Path Planning Acceleration with GPU for an Omnidirectional Mobile Robot

Alejandro Dumas León, Eduardo Arturo Mendoza Gómez, Jorge Tomás Araujo González, Ulises Orozco-Rosas, Kenia Picos*

CETYS Universidad, Av. CETYS Universidad No. 4. El Lago, C.P. 22210, Tijuana B.C., México
[alejandrodl, arturo.mendoza, jorget.araujo]@cetys.edu.mx, [ulises.orozco, kenia.picos]@cetys.mx

Abstract

Path planning in a state space using the iterative deepening method is a complex problem that can be accelerated using a GPU. In this approach, the state space is divided into smaller subspaces and iterative depth search is applied to each of these. The parallel capabilities of the GPU are utilized to process several subspaces concurrently. Furthermore, the shared memory in the GPU can be leveraged to store relevant data and reduce access time to the global memory. Implementing this approach on the GPU can provide significant acceleration compared to CPU execution. However, careful optimization and parameter tuning are required to utilize the GPU's capacity fully. In addition to a detailed description of the proposed methodology, experimental results are presented that demonstrate the superiority of our approach compared to traditional CPU-based methods. These results highlight the potential of GPUs to transform trajectory planning in mobile robots, offering a route to faster and more efficient solutions. Trajectory planning in state spaces represents a significant challenge in mobile robotics, particularly in applications that demand fast and efficient responses in dynamic and complex environments. This work introduces a novel method to accelerate route planning in an omnidirectional mobile robot fusing advances in hardware with sophisticated algorithmic techniques, a new paradigm is established in path planning for omnidirectional mobile robots, marking an important milestone in the search for more agile and capable robotic systems.

Key Words— Iterative Deepening Approach, Mobile Robots, Parallel Computing, Path Planning, State Space.

Resumen

La planeación de ruta en un espacio de estados utilizando el algoritmo de profundidad iterativa es un problema complejo que puede ser acelerado con el uso de una GPU. En este acercamiento, el espacio de estados es dividido en subespacios más pequeños y se aplica la búsqueda en cada uno a través del algoritmo de profundidad iterativa. Las capacidades del procesamiento paralelo de la GPU se utilizan para procesar una gran cantidad de subespacios concurrentemente. Además, la memoria compartida en la GPU puede ser utilizada para almacenar datos de gran utilidad y reducir el tiempo de acceso a la memoria global. Implementar este algoritmo en la GPU puede acelerar significativamente la ejecución del programa en comparación con la CPU. Donde requiere una configuración rigurosa de los parámetros para optimizar y aprovechar al máximo las capacidades de la GPU. Los resultados experimentales demuestran la superioridad de nuestro enfoque en comparación con los métodos tradicionales basados en CPU. Estos resultados resaltan el potencial de las GPU para transformar la planificación de trayectoria en robots móviles, ofreciendo una ruta hacia soluciones más rápidas y eficientes. La planificación de trayectoria en espacios de estados representa un desafío importante en la robótica móvil, particularmente en aplicaciones que exigen respuestas rápidas y eficientes en entornos dinámicos y complejos.

Palabras Clave— Algoritmo de Profundidad Iterativa, Cómputo Paralelo, Espacio de Estados, Planeación de Ruta, Robots Móviles.

1. INTRODUCTION

Pursuing faster and more efficient algorithms is a constant quest in the ever-evolving landscape of computational science and problem-solving. Among the many challenges researchers and engineers face, path planning within a complex state space is a formidable puzzle. This intricate problem, with its numerous dimensions and intricate calculations, has long demanded innovative approaches to achieve practical solutions. One such approach, at the forefront of contemporary computational strategies, seeks to harness the immense computational power of Graphics Processing Units (GPUs).

* Corresponding author.

This groundbreaking technique addresses the computational complexity of path planning and offers the tantalizing prospect of unlocking unprecedented speeds and efficiencies. The core principle revolves around dividing the intricate state space into more manageable subspaces. An interactive depth search is applied with meticulous precision within each of these subspaces.

What sets this approach apart is the remarkable parallel processing prowess inherent to GPUs. Unlike traditional Central Processing Units (CPUs), GPUs consist of numerous cores that considerably increase the number of tasks that can be performed concurrently. This inherent parallelism allows concurrently handling multiple subspaces, fundamentally transforming the path-planning process.

