPIECE could benefit from the use of parallelism. The runtimes collected for

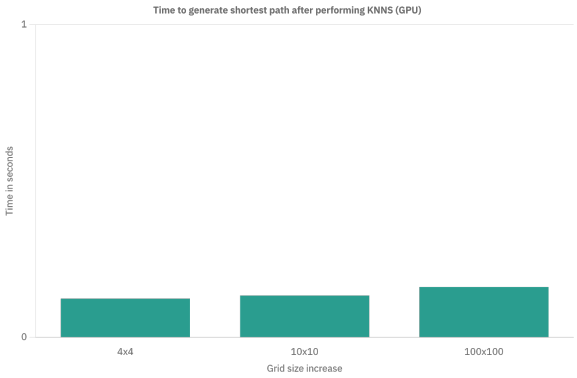

Figure 4: KNNS on GPU

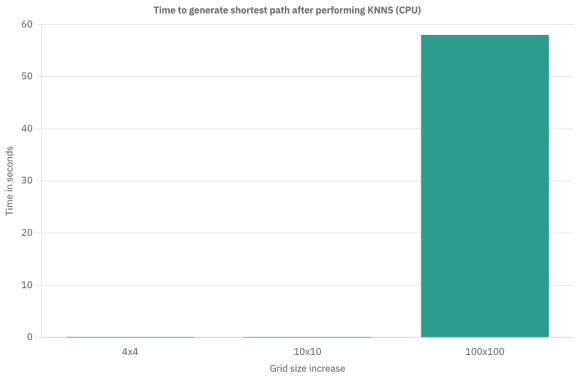

Figure 5: KNNS on GPU

KPIECE implies that the algorithm will most likely work for a medium configuration space, rather than a larger one (e.g. 100x100). On the other hand, The KNNS algorithm is an algorithm that allows you to generate all possible paths to begin with by probing the neighbors of each sample point. From there, you can perform a graph search algorithm to find the shortest path from start to finish. Given the search space of paths built off of nearest neighbors, the algorithm can greatly benefit from the use of parallelism. By just parallelizing the search of neighbors for each points, the GPU was able to run much faster for a 100x100 grid. This runtimes collected indicates that the KNNS in parallel is more likely to work better in large grid spaces (e.g. 100x100). Both implementations of these algorithms have room for improvement to better performance and decrease time complexity, potentially increasing the benefit of parallelism.

References

1) J. Pan, C. Lauterbach and D. Manocha, "Efficient nearest-neighbor computation for GPU-based motion planning," 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipei, Taiwan, 2010, pp. 2243-2248, doi:

10.1109/IROS.2010.5651449.

2) A. Sucan and L. E. Kavraki, "A Sampling-Based Tree Planner for Systems With Complex Dynamics," IEEE Transactions on Robotics, vol. 28, no. 2, pp. 308-322, April 2012, pp. 116 - 130, doi: 10.1177/0278364911429335