University of California
Santa Barbara

# The Advantage of Custom Microprocessors for Stochastic Gradient Descent in Graph-Based Robot Localization and Mapping

A Thesis submitted in partial satisfaction
of the requirements for the degree

Master of Science
in
Computer Science

by

Sung-Yee Guo

January 2018

The Thesis of Sung-Yee Guo is approved.

Professor Rich Wolski

Professor Amr El Abbadi

Professor Timothy Sherwood, Committee Chair

November 2017

The Advantage of Custom Microprocessors for Stochastic Gradient Descent in Graph-Based Robot Localization and Mapping

## Abstract

The Advantage of Custom Microprocessors for Stochastic Gradient Descent in Graph-Based Robot Localization and Mapping

by

Sung-Yee Guo

Simultaneous Localization and Mapping (SLAM) describes a class of problems facing a large and growing field of autonomous systems – from self-driving cars, to interplanetary rovers, to home automation products. Unfortunately this is a complex task where sophisticated algorithms and data structures are required to navigate a wide range of uncharted environments. Furthermore, most mobile robots need to run these tasks near real-time onboard an embedded controller with limited power and compute resources. To address this problem we explore the stochastic gradient descent (SGD) variant of graph solvers for SLAM and observe a tradeoff between various execution architectures and overall execution speed. Based on these observations, we propose a custom multiprocessor design that relaxes memory-coherency constraints between parallel cores while avoiding divergent behavior. We introduce a specialized streaming-tree interconnect that provides increased performance while using fewer resources compared to state-of-art GPU/CPU implementations of SGD. Finally, we discuss applications of unconventional architectural paradigms like over-provisioned dark processors and specialized data partitioning that provided a unique performance advantage for our particular design.

# Contents

# Chapter 1

# Introduction

Localization and mapping are crucial skills for any mobile robot's ability to navigate previously-unseen environments. Like human surveyors, a robot moves around identifying and measuring landmarks to construct a map of each landmark's location. Then the robot uses this map to deduce its location in the environment. While this task seems intuitive to humans, robots have a very hard time doing the same. They need large volumes of data from a vast array of sensors to feed a computationally-expensive simultaneous localization and mapping (SLAM) algorithm before producing a useful map. Furthermore, as robots evolve from application-specific problem domains like automotive factories to unstructured free-form settings such as homes and public roads, its SLAM algorithms become more complex to process the uncertain and noisy environments. In this work we survey the computational complexity of a popular SLAM algorithm and investigate issues that arise when implementing it on an embedded mobile robot. Then we analyze a potential avenue for optimization where custom microprocessor designs may be advantageous compared to traditional CPU/GPU processor paradigms. We conclude our findings with a novel architectural paradigm that handles asymmetric loads across parallel processors.

## 1.1 Pose Graph SLAM

We investigate a particular family of SLAM algorithms called Pose Graph SLAM. Pose graph SLAM models the robot's environment as a directed graph of pose nodes and edge constraints. Each pose node is a point in 2D space where a robot has been in the past. Edge constraints between each node encode some measurement between two robot pose nodes. In the context of pose graph SLAM, "edges" are synonymous with "constraints" while "nodes" are synonymous with "poses." As a robot moves through uncharted territory, it drops a "breadcrumb trail" of poses at fixed distance intervals. Between consecutive poses, the robot inserts a constraint that encodes the change in position from one pose to the other based on odometry data. If the robot encounters an area it has visited in the past, i.e. it encountered a previously dropped breadcrumb, it will insert a special constraint between its current pose and the encountered pose. This process is called a loop closure because the robot has returned to a place it has been before and thus completed a loop (see Figure 1.1). The robot then continues on, repeating the pattern of dropping poses and connecting them with constraints through consecutive odometry measurements or loop closures until it has a satisfactory map of the new environment.

In a perfect world, the resulting graph will line up perfectly with the environment. For example, if the robot explored a home with many rooms its resulting graph map will correspond exactly to the building's floorplan. But noise in sensing and actuation processes introduce error into the pose graph, resulting in maps that do not line up. Thus the process of "fixing" the graph map is called graph optimization (Figure 1.2). The "optimization" part comes from the fact that pose graph SLAM is formulated as a nonlinear least-squares (NLS) problem where graph error is the objective function to be minimized.

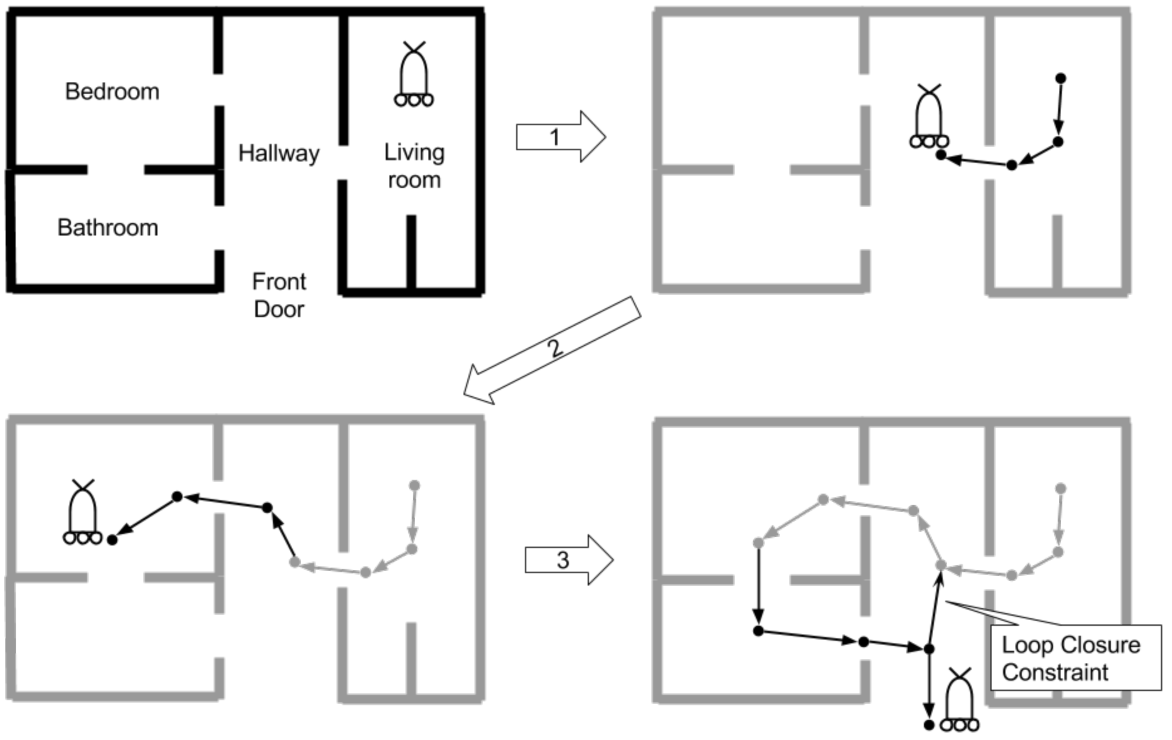

Figure 1.1: Consider a robot exploring a house. It starts in the living room and moves around adding poses and constraints to its graph. It explores the hallway, bedroom, bathroom, and eventually escapes out the front door. After step 3, the robot notices that it has already visited the hallway and adds a loop-closing constraint between the hallway nodes.

## 1.2 Stochastic Gradient Descent

There are many existing ways to solve NLS, one of which is the stochastic gradient descent (SGD) algorithm. SGD can handle graphs with very large initial error, a property other NLS solvers struggle with. We chose a technique proposed by Edwin Olson in his paper "Fast Iterative Optimization of Pose Graphs with Poor Initial Estimates."[1] Olson describes a method of adjusting each pose individually in an iterative manner so as to incrementally improve the graph with each pass. The constraints can be thought as springs that push and pull nodes around until the graph is fully aligned (Figure 1.3).

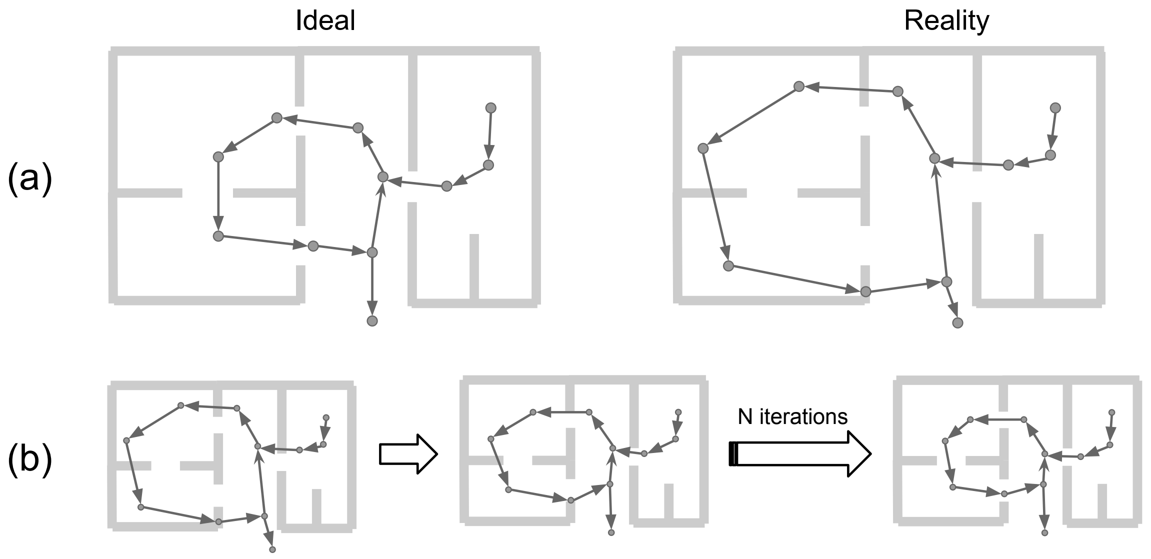

Figure 1.2: (a) An ideal robot would produce a graph that precisely reflects its exploration path. Instead, noisy odometry measurements by the robot's wheels introduce major errors in the graph, resulting in a map where the robot teleports through walls. (b) The loop closure in the hallway (i.e. robot cameras identified hallway features it has seen before) allows the robot to re-align its graph via an iterative graph optimization process.

