FPGA-AMP: FPGA Accelerated Motion Planning

Matt Ngaw, Yufei Shi, Chris Stange
Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Hardware acceleration of key robotic computations is essential to make robots more viable for tasks that require fast response times and reactivity to the environment. Motion planning is a critical step in the robotics pipeline that generates paths for the robot to follow. Motion planning algorithms, such as Rapidly-exploring Random Trees (RRT), is too slow when run on CPUs and too power-inefficient when run on GPUs. In our capstone project, we used a Field-Programmable Gate Array to accelerate motion planning and reduce its power consumption. Our FPGA implementation of RRT achieves 18x speedup over the baseline implementation at a 70 percent decrease in power and a 98 percent decrease in energy.**

***Index Terms*—Field-Programmable Gate Array, Hardware Acceleration, High-level Synthesis, Motion Planning, Rapidly-exploring Random Trees, Robotics)**

1 INTRODUCTION

Recent advancements in computing technologies have facilitated the emergence of increasingly sophisticated robots, which play an ever-growing role in our society [1], [2]. Robots are becoming more capable of performing tasks that normally risk people's safety and health. These tasks typically require quick thinking and fast reaction times. Thus, the hardware driving the computation within robots must keep up with the increasing challenges of their use cases.

Key steps in the robotics computing pipeline include perception, motion planning and dynamics. Motion planning is particularly crucial because it is one of the more compute-intensive tasks. Motion planning is the task of calculating a series of valid configurations to get from a starting position to a destination position [3]–[5]. There are various algorithms for generating motion plans, among which Rapidly-exploring Random Trees (RRT) is often used to efficiently search a high-dimensional space for collision-free trajectories. [6]–[9].

RRT has conventionally been run on central processing units (CPUs) and graphics processing units (GPUs). While providing generality in computing and ease of programming, CPUs are not performant at computing RRT. GPUs, while better suited for an algorithm with parallelism like RRT, are not power efficient enough to be viable in a robotics system with power constraints. There are many use cases for robots where performance and power efficiency are paramount. Thus, it is necessary to find a different solution.

Field Programmable Gate Arrays (FPGAs) are capable of significantly accelerating algorithms like RRT while consuming less power compared to traditional CPU or GPU implementations. Hence our project aims to leverage FPGAs to enhance the speed and efficiency of motion planning. We plan to develop an end-to-end system that uses the FPGA-AMP accelerator to guide the motion of a robotic arm.

2 USE-CASE REQUIREMENTS

2.1 Accurate Motion Planning

Motion plans that are collision-free are essential for autonomous robotics. Robotic systems that fail to avoid obstacles will likely be unsafe and be repeatedly broken. These social and economic ramifications necessitate that the system we create generate accurate motion plans. Our motion planning module consists of two steps, collision detection and path generation. Using data generated by a perception system, collision detection is needed to determine what viable, collision-free paths exist in the state space. Path generation then uses this result to find the shortest path between the start and goal position. Algorithms that are commonly used for motion planning are Probabilistic Roadmap (PRM) and Rapidly Exploring Random Tree (RRT). Both of these algorithms are non-deterministic, which means they are not guaranteed to converge on a single solution. Algorithms for path generation consist of A* search, Dijkstra Algorithm, and heuristic path smoothing steps. We targeted designing a motion planning system that will **generate accurate, collision-free paths at least 95 percent of the time.**

2.2 Rapid Motion Planning

Using robotics in dangerous environments that are typically reserved for humans involves enabling the robot to adapt and react to its surroundings. To do so, they must be able to quickly generate motion plans that account for dynamic, changing scenes. When run on modern CPUs, motion planning is generally too slow to handle such situations [1]. Academic research shows that using an FPGA for motion planning can achieve a 1000x speedup over a CPU [4], [5]. Since motion planning makes up the majority of the processing time on a robot, this speedup should be sufficient to allow the robot to interact with dynamic scenes [1]. Due to improvements in modern CPUs since the research we cite, we targeted a **10x speedup of motion planning over a baseline CPU implementation**.

2.3 Efficient Motion Planning

Power and energy efficiency are important when considering the environmental impacts of electricity generation and the long-term economics of using robots. Efficiency is also critical for robotic systems that may be used in remote environments and/or are battery-powered. Modern CPU implementations of motion planning are not only slow, but they are also extremely power inefficient. While GPUs can achieve considerable speedup over CPUs, they suffer the same problem of power inefficiency. Therefore we use an FPGA, which allows us to maintain the speedup advantages of the GPU at a fraction of the power cost. We targeted a **70 percent decrease in power and a 90 percent decrease in energy consumption when generating motion plans versus a baseline CPU implementation.**

3 ARCHITECTURE AND PRINCIPLES OF OPERATION

3.1 Architecture

The FPGA-AMP system is an end-to-end motion planning solution and we have integrated it into a complete robotics pipeline. Our robotic system is able to observe obstacles in a scene, generate a collision-free path (if one exists), and convert the path to a motion plan. This naturally separates the system architecture into three major components:

- Perception:
  Observes and maps a scene
  This includes the Kinect One Camera and the laptop in Figure 1.

- RRT Acclerator:
  Generates RRT tree, representing a set of collision-free paths through the state space
  This includes the Ultra96v2 FPGA board and the laptop in Figure 1.

- Kinematics:
  Finds the shortest collision-free path, generates a viable motion plan, sends commands to the arm
  This includes the Robotic Arm & Controller, and the laptop in Figure 1.

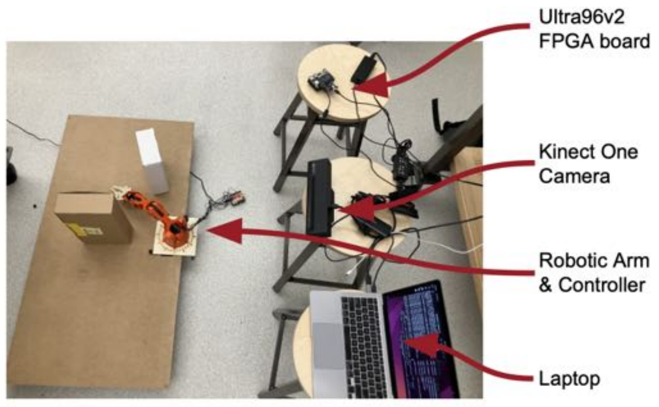

Figure 1: Overall physical system setup.

