FPGA-based Accelerator for Vector Search

Lasya Balachandran
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, United States
lasyab@mit.edu

Sanjay Seshan
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, United States
seshan@mit.edu

***Abstract*—With the application of graphs in large-scale modeling, such as social network analysis and image and video segmentation, among other applications, graphs are increasingly being used to encode and find complex relationships between data for machine learning models, leading to an increased need for optimization of these models. As a result, in order to better support model-specific algorithm efficiency, there has been work to create specialized hardware accelerators focusing on aspects such as memory accessing, latency, and resource allocation. However, current accelerators for graph problems are not scalable and can only be optimized for a single application, such as graph random walks or matrix multiplication. Existing systems also run on CPU or GPUs. Following from prior accelerator work on FPGAs, we plan to implement a graph-based vector search algorithm, based on iQAN, that runs on an FPGA to produce better algorithm performance than on existing systems and can be used for more versatile applications.**

***Index Terms*—FPGA, accelerator, graph, vector search**

I. Introduction

The goal of this project is to implement an accelerator for a general-use search algorithm. One of the main problems when working with large graphs is the large amount of computations, which is a reason why current accelerators have focused on specific optimizations on CPU or GPU systems [1]. One way we can reduce computations is by running the calculations on only a subset of the graph. For example, the novel graph-based vector search algorithm iQAN [2] uses a priority queue of certain length to approximate the most similar points and avoid brute-force-checking all of the points. Graph sampling, where we select a random subset of vertices representative of the entire graph, can also be used to reduce a graph. We are planning to implement a vector search algorithm on an FPGA to get improved results as a result of higher customizability with pipelining, parallelizing, and optimizing the architecture.

We propose a new implementation of the BFiS on an FPGA, using a custom architecture, that can take advantage of the multifaceted processing of a Xilinx Spartan-7 FPGA, to improve performance compared to current implementations.

The following sections will discuss the implementation of our system and individual components of the system.

II. System Overview

Based on the paper on iQAN, we are implementing the Best First Search (BFiS), a graph-based vector search algorithm, on the FPGA [2]. The algorithm is as follows:

```
Input: graph G, starting point P, query Q, queue
    capacity L
Output: K nearest neighbors of Q
priority queue S <- {} /* sorted based on distance
    */
index i <- 0
compute dist(P, Q)
add P into S
while S has unchecked vertices do /* stop condition
    */
    i <- the index of the 1st unchecked vertex in S
    mark v_i as checked
    foreach neighbor u of v_i in G do
        if u is not visited then
            mark u as visited
            compute dist(u,Q)
            add u into S /* u is unchecked */
    if S.size() > L then S.resize(L)
return the first K vertices in S
```

Our design [Figure 8 in Appendix section XI-A] is composed of three main components: graph storage, vector search, and SPI output. The vector search module is a much larger module that is composed of several different modules.

We have chosen to implement our system in a bottom-up approach, implementing core features such as queues, priority queues, memory interfaces, and related layouts before designing the interconnections for each module.

III. CPU Implementation

A. Design

We have created a basic implementation of BFiS in C++ as a baseline comparison. The code can be found in Appendix section XI-B. By default, priority queues in C++ have a max heap structure (can be converted to min heap), where only the first element can be accessed or removed from the queue. Since, according to the BFiS pseudocode, we must be able to access the closest visited point that we have not yet checked in the priority queue, we created two priority queues: a min heap `S` to keep track of all visited points and a max heap `checked` to keep track of checked points.

In our min heap structure, our top point is our closest unchecked point. Once we check a point in `S`, we add it to `checked` if either the size of `checked` is less than our desired priority queue capacity `L` or the distance from the query to the point is less than the distance of the query to the top point in `checked` (farthest checked point from the query). If the size of `checked` is greater than `L` after adding

a point, we pop the top element off of the queue, allowing us to maintain queue capacity `L`.