Since the network of poses and constraints are built sequentially as the robot moves through the environment, each node is necessarily constrained by the pose immediately before it. This means that any change to a given pose must imply a similar change to all subsequent poses in the graph. The spring-network analogy illustrates this idea intuitively since pushing a node in the network will force all connected nodes to shift around too. Olsons SGD algorithm thus prescribes an additional distribution routine that propagates the change from one pose to all subsequent poses in the graph. Such distribution of changes for each adjustment of each pose node requires $O(N^2)$ complexity (for each N nodes in the graph, optimize and update all subsequent N nodes in the graph) for a naive implementation.

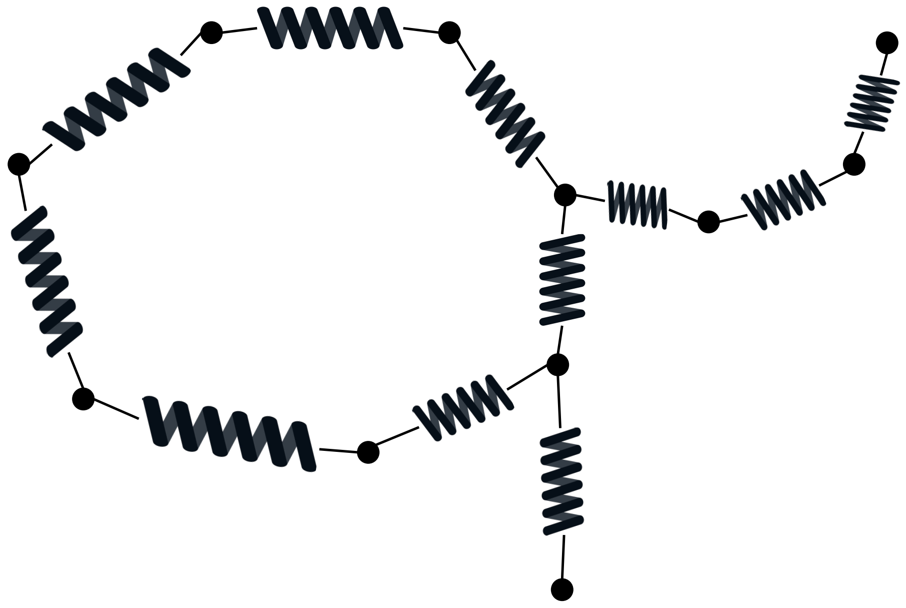

Figure 1.3: Optimization can be visualized as a spring network trying to "relax" from a stretched state. Nudging one of the nodes can cause connected neighboring nodes to shift; thus every adjustment to any single node will require adjusting all other connected node.

## 1.3 Offset-Tree Parameterization of Pose Nodes

Olson applies a special tree parameterization to reduce the $O(N^2)$ runtime to $O(NlogN)$. Instead of storing the exact coordinates of each pose as a 3D vector (x, y, heading) he suggests encoding coordinates as sum-of-offsets of intermediate poses in a walk from root-to-leaf of a binary tree. In other words, looking up the positional coordinates of a pose involves applying a binary walk across the pose array and accumulating the values of the poses in each walk step. The resultant accumulated value is the exact positional coordinates of the searched pose. Each entry in the pose array thus does not store exact x-y-heading coordinates but rather an offset from a parent pose node.

The method to this madness is the benefit of being able to efficiently update contiguous regions of the graph. An update can be distributed throughout the rest of the graph in $O(logN)$ time by walking diagonally up the tree.

## 1.4 Characteristics of Offset-Tree SGD

While SGD may be well-known for massively parallel machine learning applications [2], Olson's offset-tree imposes strong data dependencies between threads which lead to performance-destroying memory-synchronization or thread-scheduling implementations: Every update to a pose node requires modifying $O(log\ N)$ other nodes *atomically*. Every read from a pose node accesses $O(log\ N)$ other nodes *atomically*. Our baseline multi-thread SGD implementation synchronizes access to the tree via a mutex which, unsurprisingly, bottlenecked performance as the number of parallel threads scaled. Performance still suffered even after applying high-granularity synchronization by only locking subsections of the tree. These experiments led us to believe mutual exclusion unsuitable as a synchronization mechanism for Olson's SGD algorithm.

Finding a lock-free solution was challenging because SGD can diverge very easily. For instance, we implemented a multithreaded scatter-gather version on a GPU similar to [3] but we found we needed to apply an aggressive learning rate to prevent divergence. Learning rates are difficult to tune and can reduce performance if used improperly. Furthermore, scatter-gather is still bottlenecked by a $O(N)$ reduction operation in each gather phase.

In the Architecture section we introduce several properties of the offset-tree we used to design a high-performance SLAM processor.

# Chapter 2

# Architecture

We believe pose graph SGD to be a poor fit for traditional parallel programming techniques. The tightly-coupled data dependencies in the offset-tree prevent shared-memory processors from scaling their speedup with the number of threads. On the other hand, we show custom hardware platforms like FPGAs and ASICs can handle such data dependencies more gracefully with pipeline and dataflow design paradigms. All lookup and update operations are simple summations across multiple memory elements which is cheap to build with digital logic. Memory elements can be wired in the pattern of the offset-tree to streamline access to data. Furthermore, we synthesize a specialized gradient descent unit (or "GDU") that serves as our massively parallel SGD processor. The flexibility in designing the GDU allows us to fine-tune fixed-point and dataflow optimizations to minimize latency in processing each constraint. Connecting many such GDUs to our high-speed custom tree unlocks speedup rates that scale beyond CPU/GPU implementations.

In this section we will introduce our method for implementing Olson's offset-tree in digital logic. We briefly mention how our GDUs are built and how they interact with each other and the tree. Then we will discuss limitations of our custom tree hardware and our innovations for overcoming them.

## 2.1 Gradient Descent Units

Gradient descent units are specialized processors that iterate through a list of edges and computes SGD updates for each. Many GDUs are interconnected to form a massively parallel multiprocessor like a GPU. Unlike a GPU however, our processors are multiple-instruction-multiple-data (MIMD) so GDUs independently execute their own instruction and data streams. Even though all GDUs execute the same SGD program, any two GDUs can be executing different parts of the program without interfering with each other. SIMD/SIMT architectures on the other hand suffer a performance penalty when thread execution paths diverge.

We emphasize the MIMD distinction because the GDUs are synchronized by iteration, not by edge. For example, we implemented a scatter-gather version of SGD on a GPU where the scatter phase assigns a graph edge to a thread and schedules them on the GPU. After the GPU finishes executing all threads the graph updates from each thread are then gathered on the CPU and applied to the tree array. GDUs on the other hand employ coarser-grain parallelism; each GDU is instead given a list of graph edges to process. GDUs proactively apply updates to the graph in “real-time” without waiting for the CPU gather phase. Once a GDU has completed a pass through its edge list, it will wait for all other GDUs to finish processing their edge lists before continuing on to the next iteration. Thus the GDUs are not bottlenecked by a gather phase; they immediately continue SGD after synchronizing.

## 2.2 Streaming Tree

Our tree is composed of three distinct components:

1. An *update pipeline* that continuously applies changes to the tree.
2. A tree-array that accumulates changes from the update pipeline and maintains the offset values for each pose node.
3. A *lookup pipeline* that continuously sums the offsets from the tree-array and provides the memory interface with the absolute coordinate values of each pose node.

Figure 2.1 shows the overall design of the tree and its association to the GDUs. Data flows cyclically starting from the GDUs, through the FIFO, through the update pipeline, through the read pipeline, and finally distributed back via the memory interface to the GDUs.

The key goals behind making this *streaming tree* are high-performance atomic reads and writes. We want GDUs to submit an update to the tree in one clock cycle and lookup arbitrary *pose coordinates* (not offsets) in one cycle too. A streaming pipeline serves exactly this purpose because a pipeline "unrolls" a multi-step operation across several registered stages to maximize IPC. Updates from many parallel GDUs are collected into a FIFO that continually feeds the streaming tree (as long as the FIFO is non-empty). Thus the GDUs do not have to spread a update operation across multiple block memory transactions, instead writes happen in a single-cycle push to the FIFO. Similarly, lookup operations are pipelined so GDUs can get pose coordinate data in one memory transaction. Sections 2.2.1 and 2.2.2 details how update and lookup operations are constructed using digital logic primitives.

