FPGA Implementation of Feature Matching in ORB-SLAM2

Hannah Lindström
hannah.lindstrom.37@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu, Lucas Ferreira

Examiner: Erik Larsson

January 27, 2022

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Simultaneous Localization And Mapping (SLAM) is an important component in solving the problem of autonomous navigation — allowing machines such as self-driving cars and mobile robots to find their way in the world without human instruction. Though there is a steadily growing body of literature in the field of SLAM, far fewer works currently address using hardware acceleration to speed up this computationally heavy task.

That is precisely the concern of this thesis project, in which one of the largest bottlenecks in feature based visual SLAM — feature matching — is investigated for hardware acceleration. After comparing several state of the art methods, the Hamming Distance Embedding Binary Search Tree (HBST) is identified as the best candidate for a hardware-based feature matching system; and the specifics of such a system design are presented in detail.

As a means of reducing memory requirements by up to 50%, thus enabling a component of the system to reside in on-chip memory, a new way of storing binary trees was invented: the Heterogeneous Binary Tree Array (HBTA). This method enables binary trees with different sizes of data in their internal and leaf nodes to be stored in an array-based layout with significantly less overhead than a traditional approach; thereby enhancing cache performance, prefetching capabilities, and minimizing storage space.

Popular Science Summary

What if we could release a swarm of butterfly sized robots to explore and map a forest? Imagine sending a tiny helicopter, the size of a fist, to fly along a country road and check it for potholes, all without needing a human to look at it. With this thesis project, we take a step closer towards that vision of the future.

For robots to do this type of work on their own, they need to be able to move around on their own, to navigate and find their way in the world. A way of accomplishing this is known as Simultaneous Localization And Mapping (SLAM), in which a robot is able to both build up a map of its surroundings and use that map to navigate at the same time, using a camera to look at the world. This project looks at one of the most important building blocks of SLAM, called feature matching, and how to make it as efficient as possible.

There are already many systems using SLAM today, from self-driving cars to robot vacuum cleaners. These systems run on normal computer chips, programmed with software which performs all the work required for SLAM. This is all well and good when you have a car battery to run off or only drive around for half an hour before going home to charge like a robot vacuum. But if we want to shrink down to the tiny robots mentioned at the start the battery would drain far too quickly, or have to be so heavy the robot could barely lift it, when running SLAM in software.

Luckily, there are other options! Computers are great because we can program them to do anything we want. All it takes is a bit of code and the exact same computer chip can write emails, play video games, or even send commands to the Mars rover. But as the saying goes, computers are jacks of all trades and masters of none. Specialized hardware is the true king of efficiency; it's made to do one job and one job only, but it does it faster and using less battery than a computer.

So to achieve the efficiency we strive for, this project developed a specialized hardware component which speeds up one of the slowest and most energy-intensive parts of SLAM. An algorithm was found which is both very efficient and suitable for implementation in hardware, and a design based around this algorithm was created. Several tricks and optimizations were used to squeeze the best possible performance from each part of the system, and a brand new invention was created to cut the memory requirements in half for one of the components! In total, the design created in this project is expected to improve several hundreds of thousands of times on the current state of the art's performance, drastically lowering the battery requirements for the tiny robots of the future.

Preface

The thesis project detailed by this report was carried out over the period August – December 2021 from my home and via remote access to the Lund University Department of Electrical and Information Technology (EIT), as part of a M.Sc.Eng in Computer Science and Engineering programme at the Lund University Faculty of Engineering LTH.

I would like to thank my supervisor Liang Liu and my co-supervisor Lucas Ferreira for their assistance and feedback provided over the course of the project. Additionally I want to thank Steffen Malkowsky, postdoc at the EIT department, for contributing additional feedback.

Table of Contents

Acronyms

CAM Content Addressable Memory. v, 3, 4

HBST Hamming Distance Embedding Binary Search Tree. v, 5–8, 10, 11

HBTA Heterogeneous Binary Tree Array. ix, 6, 7, 10

HLS High-Level Synthesis. 11

SLAM Simultaneous Localization And Mapping. 5

List of Figures

List of Tables

Chapter 1
Introduction

In recent years the horizons of automation have been rapidly expanding. In the past we had only industrial machines, carefully tuned by experts to repeat a single, predictable task. Now we not only have machines for which the behavior can be configured by laypeople, but also machines which can modulate their own behavior: autonomous robots.

For such automated systems, the problem of navigating the world is highly relevant. For some applications movement may be unnecessary, or constrained to some area which can be specified by a human operator. However, some systems are required to operate much more freely — for example self-driving cars. For such systems it is prohibitive — or counter to the point, in the case of self driving cars — to rely on human oversight.

Hence, the problem area of autonomous navigation is of growing interest. One method of achieving autonomous navigation — which will be further explained in the Background chapter — is Simultaneous Localization And Mapping (SLAM); a process in which an agent simultaneously creates an approximate map of its surroundings, and determines its position in the map.

This project focuses on a specific visual SLAM system known as ORB-SLAM2[1]. Specifically, the project aims to implement a component of the ORB-SLAM2 system in FPGA hardware. The component in question constitutes a major bottleneck in ORB-SLAM2, hence why it was chosen as a focus.

Through hardware acceleration of key components in the ORB-SLAM2 pipeline, power and area requirements for the resulting system-on-chip are reduced. By achieving these reductions a new class of SLAM enabled systems may become feasible; for example insect sized drones operating with real-time performance.

There is an ongoing effort at the university's Department of Electrical and Information Technology (EIT) to investigate other components of the ORB-SLAM2 system and their potential avenues for hardware acceleration in a system-on-chip. This thesis project serves as a contribution to that effort, and its results are expected to inform future endeavors at the department.

The component of the ORB-SLAM2 system that this project aims to specify is called feature matching. The Background chapter will explain the concept behind feature matching and its importance to SLAM in further detail. To put it briefly, feature matching is a very computationally demanding task, which entails comparing thousands of points from one set to thousands of points from another, and pairing up the ones closest to each other.

To achieve the FPGA implementation High-Level Synthesis (HLS) is used. HLS is a paradigm of hardware design offering a greater level of abstraction than RTL[2]. It allows regular software code, e.g. C++ code, to be synthesized into hardware — provided it adheres to the constraints posed by HLS.

At the onset of this project, three goals were decided on. In order of importance, these were:

- To devise an FPGA implementation of feature matching with a throughput capable of handling at least 2,000 feature points per frame at a rate of 30 frames per second. The motivation for this goal was to feasibly handle the throughput required in an ORB-SLAM2[1] system operating with VGA video quality (640×480 pixels) at 30 frames per second.
- To minimize the power consumption of the implementation; which was anticipated to be largely influenced by the number of off-chip memory accesses. Since the motivation for implementing this hardware accelerator is to reduce power consumption, it naturally follows that the implementation should strive for such reductions, too.
- To specify the implementation through C++ based HLS. This goal was chosen both as a means for myself to learn HLS, and in the hopes of gaining further insight into aspects of HLS development that may be useful in future students' endeavors.

The remainder of this report will firstly go over further background and motivation for the work. Secondly a review of feature matching strategies found in the literature will be presented, and motivation as to why they were rejected for use in this project. Following that, the strategy which was chosen will be introduced and a feature matching system based on that strategy will be elaborated on.

As part of the devised feature matching system, a new method is presented for efficiently storing binary trees storing different sizes of data in their internal and leaf nodes; which optimizes for cache performance and achieves up to a 50% reduction in storage space.

Following the system specification, the results of the project will be presented and discussed. Lastly, a conclusion summarizing the work will be presented, and a number of avenues for future works to explore will be suggested.