Compared with the system architecture in our design report, instead of doing a full-ROS integration on the Kria KR260 board, our final system architecture uses a laptop to orchestrate all the components. This change was necessary due to the fact that we had to switch the FPGA we were using from the Kria to the Ultra96v2. We spent a significant amount of the semester trying to get the Kria to build a project but even with the help of people from AMD we were not able to do so. We switched to the Ultra96v2 because we knew it would work and were running out of time. The Ultra96v2 does not have native ROS2 support which means we needed to add the laptop to run perception. The laptop receives raw perception data from the camera which is then processed and sent to the Ultra96v2 over a local network. We run RRT on the FPGA and pass the generated tree back to the laptop. The laptop finished the pipeline by running kinematics and sending commands to the arm.

3.2 Principles of Operation

3.2.1 Engineering

We applied principles of engineering both in the formulation and approach we took to our project. Our project began by identifying a real world problem for which there existed a gap in performance when using current methods. We then applied our systems programming and hardware design skills to devise a solution that would improve the state of the art. When implementing our robotics system we broke it into three main subsystems. Dividing up the work like this made the project more manageable and allowed us to work in parallel. Despite this partitioning of the workload, we ensured that we were still knowledgeable about all the components and lent help to one another whenever necessary. In short, we used the engineering skills we developed during our time at CMU and worked collaboratively to solve a complex problem.

3.2.2 Science

We applied the principles of science through the formulation and execution of our project. The scientific method

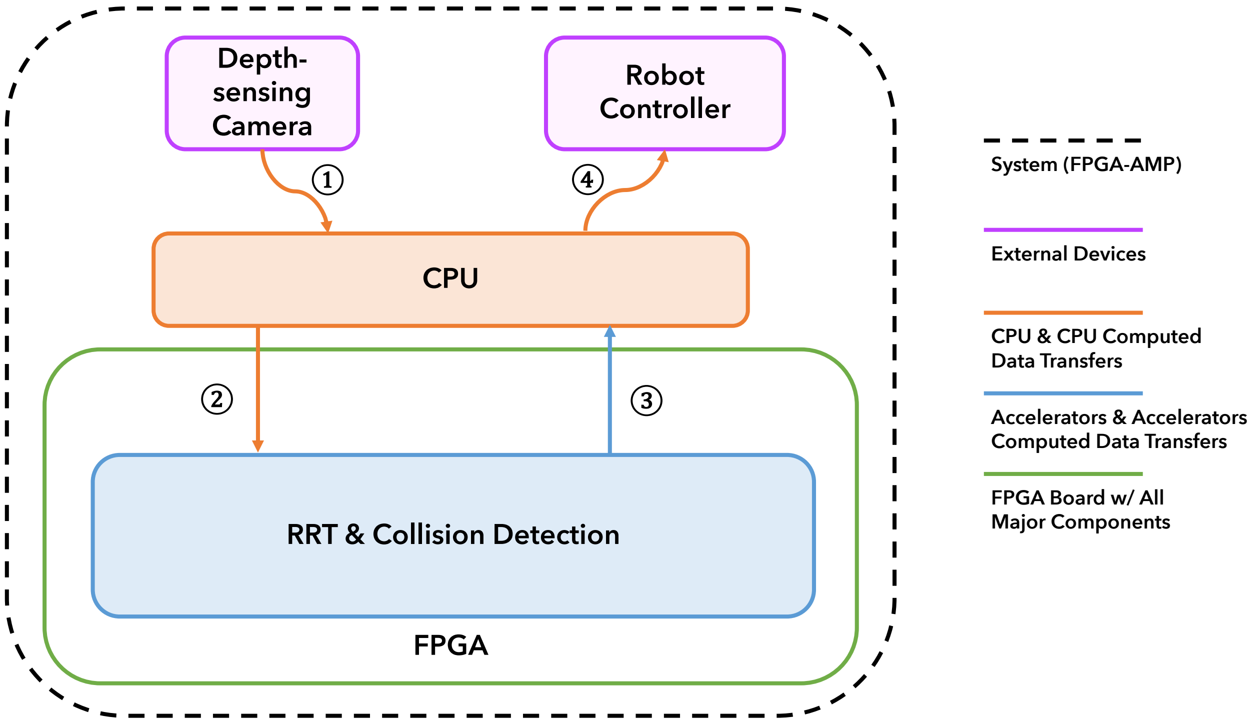
Depth-sensing Camera
Robot Controller
①
④
CPU
②
③
RRT & Collision Detection
FPGA
System (FPGA-AMP)
External Devices
CPU & CPU Computed Data Transfers
Accelerators & Accelerators Computed Data Transfers
FPGA Board w/ All Major Components

Figure 2: An overview of FPGA-AMP system pipeline. ① The depth-sensing camera sends raw point cloud data of the scene to a CPU. ② The CPU processes the point cloud data, turns it into a voxelized 3D scene mapping, and passes it to the FPGA. ③ The FPGA runs RRT, and outputs the scene's state space as well as the tree. ④ The CPU generates and optimizes the motion plan based on the generated tree, and sends the motion plan to the robotic arm controller.