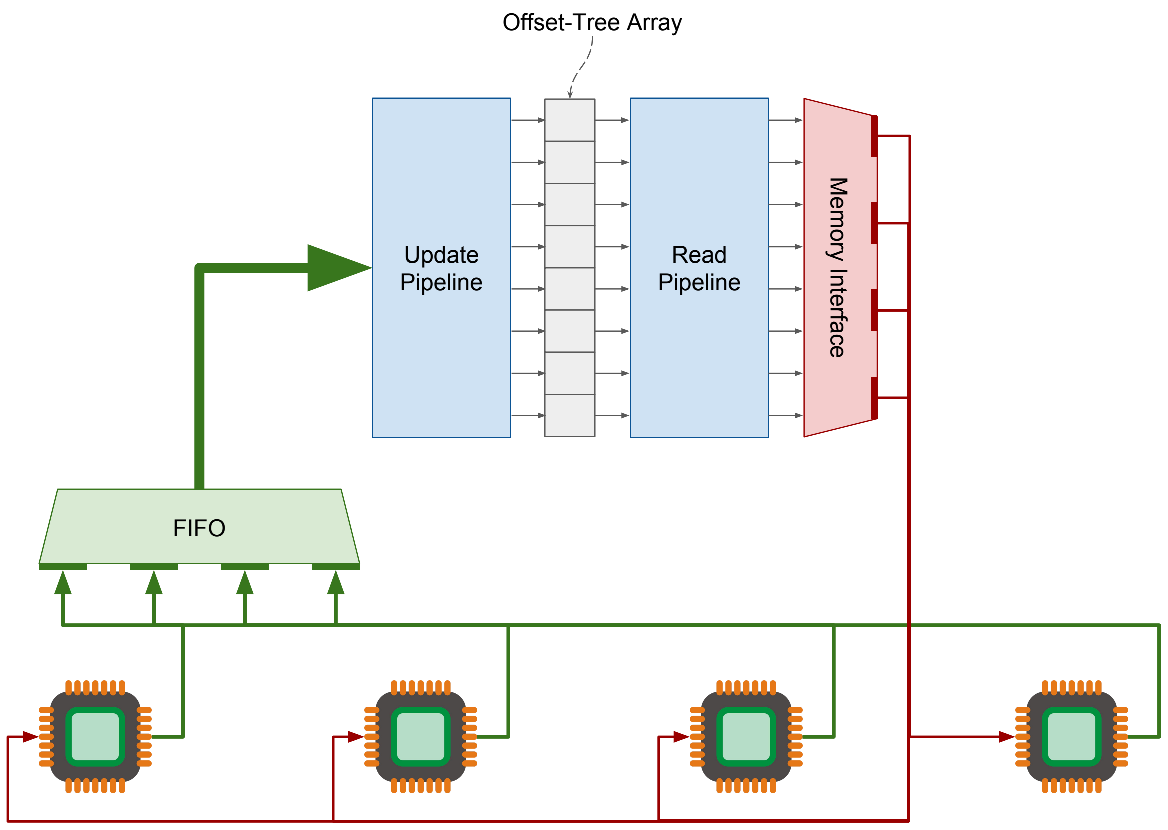

Figure 2.1: GDUs continuously compute constraint edges and generate updates that are streamed through a FIFO into the tree. The update pipeline then applies changes to the offset-tree which propagates its new values through the read pipeline. Each GDU has a dedicated memory interface to the output of the read pipeline so it can access the pre-computed coordinates of any pose.

### 2.2.1 Update Pipeline

Building the update pipeline involves noticing the pattern in which the SGD algorithm walks through the tree. In Figure 2.2, pose nodes are numbered at the leaves and arrows are drawn between nodes that represent the update path. For example, applying an update to pose node 3 also updates pose node 4 and 8 by the same amount. Repeating this process of tracing the update path for every node reveals what we call a "path-flow" graph, which is conceptualized as the dataflow of updates across pose nodes.

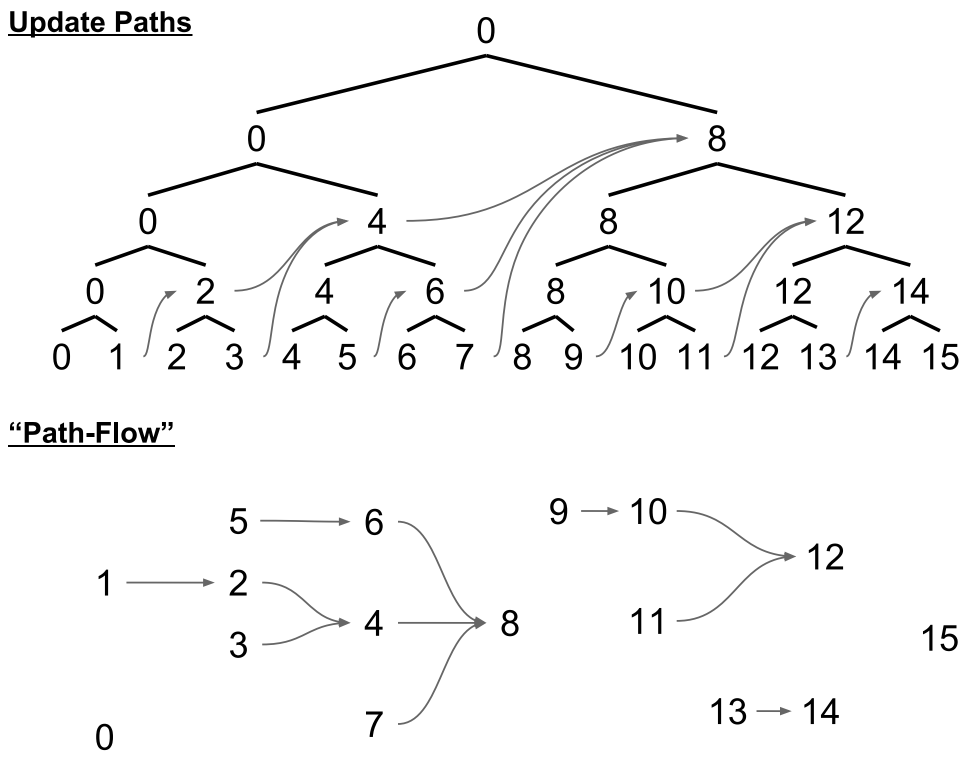

Figure 2.2: Plotting the update paths for a 16-node graph reveals a "path-flow" used to unroll and pipeline tree operations in hardware.

The path-flow graph shows how update operations can be pipelined: For each leaf node, apply its update to the parent node and save the result in the next pipeline stage. Remove the leaf nodes, and repeat the process for the nodes who now have no children. Each pipeline stage is an array of registers that stores the intermediate state of the tree so each stage has as many registers as nodes in the tree. Figure 2.3 shows the result of this process on a 4 stage, 16-node pipeline.

For example, all leaf nodes in Figure 2.2 are 0, 1, 3, 5, 7, 9, 11, 13, and 15. In Figure 2.3, Stage 1 sums nodes 1, 3, 5, 7, 9, 11, and 13 with their respective parent nodes and saves the result in Stage 2. Since nodes 0 and 15 do not have a parent, they are simply passed-through to the next stage.

Removing nodes 0, 1, 3, 5, 7, 9, 11, 13, and 15 reveals nodes 2, 6, 10, and 14 as the

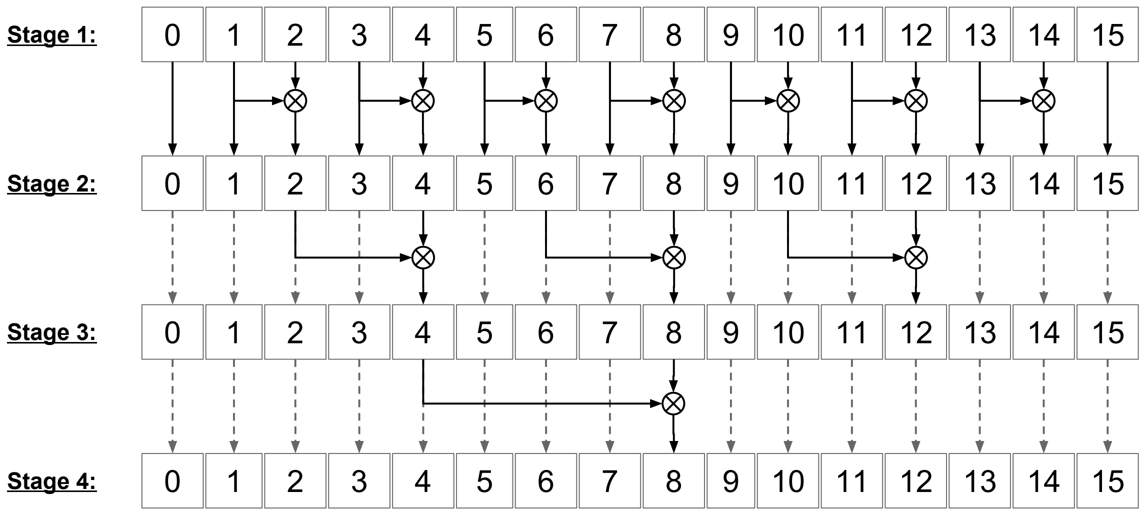

Figure 2.3: Applying the path-flow graph from Figure 2.2 to a pipeline with 4 stages produces an entire copy of the offset-tree each clock cycle.

new leaf nodes. Repeating the process generates the inputs to Stage 3. Again, 14 has no parent and is simply passed-through the pipeline. The pipeline depth is always $log\ N$ because each stage computes one step of the $O(log\ N)$ update algorithm.