Chapter 2
Background

As the Introduction chapter highlighted, the growing need for automation in the modern world necessitates increasingly intelligent systems. Autonomous navigation was identified as a highly relevant capability for such systems.

While there are different types of autonomous navigation — approachable with different techniques — we will focus on SLAM for this project. In SLAM a system simultaneously creates a continuously updated estimate of the world around it, and determines its location within this estimate[3]. SLAM not only enables a robot to navigate a new environment without any prior knowledge of it, but also to create a map for future use. For example, a SLAM enabled drone could be deployed in a warehouse and tasked to explore it until a complete 3D map depicting every wall, aisle, and shelf is obtained.

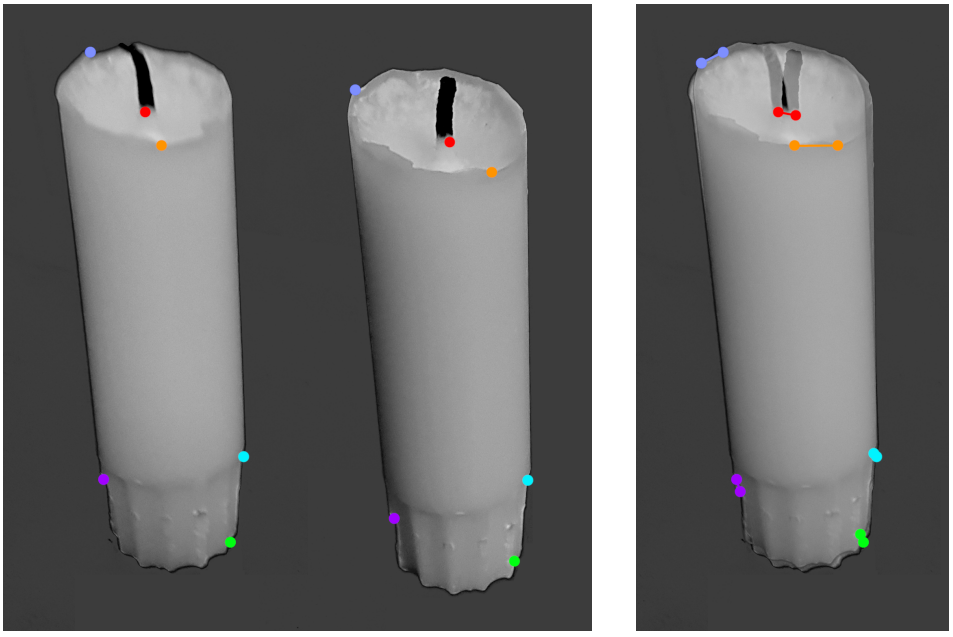
SLAM is a general technique applicable for use with several different types of sensors. For this work we are particularly interested in visual SLAM — a subset of SLAM in which visual input from one or several cameras is used — since cameras are both cheap and able to provide rich information about the recorded environment[1].

An in-depth explanation of visual SLAM is beyond the scope of this paper, but the general concept can be intuitively understood. If you close your right eye and focus your gaze on some object a few meters away, then close your left eye and open the right instead, you will see the object jump a small distance to the left — and the rest of the world for that matter. Of course, we realize that our point of view has simply moved a small distance to the right, rather than the entire world moving the opposite way. Through comparing images from its camera, a system implementing visual SLAM can come to this same conclusion; with math substituting for the role of a visual cortex.

Of course, the camera doesn't see objects, it sees pixels. We may get pixels with the same values in completely opposite corners of an image, or all the pixels may change intensity from one image to another due to changes in lighting. Clearly we can't compare lone pixels from one frame to another and hope for sensible results. A popular and robust method for solving this issue is feature-based visual SLAM[3]; in which feature points are identified in each image.

Feature points are regions of visual interest in an image, parts of it that can be assumed to hold some meaning. An image of a completely white wall isn't interesting. But if there's a dark rectangle to the side of the white wall, you've just found a doorway. Hence, the edge between the doorway and the wall is

interesting, and should have several feature points along it. Beyond just marking a region of interest, feature points all come with their own sort of fingerprint: a feature point descriptor. Comparing these fingerprints from one image to another is what allows a visual SLAM system to recognize movement, and calculate how the camera must have moved within the world to result in such movement. Figure 2.1 illustrates this concept, in which corresponding parts of the same object are marked with the same color of dots, analogous to feature points.



(a) A candle pictured from two different angles with faux feature points marking corresponding parts.

(b) The two pictures overlaid with each other, with lines marking correspondence between the faux feature points.

Figure 2.1: An illustration of the concept behind feature matching.

Just like a regular fingerprint may differ slightly depending on the amount of pressure applied or angle of holding something, feature point descriptors are rarely an exact match; as they too end up taken from different angles or varying lighting conditions. Therefore, when comparing feature point descriptors, we are concerned with finding closely matching points. This process is known as feature matching, and is the focus of this thesis.

Before we can start matching features, we need to define what we mean by a close match, and to do that we need to know what our feature points look like. In this project we target the ORB feature descriptor[4]. The ORB descriptor is used in ORB-SLAM2[1], which is the visual SLAM system this project aims to accelerate the feature matching component of.

An ORB feature descriptor is a binary sequence of 256 bits. The distance

between ORB descriptors is determined by their Hamming distance, which is a measure of how many bit positions the binary sequences differ in. Feature matching on ORB descriptors is then a matter of finding the descriptor with the lowest Hamming distance for each given query, also known as the nearest neighbor problem.

Feature matching is a computationally demanding task, and often becomes a bottleneck in systems dependent on it[5], [6]. In particular, for an ORB-based visual SLAM system such as ours, feature extraction and feature matching are by far the two slowest tasks[3]. Thus, this thesis project aims to mitigate one of these bottlenecks; by devising a hardware based system for feature matching that can leverage the increased processing power of a dedicated circuit.

Chapter 3

Rejected feature matching strategies

This section presents several methods for feature matching that were considered, but ultimately rejected, for use in the system. The core idea behind each method will be briefly explained, followed by the reasoning for choosing not to use it. For further details the reader is referred to the respective original publications.

3.1 Brute force

Brute force is the standard against which all other strategies are judged. As the name suggests, brute force feature matching involves linear iteration over all candidates when finding a match for a query. Though the method requires no overhead beyond placing all the candidate features in contiguous memory, the linear complexity of querying — amounting to quadratic complexity when the candidate set and query set are of the same size — renders this method infeasible when scaled up.

3.2 Locality-Sensitive Hashing

To reduce the number of feature descriptors that need to be retrieved from memory and iterated through, Locality-Sensitive Hashing (LSH) organizes features into a hash table[7], [8]. As the name suggests, LSH leverages a hashing function which tends to map points which are close to each other to the same bucket.

LSH suits itself well to hardware implementation, as previous works have demonstrated[7], [8]. These works both concern themselves with feature matching in an object recognition setting, however.

Whereas the database to be queried is unchanging in object recognition, SLAM requires a constantly changing database; as the frames against which to match change. This introduces an additional complexity which the system must address.

The additional problem to be solved in adapting LSH for SLAM was not reason enough to reject the strategy. Rather, LSH was rejected as the lesser of two good options. Compared to the feature matching strategy which was chosen, LSH has significantly worse accuracy and three orders of magnitude slower processing times[9].

3.3 Hamming Weight Tree

Hamming weight, also known as population count, is a measure on a binary sequence equal to the number of set bits. Hamming weight is closely related to Hamming distance, since Hamming distance between sequences a and b may be calculated as the Hamming weight of $a \oplus b$.