We exit the `while` loop when the capacity of `checked` is `L` and the closest point in `S` is farther than the farthest point in `checked` since this denotes that the `L` closest points (based on our approximation) have all been checked. We then returned the bottom `k` elements in `checked` to represent the approximate `k` closest points to the query.

We also use the square of the distance in our CPU implementation instead of the actual distance since the square would be easier to implement on the FPGA, making it easier to directly compare the performance of the two implementations.

We designed system on the FPGA such that it is similar to this CPU design. For example, since we can remove the top element of priority queue `S` and access its size, we designed our priority queue module to have similar capabilities.

B. Verification and Performance

The C++ implementation of BFiS was tested against a data set containing 10,000 points of 128 dimensions each and 100 queries. Table I shows the algorithm performance in terms of accuracy and runtime with varying sizes of `L` and `k`$= 100$. The accuracy was compared to the actual `k` closest points from a brute force approach. A C++ implementation of the brute force approach runs in about 0.8 seconds.

TABLE I
PERFORMANCE OF BFIS (C++) ON GRAPH WITH 10,000 POINTS

PQ Capacity L	Accuracy	Runtime (sec)
200	0.8434	0.072317
500	0.9482	0.121287
700	0.9892	0.144219
900	0.9893	0.163350
1000	0.9993	0.172315

Based on the accuracies, the algorithm seems to approximate the `k` closest points well. As the size of `L` increases, the accuracy of the approximation approaches 1 since the algorithm visits more points. The runtime of BFiS is also significantly faster than the runtime of the brute force approach. We plan on further testing the CPU implementation with graphs of up to 1,000,000 points for eventual comparison with our FPGA version, but for the scope of this project, we did not use graphs that large and used small priority queue sizes (< 10) due to size limitations of our FPGA.

We also tested the system on a series of smaller predefined datasets (tester and citeseer) and configurations as a preliminary baseline comparison to the FPGA results. The CPU results are seen in Table II. The number following the dataset name denotes the number of dimensions used. The dataset tester contains 8 vertices and 24 edges, while citeseer contains 3312 vertices and 9072 edges.

These results are from a server with an Ubuntu Linux server with a 64-bit Intel Xeon E5-2690 CPU and sufficient memory for the graph. The code is running on a single core.

TABLE II
PERFORMANCE OF BFIS ON AN CPU FOR SELECT DATA

Dataset	PQ Capacity L	Runtime (sec)
tester_4	5	$5.3 \cdot 10^{-5}$
tester_8	5	$5.6 \cdot 10^{-5}$
citeseer_4	5	0.000113
citeseer_4	8	0.000116
citeseer_4	10	0.000112

IV. CORE COMPONENTS

We designed 6 smaller modules to create a working system on the FPGA: n-dimensional distance calculator, FIFO queue, specialized priority queue, graph memory storage, interface for finding if a point has been visited, and graph fetcher for finding neighbors of a given point and position data for the neighbors. The FIFO queue was used to build the specialized priority queue and graph fetcher, while the other 5 modules were integrated together in a larger BFiS module to run the algorithm. We also used Manta to transmit inputs to and receive outputs from the FPGA. A block diagram of the entire system is shown in Figure 8 in Appendix section XI-A.

A. Distance Calculator

We implemented a distance calculator that computes the square of the Euclidean distance between two points, according to the following equations:

$$\text{dist}(\vec{v}) = ||\vec{v}||$$

$$= \text{dist}(x_1, y_1, x_2, y_2 + \dots) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots}$$

$$\text{dist}^2(x_1, y_1, x_2, y_2, \dots) = (x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots$$

To avoid implementing a square root, we decided to consider the square of the distance anywhere dist is used since this calculation will provide the same result to our algorithm.