### 2.2.2 Lookup Pipeline

Similar to the update pipeline described in Section 2.2.1, the lookup pipeline is built according to the "path-flow" walks across the tree. However, the algorithm for mapping the path-flow to a physical pipeline is different.

The way to conceptualize path-flow for lookups is to imagine data flowing right-to-left starting from the root node 0. Figure 2.4 for example shows that any change made to pose node 0 will necessarily cause a change to 1, 2, 4, and 8. Nodes 1, 2, 4, 8 then causes changes to happen to their child nodes, and so forth. Thus the pipeline is constructed from right-to-left with a stage for each level in the path-flow graph.

Figure 2.5 shows the path-flow from Figure 2.4 applied to a 5-stage, 16-node pipeline.

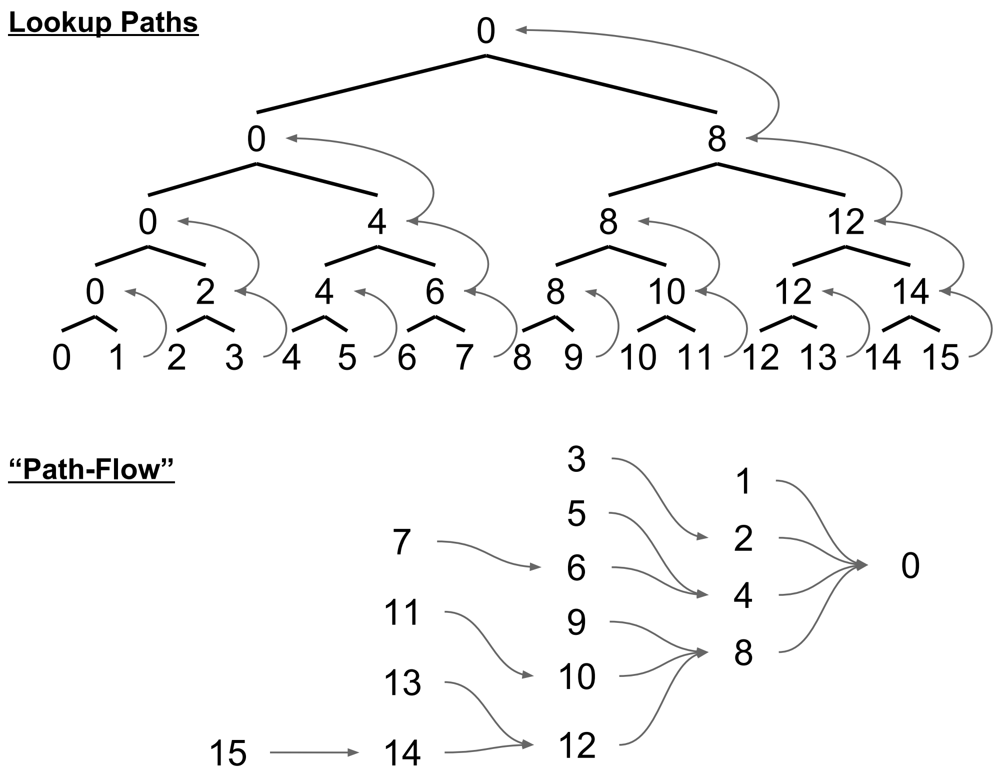

Figure 2.4: Path flow for lookup operations are generated from tracing the steps made for all lookups for each leaf pose node.

The first stage corresponds to the first level in the path-flow graph, with node 0 propagating its value to nodes 1, 2, 4, and 8 in the next stage. Remove node 0 from the path-flow and repeat the process for all nodes without a parent. This means nodes 1, 2, 4, and 8 propagate their values to their children in the next stage. Note that since pose node 1 does not have any children, its value is simply forwarded to the next stage.

The lookup pipeline has an additional register array (Stage 5 in Figure 2.5) compared to the update pipeline. This extra stage is for summing the last two nodes at the left-most end of the path-flow graph.

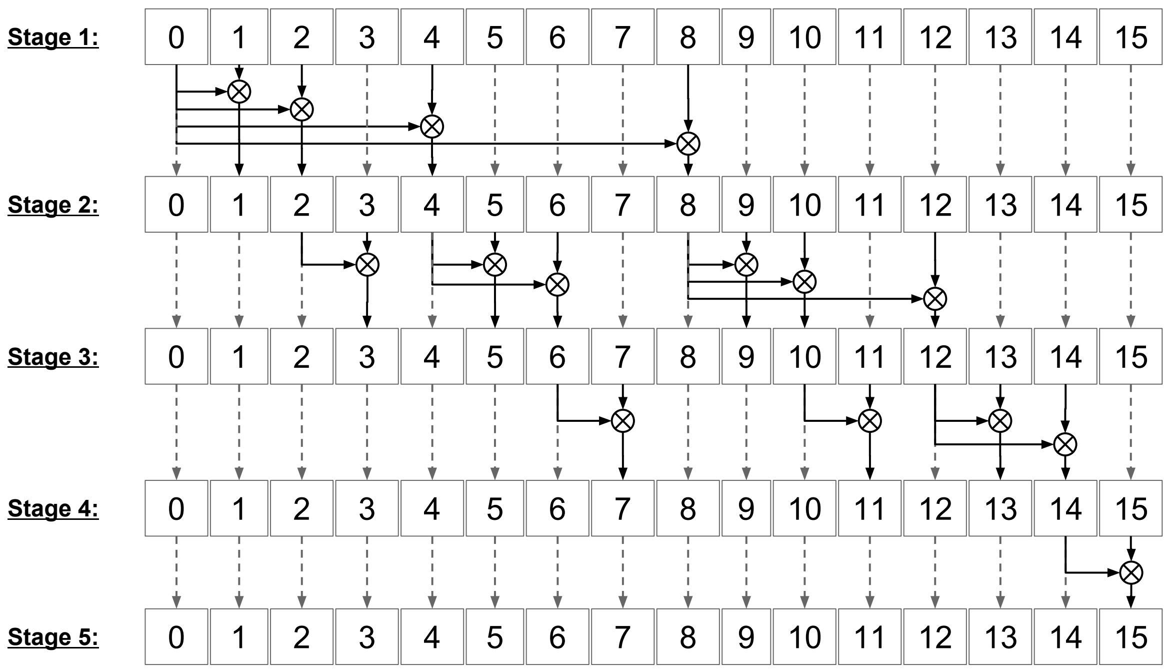

Figure 2.5: The lookup pipeline requires an additional final stage to apply the last summation.

## 2.2.3 Streaming-Tree Characteristics

With both the update and lookup pipelines sandwiching the tree register array (see Figure 2.1) the total clock latency from the front of the update pipeline to the back of the read pipeline will be $2 + 2 \times log\ N$. Two pipelines account for $2 \times log\ N$ while the offset-tree array and the extra lookup stage account for two more cycles.

The streaming tree trades memory coherency with throughput. While tree data is never out-of-date if the GDUs use block memory, overall memory throughput is less since each operation needs $O(log\ N)$ cycles for each update or lookup. Streaming tree, on the other hand, can process one update per clock cycle but suffers from a $2 + 2 \times log\ N$ cycle delay before changes are propagated through the pipeline. Even though this means GDUs are nearly always reading out-of-date information, we've found SGD still works well across many GDUs despite incoherent knowledge of the tree.

| **Tree Size** | **Flip Flops** | **LUTs** | **LUTRAMs** | **FPGA Usage** |
|---|---|---|---|---|
| 16 nodes | 24.8K | 17.4K | 2.88K | 27% |
| 32 nodes | 50.88K | 39.14K | 9.02K | 62% |
| 64 nodes | 102.5K | 84.1K | 23.2K | 132% |
| 128 nodes | 204.5K | 177.7K | 54.5K | 287% |

Table 2.1: Estimated costs of implementing streaming-trees of different sizes. Artix-7 FPGA; each memory element is 32-bits wide.

The astute reader may notice such a streaming tree simply micro-optimizes a $O(log\ N)$ memory operation to $O(1)$. This may not seem like a huge win since logarithmic-time and constant-time algorithms are considered extremely efficient. Our results (see Section 3) show a performance difference at-scale, when there are large numbers of parallel GDUs competing with each other for access to the tree. Furthermore, since the streaming tree was built using custom logic we were able to provide a dedicated read-only memory interface for each GDU connected to the tree. Thus the constant-time, dedicated memory channels for the GDUs

Pipelining is an architectural technique that trades resource usage with performance. The proposed streaming tree is no exception: Handling one update operation per clock cycle costs $O(N\ log\ N)$ memory elements on the chip.

Table 2.1 samples several synthesis runs of the streaming-tree in verilog on an Artix-7 FPGA. Details about our hardware implementation will be listed in Section 3.2; however these preliminary tests indicate significant constraints on the size of our tree design. The way this streaming tree grows in size creates a cause for concern when we consider larger-scale implementations of this design on physical hardware. To this end we propose a hybrid design that compromises between streaming tree and block memory.