The Hamming Weight Tree (HWT) is a structure introduced by Eghbali, Ashtiani, and Tahvildari [10], which leverages some observations they make on the relationship between Hamming weights and Hamming distances:

- The difference between two sequences' Hamming weights is at most the Hamming distance between them.
- If the sequences are split into several subsequences, the sum of the Hamming distances between corresponding pairs of subsequences is equal to the Hamming distance between the original sequences.

Making use of these observations, a depth one HWT is constructed by creating a node for each possible Hamming weight a binary sequence can have. Since ORB feature descriptors are 256 bits wide, anywhere between zero bits and all 256 bits can be set. Because this is an inclusive range, the HWT ends up with a total of 257 nodes at depth one. The input sequences are assigned accordingly to whichever node corresponds to each sequence's Hamming weight.

To query the structure a radius r must be selected, for which all sequences within Hamming distance r to the query sequence will be returned. This is achieved by first finding the Hamming weight of the query, and then searching only those nodes in the tree corresponding to Hamming weights within r of the query's weight. For example given $r = 2$ and a query with Hamming weight 100, nodes 98, 99, 100, 101, and 102 would be searched.

Eghbali, Ashtiani, and Tahvildari proceed to extend the HWT to arbitrary depths by defining the procedure by which a node is split into descendants. This involves splitting the sequences and creating a new node for each combination of subsequence Hamming weights that add to match the parent. For example node 100 would split into nodes [0,100], [1,99] $\cdots$ [99,1], [100,0]. Similarly at the next depth level node [50,50] would split into [[20,30],[5,45]] among others.

Revisiting our example query of Hamming weight 100 with $r = 2$, let's assume the first half of the query has weight 60 and the second half has weight 40, adding to 100. When the search goes down node 100, the candidates among its children would be [59,41], [60,40], and [61,39]. If these aren't leaf nodes, the query's subsections are split again for a total of four subsections, and the Hamming weights of these are considered when searching the children of the three candidate nodes.

The following were the chief reasons for deciding not to pursue the HWT method further:

- The data structure is a tree with both a high and variable branching factor, which was anticipated to require an excessive amount of off-chip memory accesses to construct and traverse.
- As the original paper discusses[10], HWT is primarily suited to finding neighbors within a given Hamming radius, while nearest neighbor queries — which we are interested in — require multiple searches at iteratively larger radii.

- There is a quadratic dependence on the radius r in the time complexity for search. Since the radius can often be large and vary significantly within a single dataset[10] this risks both a slow and inconsistent implementation.

3.4 Randomized CAM lookup

One idea which was under consideration was to store feature vectors in a Content Addressable Memory (CAM) and repeatedly generate mutations of a query vector in hopes that the mutated query would then correspond to the actual query's nearest neighbor. The probability of generating an appropriate mutation was however found to be far too low for practical use.

Let L be the length of a feature vector, d be the the Hamming distance from a query vector to its nearest neighbor, and n be the number of bits randomly selected for mutation.

There are $\binom{d}{d}\binom{L-d}{n-d}$ ways to select n bits which include all d bits in which the nearest neighbor differs from the query. With a total $\binom{L}{n}$ ways to select n bits, the probability of selecting a superset of the differing bits is:

$$P(superset) = \frac{\binom{d}{d}\binom{L-d}{n-d}}{\binom{L}{n}} = \frac{n!(L-d)!}{L!(n-d)!} \tag{3.1}$$

Let us choose to flip a bit contained in the chosen superset with probability p_f. To achieve the desired mutation only the exact bits in which the nearest neighbor differs from the query must be flipped. This gives us:

$$P(desired \mid superset) = p_f^d(1-p_f)^{n-d} \tag{3.2}$$

Hence the overall probability of generating the desired result, a mutation of the query vector that equals the query's nearest neighbor, is:

$$P(desired) = \frac{n!(L-d)!}{L!(n-d)!}p_f^d(1-p_f)^{n-d} \tag{3.3}$$

Let T be the number of trials required to successfully match a query with its nearest neighbor with probability p_m.

$$p_m = 1-(1-P(desired))^T \iff T = \frac{\ln(1-p_m)}{\ln(1-P(desired))} \tag{3.4}$$

Numerical methods were used to analyze equations 3.3 and 3.4 for different values of d, n, and p_f with the targeted length $L = 256$. Through this analysis the randomized CAM lookup strategy was found to be infeasible. To minimize T the optimal parameters found by the numerical analysis were always $n = d$ and $p_f = 1$. However, $P(desired)$ decreased rapidly in value with increasing nearest neighbor distances d. Even for a nearest neighbor with Hamming distance $d = 6$ from the query $P(desired) \approx 2.7e^{-12}$, which for a target matching probability $p_m = 90\%$ would require $T \approx 8.5e^{11}$ trials. For the same p_m this increases to $T \approx 9.4e^{14}$ for a nearest neighbor at distance $d = 10$.

As the above analysis demonstrates, the randomized CAM lookup strategy would only work for matching features within very low Hamming distances of each other. For a more thorough analysis of the strategy one would have to consider finding a nearest neighbor within Hamming distance d_{max}, rather than one at exactly distance d. Consideration would also need be taken to ensure that the query's nearest neighbor is actually matched, rather than any neighbor within distance d. Both of these issues could be resolved by performing consecutive attempts with $1 \leq d \leq d_{max}$ until a match is found. The mathematical analysis for this method is however uninteresting, as it would obviously require even more trials than the simpler method.

Chapter 4

The chosen approach

This chapter will introduce the feature matching strategy which was deemed the best option out of the methods investigated over the course of this project, based on both performance and the simplicity of implementation in hardware.

The chosen strategy will first be introduced on a conceptual level, after which the system that implements it will be discussed in more detail.

4.1 Hamming Distance Embedding Binary Search Tree

Schlegel and Grisetti [9] introduced the Hamming Distance Embedding Binary Search Tree (HBST) data structure, which was integrated into the feature matching hardware implementation.

The essential concept underpinning HBST is to reduce the linear search required for brute force matching to a logarithmic search by organizing feature descriptors into a binary tree.

Figure 4.1 illustrates some of the specifics of an HBST constructed from the set of input feature descriptors $\mathbf{d}_j$.

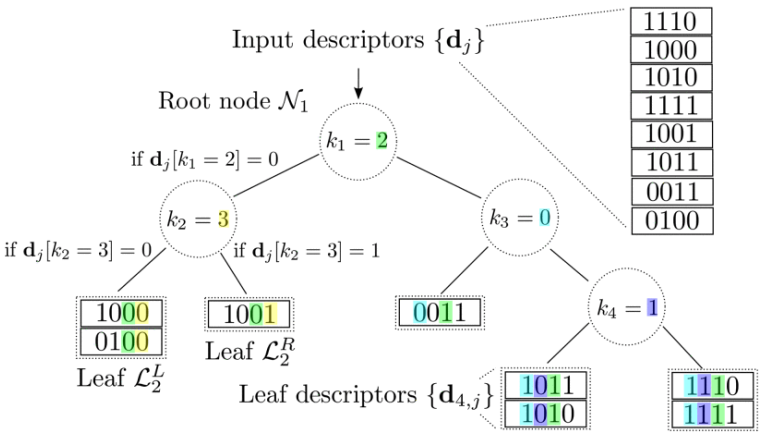


Figure 4.1: An example of an HBST taken from [9].

Each feature is found in exactly one of the tree's leaf nodes, however multiple distinct features may belong to the same leaf node.

Rather than containing features, each internal node of the HBST holds an index $k_i \in [0, dim(\mathbf{d}) - 1]$ specifying a bit position in the feature descriptors. For this work $dim(\mathbf{d}) = 256$, the width of an ORB feature.