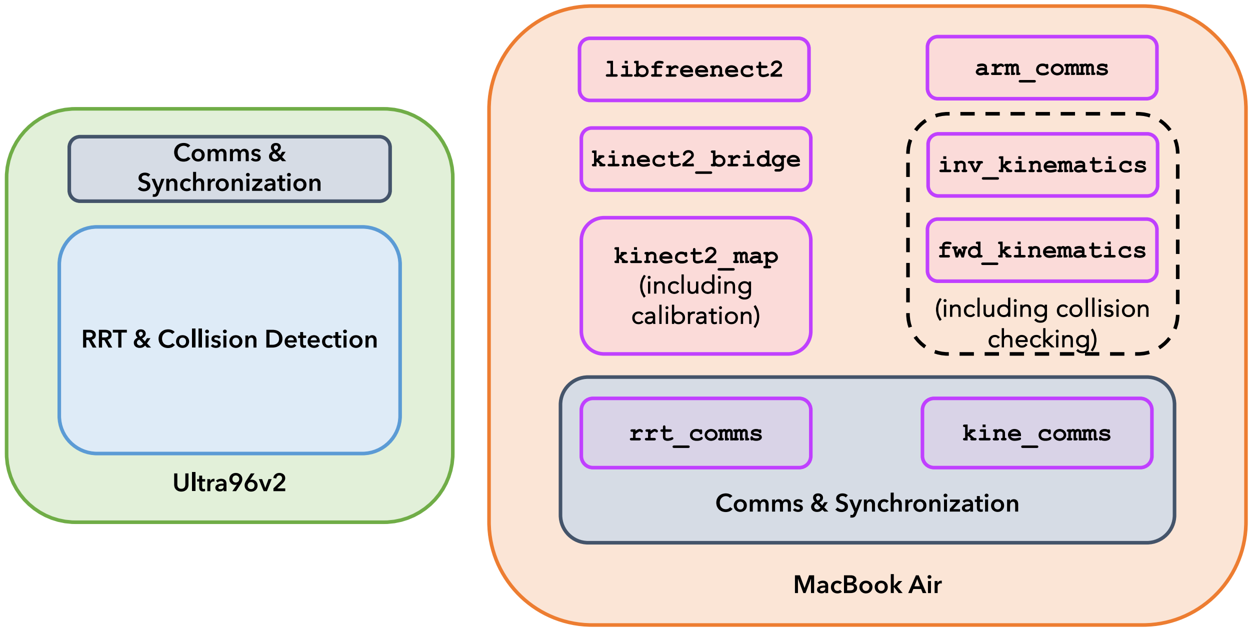

Figure 3: An overview of FPGA-AMP system components.

is defined by doing research, formulating a hypothesis, running an experiment, and finally reaching a conclusion. Our project began by reading multiple research papers on the application of hardware acceleration to robotics. We then hypothesized that implementing motion planning on a FPGA would improve upon the state of the art. From here we implemented our accelerator, ran tests, and compared it to benchmarks. Our initial hypothesis has been proven correct, using FPGAs for motion planning results in significant speedup and reduces the power and energy cost.

3.2.3 Mathematics

Throughout our project we relied heavily on linear algebra and geometry. The perception camera is located behind and above the arm so that it can get a clear view of the state space. The camera is angled down which we must account for by applying rotation matrices to the voxelized data. Rotation matrices along with translation matrices are applied during forward kinematics when converting between the link's reference frames. Geometry is used during inverse kinematics when we repeatedly apply the law of cosines as well as other trigonometric identities. RRT can be considered a Monte Carlo method [10] and A* is a graph search algorithm [11].

4 DESIGN REQUIREMENTS

4.1 Perception System

Having a precise perception of the scene is the cornerstone of accurate motion planning. This is because having an inaccurate mapping of the scene would lead to inefficient or even hazardous decisions in the motion planning process. In addition, since the perception system is the "front end" of the entire motion planning system, its accuracy directly impacts all subsequent systems. Hence, we aimed to make the perception system have a high resolution and a high accuracy.

Given that the Kinect One camera's IR sensor's maximum resolution is 5mm at 50cm and $>$ 5cm at $>$ 5m, we estimated that for our use case (in which the camera will be 1 meter to 2 meters away from the scene), 1cm is the finest resolution we could theoretically achieve. Hence, the perception system should have a resolution of 1cm. **Quantitatively, the perception system should be able to recognize a 1cm $\times$ 1cm $\times$ 1cm standard cube at least 90% of the time.**

Having set the resolution of the perception system, we further derived the requirement for mapping accuracy. **For all mapped objects in the scene, the mapped dimensions should be within +/- 1cm range of its actual physical dimensions at least 90% of the time.**

4.2 RRT Accelerator

By virtue of being a non-deterministic algorithm one of the inputs to RRT is a K value, describing the number of iterations it performs of picking a random point in the space and attempting to grow the tree towards that space. This K can vary depending on the scene, and it has impacts on the effectiveness of RRT. Too low of a K value, and the tree is less likely to grown enough to connect the start and end voxels together. Too high of a K, and computation done by the accelerator is wasteful. Since our accelerator is concerned with speedup, **given a scene with previously stated resolution and an input K value, our accelerator running on the Ultra96v2 should achieve a greater than 10x speedup against the software version of RRT.**

4.3 Kinematics System

Kinematics is responsible for generating commands for the robotic arm, in the form of servo motor angles, that allow it follow the motion plan. If the commands kinematics generates are inaccurate, it no longer matters how good of a motion plan we have created; the arm will deviate from the path and possibly result in a collision. Kinematics is the "back end" of the motion planning system and is responsible for the successful execution of the plan, therefore it is imperative the commands we generate are accurate. The resolution of our states space informs us on how far we can deviate from the path and with a high level of certainty not encounter a collision. **For a path consisting of poses within the arms reach, the commands kinematics generates should result in no more than a 1cm divergence from the path in any direction.**

5 DESIGN TRADE STUDIES

While implementing our system we made countless design decisions. Here are a few key decisions.

5.1 Perception System: Guided Auto Calibration vs. Measurement-based Calibration

One of the biggest challenges in implementing the perception system is designing a camera calibration approach that is consistent and reliable for different environment settings.

We initially implemented a guided auto calibration system. The auto calibration works by telling the user to place the 1cm $\times$ 1cm $\times$ 1cm unit cube at different places in the scene and taking differences in the snapshots. The system then compares these snapshots against a pre-stored snapshot to estimate and adjust the camera's parameters then compose a transformation matrix accordingly. However, this guided auto calibration method, while adaptable, relies heavily on user interaction and can be time-consuming. It also poses precision issues, as the way that the point cloud is transformed into voxels involves some heuristic process, so it is not always deterministic.

To mitigate these issues, we explored an alternative approach: measurement-based calibration. This method involves asking the user to measure (or estimate) the difference between an uncalibrated mapped scene and a real-world scene and using the user input to compose transformation matrices. This method tends to be time-consuming, as it is an iterative process that requires the user to keep adjusting the parameters until the mapping has been correctly calibrated. On the other hand, it did appear to be more precise. Since accuracy and precision are what we care about the most in the perception system, and calibration only needs to be done once for a new environment, we eventually opted to integrate the measurement-based calibration approach into our perception system.

5.2 Robot

We chose to work with a robotic arm because it has multiple degrees of freedom and motion planning is more computationally expensive in higher dimensional state spaces. This means we tested our accelerator in an environment where it is most likely to be useful. This also allows us to provide a more accurate assessment of the accelerator's performance.

5.3 Motion Planning Algorithm

The two motion planning algorithms used most often in robotics are Probabilistic Roadmap (PRM) and Rapidly Exploring Random Tree (RRT) [1]. Both algorithms are non-deterministic and involve taking random samples of the state space. These random samples are then connected via edges to form a graph. Eventually, the graph represents a subset of the viable, collision-free trajectories. Our decision to use RRT was primarily based on the host of high-quality literature on its acceleration [4], [5]. RRT contains intrinsically parallel properties which make it ideal for hardware acceleration.