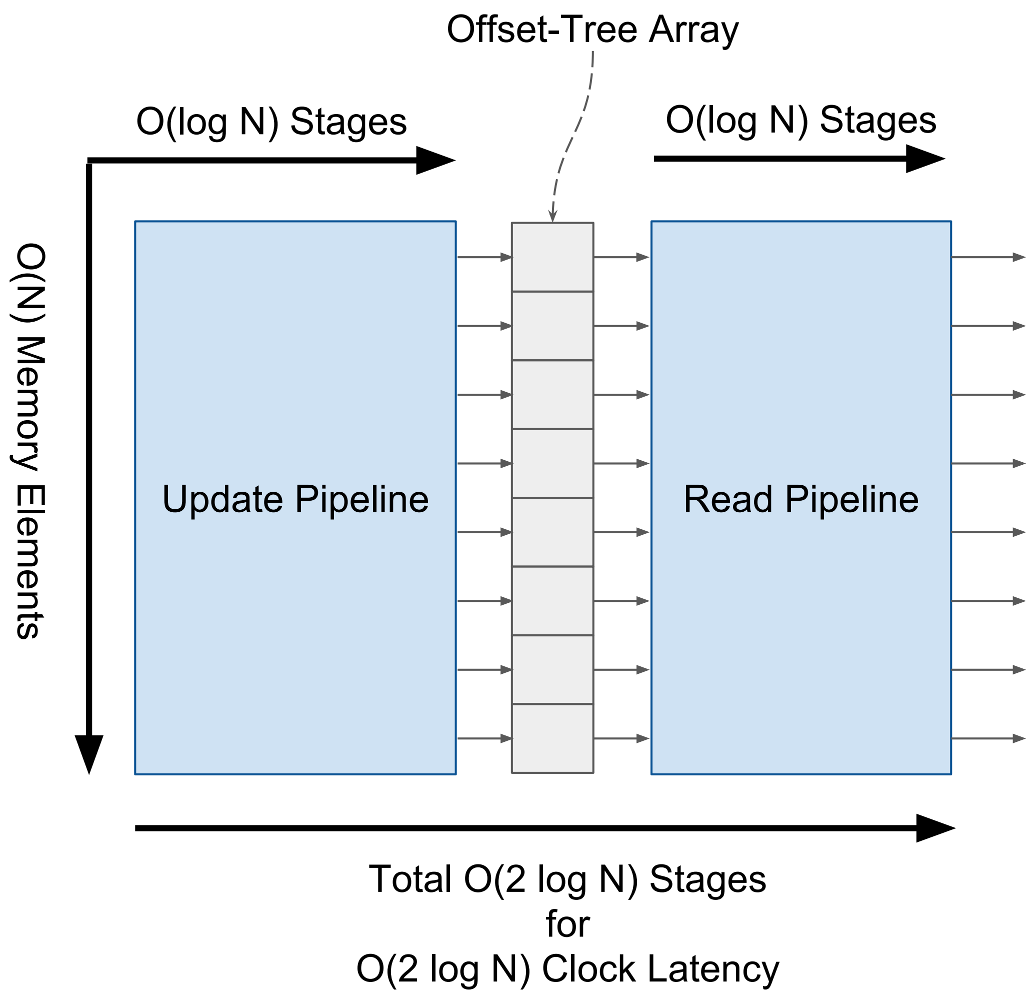

Figure 2.6: Each pipeline is $O(N)$ by $O(log\ N)$ which makes for a $2 + 2 \times log\ N$ total clock latency

## 2.3 Specialized Partitioning

Section 2.2.3 established the scaling issue with the streaming tree. If the entire pose graph can not be stored in the streaming-tree, we are forced to use high-density block memory again.

Block memory does not spell the end of the world, but there is now an additional question of what portions of the graph data is stored in block memory versus the streaming-tree. Fortunately, the path-flow graphs in Section 2.2 show the nodes closest to the root

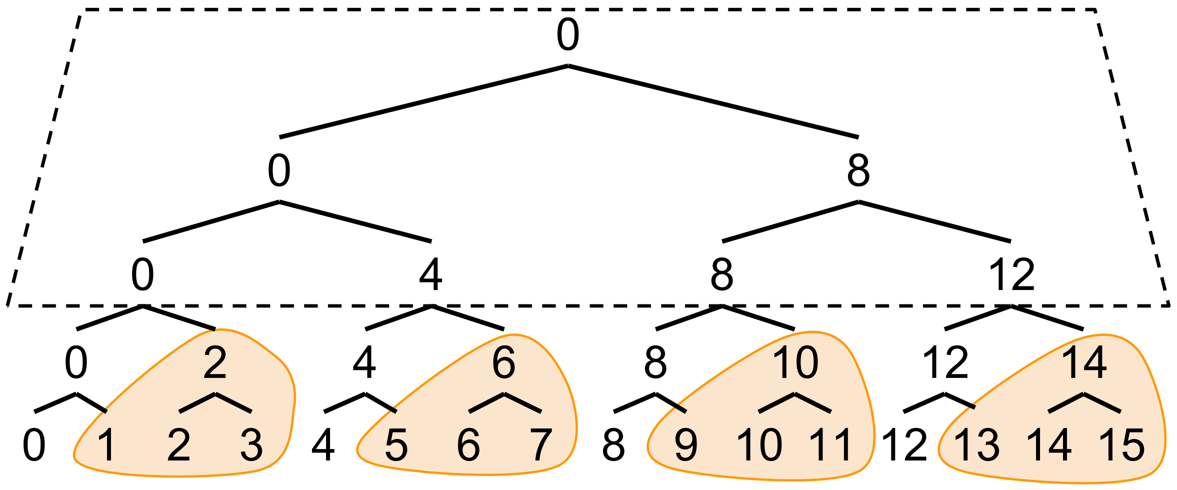

Figure 2.7: Nodes close to the root are grouped into the streaming-tree (dotted trapezoid) while the rest are stored in block memory (encircled blobs).

as the most "popular" with high edge connectivity. In other words, the nodes closest to the root have high update/lookup activity since many paths flow through them. Therefore the streaming tree will be most effective when it is used on the shallow nodes since those nodes bottleneck lookup and update tree operations.

We propose a "skew-tree" partitioning setup to organize the pose graph data between block memory and the streaming tree. Nodes deeper within the tree are stored in block memory while the remaining nodes will be stored in the streaming tree. Figure 2.7 diagrams a setup where nodes 0, 4, 8, and 12 are stored in the streaming-tree accelerator while the rest are grouped into clusters of 3 adjacent nodes in block memory.

Figure 2.8 illustrates a method to this madness by conceptualizing a hybrid block-memory/streaming-tree processor. Each GDU is now paired with some block memory to augment the capacity of the streaming tree. The four slanted blobs in Figure 2.7 correspond to the four GDUs' local block memory; i.e. nodes 1, 2, 3 are placed in GDU 1's block memory, 5, 6, 7 are in GDU 2, etc... Nodes 0, 4, 8, and 12 are left in the streaming-tree.

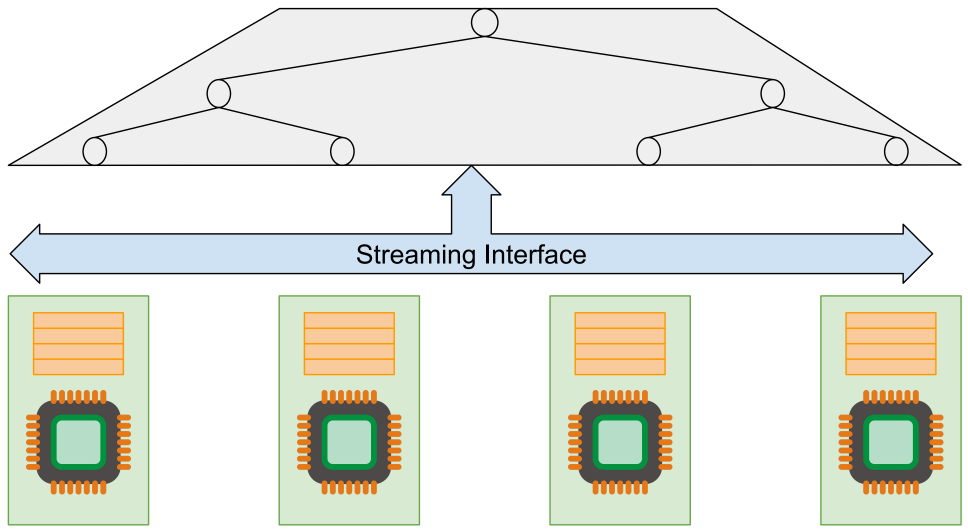

Figure 2.8: Imagine a typical multiprocessor where each core has some local cache memory and a high-performance connection to global RAM. Our proposed design is similar, where each GDU has local block memory while also connected to the global streaming-tree accelerator.

Our skewed partitioning idea is designed to leverage performance characteristics of the local-memory GDU architecture by minimizing the need for GDUs to communicate with other GDUs' local memory. We postulate several properties ("rules") about this architecture:

1. GDUs access to local memory is fast; similar to low-level caches in multicore processors.

2. GDUs access to the streaming tree is fast since each GDU has a dedicated memory bus to it.

3. GDUs have access to each other's local memory, but doing so is a slow operation. "Snooping" each other's local memory requires slow synchronization mechanisms to avoid race conditions.