Furthermore, using shared memory within the GPU is a crucial catalyst in this computational alchemy. By storing essential data within this shared memory, access times to the global memory are dramatically reduced, optimizing the computational workflow. The cumulative effect of these innovations results in a seismic shift in path planning speed, offering a quantum leap beyond what traditional CPU-based executions can achieve.

Nevertheless, as with any transformative technology, the full realization of GPU-accelerated path planning demands meticulous optimization and parameter tuning. The harnessing of the GPU's vast computational potential is a nuanced art that requires finesse and expertise. Researchers and practitioners alike must navigate the intricate terrain of algorithmic optimization and hardware utilization to unlock the GPU's unparalleled capacity fully.

In this realm of computational innovation, the fusion of state space, iterative deepening approach, and GPU acceleration represent a powerful trifecta that has the potential to redefine the boundaries of what is computationally achievable. It is a testament to the relentless pursuit of efficiency and the ceaseless exploration of novel approaches to complex problems. As we delve deeper into the intricacies of this groundbreaking technique, we embark on a journey into the heart of computational discovery, where the boundaries of what is possible continue to expand, setting new standards in path planning and beyond.

The task is to perform pathfinding in a state space, initially sequentially and ultimately in parallel, using the GPU. Therefore, path planning in state space using the iterative deepening approach on a GPU (specifically the CUDA programming language) can provide a faster and more efficient solution for this complex problem. This could be highly beneficial in robotics and autonomous systems applications such as route planning [3,4]. Using the iterative depth method, the main objective of this work is to employ parallel computing with the GPU in an omnidirectional mobile robot to perform path planning in state space.

The article's structure follows: Section 2 starts with a theoretical background. Section 3 presents the proposal for this work, which describes the software, hardware, and mechanical implementation. The results will be presented in Section 4, where a comparison between sequential and concurrent execution shows the best path in a minimum time. Finally, Section 5 presents a conclusion of this proposal.

2. THEORETICAL FRAMEWORK

This section comes into the theoretical foundations that underpin the study, concentrating on a spectrum of concepts crucial for a comprehensive understanding of the research approach. Central to this exploration is "Path Planning," a key element in disciplines like robotics and computer science, which involves identifying an optimal route or strategy [10].

The "Iterative Deepening Approach" is also examined, an algorithmic technique that skillfully balances depth and breadth in search processes, particularly pertinent in environments with vast state spaces. The notion of "State Spaces," embodying all conceivable configurations or conditions in a given problem, is fundamental to this discourse, providing a structure for algorithmic problem-solving.

Additionally, the increasing importance of "Parallel Programming" in enhancing computational efficiency is acknowledged as a vital facet of contemporary computing paradigms. Within this context, "CUDA," NVIDIA's parallel computing platform and programming model, is highlighted for its transformative impact on handling computing tasks, especially in high-performance computing scenarios. This section aims to provide a thorough background, laying the groundwork for the subsequent application and examination of these concepts within the specific research field.

Trajectory planning: is a fundamental concept in robotics and control systems, focusing on determining the optimal path a robot should follow over time. It involves not only the selection of a path from one point to another but also the consideration of how an entity moves along this path, factoring in its speed, acceleration, and other dynamic aspects [7]. The main objective of trajectory planning is to develop a sequence of movements or a trajectory that allows the robot to perform its task efficiently and in compliance with its physical and operational constraints [5].

As Steven LaValle's book "Planning Algorithms" detailed, trajectory planning primarily involves solving two interrelated problems. First, it requires the determination of a path that the robot can follow, avoiding obstacles and ensuring feasibility within its operational environment. This path is often calculated in a state space, denoted as X, where each state x comprises the robot's configuration (q) and its velocity (q'). The second aspect of trajectory planning involves figuring out the appropriate velocities and accelerations (q') at each point along the path, ensuring that these movements respect the robot's mechanical limitations and differential constraints.

Iterative Deepening Approach: A tree search algorithm that explores different depths to find an optimal or satisfactory solution. Starting from a root node, successor nodes are recursively explored. Iterative depth involves conducting a depth search iteratively, starting with a given maximum depth and gradually increasing this maximum in each iteration until the desired solution is found. This efficient and memory-conservative method makes it suitable for large search spaces. The idea is to use a depth-first search and find all states that are distance i or less from $x1$. If the goal is not found, the previous work is discarded, and depth-first is applied to find all states of distance $i+1$ or less from $x1$[1].