5.4 Shortest Path Algorithm

The two shortest path algorithms we considered are Dijkstra's Algorithm and A* search. Dijkstra's is ideal when using PRM [12]. However, research has been done to combine RRT and A* into a more efficient algorithm called RRT* [8]. For this reason, we decided on using A*. Doing this left the door open to later integration of RRT and A* into RRT* which would have resulted in even larger performance gains.

5.5 Acceleration Device

5.5.1 CPU

Running motion planning on a CPU is relatively simple as open-source implementations of it are widely available in robotics libraries. As discussed before, there is parallelism to be reaped from RRT but it is hard to take advantage of on CPUs. Collision detection entails many independent, parallel calculations, while conventional CPUs only have around 8-16 threads available. The only benefit of keeping the motion planning computation on a CPU is that the CPU is responsible for orchestrating the rest of the robotics system, there is no overhead associated with transferring data to some external processor.

5.5.2 GPU

Graphics Processing Units (GPUs) are able to exploit the parallelism in algorithms like RRT and achieve significant speedup over CPUs. This being said, they suffer from extreme power inefficiency. Modern consumer GPUs can consume up to 450 Watts. The long-term economics and environmental impact of such a solution do not make sense.

5.5.3 ASIC

Application-specific integrated circuits (ASICs) take hardware acceleration to the extreme, providing even more performance gains and power efficiency than the solution option that we settled on. However, ASICs also have drawbacks, namely the high upfront development costs and long design cycles. In a scenario where hardware acceleration of robotics is operating at scale, fabricating ASICs may make sense, but for the purpose of a semester-long design challenge, we need a platform that is not as expensive and better suited for prototyping.

5.5.4 FPGA

We settled on FPGA acceleration because of the hardware acceleration and efficiency it provides while also being easily reconfigurable, striking a middle ground between CPUs, GPUs, and ASICs. As stated before, hardware acceleration is great for computation that has lots of parallelism, thus making motion planning a great use case for an FPGA. FPGA accelerated motion planning and robotics in general is also an active area of research, and we have obtained much guidance from the literature we have read so far on the subject.

5.6 Hardware Development

5.6.1 HDL

The traditional method for designing an accelerator is to use a hardware description language (HDL) like SystemVerilog. Using such a language allows for much lower and finer control over the accelerator that we build. Having full control is double-edged since the ability to control all the details in our system means we have to worry about all the details in our system. While all three of us in our group have experience with SystemVerilog, for a semester-long project we decided that we needed something that would enable us to develop our accelerator faster.

5.6.2 High-level Synthesis

High-level synthesis (HLS) came to mind as a better alternative to HDLs. For some background, HLS takes a software description of some computation and extracts the dataflow within it to get a hardware description. The translation from software to hardware preserves the correctness of the design. When making a change to

HLS is a better choice for this project for a few reasons. First, it is a more accessible development method at the cost of control over the finer details of the design. For example, it is hard to force the HLS compiler to put a register in between two signals, while it is trivial in an HDL. However, this level of control is not necessary for us to get speedup in our accelerator. The second reason is the fast design cycles we get from using HLS.

5.7 Writing our own kinematics solvers vs. using an open source implementation

The main reason we decided to write our own kinematics solvers was because it gave us significantly more control with regard to calibration and simulation. Having a simulator that maps the dimensions and the angles 1:1 made it easier to reason about and visualize the way our arm was operating. Another reason was that the good open-source kinematics implementations are packaged in ROS and normally work with arms that have more DOF than what we had available. Working with the RRT output directly minimizes the data transfer overhead versus the extra post-processing needed to work with it in ROS. Implementing forward and inverse kinematics was non-trivial but it ended up being a good learning experience.

6 SYSTEM IMPLEMENTATION

6.1 Perception System Implementation

The physical components of the perception system are the Xbox Kinect One (v2) camera and the laptop. The software components include **`libfreenect2`**, **`kinect2_bridge`**, and **`kinect2_map`**.

6.1.1 `libfreenect2`

`libfreenect2` is an open-source driver library for the Kinect for Windows v2 (K4W2) devices. It is used to retrieve the raw sensor data and is required by **`kinect2_bridge`**.

6.1.2 `kinect2_bridge`

`kinect2_bridge` is the ROS 2 version of the **`iai_kinect`**, an open-source software bridge between the Kinect camera and the Robotic Operating System, that reads raw sensor data from Kinect via USB port, converts the raw sensor data to the point cloud, and publishes the point cloud as a point cloud message to ROS 2.

6.1.3 `kinect2_map`

We developed **`kinect2_map`** to process the point cloud message published by **`kinect2_bridge`**. **`kinect2_map`** has multiple layered features:

- Calibration: **`kinect2_map`** can calibrate the camera's coordinate system such that the mapped coordinate system is consistent with the real-world coordinate system in the scene. This is implemented by using **`Eigen`** to generate fully parameterized transformation matrices and applying the transformations to the points in the point cloud.
- Scene Cropping: **`kinect2_map`** can crop the point cloud mapping to the size of the scene, which is fully parameterized.
- Pruning: **`kinect2_map`** can prune the scene state space by marking the unreachable voxels as occupied. Reachability is determined by the position of the base of the robotic arm and the maximum reach of the robotic arm.
- Filling (in blanks): One of the fundamental limitations of using a single IR-ray-based camera to generate a 3D mapping of a scene is that the camera can only observe the surfaces of the objects. This causes all objects to be hollow, or only have their camera-facing surface to be mapped. To solve this issue, we added a feature to **`kinect2_map`** such that it traces the projection of the rays and fills the path along the rays with occupied voxels if the ray hits an obstacle.

6.2 RRT Accelerator Implementation

The RRT accelerator is built on the Ultra96v2, an Arm-based, Xilinx Zynq UltraScale+ FPGA development board. Development of the kernel was done using Vitis and Vitis HLS. The kernel source code is comprised of two parts: the host code written in OpenCL C++, and the HLS description of the accelerator in Vitis HLS C. The host code is responsible for data transfer between the laptop and Ultra96 board, as well as data transfer on the board, between itself (the Arm core) and the kernel on the actual FPGA. The HLS C is where hardware optimizations are applied to our kernel achieve our desired speedup, and there three main optimizations that we used that helped us achieve speedup.