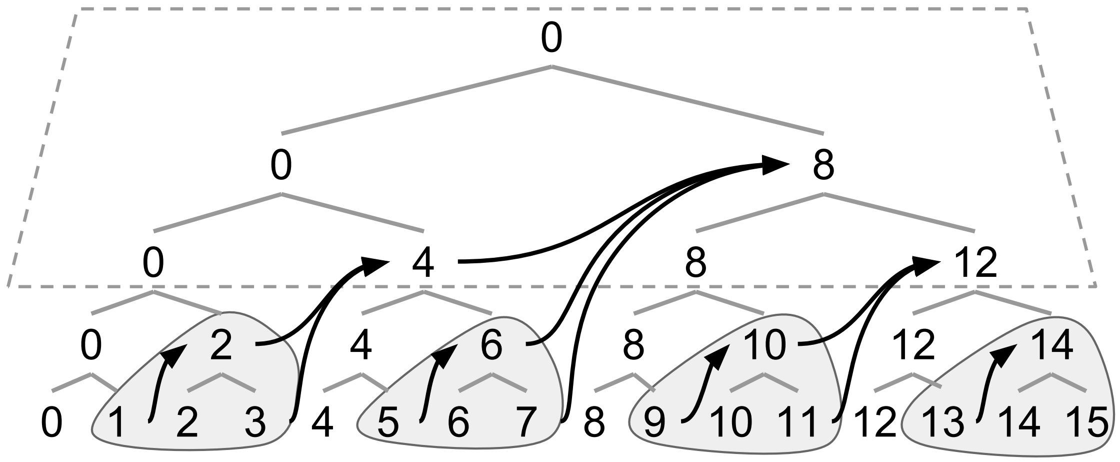

Figure 2.9: Skewed partitioning prevents update paths from crossing between partitions (grey blobs).

Figure 2.9 superimposes the update path-flow on top of the partitioning diagram to show how paths are bounded within the blobs and the streaming-tree. If the paths cross from one blob to another, that means the source GDU must access the destination GDU's local memory. If the paths stay within the local-memory partitions and only leave to read a streaming-tree node, then the operation is assumed to be fast. This policy obeys rules 1 and 2 from the above list and avoid the problem with rule 3.

Consider a naive partitioning scheme (without the streaming tree) where each GDU is simply assigned sequential, equally-sized contiguous portions of the tree (i.e. GDU 1 gets nodes 0 through 3, GDU 2 gets 4 through 7, etc...). The update paths will necessarily cross between partitions and require more synchronization between GDUs i.e. any update GDU 1 does must propagate through to GDU 2's node 4 data.

The rules also apply to the lookup paths in Figure 2.10. None of the paths cross between partitions; paths only leave the partition to access the streaming-tree. Furthermore, if we compare the naive partitioning scheme mentioned earlier we realize nearly all lookup paths would have crossed into some other partition to the left. For example, if

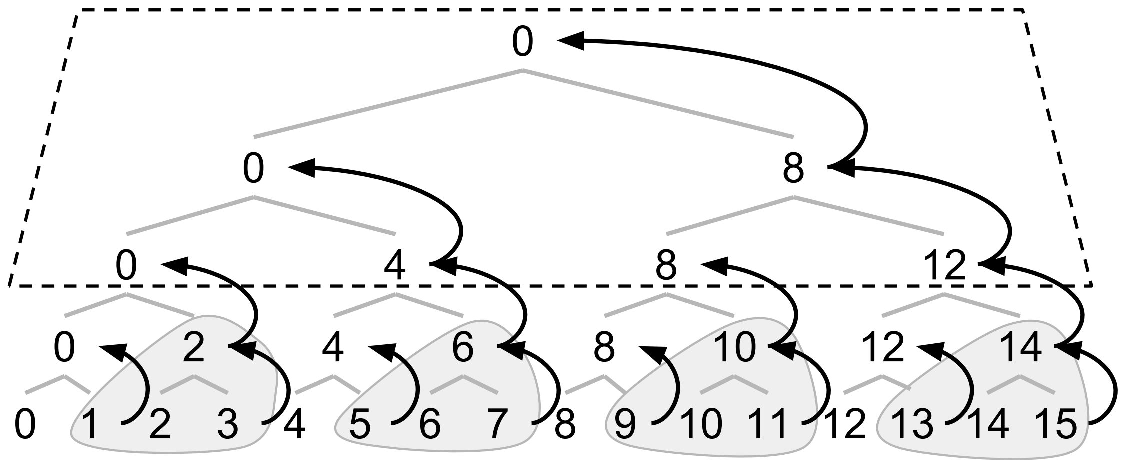

Figure 2.10: Lookup paths also avoid crossing between partitions.

GDU 3 looks up node 11, it will need to cross into GDU 1's local memory to read pose node 0 data. That means more synchronization between GDUs and necessarily lower performance.

Unfortunately, the skewed partitioning scheme does not protect against large loop closures. For example, if there is an edge that connects two nodes far apart from each other (say node 1 and 13 in Figure 2.10) then the GDUs must crosstalk with each other. This unavoidably violates rule 3 and thus serves as the bottleneck of this architectural design. However, if there are relatively few such large loop-closing edges compared to shorter edges then it should not be a big problem. Our experiments have not indicated major performance issues due to these large loop closures.

While this skewed partitioning scheme leverages the best of both streaming-tree and block memory worlds, it opens up the possibility of inconsistency between a GDU's local memory and the streaming-tree. The streaming-tree requires $2 + 2 \times log\ N$ clock cycles before a change is reflected on a subsequent lookup operation. Therefore reads from GDU's local memory will always be up-to-date with the last write while the streaming-tree may return outdated information w.r.t. local memory.

We are hoping this inconsistency delay will not force SGD to diverge since the delay is very short i.e. on the order of tens of clock cycles. So as part of our evaluation of the design we built a simulator to determine if this will cause SGD to diverge. In short, we were able to avoid divergence by adjusting the learning rate. We detail our findings in Section 3.

# Chapter 3

# Evaluation

To evaluate the effectiveness of our design, we implemented a detailed cycle-simulator to model the behavior of our processor. We also implemented a small-scale FPGA prototype of the GDUs and streaming-tree to prove the feasibility of our system. Performance characteristics of the prototype were then fed into the cycle-simulator to predict the performance of a large-scale ASIC-sized version of our processor. We observe several characteristics of the skew partitioning and introduce a over-provisioning design to handle asymmetric loads.

## 3.1 Graph Convergence

The iterative nature of SGD means there is no “stopping point” for the algorithm. In practice the algorithm is run until the graph’s quality is “good enough”; exact stopping conditions for SGD are tuned for specific applications. In our case, we’ve found 100 iterations to sufficiently optimize our pose graphs such that resulting graphs from multiple runs are visually similar. Characterizing the exact “goodness” of a pose graph is difficult to rigorously define and compare so we leave that analysis outside the scope of this project.

## 3.2 FPGA Prototype

We built a proof-of-concept design of the streaming-tree and GDU on an Artix7 FPGA platform. The GDU was written using Vivado High-Level Synthesis (HLS) tools while the streaming-tree was written in Verilog. A MicroBlaze soft-processor acts as the central controller to coordinate the GDUs across an AXI-4 bus. A small design has the advantage of rapid turnaround times after making changes to the hardware description. Thus the prototype only concerns one GDU and one streaming-tree. However we standardized the interfaces to our streaming-tree and GDUs to use AXI-4 bus protocols so the block design has the capacity to include arbitrary parallel GDUs.

The prototype consists of one GDU connected to one streaming-tree containing a small graph ( 10 nodes). We ran simple tests to confirm that the GDUs and tree were functionally complete and capable of optimizing an example pose graph.

Addition operations between each pipeline stage must happen within a clock cycle; thus we opted for fixed-point approximations of the pose node data since fixed-point arithmetic complete in one cycle. In our prototype we used signed 32-bit words with 11 fractional bits but the design can arbitrarily scale to whatever bitwidth is appropriate for the application. Consequently the GDUs apply appropriate conversion routines whenever it interacts with the streaming tree.

In terms of timing, we were most concerned with each GDU's cycle delay when it processes one graph edge. According to the HLS synthesis reports we estimated SGD itself–without walking the tree data structure–requires around 744 clock cycles. The simulator thusly delays the GDUs' state machines for 744 cycles every time the GDUs enter a computing state.

| LUTs | LUTRAMs | Flip-Flops | BRAMs | DSPs |
|---|---|---|---|---|
| 34.11K | 1.6K | 41.4K | 75.5 | 76 |

Table 3.1: Post-implementation logic resource usage for the FPGA prototype of a GDU and streaming-tree on an Artix-7. Note that these values include the usage of the MicroBlaze controller. The streaming-tree stores 16 poses.

### 3.2.1 Utilization

As a sanity-check, we reported resource utilization estimates to ensure our design can be reasonably implemented on a physical chip. Table 3.1 shows how many FPGA logic elements are used to build our prototype design. Since we already knew that the streaming-tree would be expensive in Section 2.2.3 the focus of this analysis is on the cost of the GDUs.

According to HLS results, a single GDU takes up about 24K flip-flops and around 36K lookup tables (combinational logic elements). While that GDUs footprint is not trivial for a small FPGA like the Artix-7, we believe its small enough to scale to ASIC-sized chips. In other words, for a GDU that takes less than 100K logic elements we could theoretically build a SLAM processor with hundreds of GDUs that will fit on a typical integrated circuit.