State Space: It encompasses all conceivable scenarios within a given context. This could range from the position and orientation of a robot, the locations of tiles in a puzzle, to the position and velocity of a helicopter. LaValle emphasizes the versatility of state spaces, acknowledging that they can be either discrete—finite or countably infinite—or continuous, which means uncountably infinite.

A critical aspect highlighted by LaValle is the implicit representation of the state space in most planning algorithms. Due to the vast number of states or their combinatorial complexity, it is impractical to explicitly represent the entire state space in most applications. Nevertheless, defining the state space remains a crucial step in formulating planning problems and designing and analyzing algorithms to solve them.

LaValle advises careful definition of the state space in specific applications, ensuring that irrelevant information is not encoded into a state, thereby maintaining the efficiency and relevance of the planning algorithm. This thoughtful approach to defining state spaces underscores their importance in the development and application of planning algorithms.

Parallel Programming: This involves designing computer programs and algorithms for concurrent execution on multiple processors or cores, enhancing speed and efficiency. Traditional sequential programming executes instructions one after the other, limiting performance in data-intensive or computationally heavy tasks. Parallel programming leverages multiple processing units concurrently to expedite task execution [2].

One of the fundamental concepts in parallel programming is task decomposition. It involves breaking down a larger computational task into smaller, more manageable subtasks that can be executed concurrently. The key challenge here is to ensure that these sub-tasks can run independently and efficiently without causing conflicts or bottlenecks. Effective task decomposition is essential for harnessing the full potential of parallelism [11].

Parallel programming encompasses various forms of parallelism, each suited to several types of problems. **Task-level** parallelism divides a program into smaller units of work that can run in parallel. **Data-level** parallelism involves distributing data across multiple processing units and performing identical operations on different data elements simultaneously. **Instruction-level** parallelism exploits parallelism within a single instruction stream, often seen in modern processors with pipelining or superscalar execution capabilities [12].

Parallel programming models can be categorized into two main types: **shared memory** and **distributed memory**. Shared memory parallelism involves multiple threads or processes sharing a common memory space, allowing them to communicate by reading and writing to shared data structures [9]. In contrast, distributed memory parallelism relies on message passing, where processes or nodes communicate through explicit messages, each having its private memory space. Distributed memory models are commonly used in high-performance computing (HPC) environments with frameworks like MPI. In parallel programming, **synchronization** mechanisms are crucial to coordinate the execution of concurrent tasks. These mechanisms ensure that tasks do not interfere with each other or access shared resources simultaneously, which could lead to data corruption or race conditions. Common synchronization primitives include locks, semaphores, barriers, and atomic operations, helping to maintain the order and integrity of parallel execution.

Efficient parallel programs require **load balancing** to distribute the workload evenly among available processors or cores. Load imbalances can lead to situations where some processors are idle while others are overwhelmed, resulting in suboptimal performance. Achieving good scalability, where the program's performance scales with the number of available processing units, is a significant challenge in parallel programming and often requires careful design and optimization [6].

CUDA: NVIDIA's parallel computing platform, allowing general-purpose computing on graphics processing units (GPUs). Modern GPUs are highly parallel architectures capable of executing multiple computational tasks simultaneously, unlike traditional GPUs designed solely for graphics rendering.

3. PROPOSAL

Figure 1 delineates the sequential stages involved in CUDA-based computation, starting from the allocation of memory on the host (CPU) and the device (GPU), through to the transfer of data between host and device.

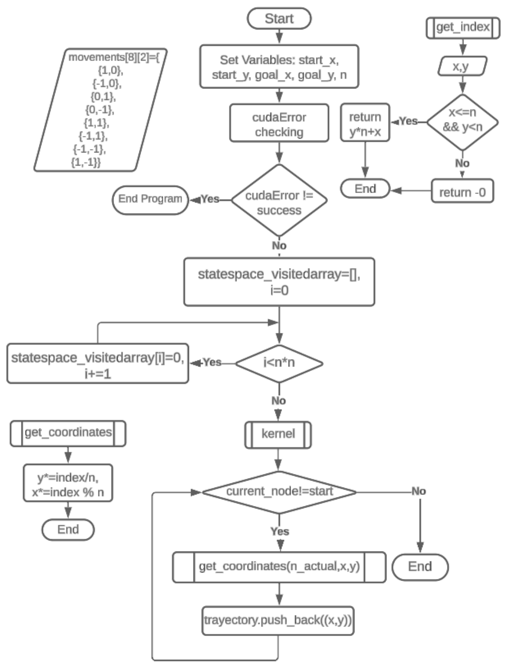
Start
Set Variables: start_x, start_y, goal_x, goal_y, n
movements[8][2]={ {1,0}, {-1,0}, {0,1}, {0,-1}, {1,1}, {-1,1}, {-1,-1}, {1,-1}}
cudaError checking
cudaError != success
Yes
End Program
No
statespace_visitedarray=[], i=0
i<n*n
Yes
statespace_visitedarray[i]=0, i+=1
No
kernel
current_node!=start
Yes
No
get_coordinates(n_actual,x,y)
trayectory.push_back((x,y))
End
get_index
x,y
x<=n && y<n
Yes
return y*n+x
No
return -0
End
get_coordinates
y*=index/n, x*=index % n
End
Figure 1. Main program diagram.