6.2.1 Pipelining

Pipelining takes a long sequential computation and cuts it up into smaller chunks, inserting registers between each chunk. By cutting up the computation and drawing it out over multiple cycles, the execution of multiple loop iterations can be overlapped. Cutting up long computations in hardware also improves the critical path of the circuit, increasing the clock frequency.

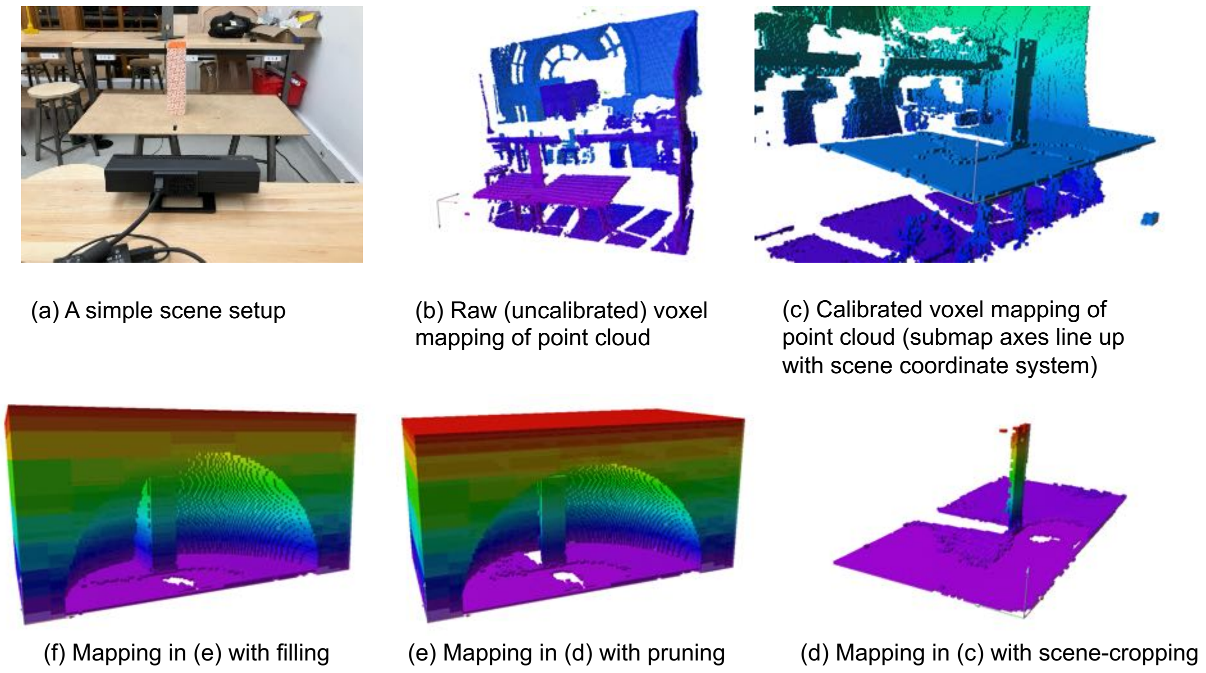
(a) A simple scene setup
(b) Raw (uncalibrated) voxel mapping of point cloud
(c) Calibrated voxel mapping of point cloud (submap axes line up with scene coordinate system)
(f) Mapping in (e) with filling
(e) Mapping in (d) with pruning
(d) Mapping in (c) with scene-cropping

Figure 4: An overview of **`kinect2_map`**'s calibration, mapping, post-processing pipeline.

Throughout the HLS code, there are many sequential computations where pipelining was applied such as generating random numbers via linear feedback shift registers, calculating unit vectors based on the randomly sampled point and the nearest tree voxel, and reads and writes to and from DRAM.

6.2.2 Loop Unrolling

In many algorithms, there are sections of code that are repeatedly executed while being independent of each other. Such loops can be unrolled so that their execution is laid out in parallel on hardware. This allows the computation to be done more quickly while making use of more hardware resources.

In RRT, the most parallelizable portion of our implementation of RRT was the search for the nearest tree voxel given a randomly samples point. In C code, this search looked like a triply-nested loop that searched over the state space for tree nodes and calculated distances, keeping track of the minimum. The most speedup was gained by unrolling some of these loops so that rather than iteratively calculating distances and minimums, the computations for each point are done in parallel. There is an upper limit to how much can be unrolled, as each time a loop is unrolled it turns into more logic and area on the FPGA. We ultimately settled on unrolling across the entire Z dimension, leaving the X and Y dimensions alone (still operating iteratively).

6.2.3 Memory Reshaping

On-chip buffers (BRAM) were used to copy the input data from DRAM into more local, faster-to-access buffers. Vitis HLS allows the user to reshape the dimensions of the buffers, so that instead of receiving a single word on an access of BRAM, multiple words are received. Deciding how to reshape the BRAM is tricky, as a poorly chosen size can slow down the kernel. By reshaping the BRAM, one can ensure that any parallelized portions of the kernel receive enough data to feed their compute units.

Since we partially parallelized the search over the state space, we reshaped our state space BRAM acrosss the entire Z-dimension as well, so that given X and Y coordinates an entire Z row's worth of data is output and immediately fed to the unrolled search computation, rather than requiring multiple cycles to get all the Z data needed to feed the unrolled compute.

6.3 Kinematics System Implementation

The kinematics system takes the RRT tree as input. The system is controlled via a python script and runs the following steps in sequence.

6.3.1 A*

A* is implemented in C and is called via python bindings. A* is used to find the shortest, collision free path through the state space. Further discussion on how A* works can be found in subsubsection 3.2.1.

6.3.2 Inverse Kinematics

Inverse kinematics is the process of mapping the points in the path to servo motor angles. The points represent the location of the arm's end effector and the servo motor angles are used to control the links of the arm. Our robotic arm has 4 key links when it comes to solving inverse kinematics. The base link has length zero and rotates around the Z-axis, aligning the arm with the target point in the XY-plane. The other 3 links have non-zero length and rotate around a translated Y-axis. These links work in conjunction to align the end effector with the target point in the YZ-plane. Solving for the base link is trivial while solving for the remaining links yields a nonlinear system of equations. We solve this via an analytical method, utilizing geometric intuition and algebraic representation [13]. We set one of the link's angles to a known value and solve for the remaining two, iterating until we find a valid solution. In some cases, the arm is not physically capable of reaching a point in the state space and there will be no solutions for the angles. If a point in the path cannot be reached then the path is not valid. Significant work was done in the perception system to pre-process the sensor data and bias RRT towards reachable paths.