## 3.3 Simulator

The simulator is based off the synthesis and timing data as described in Section 3.2. We scale the number of GDUs in the simulator to extrapolate an estimation of the design performance.

Figure 3.1 provides a top-level overview of the simulation results. We ultimately experimented with three separate designs: One where all GDUs synchronize access to a shared block memory resource. One where the block memory is augmented with the

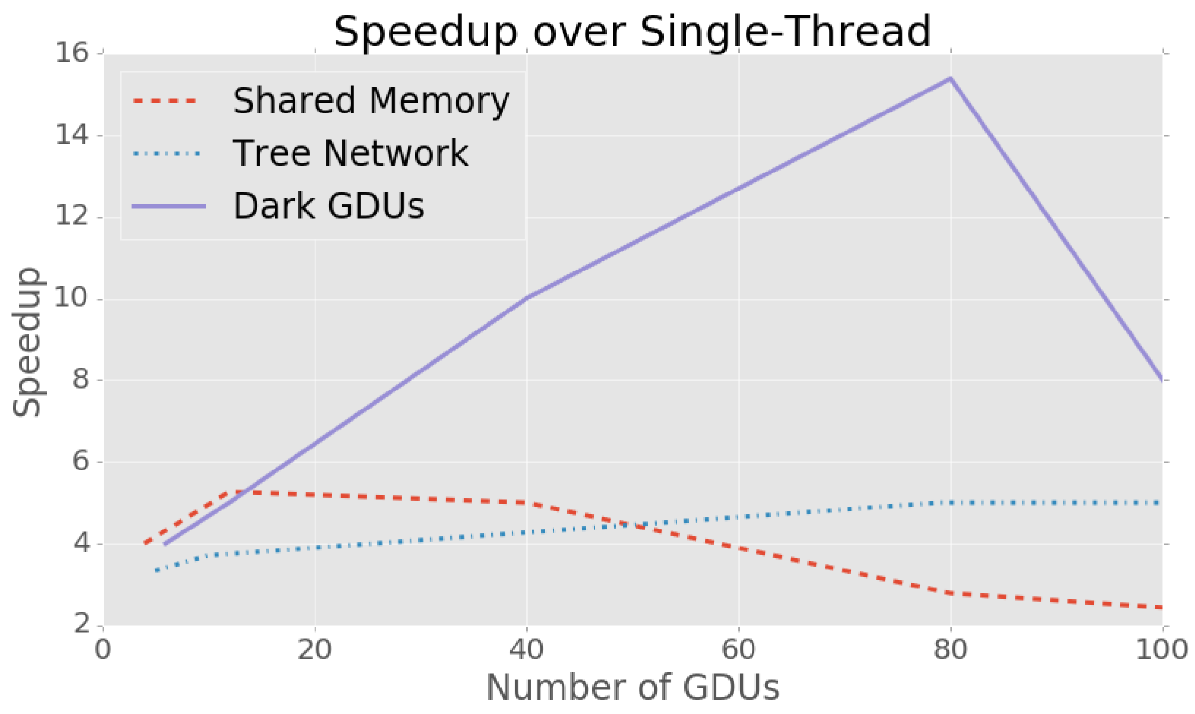

Figure 3.1: Speedup curves for a range of parallel GDUs. The "Dark GDU" design provides much better performance scaling thanks to specialized streaming-tree accelerators and load balancing.

streaming tree accelerator (as illustrated in Figure 2.8). And one where GDUs are selectively turned on or off similar to the "dark silicon" paradigm. So-called "dark GDUs" were a direct result of our simulation data and are detailed in Section 3.3.2.

### 3.3.1 Skewed Partitioning

The results of the skewed partitioning is shown on Figure 3.1, where "Tree Network" curve shows slightly better speedup on larger numbers of parallel GDUs since there is less memory contention between GDUs. However, there are still issues with the performance bottlenecking because the way GDUs are allocated edges to process.

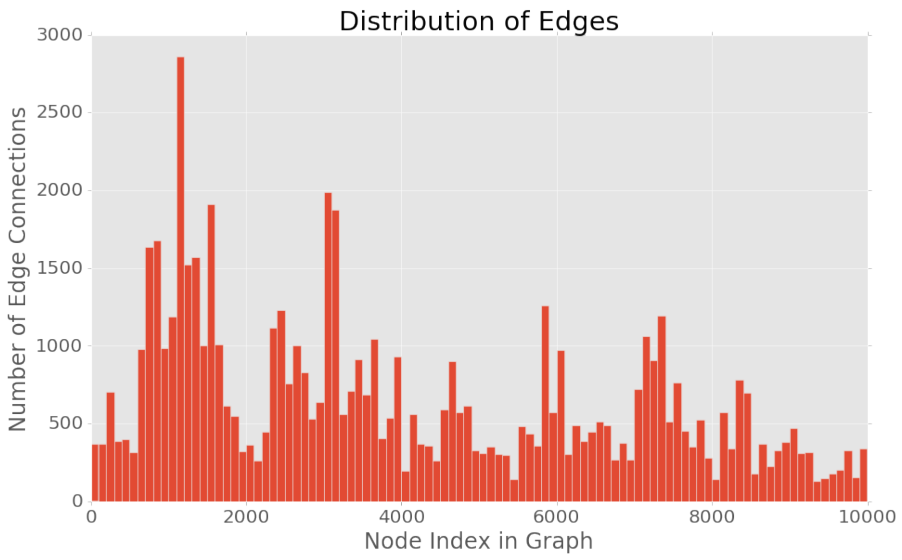

Figure 3.2: Very large pose graphs (in this case 10K nodes) do not have uniform distribution of edges across all nodes. This histogram shows how some nodes have vastly more edges connected than others, which means the GDU working on nodes between 0 and 2000 will spend more time optimizing its edges than all other GDUs.

### 3.3.2 Dark GDUs

One of the downsides of using the skewed partitioning technique described in Section 2.3 are asymmetric amounts of work delegated to the GDUs. One of our datasets had a very large graph with many edges connecting between nodes. Some pose nodes have more edges connecting them than other nodes. Thus, if GDUs can only optimize edges that connects nodes in their partition, then for large graphs some GDUs will have more edges to process than others. If one processor is given disproportionately more work all other processors will be blocking and waiting for the slowest processor to complete. Therefore the parallel performance of the overall processor is damped due to a single GDU working on many edges while the others wait for it to complete. That was our suspicion for the downward slant in the speedup curve for Figure 3.1.

Our solution to this problem was to draw from the “dark silicon” paradigm and insert extra GDUs into the processor even though not all of them may be turned on in

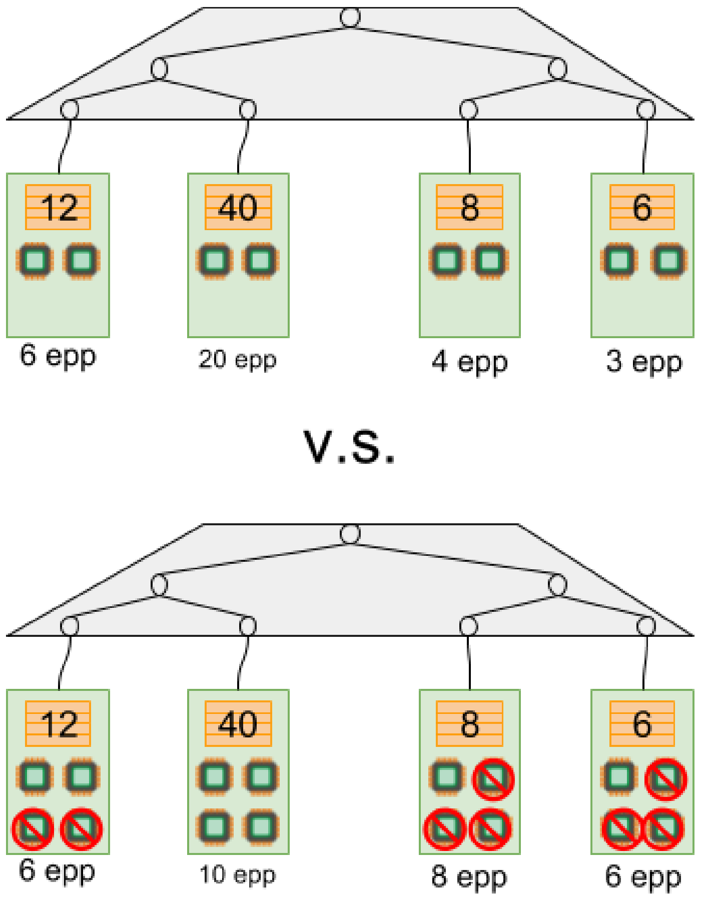

Figure 3.3: Clustering many extra GDUs together can result in a more uniform distribution of edges at the cost of more hardware used. Both top and bottom configurations have 8 GDUs actively participating in SGD. However, the bottom setup has better parallel performance because the maximum number of edges any GDU has is 10 (instead of 20).

practice. These so-called “dark GDUs” are grouped into clusters that share the same skew partition and local block memory but are selectively enabled during runtime to help load-balance overworked GDUs.