The input vertex and query to our distance module represent two vectors (a point in the graph and the query), each of dimension DIM, which is set at compile time. Our graph fetcher module (Section IV-F) provides the position vector of the vertex, while our query is an input to the system. Each element of the vertex is provided sequentially on subsequent cycles (i.e. P_0 and Q_0 on the first cycle, P_1, Q_1 on the second cycle, etc). We assume that the elements of the input vectors are in order in memory for all practical purposes.

```
module distance #(parameter DIM = 2)(
  input wire clk_in,
  input wire rst_in,
  input wire data_valid_in [DIM-1:0],
  input wire [31:0] vertex_pos_in [DIM-1:0],
  input wire [31:0] query_pos_in [DIM-1:0],
  output logic [31:0] distance_sq_out,
  output logic data_valid_out
  );
```

This system is pipelined by parallelizing the subtraction, multiplication, and addition for the distance calculation, as shown in Figure 1. On the first cycle, we calculate $Q_0 - P_0$ and on the second cycle, we calculate square the result. On the second cycle, we also calculate $Q_1 - P_1$ and similarly parallelize our subtraction and multiplication on future cycles.

We originally designed this module using two states: one for subtraction and multiplication and a second to recursively add the squares of the differences. However, since the recursive adder took 2 cycles, we realized that we could parallelize the addition similarly to how we parallelized the subtraction and multiplication to reduce our cycle count by 1 cycle. With many distance calculations in the overall BFiS module, this change could significantly reduces the overall cycle count.

To meet timing requirements, we later implemented a 32-cycle multiplier that multiplies two 32-bit numbers using each power of two instead of using the built-in multiplication functionality. Each multiplier runs in parallel so that we have a total delay of only 32 cycles.

```
int multiply(int x, int y)
{
  int ret = 0;
  for (int i = 0; i < 32; i++)
  {
      if (((y >> i) & 1) == 1)
      {
          ret += (x << i);
      }
  }
  return ret;
}
```

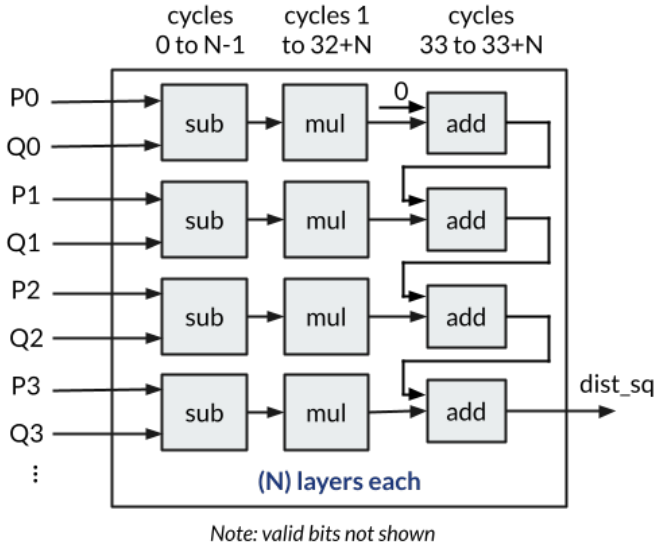


Fig. 1. Distance calculator tree

We created a testbench to simulate our distance calculator with dimensions 3, 4, and 9. Our module correctly returned the square of the distances for each with 36, 37, and 42 cycles, respectively.

B. FIFO Queues

We developed an effective FIFO [Figure 2] that is versatile for use throughout our system. This FIFO is important for maintaining ordering for memory requests, buffering outputs, buffering inputs for distance calculations, and more.

We provide an easy interface for our FIFOs:

```
module FIFO #(parameter DATA_WIDTH = 32, parameter
    DEPTH = 8)(
  input wire clk_in,
  input wire rst_in,
  input wire deq_in,
  input wire [DATA_WIDTH-1:0] enq_data_in,
  input wire enq_in,
  output logic full_out,
  output logic [DATA_WIDTH-1:0] data_out,
  output logic empty_out,
  output logic valid_out
);
```