Figure 2 details the execution of parallel kernels on the GPU, displaying the distribution of tasks across multiple threads and blocks within the GPU's architecture. Following the kernel execution, the diagram illustrates the process of transferring the computed results back to the host.

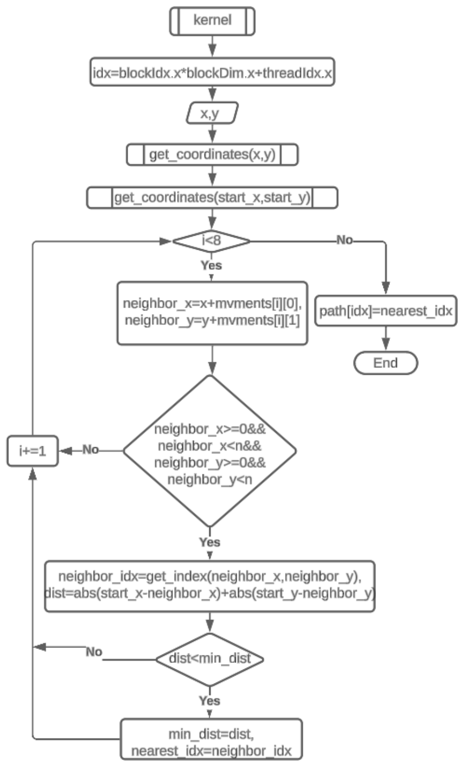
kernel
idx=blockIdx.x*blockDim.x+threadIdx.x
x,y
get_coordinates(x,y)
get_coordinates(start_x,start_y)
i<8
No
Yes
neighbor_x=x+mvments[i][0],
neighbor_y=y+mvments[i][1]
path[idx]=nearest_idx
End
i+=1
No
neighbor_x>=0&&
neighbor_x<n&&
neighbor_y>=0&&
neighbor_y<n
Yes
neighbor_idx=get_index(neighbor_x,neighbor_y),
dist=abs(start_x-neighbor_x)+abs(start_y-neighbor_y)
No
dist<min_dist
Yes
min_dist=dist,
nearest_idx=neighbor_idx

Figure 2. CUDA kernel diagram.

Set Initial State, Goal State, and Obstacles. In Figure 3, the initial state marks the beginning point, defining the exact location and orientation from which the journey or operation will start. This state is pivotal as it serves as the cornerstone for all subsequent path calculations. Following this, the goal state is established, delineating the intended destination or endpoint. This goal is crucial as it guides the trajectory and final aim of the path-planning algorithm. Lastly, identifying obstacles forms an integral part of the process. These obstacles, which may vary in nature and dynamics, create constraints within which the path planning must efficiently navigate. By accurately setting these elements, the system is equipped to utilize parallel computing techniques, enabling it to concurrently explore multiple pathways and scenarios, significantly enhancing the efficiency and effectiveness of the path-planning solution.

Creation of State Space. The creation of state space is a fundamental step that involves defining all possible states that the system or agent can occupy within its environment.

Neighbor Calculation on Each State. This involves analyzing the connectivity and accessibility between states based on the system's movement rules and environmental constraints. For each state, the algorithm computes a set of neighboring states, which are the potential next steps the entity can take. This calculation considers several factors such as distance, direction, possible obstacles, and specific movement capabilities of the entity [8].

Concurrently choose the best neighbor for each state. This method involves evaluating all adjacent states or neighbors for each state in the system simultaneously, rather than sequentially. By leveraging parallel processing, the algorithm can assess multiple paths and options at once, significantly speeding up the decision-making process.