6.3.3 Arm Control

Finally the commands are streamed over UART to the robotic arm where we had written code to parse the packets and set the motor angles. Unfortunately the day before the demo two of the servo motors on our arm broke, rendering it effectively inoperable. When the motors broke, we were in the process of calibrating and aligning the arm with the perception system. Before the arm broke, inverse kinematics was successfully placing the end effector at targeted positions.

6.3.4 Forward Kinematics and Simulation Environment

Forward kinematics is the process of mapping servo motor angles, measured between links, to a point that represents the coordinates of the end effector. During this process we also get information about the positions of the links with regards to the global coordinate system. Forward kinematics is implemented via a series of translation and rotation matrices that are used to commute the links between local reference frames. Forward kinematics and its simulation environment were implemented in order to test the validity of inverse kinematics as well as aid our understanding and debugging.

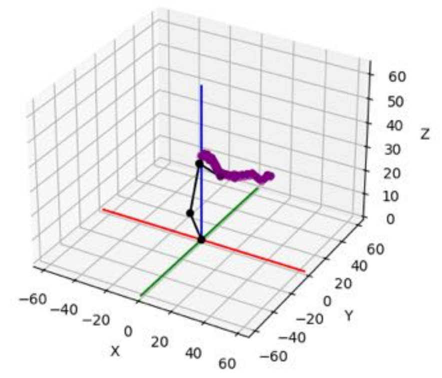

Figure 5: Arm following path in simulation environment.

7 TEST & VALIDATION

7.1 Results for Perception System's Resolution and Mapping Accuracy

We evaluated the perception system's resolution by having the camera calibrated and fixed, and varying the resolution parameter used when converting different point clouds to octal trees of voxels. We used 10 different sets of point clouds, each has the same middle-sized object in it at different positions. We used **`octovis`** to inspect if the mapped position of the object is correct in its resolution (+/- 1 resolution unit). To our surprise, the camera was able to output quality point cloud data one meter away from the scene such that using {10cm, 5cm, 1cm, 0.5cm} resolutions all seemed to work. The only failed resolution was 0.1cm, which produced no matched mapped position.

This was then explained by the other experiment. In the other experiment, we fixed the resolution to 1cm, the scene to the same static scene, but varied the distance between the camera and the scene. We randomly sampled 20 nodes in the mapped scene by clicking in **`octovis`**, and compared their mapped coordinates to their real-world coordinates, and recorded the number of correctly mapped nodes. As the camera moves away from the scene, the number of correctly mapped nodes significantly decreases. While the camera had only one mismatch when it was one meter away from the scene, it had almost half of the sampled nodes were mismatched when it was three meters away from the scene.

Overall, at one meter to two meters, the perception system was able to achieve 1cm resolution and mapping +/- 1cm accurate > 90% of the time. These results align with our predictions and validate the design requirements mentioned in subsection 4.1 and hence also use case requirements mentioned in subsection 2.1.

As a side note, we have also found that the camera's tilt angle and the material of the test object affects the accuracy of the perception system. One possible explanation is that having a too-large or too-small tilt angle, as well as having a translucent object, could make the IR rays tend to scatter more, which complicates the detection of reflected IR rays.

7.2 Results for RRT Accelerator's Accuracy and Performance

We evaluated the accuracy of our RRT implementation through the use of A*. Given a high enough K value such that the start and end voxels connect to the tree, the tree is correct only if A* can find a path through the tree between the start end.

Regarding performance, we timed the execution of RRT including all necessary data transfers. In the case of our software RRT, which already had its input data in memory, this meant simply timing the execution of the algorithm in C. However, in our accelerator included in our timing the sending of input data from the host to the kernel, and the receiving of RRT data from the kernel to the host. This timing was done on the host side—timing began as soon as the host fired off the kernel to run RRT, and it ended as soon as the kernel returned with the data.

We compared both the software and accelerated versions of RRT on a scene with a size of 128x64x64 voxels, with 1cm of resolution, and using $K = 10000$. The software version took 11.53 seconds, and our accelerated kernel took 1.58 seconds, resulting in a **speedup of 18.22x**.

7.3 Results for Kinematics System's Accuracy

Our initial plan was to evaluate the kinematics system's accuracy with regards to the robotic arm but due to the failure of the servo motors we were not able to conduct a significant amount of test. That being said, the forward kinematics and simulation environment were developed while the arm was still functional and designed to model its behavior. Therefore, using the simulator we were able to test the commands generated by inverse kinematics and determine its accuracy. With this method we were able to verify that if a path consists of poses within the arms reach, the commands inverse kinematics generates result in no more than 1cm divergence from the path in any direction as we required.

8 PROJECT MANAGEMENT

8.1 Schedule

The schedule is shown in Figure 6. There are multiple differences from our schedule in our design review. The biggest difference is that the setup of our FPGA was drawn out over the entire semester due to our struggles with the Kria board, resulting in our switch to the Ultra96. This had ripple effects on when porting RRT to the FPGA finished and when integration began, since the FPGA is at the center of our system. Regardless of these troubles, integration and testing nonetheless took longer than expected.

8.2 Team Member Responsibilities

- Baseline RRT: Chris
- HLS-FPGA Environment: Matt, Yufei
- Porting RRT to HLS: Matt, Chris
- Optimization: Matt
- Perception: Yufei
- Kinematics: Chris
- ROS: Yufei
- Full System Integration: Matt, Yufei, Chris

8.3 Bill of Materials and Budget

We have a total budget of \$600.00 and we used \$348.98 for this project.

Although we requested the Ultra96v1 board and Kria KR260 board, we did not end up using them because we only have the Xilinx toolchain set up for Ultra96v2. We planned to run everything as ROS 2 nodes on the KR260 board in our design review. However, since we weren't able to set up the development environment for KR260, we realized that we had to use a ROS 2-compatible laptop to run all the ROS 2 nodes. Hence we added Chris' old MacBook Air to our bill of materials.

For the complete bill of materials, see Table 1.

8.4 Risk Management

8.4.1 Design

One of the primary design risks involved designing the system to be versatile under varying scenes and environments. To make the solution finders as generic as possible, we adopted existing open-source libraries, parameterized as much as possible, and decoupled functionalities when designing the individual system sub-components. For example, the perception system is fully parameterized and can be adopted for scenes with different dimensions. All that's needed to use it on a different-sized scene is to change the dimension sizes in the ROS launch file. Because the individual functionalities are decoupled, doing this won't affect how mapping, or post-processing work.