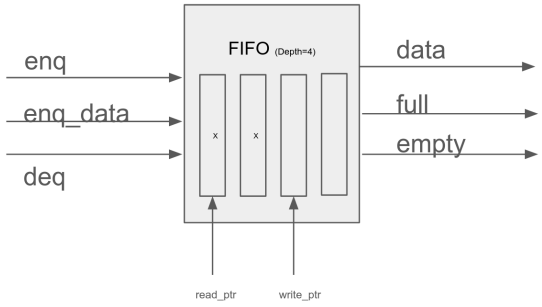


Fig. 2. Core FIFO Implementation

Our FIFO consists of four main components – the queue array, a read pointer, a write pointer, and an array of valid bits. The FIFO is fixed size and is allocated a depth and width at initialization. The read pointer gets incremented upon dequeueing and the write pointer gets incremented upon enqueueing. Pointers wrap around when reaching the maximum depth of the queue, assuming it is not full. The valid bits are used to determine if the queue is full or empty, on the condition that the read and write pointers are at the same location. Such a queue allows us to effectively buffer all inputs and outputs to our system, making pipelining easier.

We created a testbench enqueuing and dequeuing elements from the FIFO to verify that it is working. Enqueing and dequeuing elements takes 1 cycle each.

C. Specialized Priority Queue

Based on our CPU implementation that used a priority queue `S` with a min heap structure and a priority queue `checked` with a max heap structure, we implemented a priority queue that is capable of dequeueing either the maximum or minimum element at any given cycle.

1) Structure: Our modified priority queue [Figure 3] is implemented as a searchable FIFO that functions as a min heap and a max heap. We have a data array, which contains the vertex ID, and a tag array, which contains the corresponding

distances. If we request the smallest element, we dequeue the element with the smallest tag; if we request the largest element, we dequeue the element with the largest tag.

We provide a simple interface for a priority queue that is similar to a FIFO, but allows tagging entries with a distance.

```
module PriorityQueue #(parameter DATA_WIDTH = 32,
    parameter TAG_WIDTH = 32, parameter DEPTH = 8)(
  input wire clk_in,
  input wire rst_in,
  input wire deq_smallest_in,
  input wire deq_largest_in,
  input wire [DATA_WIDTH-1:0] enq_data_in,
  input wire [TAG_WIDTH-1:0] enq_tag_in,
  input wire enq_in,
  output logic full_out,
  output logic [DATA_WIDTH-1:0] data_out,
  output logic [TAG_WIDTH-1:0] tag_out,
  output logic [$clog2(DEPTH):0] size_out,
  output logic empty_out,
  output logic valid_out,
  output logic [TAG_WIDTH-1:0] max_tag_out,
  output logic deq_stall_out
);
```

In order to determine the smallest and largest tags, we start with default values of `32'hFFFFFFFF` for the smallest tag and `32'h0` for the largest tag if the queue is empty. If we enqueue an element, we compare the current minimum and maximum values to the tag `enq_tag_in` of the new element. If `enq_tag_in` is less than the current minimum tag, our new minimum tag is `enq_tag_in`, and similarly if `enq_tag_in` is greater than the current maximum tag, our new maximum tag is `enq_tag_in`. If we dequeue an element, we search through the queue for the new minimum and maximum tags after successfully dequeuing the element. We sequentially search through the queue by iterating through the elements in `DEPTH` cycles and comparing the tags of the elements.

A FIFO is used to maintain the least recently used (LRU) ordering in the cache, so that we can insert elements to only empty slots in the priority queue array(s). The LRU FIFO is largely the same as a regular FIFO but allows synchronous dequeueing (i.e. can read and dequeue in the same cycle) and preinitializing values in order. The preinitialized values correspond to each empty index in the priority queue array. Dequeueing from this FIFO allows a quick identification of empty positions and a way to easily recycle positions, once dequeued from the priority queue. The smallest tag and its associated data are returned on dequeue and its index is freed in the LRU FIFO.