In figure 4.1 we see that features are organized according to these indices, with the position indicated by a particular node ensuring that all features to the left of the node have a zero in that position, and vice versa for features to the right.

Algorithm 1 Querying the HBST.

```
1:  procedure NearestNeighbor(query)
2:      node ← root
3:      while node.features = NULL do                    ▷ Not a leaf
4:          bit ← query[node.index]
5:          if bit = 0 then
6:              node ← node.left
7:          else
8:              node ← node.right
9:          end if
10:     end while
11:     return BruteForce(node.features, query)
12: end procedure
```

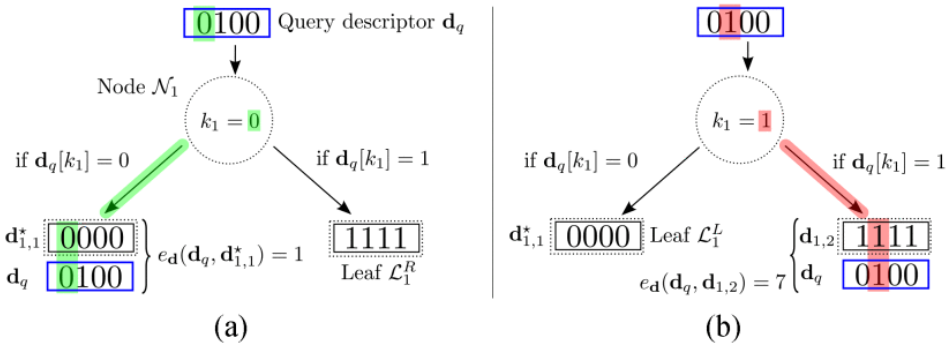


Figure 4.2: An example of query lookup in HBST taken from [9].

Algorithm 1 describes the procedure by which a nearest neighbor is found for a query from among the descriptors contained in an HBST. Intuitively this procedure can be understood as finding the leaf node to which the query would belong to if it were included in the tree. Figure 4.2 illustrates the querying behavior for a single node. The query descriptor **0100** has a nearest neighbor **0000** in one of the tree's leaves, while the other leaf containing **1111** is not the nearest neighbor. Cases (a) and (b) exemplify a shortcoming of HBST: the choice of index k_i can impact whether the nearest neighbor is found as in case (a), or not found as in case (b).

While there exists no known method of choosing indices such that the true nearest neighbor is guaranteed to be findable for any arbitrary query, Schlegel and Grisetti point out that for practical applications it is usually sufficient to find a feature descriptor "*similar enough*" to a query.

Schlegel and Grisetti specify two criteria for choosing a node's index. Firstly, the same index may occur at most once along any path from the root to a leaf. Secondly, indices should be chosen such that the features are split as evenly as possible; with as close to a 50% bit frequency as possible in that position, across the population of features belonging to the given node's subtree. This metric strives to keep the tree balanced.

Though a method is defined for insertion into a preexisting HBST, no such method is given for deletion. This serves as an obstacle for a visual SLAM system, in which the set of feature points for which to perform feature matching may change between frames. To resolve this, the proposed system constructs a new HBST any time the set of features changes.

As we shall see later on, HBST achieves a competitive degree of accuracy on industry standard benchmarks with its approximate nearest neighbor finding. Furthermore, the drawback of having to construct a new HBST each time the set of feature points included changes will be shown to constitute a manageable cost.

4.2 System design

Figure 4.3 shows a simple block diagram of the proposed system design. The solid arrows indicate the flow of data through the system, while the dotted arrows indicate the external I/O interface of the system. Controller blocks have been omitted for the sake of brevity.

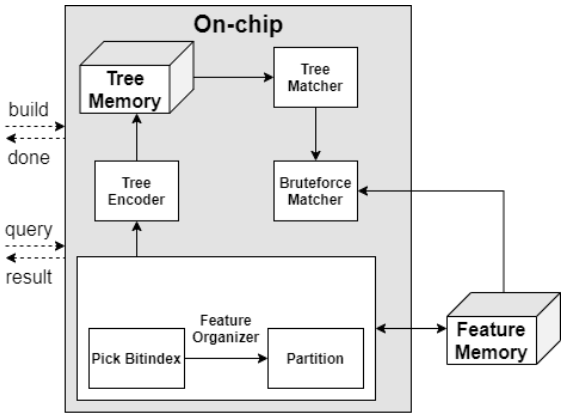


Figure 4.3: A block diagram overview of the system.

As the interface suggests, the system has two separate modes of operation. The first mode is that of constructing the HBST, which is invoked once some external process has populated the feature memory with feature descriptors.

Construction of the HBST is the more complex of the two modes. The Feature Organizer block plays the main part in this procedure; in which it is responsible for accessing and rearranging the descriptors of the Feature Memory, as well as supplying the information to be stored in the Tree Memory to the Tree Encoder block.

The second, simpler mode of operation is querying the HBST once constructed.

The Tree Matcher block's function is to traverse the tree represented in the Tree Memory — according to the procedure introduced in algorithm 1 — until a leaf of the tree is reached. The Bruteforce Matcher block is then delegated the task of finding the feature descriptor nearest to the query from among those belonging to the leaf.

With this brief overview in mind, we will now delve deeper into the specifics of the system design. Firstly discussing the conceptual and practical separation of the feature and tree memories, and in particular a newly devised method of storage utilized by the tree memory. Thereafter the procedures by which the HBST is both constructed and utilized will be specified, and insights on how they may be implemented in hardware will be discussed.

4.2.1 Feature organization

The ORB feature descriptors to be matched against are stored in a contiguous block of memory, to improve cache performance by locality of references. Depending on the design of other components involved in the SLAM pipeline it may be of interest to utilize a ping-pong memory organization for the feature memory; while one memory block is being used for feature matching another can be populated with the next batch's features, after which the roles of the blocks are switched to use the newly populated memory for feature matching while the old features are replaced.

For the tree structure's leaf nodes to reference a set of features from the feature memory, all features belonging to a particular node must be grouped together. To achieve this the features are partitioned into contiguous subsections of the feature memory, the lengths of which are encoded in the tree structure along with the memory location. No additional information is stored in the feature memory, only the feature descriptors themselves. The uniformity of the memory is important for the procedure by which the features are partitioned, which will be described after introducing the organization of the tree structure.

4.2.2 Tree organization

A well known way to represent binary trees is the Eytzinger layout[11], [12]. As illustrated in figure 4.4, the Eytzinger layout stores a binary tree in a one dimensional array. The indices of a node's children are determined by the parent node's index, for a parent node at index i the left child will have index $2i+1$ and the right child index $2i+2$. This indexing scheme starts at zero and goes through every natural number, ensuring that the underlying array may be fully utilized.

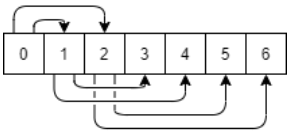


Figure 4.4: Eytzinger layout.

Let us consider the most obvious alternative to the Eytzinger, a tree representation where each node contains a memory reference to its left and right child, in addition to the data of interest. Compared to this "pointer chasing" layout, the Eytzinger layout provides the following benefits:

- Superior locality of references, thus enabling better cache performance.

- No extra memory is required for a node to reference its children, since they are inferred by the parent's index.
- The predictable memory locations of a node's descendants allows for a simple and efficient prefetching scheme to be utilized. All descendants of the same node in the same generation (i.e. depth) are stored contiguously, so they may all be prefetched as part of the same block.

As explained above, the Eytzinger layout has many desirable properties for use in a hardware implementation. However one drawback of using an array-based scheme such as the Eytzinger layout is that all components of an array must occupy the same amount of memory. As we have seen, the HBST contains different information in its internal and leaf nodes; the discriminating bit index and the feature descriptors associated with the leaf, respectively.