8.4.2 Schedule

Schedule risks were mitigated by implementing an agile project management approach, which involved breraeaking down the project into small milestones for individual team members, and sprinting altogether to achieve the milestones if they were at risk of not being met by their deadlines. We also had regular scrum meetings scheduled for each week, and flexible ad-hoc meetings whenever necessary. This collaborative approach allowed for rapid adjustments and proved to be helpful to pivot quickly when needed, especially in the final two weeks of the project.

However, we still had some scheduling problems caused by the integration of the Kria KR260 board. For more, see subsection 11.2.

8.4.3 Resources

From a resources perspective, both budget and personnel were considered. We established clear budget constraints early in the project and regularly monitored expenditures against this budget. For personnel, roles were clearly defined, and responsibilities were distributed based on expertise and workload capacity. Regular team meetings and updates ensured that all members were aware of their tasks and deadlines.

However, we encountered some challenges in mitigating resource-related issues during the final week of the project. Specifically, we were unexpectedly confronted with the malfunction of the motors in the robotic arm. For more, see subsection 11.2.

9 ETHICAL ISSUES

The deployment of robotic systems, particularly those capable of autonomous task execution, poses ethical challenges that must be addressed to ensure safe, fair, and responsible use.

9.1 Safety and Reliability

The primary ethical concern in robotic motion planning is safety. Robots operating in human-centric environments must ensure the well-being of humans. An edge case would be a malfunction or misinterpretation by the motion planning system, especially in complex or dynamically changing environments, which could lead to accidents. People working around the moving robots would be affected adversely by this edge case. To mitigate such risks, our system incorporates rigorous testing and validation to ensure reliability and accuracy. Additionally, implementing redundant safety mechanisms and fail-safes can further protect against unexpected failures.

9.2 Privacy Concerns

Robots equipped with perception systems may collect sensitive visual and spatial data, which could raise privacy concerns. An edge case would be cameras that record video could inadvertently capture private moments without consent. To address this, data collection and storage protocols that comply with privacy laws and ethical standards are essential. Placing labels notifying camera presence near the working environment can help minimize privacy risks.

9.3 Job Displacement

The increasing capabilities and deployment of robots can lead to job displacement, particularly in industries where automation is feasible. While this can enhance efficiency and safety, it also raises concerns about the economic impact on individuals whose jobs are affected. Our FPGA-AMP system is not designed to fully replace human workers in robotic motion planning tasks, as it still requires human-guided calibration in various steps during system deployment. Additionally, promoting policies that encourage the retraining and education of workers can help further mitigate job displacement risks.

9.4 Accessibility and Inequality

Advanced robotics technology, such as FPGA-accelerated motion planning, could exacerbate existing inequalities if only accessible to well-funded organizations or countries. An extreme case scenario would be having the solution product be so expensive that only wealthy organizations can afford to use it, which would further widen the socio-economic gap between the affluent and the less affluent segments of society. To mitigate such risks, we intentionally selected low-priced, easily-accessible components that can be easily bought from online markets. We also hope to open-source the software we developed for this project once we have integrated them into a unified repository, which could help reduce the socio-economic gap.

10 RELATED WORK

Robotic motion planning using RRT is a well-researched topic [6]–[9], yet there isn't much work specifically accelerating motion planning using FPGAs. The work by Murray *et al.* [4], [5] is the closest we could find. Our work differentiates by not using pre-computed collision data and targeting a robotic arm, which has a higher degree of freedom in motion.

There is some work [14], [15] that accelerated motion planning using RRT and targeted robotic arms. But our work differentiates by using FPGA acceleration whereas theirs used neural networks.

11 SUMMARY

Overall, the final FPGA-AMP system was able to meet the design requirements.

We are happy to report that we gained 18.2× speed up in RRT generation by using FPGA while using 77.1% less power and 98.7% less energy consumption. Our design goal was $> 10\times$ speed up in RRT generation, 70% less power, and 98% less energy consumption.

The perception and kinematics systems also met the design requirements.

11.1 Future work

We are hoping to open-source the software we developed for this project after we have unified them into one single code repository.

If we were to work on this project further, we intend to reduce the perception system latency and improve the integration for all sub-components in the system.

11.2 Lessons Learned

- Be vigilant, pivot early.
  If something does not work smoothly in the first week, it is highly likely that it will not work smoothly for many weeks too. In that case, evaluating if it is worth trying just to make it work is very critical, as there are likely alternatives that can do the same job well. This happened to us with the Kria KR260 board. We spent many weeks setting up the Linux environment, ROS 2 environment and libraries, and the HLS toolchain for the board. This caused major delays in meeting the project milestones, and it would have been better if we decided to switch to using the Ultra96v2 board earlier.

- Always have at least one backup.
  When our faculty advisor, Prof. Hyong Kim, told us to buy another robotic arm just to make sure, none of us really thought that the arm failing the night before the demo would happen to us. However, this truly happened, and it was awful to not have a readily available backup plan to switch to. In the end, we had to stick with the broken robotic arm, demo'ed it as if it were still functional, and shown the kinematics simulation to prove that we indeed have a working end-to-end system.

Glossary of Acronyms

- FPGA - Field-Programmable Gate Arrays
- ROS - Robotic Operating System
- RRT - Rapidly-exploring Random Trees
- DOF - Degrees of Freedom

References

[1] S. Liu, Z. Wan, B. Yu, and Y. Wang, *Robotic computing on fpgas*. Springer, 2021.

[2] C. R. Garrett, R. Chitnis, R. Holladay, *et al.*, "Integrated task and motion planning," *Annual review of control, robotics, and autonomous systems*, vol. 4, pp. 265–293, 2021.

[3] Z. Wan, A. Lele, B. Yu, *et al.*, "Robotic computing on fpgas: Current progress, research challenges, and opportunities," in *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, IEEE, 2022, pp. 291–295.

[4] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[5] S. Murray, W. Floyd-Jones, G. Konidaris, and D. J. Sorin, "A programmable architecture for robot motion planning acceleration," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, vol. 2160, 2019, pp. 185–188.

[6] S. M. LaValle, J. J. Kuffner, B. Donald, *et al.*, "Rapidly-exploring random trees: Progress and prospects," *Algorithmic and computational robotics: new directions*, vol. 5, pp. 293–308, 2001.