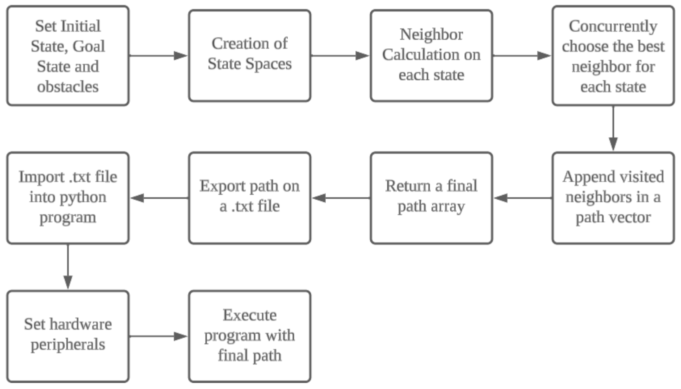
Set Initial State, Goal State and obstacles
Creation of State Spaces
Neighbor Calculation on each state
Concurrently choose the best neighbor for each state
Append visited neighbors in a path vector
Return a final path array
Export path on a .txt file
Import .txt file into python program
Set hardware peripherals
Execute program with final path

Figure 3. System block diagram.

Append visited neighbors in a path vector. As the algorithm progresses, each neighbor that is visited or deemed a viable step towards the goal is added to the path vector. This vector acts as a dynamic record, documenting the sequence of states traversed by the entity. By maintaining this record, the algorithm not only keeps track of its current position concerning the start and goal points but also avoids revisiting the same state, thereby preventing redundant calculations and potential loops.

Return a final path array. This final path array is a structured collection of states or steps, sequentially arranged to guide the entity through the most efficient, safe, or otherwise specified route. It represents the algorithm's solution to the navigational challenge, distilled from the complex exploration of state spaces and neighbor evaluations.

Export path on a *.txt file. This process involves translating the final path array, which comprises a sequence of coordinates or steps, into a text format that is easily readable and accessible in Python.

Import *.txt file into python program. This step implies importing the final path into the Python program to tell the mobile robot the path to follow.

Set hardware peripherals. Configuring and initializing external devices – motors and communication modules – that the main processor or microcontroller will interact with. Each peripheral must be correctly connected, and its communication protocols firmly established, ensuring seamless data exchange and control.

Execute the program with the final path. The mobile robot performs the best-established path from the initial state to the final state avoiding obstacles.

Figure 4 presents a layered arrangement of various robotic components, each distinct in its function and appearance. On the top, there is a Jetson Nano, notable for its compact black casing, serving as the brain of the setup with its powerful computing capabilities. In the middle layer, we find an array of components: a Stepper Motors Driver, easily identifiable by its purple color, which plays a crucial role in controlling the precision of motor movements; alongside there are Antennas, represented as black cylinders, crucial for wireless communication; and a Robot Handler, comprising white semi-ellipses, which is likely involved in manipulating or interfacing with other parts of the robot.

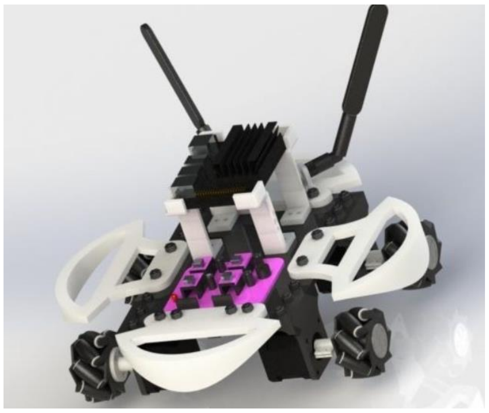
Figure 4. 3D Render view.

On the bottom, the setup is completed with Mecanum wheels, known for their unique ability to move in any direction, which adds a versatile range of motion to the robotic assembly. This layered configuration illustrates a sophisticated blend of electronic and mechanical engineering, each component working in harmony to create a versatile and functional robotic system.

4. RESULTS

The implemented algorithm is iterative depth, which was achieved using object-oriented programming where two classes were created: a graph class with attributes including value, neighbors, and end flag, and a node class with attributes such as start, end, grid, depth, cost, path, prohibited states, and stack. The program operates through the value of an initial node that calls the neighbor calculation function, where the neighbors are calculated. It is ignored if any of the neighbors is prohibited; otherwise, all other neighbors are added to the stack. The next node is the stack's last element, and the current node's neighbors are recalculated recursively until the current node becomes the goal state. If a node has no neighbors and is not the goal state, it is removed from the stack and marked as visited.