It would seem that to utilize an array-based scheme such as the Eytzinger layout, each node would need space to store both the data relevant to internal nodes and leaf nodes, even if only one or the other is used. This redundant storage is obviously a waste of memory, albeit a lesser waste than a "pointer chasing" scheme would incur.

To even further optimize memory efficiency, a new method was devised for encoding asymmetrical data, such as the kind required by HBST, into a compact Eytzinger array representation. This method is the Heterogeneous Binary Tree Array (HBTA).

4.2.3 Heterogeneous Binary Tree Array

As the name suggests HBTA is a method of encoding a binary tree containing asymmetrical, or heterogeneous, information into an array representation; specifically a tree in which the internal nodes contain smaller pieces of data than the leaf nodes.

Figure 4.5 illustrates the core concept behind HBTA. A precondition for the encoding is that only leaf nodes contain the larger type of data, while the internal nodes contain the smaller data — labeled **data** and **index** respectively in the figure.

To differentiate between the nodes a single bit of metadata is introduced for each node, labeling internal nodes with a zero and leaf nodes with a one. Then to ensure the heterogeneity of the structure, leaf nodes are expanded into "leaf subtrees"; a subtree of some implementation-defined depth sufficient to encode the larger data.

Encoding data into a leaf subtree follows a straightforward procedure. First store as many bits of the data as will fit into the root of the leaf subtree, then move onto the next node in turn and do the same for it. In figure 4.5 a breadth first traversal has been used. Breadth first ordering has the benefit of nodes at the same depth being adjacent in memory, which may simplify access patterns for storage and reconstruction. However the concept is equally applicable for depth first traversal — which may entail simpler controller logic than breadth first — or any other traversal order.

Because we assume that the leaf subtree is of a predefined depth only the subtree root's metadata bit is required to encode the structure of the tree. This

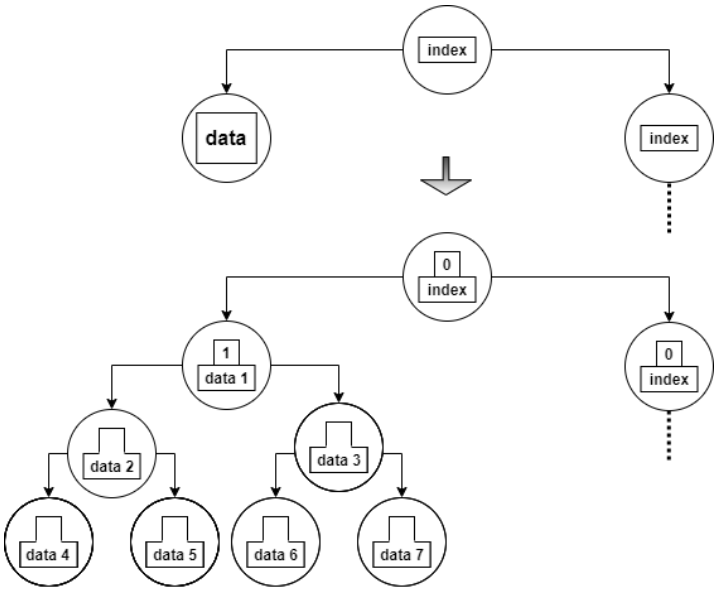


Figure 4.5: The conceptual transformation behind HBTA.

means that all other nodes in the subtree may encode the actual data in their metadata bits, which reduces the total amount of overhead required for the structure; especially as the leaf subtree depth increases.

4.2.4 Construction

The procedure utilized for construction of the HBST in the specified hardware implementation follows a conceptually simple, recursive approach. The details of this procedure will be illustrated in algorithm 2, after which the procedures utilized by the algorithm will be explained.

As previously stated, the feature descriptors constituting the HBST are presumed to exist in a contiguous block of memory as a precondition to the following procedure.

Algorithm 2 HBST construction.

▷ *initial_range*: The range of features to construct the HBST from.
1: **procedure** CONSTRUCT HBST(*initial_range*)
2: *root_index* ← 0
3: PROCESS NODE(*root_index*, *initial_range*)
4: **end procedure**
▷ *node_index*: Node's index in Eytzinger layout.
▷ *range*: Range of features to process.
5: **procedure** PROCESS NODE(*node_index*, *range*)
6: *bit_index* ←PICK BITINDEX(*range*)
7: **if** *bit_index* = *NULL* **then** ▷ No suitable partitions left.

```
        ENCODE LEAF(node_index, range)
        return
    end if
    ENCODE INTERNAL(node_index, bit_index)
    [range_l, range_r] ←PARTITION(range, bit_index)
    node_index_l ← 2 * node_index + 1
    node_index_r ← 2 * node_index + 2
    PROCESS NODE(node_index_l, range_l)
    PROCESS NODE(node_index_r, range_r)
end procedure
```

Pick Bitindex

This procedure is implemented by the Feature Organizer block and its subblock of the same name, shown in figure 4.3.

As illustrated by algorithm 2, construction of the HBST starts by considering the entire collection of feature descriptors that specify the tree. From this range of features a bit index is selected according to the criteria specified by Schlegel and Grisetti [9]; i.e. the index for which the frequency of that bit being set is as close to 50% as possible in the given population of features.

To ensure the HBST scheme remains favorable, a parameter δ_{max} is defined. If the bit selection procedure finds the frequency closest to 50% to be at a distance greater than δ_{max} a selection is not made, and thus the recursion on that branch of the tree ends. Schlegel and Grisetti [9] present good results for $\delta_{max} = 10\%$, but offer no analysis or comparisons to justify this choice.

The specification of HBST requires that no bit index is used more than once along the same path from the root to any given leaf. Because this implementation doesn't concern itself with insertions into the tree after construction, we can easily convince ourselves that this criteria is upheld without any additional hardware required.

Consider that all features partitioned into the left subtree of a given parent node all must have a zero in the bit index used by the parent, and respectively all features in the right subtree have a one in that position. This implies a frequency of 0% and 100% respectively for that bit, neither of which will allow the index to be selected by the procedure a second time (barring a nonsensical choice of δ_{max}).

Implementation of this logic in hardware would seem to require a number of registers equal to the feature descriptor width (i.e. 256), and an accompanying set of incrementers; with an appropriate tradeoff between speed and area made through a resource sharing pipeline. Each feature in the given range would then be iterated over, and each register incremented in accordance with its corresponding bit in the feature. Then the index of the register closest to half the range length is selected, and checked to be within δ_{max}.

Partition

This procedure, too, is implemented by the Feature Organizer block, in its subblock Partition, as illustrated by figure 4.3.

Once a suitable bit index has been selected, the given range of features is partitioned accordingly. We arbitrarily choose to partition the range such that features with the selected bit set to zero come before those with the bit set to one. Hence the predicate used in the following algorithm should return **true** if the input's bit is zero at the selected index.

A well known partitioning algorithm suitable for a hardware implementation is Hoare's[13] partition, exemplified in algorithm 3. Compared to its equally well-known counterpart, Lomuto's partition[14], Hoare's algorithm requires half as many memory swap operations[15]. With memory accesses being a bottleneck in hardware implementations, particularly in the case of the feature memory which is likely too large to fit on-chip, this is an important metric to optimize for.

Algorithm 3 Hoare's Partition.