[7] M. Mohanan and A. Salgoankar, "A survey of robotic motion planning in dynamic environments," *Robotics and Autonomous Systems*, vol. 100, pp. 171–185, 2018.

[8] I. Noreen, A. Khan, and Z. Habib, "Optimal path planning using rrt* based approaches: A survey and future directions," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 11, 2016.

[9] M. Elbanhawi and M. Simic, "Sampling-based robot motion planning: A review," *Ieee access*, vol. 2, pp. 56–77, 2014.

[10] Wikipedia, *Rapidly exploring random tree — Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Rapidly%20exploring%20random%20tree&oldid=1211274166`, [Online; accessed 04-May-2024], 2024.

[11] Wikipedia, *A* search algorithm — Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=A*%20search%20algorithm&oldid=1221291584`, [Online; accessed 04-May-2024], 2024.

[12] Wikipedia, *Probabilistic roadmap — Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Probabilistic%20roadmap&oldid=1209859792`, [Online; accessed 01-March-2024], 2024.

[13] R. Tedrake, *Robotic Manipulation, Perception, Planning, and Control*. 2023. [Online]. Available: `http://manipulation.mit.edu`.

[14] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, “OctoMap: An efficient probabilistic 3D mapping framework based on octrees,” *Autonomous Robots*, 2013, Software available at `https://octomap.github.io`. DOI: `10.1007/s10514-012-9321-0`. [Online]. Available: `https://octomap.github.io`.

[15] Q. Gao, Q. Yuan, Y. Sun, and L. Xu, “Path planning algorithm of robot arm based on improved rrt* and bp neural network algorithm,” *Journal of King Saud University-Computer and Information Sciences*, vol. 35, no. 8, p. 101 650, 2023.

Description	Model #	Manufacturer	Quantity	Cost @	Total
Xbox Kinect One Camera	v1	Microsoft	1	$49.99	$49.99
Kinect Adapter for Xbox One	B-SPQxbox0001	KABCON	1	$24.99	$24.99
Ultra96 Development Board	v1	AVNET	1	from Capstone inventory	$0.00
Ultra96 Development Board	v2	AVNET	1	from 18-643 inventory	$0.00
Kria KR260 Robotics Starter Kit	SK-KR260-G	AMD	1	from AMD	$0.00
Arduino Braccio Robotic Arm	RB-Ard-81	RoboShop	1	$274.00	$274.00
MacBook Air	MVH22LL/A	Apple	1	from Chris	$0.00
					$348.98

Table 1: Bill of materials

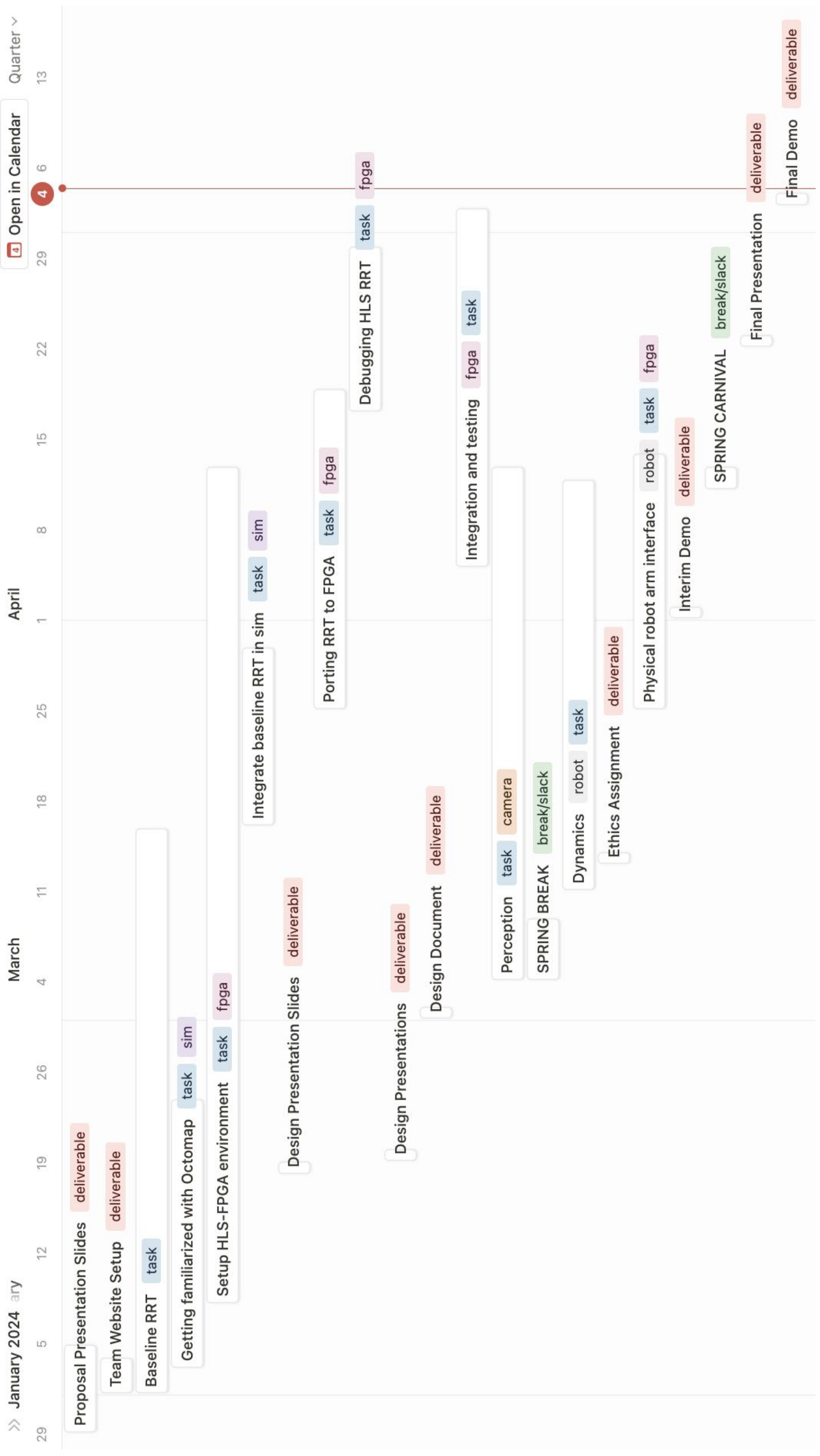

Figure 6: Gantt Chart