Figure 3.3 illustrates an example where some GDUs are given more edges to process than others. If all GDUs are required to be turned on and in use, then the cluster of GDUs given the most edges (in this case the second from left with 40 edges per processor) will significantly slow down overall performance. However, if there are twice as many processors in each cluster but most of them are not turned on, then the maximum number of edges processed by any given GDU is reduced (10 edges per processor in the example) thereby improving performance.

Adding the extra unused GDUs can be rationalized by the low resource cost of each GDU. A GDU only costs tens of thousands of logic elements to implement which is very cheap on a modern ASIC. Furthermore since most of the GDUs are not turned on, there is no concern for exorbitant power usage from hundreds of processors simultaneously running on a single processor.

With dark GDUs enabled on the simulator, we observe much better speedups in Figure 3.1. The GDUs are no longer spending time waiting for one over-worked GDU to complete which translates to better parallel performance as the design scales.

# Chapter 4

# Related Works

Hardware-acceleration of robot algorithms were explored quite extensively in the past: Path planning saw FPGA and GPU applications in [4] and [5] respectively. Vision and reconstruction algorithms are quintessential users of FPGAs and GPUs on mobile robots [6]. Mobile robots are common targets for these works because of their embedded characteristics.

SLAM however, eludes mainstream micro-architecture efforts due to its sophisticated problem definition and implementation. The "chicken-egg" interdependency of mapping while localizing motivate engineers to focus on optimizing single-thread performance on classic processors instead of using highly parallel architectures [7, 8, 9].

### 4.0.1 Multithreading and GPU

The intuitively parallel appearance of Olson's SGD algorithm belies the subtleties of distributing data throughout the map in parallel. Our attempts at a multithreaded implementation of SGD yielded unpromising performance results, in addition to several background works that have encountered similar roadblocks.

First we established that Olson's SGD does not fall into the domain of lock free approaches as described by “Hogwild” [2]. While the pose graph’s adjacency matrix is sparse, each edge update requires data to be propagated through the rest of the graph. We tried implementing a lock-free Hogwild-like version but it simply diverges. The only way we got it to not diverge was with extremely aggressive scaling factors that negated any parallel performance gains.

Non-SGD methods include [10] where they used a more traditional solver called CNC to optimize pose graphs with less-impressive results than [3]. The authors considered the iterative nature of solvers like CNC to be the main performance bottleneck for their implementation. Thus they concluded that traditional GPU sparse solvers provide small incentive compared to highly-optimized CPU solvers.

However, the primary issue with current state-of-the-art GPUs is power consumption: The NVIDIA 570 GTX used in [3] for example can draw up to 200 watts of power which excludes its use on low-power embedded robots.

SGD is very popular in machine learning fields, and their sparse nature paved the way to large scale parallel implementation on GPUs. However, our findings have failed to provide substantial evidence that pure multithread-GPU approaches to pose graph optimization are viable in the long term. We think the cost of using traditional multi-threading for performance enhancement do not outweigh the performance-power benefits of multicore and GPU systems. Thus, we conclude that SLAM on mobile robots imply first-class micro-architectural support.

### 4.0.2 FPGA

FPGAs have been a somewhat elusive target for SLAM because there are not many subroutines within SLAM implementations that intuitively fit FPGA paradigms.

[11, 12] for example built matrix multiplication co-processors to accelerate EKF and

VO-SLAM subroutines, but $O(n^2)$ scaling of matrix multiplication impose limitations on map size and capacity.

[13] on the other hand, exported CORDIC and random-number-generation operations to the FPGA while the rest of the particle filter SLAM is done on a MicroBlaze. Particle filters are a popular choice for SLAM applications, however modern robots require robust graph-based systems to handle complex environments.

### 4.0.3 GDU

Our approach synthesizes the SGD algorithm into a physical architecture on a chip. In other words, instead of co-processing subcomponents of a larger SLAM algorithm we rebuilt the entire SLAM backend itself as a fully independent processor. This allowed us to rethink SGD as a totally parallel process that maps and localizes at the same time as opposed to iteratively alternating between the two. In our system, the streaming tree is constantly incorporating map updates while the GDUs are computing pose locations. Both the streaming tree and the GDUs are constantly streaming data between each other in a highly efficient and parallel manner. This is where our convergence performance beats GPU and CPU methods.

A fully custom processor also gave us superior flexibility in power-performance-cost tradeoffs compared to previous attempts to accelerate SLAM on hardware. Everything from fixed-point to GDUs utilization can be fine-tuned for each particular platform from high-performance PCIe-FPGA computing platforms to small embedded FPGA-SoCs.

# Chapter 5

# Conclusion

While embedded systems are traditionally thought of as being dominated by tiny micro-controllers, large scale autonomous systems present a new set of real-time computational challenges of significant scale for embedded systems architecture. Dealing with noisy data, integrating information from multiple sensors, managing continuous operation with limited interruption, balancing load across multiple resources, reducing communication, and minimizing error are often times goals in direct conflict with one another. The work we present attempts to strike this balance through the use of our skewed tree data structure, our computational GDU specialized to the applications, our new application-specific streaming-tree, and finally through new techniques on load balancing. More concretely, however, we do show that a) there does exist a tradeoff between synchronization accuracy and algorithm convergence where eager computation can introduce error through asynchronous operation and that this is an important limiter to scaling in a traditional design, b) that pipelined computations performed in the streaming-tree can reduce error introduced by this eager parallelism in particular through the elimination of the most contended-for resources at the top of the tree data structure, and c) that it is possible manage variable loads but that doing so requires the addition of extra resources which are only turned on when needed and helps avoid the significant amount of network overhead that would required to manage the load otherwise. In the end we find that the resulting system exhibits superior parallel performance scaling compared to traditional state-of-art processor designs.

### 5.0.1 Future Work

We believe there is much promise in the areas of robotics and computer architecture due to the embedded and realtime constraints required for robot control. We investigated pose graph SLAM as an example of an application where custom chip designs are a viable alternative to traditional computing infrastructures.

With regards to this particular work, the next step would be towards realizing a full-scale prototype on a large FPGA. Integrating the full-scale design with an existing SLAM framework like RTAB-Map would reveal the true performance of our processor. The profiler within RTAB-Map will provide details on the speedup our design provides over a typical CPU. Furthermore, a next-generation simulator can provide more accurate cycle timing and perform detailed design space exploration to look for pareto-optimal processor configurations.

# Bibliography

[1] E. Olson, J. Leonard, and S. Teller, *Fast iterative alignment of pose graphs with poor initial estimates*, in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pp. 2262–2269, May, 2006.

[2] B. Recht, C. Re, S. Wright, and F. Niu, *Hogwild: A lock-free approach to parallelizing stochastic gradient descent*, in *Advances in Neural Information Processing Systems 24* (J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, eds.), pp. 693–701. Curran Associates, Inc., 2011.

[3] A. Ratter, C. Sammut, and M. McGill, *Gpu accelerated graph slam and occupancy voxel based icp for encoder-free mobile robots*, in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 540–547, Nov, 2013.

[4] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, *Robot motion planning on a chip*, in *Proceedings of Robotics: Science and Systems*, (AnnArbor, Michigan), June, 2016.

[5] J. Pan, C. Lauterbach, and D. Manocha, *g-planner: Real-time motion planning and global navigation using gpus*, in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pp. 1245–1251, AAAI Press, 2010.

[6] Q. Gautier, A. Shearer, J. Matai, D. Richmond, P. Meng, and R. Kastner, *Real-time 3d reconstruction for fpgas: A case study for evaluating the performance, area, and programmability trade-offs of the altera opencl sdk*, in *2014 International Conference on Field-Programmable Technology (FPT)*, pp. 326–329, Dec, 2014.

[7] S. Lee and S. Lee, *Embedded visual slam: Applications for low-cost consumer robots*, *IEEE Robotics Automation Magazine* **20** (Dec, 2013) 83–95.

[8] A. Dine, A. Elouardi, B. Vincke, and S. Bouaziz, *Enhancing processing time for graph-based slam applications*, in *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, pp. 706–711, April, 2014.

[9] M. Abouzahir, A. Elouardi, S. Bouaziz, R. Latif, and T. Abdelouahed, *An improved rao-blackwellized particle filter based-slam running on an omap embedded*

*architecture*, in *2014 Second World Conference on Complex Systems (WCCS)*, pp. 716–721, Nov, 2014.

[10] D. Rodriguez-Losada, P. S. Segundo, M. Hernando, P. de la Puente, and A. Valero-Gomez, *Gpu-mapping: Robotic map building with graphical multiprocessors*, *IEEE Robotics Automation Magazine* **20** (June, 2013) 40–51.

[11] D. T. Tertei, J. Piat, and M. Devy, *Fpga design and implementation of a matrix multiplier based accelerator for 3d ekf slam*, in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pp. 1–6, Dec, 2014.

[12] M. Gu, K. Guo, W. Wang, Y. Wang, and H. Yang, *An fpga-based real-time simultaneous localization and mapping system*, in *Field Programmable Technology (FPT), 2015 International Conference on*, pp. 200–203, Dec, 2015.

[13] B. G. Sileshi, J. Oliver, R. Toledo, J. Gonçalves, and P. Costa, *Particle Filter SLAM on FPGA: A Case Study on Robot@Factory Competition*, pp. 411–423. Springer International Publishing, Cham, 2016.