Require: A function PREDICATE operable on the range's elements.
Ensure: Elements for which PREDICATE is true are partitioned ahead of others.

```
 1: procedure PARTITION(range)
 2:     lo ← 0
 3:     hi ← LENGTH(range)−1
 4:     while lo < hi do
 5:         while PREDICATE(range[lo]) and lo < hi do
 6:             lo ← lo + 1
 7:         end while
 8:         while not PREDICATE(range[lo]) and lo < hi do
 9:             hi ← hi − 1
10:         end while
11:         if lo < hi then
12:             SWAP(range[lo], range[hi])
13:             lo ← lo + 1
14:             hi ← hi − 1
15:         end if
16:     end while
17: end procedure
```

The hardware implementation of the partitioning procedure is straightforward. Testing any given bit of a feature descriptor is trivially accomplished, as is the swapping of elements. In theory the iteration from front-to-back and back-to-front could be performed in parallel, though in practice the bottleneck of off-chip memory accesses may limit the usefulness of this parallelism.

An alternative to Hoare's algorithm which exhibits better performance on general purpose modern computers is Alexandrescu's partition[16]. This algorithm utilizes a sentinel value — a special value inserted among the actual data to simplify some aspect of an algorithm — which eliminates bounds checking (i.e.

$lo < hi$) from the algorithm. However for a hardware implementation bounds checking is cheap, and the additional memory manipulation required to insert and clean up the sentinel value is expensive, so Alexandrescu's algorithm is rejected for the purposes of this system.

Encode Internal and Leaf

These procedures are implemented in the block Tree Encoder of the design shown in 4.3. The Feature Organizer block is responsible for signaling the information to be encoded, as the Feature Organizer is internally aware of the bitindex and partition information which the Tree Encoder must encode.

As the names of these procedures imply, they concern themselves with encoding information into the HBTA representation of the tree. For the sake of brevity it is assumed in algorithm 2 that these procedures may directly access the tree memory.

The details of the encoding can be found in the section Heterogeneous Binary Tree Array on page 15. Naturally, the **index** for this application of HBTA is the bit index associated with each respective internal node. The **data** encoded in the leaf subtrees specifies the feature descriptors associated with the conceptual leaf node.

Through algorithm 2 we achieve a feature memory recursively partitioned into several contiguous subsections conforming to the constraints imposed by the constructed HBST — the leftmost subsection of the feature memory corresponds to the contents of the leftmost leaf node in the HBST, the following subsection corresponds to the second leftmost leaf node, and so on.

The data that must be encoded into a leaf subtree in the HBTA to represent the contents of a leaf node in the HBST, then, is the position and length of the corresponding subsection of the feature memory. Simply concatenating the memory address and length and storing the result suffices.

Much like for a conventional function call in software, the recursive processing found on lines 15 and 16 of algorithm 2 is appropriately realized in the hardware implementation with a stack. The start and end point of the right side of the partition must be preserved on the stack before continuing to process the left side — recursing as needed — following which the right side information may be popped from the stack and utilized.

In building the HBTA's leaf subtree it may be unnecessary to use a stack, since the depth of the leaf subtree is known as a design parameter. A simpler and perhaps more effective approach would be to embed the sequence of index calculations required for the specified traversal order directly into the design; e.g. $i \to 2i+1 \to 2i+2 \to 4i+3 \cdots \to 4i+6$ for leaf subtree depth of 2 with breadth first traversal. Of course, this approach may be counterproductive for particularly deep leaf subtrees, in which case a stack should be used.

4.2.5 Querying

Querying the HBST follows algorithm 1, specified on page 12. The procedure corresponds to the Tree Matcher and Bruteforce Matcher blocks illustrated in figure 4.3, with the former responsible for signaling the latter.

Once the HBST has been constructed to represent a set of input descriptors it may be utilized to, for a given query descriptor, approximate a nearest neighbor from the input set. Starting from the root node, the bit index specified by the node is inspected in the query descriptor. If the query has a zero in that position the left child is visited next, otherwise the right child is visited. This continues until a node with a one in its first bit is reached, indicating the start of a leaf subtree. The leaf subtree is then traversed in the order specified by the implementation, whereupon the data stored in the leaf subtree may be retrieved.

With the data of the subtree obtained — the range's position and length — the range in feature memory belonging to the reached node may be accessed. A linear search is performed on this range, finding the feature in the range with the lowest Hamming distance to the query. This is the resulting approximate nearest neighbor returned by the procedure.

Hamming distance may be computed in hardware simply by **XOR**ing the query and candidate feature descriptors, followed by counting the number of set bits in the result; e.g. with an adder tree. The linear search is then simply a matter of storing the feature with the least distance and said distance in registers until a candidate with a lesser distance is found to replace it. Whatever the feature register holds at the end of the iteration is the result of the procedure.

As touched on previously, all nodes at the same depth level of the Eytzinger layout are adjacent in memory. More specifically, any given node's descendant at the same depth level are adjacent in memory. Per definition a node at index i has its children at $2i + 1 \cdots 2i + 2$, while grandchildren are at $4i + 3 \cdots 4i + 6$. In general, descendants at depth d from an ancestor at index i inhabit the range $2^d(i + 1) - 1 \cdots 2^d i + 2^{d+1} - 2$. Taking this strong locality into consideration, possibilities arise for cache optimizations and prefetching during traversal of the tree.

Chapter 5
Results and discussion

This chapter will firstly present the practical implementation results, followed by a discussion of the performance and efficiency of the HBST as implemented with HBTA. Finally, the stated goals of the thesis project will be addressed and a discussion on the degree to which they have been achieved will be presented.

5.1 Implementation results

One of the stated goals of this thesis project at the onset was for me to learn HLS and use it to implement the feature matching system. While I was able to absorb a great deal of knowledge reading about HLS, due to shifting priorities and unforeseen developments over the course of the thesis there was not enough time to fully implement the system. The Tree Encoder block, shown in figure 4.3, was chosen for implementation; as this block was deemed sufficiently independent of the other blocks and of such a size that the implementation could be finished within the remainder of the project's timeframe.

Xilinx's Vitis HLS tool was used to implement the block in C++. For the sake of example an HBTA leaf subtree depth of two was specified for the implementation, allowing for 62 bits of partition data to be encoded. A depth first leaf subtree encoding was used for the sake of simplicity.

The implementation target was a a Xilinx FPGA (XCZU7EV-2FFVC1156E), for which the implementation results after place and route can be seen in table 5.1.

Table 5.1: Results of HLS implementation.

	$\mathbf{f}_{max}$	Latency	LUT	FF	CLB
Case A	391.5 MHz	4 cycles	200	105	35
Case B	383.6 MHz	3 cycles	230	88	41

The two different cases presented in table 5.1 differ in the array index calculations performed during encoding of the leaf subtree. In Case A the same pair of calculations — $2i + 1$ and $2i + 2$ — were performed multiple times for different values of i in an attempt to minimize area by resource sharing. This created a

data dependency in the subtree root's grandchildren, which were using the indices calculated for the children as inputs to the aforementioned calculations.

Case B, then, eliminated this dependency by calculating all descendants' indices solely based on the subtree root's index. This reduced the latency of the block, and interestingly had a smaller impact to the FPGA resource usage than anticipated. The 2% reduction in maximum frequency can be safely ignored, as the reduced latency — which is directly related to the initiation interval since the implementation isn't pipelined — allows for a higher overall throughput. Furthermore it is highly unlikely that this component would be the determining factor for clock frequency in an implementation of the system as a whole.

5.2 Use of Heterogeneous Binary Tree Array in Hamming Distance Embedding Binary Search Trees

Firstly, as alluded to in the introduction of the concept, let us briefly examine the efficacy of using the HBST. For a more in-depth analysis and discussion of the method, the reader is referred to the original publication[9].