Regarding the hardware, a development graphics card was implemented, capable of carrying out parallel processing in its future version. The unit can process and execute the actions required to move from an initial state to its goal state. On the software side, the unit executes the algorithm in C++, while the hardware runs in Python for its simplicity, using libraries like time and GPIO to facilitate pulse control. A bash file executes all the necessary files to carry out the algorithm and enable the robot to execute its trajectory.

Path planning results. Figure 5 shows the resulting path (yellow crosses) beginning from the initial state: (3,3) and final state: (7,7). Black crosses represent all the visited states across the 10×10 grid. Obstacle coordinates are (4,4), (5,4), (6,4), (4,5), (5,5).

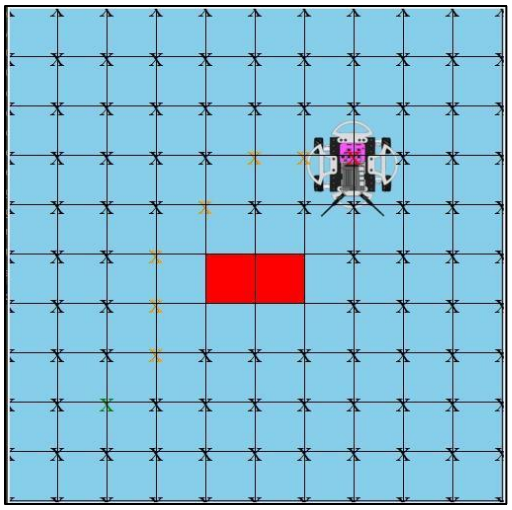
Figure 5. Resulting path with 10x10 grid size.

Figure 6 shows the resulting path beginning from the initial state: (4,4) and final state: (14,14). Black crosses represent all the visited states across the 20×20 grid. Obstacle coordinates are (8,8), (9,8), (10,8), (11,8), (12,8), (8,9), (9,9), (10,9), (11,9), (12,9), (8,10), (9,10), (10,10), (11,10), (12,10).

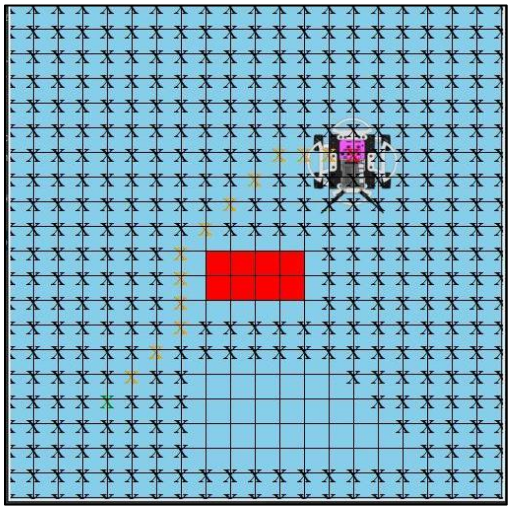
Figure 6. Resulting path with 20x20 grid size.

Performance results. Figure 7 shows a graphical representation of the execution time, measured in seconds, for computing paths in parallel computing vs sequential computing. The graph specifically illustrates how this execution time varies concerning the size of the computational grid. This visual depiction allows for an easy comparison of performance across different grid sizes, highlighting the efficiency and scalability of the parallel computing approach in path calculation tasks.

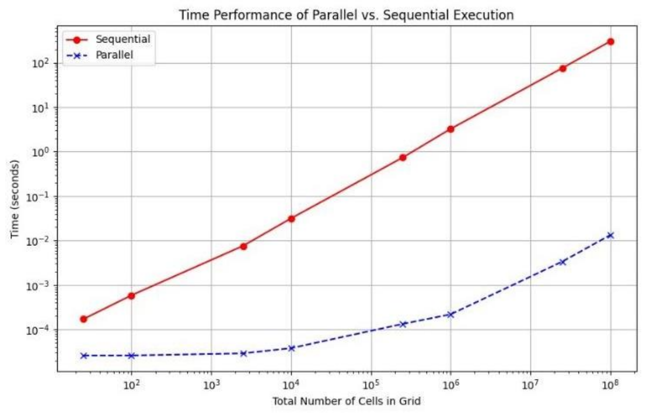

Figure 7. Execution Time vs. Grid Size.