In the original work, a few alternative state of the art methods are compared against HBST, alongside brute force. While the authors did find one of the competing methods — DBoW2[17] with direct indexing — to outperform HBST in terms of accuracy, the processing speed was found to be two orders of magnitude faster for HBST. Compared to brute force, which of course is as accurate as can be, HBST was five orders of magnitude faster.

As was touched on in section 4.2.2, the most obvious alternative to the Eytzinger layout that HBTA is based on is a pointer based tree representation. Examining the open source library[18] published alongside HBST[9], we can see that this is indeed the representation implemented and measured by Schlegel and Grisetti. Knowing the importance of cache locality, we can speculate that a software implementation using HBTA would perform even better.

Also alluded to in section 4.2.2 is the space requirement reduction obtained by using HBTA as compared to the redundant storage of data pertaining both to internal nodes and leaf nodes to achieve homogeneity. Let us now quantify these savings.

Consider the data to be stored in the internal nodes to be of size d bits, and the leaf data to be D bits. For a tree with N nodes the naive approach's memory requirement is as follows; keeping in mind each node must have space for both the internal and leaf data to function with the Eytzinger layout.

$$Size_{Naive} = N(D + d) \tag{5.1}$$

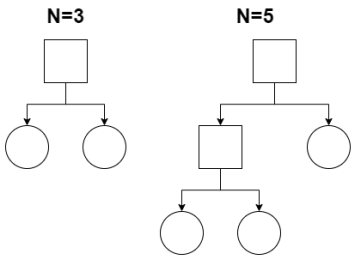


Figure 5.1: Full binary trees.

An HBST is always a full binary tree; i.e. one in which all nodes either have two or zero children. This is readily apparent, as internal nodes in HBST always branch to a *zero side* and a *one side*. A full binary tree

containing N nodes always contains $\frac{N-1}{2}$ internal nodes, and $\frac{N+1}{2}$ leaf nodes. We can convince ourselves of this fact by examining figure 5.1. The only permissible way to expand a full binary tree is to add two children to a leaf node, which turns that leaf node into an internal node. Hence the net gain is two new nodes, one new internal node, and one leaf node; which is in line with the stated formulae.

With this in mind, we can now calculate the memory requirement for HBTA. Though in practice there are more than N nodes in an equivalent HBTA representation, we can conceptually consider the leaf subtrees as a single leaf node; with one bit of overhead per leaf subtree, since the metadata bit can store useful data in all nodes but the subtree root. The internal nodes also require one bit of overhead each, for a total of N bits of overhead.

$$\begin{aligned} Size_{HBTA} &= \frac{N-1}{2}(d+1) + \frac{N+1}{2}(D+1) \\ &= \frac{N-1}{2}d + \frac{N+1}{2}D + N \end{aligned} \tag{5.2}$$

Putting together equations 5.1 and 5.2, we get the following ratio.

$$\begin{aligned} \frac{Size_{HBTA}}{Size_{Naive}} &= \frac{\frac{N-1}{2}d + \frac{N+1}{2}D + N}{N(D+d)} \\ &= \frac{1-\frac{1}{N}}{2(D+d)}d + \frac{1+\frac{1}{N}}{2(D+d)}D + \frac{1}{D+d} \\ &= \frac{1}{2} + \frac{1}{D+d} + \frac{D-d}{2N(D+d)} \end{aligned} \tag{5.3}$$

Table 5.2: Examples of HBTA memory savings relative to the naive approach.

N	D	d	Ratio
∞	∞	∞	50%
32,768	62	8	51.43%
32,768	26	8	52.94%
512	26	8	52.99%

Shown in table 5.2 are some illustrative examples of equation 5.3. Firstly the asymptotic case, showing that HBTA approaches 50% memory usage for large enough data. As the subsequent examples show however, infinity is not necessary to get reasonably close to the 50% mark. The values of d and D are therein chosen as applicable to the project; 8 bits required for the indexing, while 26 and 62 correspond to fully utilizing leaf subtrees of depth one and two, respectively. The choices of N are somewhat arbitrary, but do well to illustrate the comparable savings for both small and large trees.

Presupposed by equation 5.3 and the values used in table 5.2 is that the data of D bits precisely fits in a leaf subtree of some depth, or more formally there

exists a k such that $D = (2^k - 1)(d + 1) - 1$. Without this presupposition the analysis becomes more complex, since the nature of the Eytzinger layout requires each leaf subtree to be full and complete, whether there is enough useful data to fill the whole tree or not.

For the purposes of the feature matching system, this might mean that the true fraction of memory used by HBTA as compared to the naive approach is greater than the 51-53% shown in table 5.2. This depends on the implementation of addressing into the feature memory, and the maximum size of the feature memory used in a full implementation of the system.

Consider as an example of extremely poor luck, if we required 27 bits to specify the partition data. The leaf subtrees would have to be at a capacity of 62 bits, and through applying equations 5.1 and 5.2 once more with $D_1 = 27$ and $D_2 = 62$ respectively, the memory ratio would be 102.86% for $N = 32,768$, requiring more space than the naive approach. While this is obviously disastrous, we can conjecture from observing the relatively little excess of 2.86% in a worst case scenario, that for *most* cases — ones with less of an extreme discrepancy between D_1 and D_2 — the method should still require less space.

Because the tree memory and feature memory are logically separate, they may be physically separate as well. While the feature memory is likely to require a prohibitive amount of memory for storing on-chip, it's plausible for an implementation to require a small enough tree memory that it can be integrated on-chip — a plausibility made more likely by the size reduction introduced by HBTA.

5.3 Fulfillment of project goals

As introduced in the Introduction chapter, there were three goals set at the start of this project. Restated briefly, these were:

- Devising an FPGA implementation of feature matching capable of processing 2,000 feature points per frame at 30 frames per second.
- To minimize the power consumption of the system; chiefly by way of minimizing off-chip memory accesses.
- To implement the system with C++ based HLS, and in so doing garnering insights useful to future students' endeavors.

Since the part of the system which was implemented does not serve as a throughput bottleneck, nor requires off-chip memory accesses, there is no quantifiable result to meaningfully address either of the first two goals. From a qualitative perspective, however, I consider these goals achieved to a satisfactory degree.

In basing the system on the HBST data structure, the time complexity of feature matching became logarithmic; rather than linear as is the case for brute force matching. An FPGA implementation, eSLAM[3], which utilizes brute force matching, achieves 31.45 frames per second throughput.

Although eSLAM limits itself to 1,024 feature per frame, the reduction from linear to logarithmic complexity should more than make up for the double number of feature points. Another caveat is that eSLAM implements ORB-SLAM, not ORB-SLAM2. However, since ORB-SLAM2 does not make changes to the

properties of feature matching found in its predecessor[1], this does not affect my argument.

As was argued previously, the introduction of the tree memory separate from feature memory, and the reduced storage requirements obtained by the introduction of HBTA in the tree memory, may allow the tree memory to be stored on-chip. This, naturally, reduces the number of off-chip memory accesses, and thereby helps to minimize power consumption.

Additionally, traversing an HBST is of logarithmic complexity; compared to the linear complexity of brute force. Thus, even if an implementation requires a maximum tree depth larger than can be accommodated with on-chip memory, an overall reduction in off-chip memory accesses is still achieved.

The process of constructing the HBST introduces additional memory accesses in terms of picking a bitindex and partitioning the feature memory for each branch. However, the complexity of these operations is $\mathcal{O}(n \log n)$, meaning that the system taken as a whole still outperforms brute force in terms of memory accesses — assuming, of course, a nontrivial number of queries are made after construction.

With regards to the third goal, although only part of the system was implemented in practice, I was able to absorb a great deal of HLS knowledge from literature, which helped to inform the work process and design decisions I undertook over the course of the project. The greatest insight I had in working with HLS was the relative ease with which a testbench could be implemented, and the great sense of security in the correctness of my implementation it wrought.

Importantly, the presence of the testbench made it easy to make changes to the design in an exploratory fashion; allowing me to test alternative ways of coding the same behavior to find improvements in the synthesized result. While I may have arrived at the more optimal implementation from the beginning had I been working at a lower level, an HLS workflow allowed me to both implement the original version and optimize it at a faster rate than I would have working at a lower level.

Chapter 6
Conclusion

This work has introduced the domain of autonomous navigation as an important field of study to enable the increasing demands of automation and intelligent systems in the modern world. SLAM was introduced as a state of the art method for solving one type of autonomous navigation problem; navigating in an unknown environment. With applications in a wide variety of systems, such as self-driving cars and autonomous drones, SLAM was shown to be an important research area.

Feature matching was introduced as one of the main performance bottlenecks in SLAM, to which end this thesis project aimed to resolve the bottleneck through hardware acceleration in an FPGA. To achieve a scalable solution, several strategies for feature matching were investigated to achieve better performance than a linear brute force strategy.

The HBST was found to be the most suitable method. It demonstrated the fastest processing times among its competitors — five orders of magnitude faster than brute force — and an accuracy only outperformed by one of its competitors. Furthermore, HBST had good properties for translating into hardware.

One of the main features of HBST leveraged in this work was the binary tree structure. Using a newly devised method of storing binary trees, HBTA, the binary tree structure could be efficiently stored in an array-based layout. This enabled superior caching and prefetching as compared to a pointer-based layout, and through the use of HBTA the array's memory requirements are up to 50% less than for a naive implementation.

The end result of the project was a detailed specification of a feature matching system leveraging the HBST structure, and the introduction of HBTA to reduce memory requirements and enable part of the structure to reside in on-chip memory. Additionally, one of the functional blocks of the system was implemented in C++ based HLS; leading to a discussion of insights into the HLS workflow.

While the project encountered some unexpected challenges and changes of course, the central goals that were decided on at the onset were still largely achieved. Furthermore, the invention of HBTA and the various other optimizations introduced throughout the system design specification serve as valuable additions to the results of the project.

Chapter 7
Future work

In this chapter three suggestions for future work are presented, which can elaborate on what has been presented in this project.

Since the feature matching system specified in this work was only partially implemented, a natural continuation for a future work would be to implement the remaining blocks of the design. Of particular interest would be to explore how the finished system could be tested; and ideally benchmarked with the KITTI dataset, to compare against the state of the art.

The Pick Bitindex design block introduced on page 17 requires a number of register and adders equal to the feature width: 256. A more efficient implementation could not be devised, nor found in literature, in this project. However, improving on the high resource cost of this seemingly simple operation seems like a worthwhile future endeavor.

Lee et al. [7] introduce a Huffman encoding of the feature descriptor database in their LSH implementation. As this is an object recognition system, rather than a SLAM system, the method relies on analyzing symbol frequencies in a static database ahead of time. To extend this idea to SLAM, it would be interesting for a future work to investigate whether certain symbols within feature descriptors are more frequent than others in general. Such findings could then be used to devise a compression scheme, thereby minimizing off-chip memory requirements.

References

[1] Raúl Mur-Artal and Juan D. Tardós. "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras". In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262. DOI: 10.1109/TRO.2017.2705103.

[2] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010. ISBN: 1450097243.

[3] Runze Liu, Jianlei Yang, Yiran Chen, et al. "eSLAM: An Energy-Efficient Accelerator for Real-Time ORB-SLAM on FPGA Platform". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.

[4] Ethan Rublee, Vincent Rabaud, Kurt Konolige, et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[5] Leibo Liu, Wenping Zhu, Shouyi Yin, et al. "A Binary-Feature-Based Object Recognition Accelerator With 22 M-Vector/s Throughput and 0.68 G-Vector/J Energy-Efficiency for Full-HD Resolution". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.7 (2019), pp. 1265–1277. DOI: 10.1109/TCAD.2018.2846634.

[6] Marius Muja and David G. Lowe. "Scalable Nearest Neighbor Algorithms for High Dimensional Data". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11 (2014), pp. 2227–2240. DOI: 10.1109/TPAMI.2014.2321376.

[7] Seungjin Lee, Joonsoo Kwon, Jinwook Oh, et al. "A 92m W 76.8GOPS vector matching processor with parallel Huffman decoder and query re-ordering buffer for real-time object recognition". In: *2010 IEEE Asian Solid-State Circuits Conference*. 2010, pp. 1–4. DOI: 10.1109/ASSCC.2010.5716616.

[8] Gyeonghoon Kim, Jinwook Oh, and Hoi-Jun Yoo. "A 86mW 98GOPS ANN-searching processor for Full-HD 30fps video object recognition with zeroless locality-sensitive hashing". In: *2012 Proceedings of the ESSCIRC (ESSCIRC)*. 2012, pp. 450–453. DOI: 10.1109/ESSCIRC.2012.6341352.

[9] Dominik Schlegel and Giorgio Grisetti. "HBST: A Hamming Distance Embedding Binary Search Tree for Feature-Based Visual Place Recognition". In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 3741–3748. DOI: 10.1109/LRA.2018.2856542.

[10] Sepehr Eghbali, Hassan Ashtiani, and Ladan Tahvildari. "Online Nearest Neighbor Search Using Hamming Weight Trees". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.7 (2020), pp. 1729–1740. DOI: 10.1109/TPAMI.2019.2902391.

[11] Michael Eytzinger. *Thesavrvs principvm hac ætate in Evropa viventivm: qvo progenitores eorvm, tàm paterni quàm materni, simul ac fratres & sorores, indè ab origine reconduntur, vsque ad annum à Christo nato 1590 ... per Michaelem Eyzinger Avstriacvm.* 1590. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true%5C&db=cat07147a%5C&AN=lub.1319803%5C&site=eds-live%5C&scope=site.

[12] Paul-Virak Khuong and Pat Morin. "Array Layouts for Comparison-Based Searching". In: *ACM J. Exp. Algorithmics* 22 (May 2017). ISSN: 1084-6654. DOI: 10.1145/3053370. URL: https://doi.org/10.1145/3053370.

[13] C. A. R. Hoare. "Algorithm 63: Partition". In: *Commun. ACM* 4.7 (July 1961), p. 321. ISSN: 0001-0782. DOI: 10.1145/366622.366642. URL: https://doi-org.ludwig.lub.lu.se/10.1145/366622.366642.

[14] Jon Louis Bentley. *Programming pearls.* Addison-Wesley, 1986. ISBN: 0201103311 ; URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat07147a&AN=lub.425821&site=eds-live&scope=site.

[15] D Abhyankar and M Ingle. "A Performance Study of Some Sophisticated Partitioning Algorithms". In: *IJACSA Editorial* (2011).

[16] Nicholas Vadivelu. *Comparing Array Partition Algorithms.* 2021. URL: https://nicholasvadivelu.com/2021/01/11/array-partition/ (visited on 12/17/2021).

[17] Dorian Galvez-López and Juan D. Tardos. "Bags of Binary Words for Fast Place Recognition in Image Sequences". In: *IEEE Transactions on Robotics* 28.5 (2012), pp. 1188–1197. DOI: 10.1109/TRO.2012.2197158.

[18] Dominik Schlegel and Giorgio Grisetti. *HBST: Hamming Binary Search Tree (Gitlab repository)*. 2018. URL: https://gitlab.com/srrg-software/srrg_hbst (visited on 12/27/2021).