5. CONCLUSIONS

The work's primary goal, which was to implement iterative depth for a mobile agent operating within a state space using CUDA C/C++, has been accomplished. The utilization of parallel programming on the CUDA-enabled device has enabled the efficient execution of the algorithm, although with some minor margins of error that have not hindered overall functionality. This work serves as a compelling demonstration of the feasibility of applying data parallelism techniques to control and robotics projects. The use of CUDA libraries has proven instrumental in achieving parallelism within the source code, displaying the potential for leveraging GPU computing to enhance the performance and capabilities of systems in the realm of control and robotics. This achievement highlights the promising future of parallel programming in pushing the boundaries of computational efficiency in various domains.

Furthermore, this work underscores the growing importance of parallel programming and GPU acceleration in addressing complex computational challenges. With the continuous advancement of hardware technology, such as GPUs, the potential for leveraging parallelism to tackle intricate problems across various fields continues to expand. The successful application of CUDA C/C++ in this context not only demonstrates its versatility but also encourages further exploration of parallel computing solutions for even more ambitious projects in control, robotics, and beyond. As parallel programming techniques continue to evolve and mature, they hold the promise of unlocking new horizons in terms of computational power and efficiency, driving innovation and breakthroughs in numerous domains.

6. REFERENCES

[1] S. M. LaValle. (2006). "Planning Algorithms," Cambridge University Press.

[2] W. W. Hwu and D. B. Kirk. (2010). "Programming Massively Parallel Processors," Morgan Kaufmann.

[3] Orozco-Rosas, U., Picos, K., Montiel, O. (2020). Acceleration of Path Planning Computation Based on Evolutionary Artificial Potential Field for Non-static Environments. Intuitionistic and Type-2 Fuzzy Logic Enhancements in Neural and Optimization Algorithms: Theory and Applications. Studies in Computational Intelligence. Volume 862. Springer.

[4] Orozco-Rosas, U., Picos, K., Montiel, O., Castillo, O. (2020). Environment Recognition for Path Generation in Autonomous Mobile Robots. Hybrid Intelligent Systems in Control, Pattern Recognition and Medicine. Studies in Computational Intelligence. Volume 827. Springer.

[5] W. Kowalcyzk, M. Przybyla, K. Kozlowski. (2017). "Set-point Control of Mobile Robot with Obstacle Detection and Avoidance Using Navigation Function - Experimental Verification". Journal of Intelligent & Robotic Systems. Volume 85, pp. 539-552, Springer.

[6] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, St. Louis, MO, USA, 1985, pp. 500-505.

[7] V. Sezer, M. Gokasan. (2012). "A novel obstacle avoidance algorithm: Follow the Gap method". Robotics and Autonomous Systems. Volume 60, pp. 1123-1134. Elsevier.

[8] S. S. Ge and Y. J. Cui, "New potential functions for mobile robot path planning". IEEE Transactions on Robotics and Automation. Volume 16, no. 5, pp. 615-620, Oct. 2000.

[9] R. Trobec, M. Vajtersic, P. Zinterhof. (2009). "Parallel Computing: Numerics, Applications, and Trends". pp. 471-510, Springer.

[10] Orozco-Rosas, U., Montiel, O., Sepúlveda, R. (2017). An Optimized GPU Implementation for a Path Planning Algorithm Based on Parallel Pseudo-bacterial Potential Field. Nature-Inspired Design of Hybrid Intelligent Systems. Studies in Computational Intelligence. Volume 667. Springer.

[11] Orozco-Rosas, U., Montiel, O., Sepúlveda, R. (2015). Parallel Evolutionary Artificial Potential Field for Path Planning—An Implementation on GPU. Design of Intelligent Systems Based on Fuzzy Logic, Neural Networks and Nature-Inspired Optimization. Studies in Computational Intelligence. Volume 601. Springer.

[12] Orozco-Rosas, U., Montiel, O., Sepúlveda, R. (2018). Parallel Bacterial Potential Field Algorithm for Path Planning in Mobile Robots: A GPU Implementation. Fuzzy Logic Augmentation of Neural and Optimization Algorithms: Theoretical Aspects and Real Applications. Studies in Computational Intelligence. Volume 749. Springer.