We tested our logic implementation by enqueuing a set of inputs and verifying that the system outputs the value in increasing or decreasing order of tag based on the values of `deq_smallest_in` and `deq_largest_in`. Enqueuing and dequeuing elements take 1 cycle each, and after dequeuing, finding the new minimum and maximum tags takes an additional `DEPTH` cycles.

2) Use as Priority Queue S: The priority queue, S, strictly uses the min-heap style, i.e. only dequeues the smallest element. As such, its interface is restricted to `deq_smallest_in` and its associated ports.

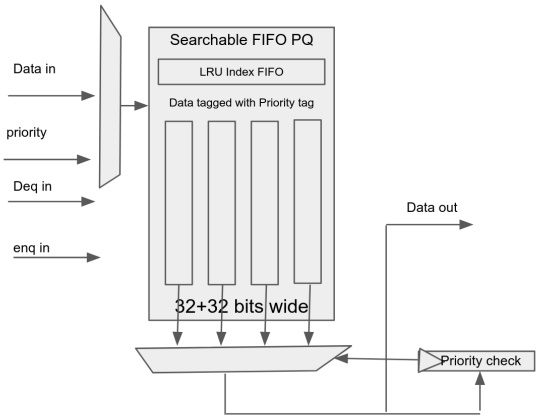


Fig. 3. Implemented Priority Queue using a Searchable FIFO.

3) Use as Checked Queue : The checked queue requires dequeuing both minimum and maximum elements. During the main stages of BFiS, we use `deq_largest_in` when we need to replace the element with the maximum tag for one with a smaller tag. The interface also allows us to read the maximum tag to determine if we should end the algorithm. When we are ready to release the top `k_in` results, we dequeue the k minimum elements in order using `deq_smallest_in`.

D. General Memory Storage

Our Memory storage is two BRAMs right now [Figure 4], containing the graph in compressed sparse row (CSR) format.

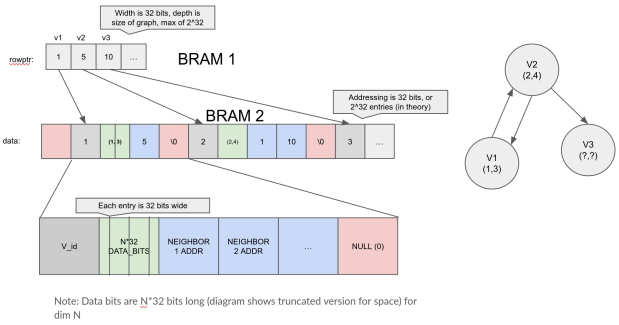


Fig. 4. Memory layout for graph storage

BRAM 1 is a lookup table between vertex IDs and its associated address (ADDR) in the CSR data array. BRAM 2 has the position vector at ADDR+1 to ADDR+1+DIM and the neighbors from ADDR+2+DIM to the next 0 (NULL) value. BRAM 2 is dual port to allow two lookups at the same time.

```
module graph_memory #(parameter DIM = 2, parameter
    PROC_BITS = 4)(
  input wire clk_in,
  input wire rst_in,
  input wire [31+PROC_BITS:0] idx_addr,
  input wire idx_validin,
```

```
  input wire [31+PROC_BITS:0] data_addra_in,
  input wire [31+PROC_BITS:0] data_addrb_in,
  input wire data_validina,
  input wire data_validinb,
  output logic [31:0] rowidx_out,
  output logic [31:0] data_outa,
  output logic [31:0] data_outb,
  output logic [31:0] data_outc,
  output logic data_valid_outa,
  output logic data_valid_outb,
  output logic rowidx_valid_out
  );
```

We created a wrapper around the BRAM to maintain ordering and allow control signals, i.e. valid in and valid out. BRAMs are delayed two cycles, which is identified using control signals for appropriate buffering in a FIFO. The control signals are outputted using a state machine that returns a valid out two cycles after a valid input.

This memory is strictly read only. Currently, we operate on small graphs that can fit in BRAM alone.

To look up the address of some vertex ID, the memory abstraction provides a port to access the rowidx BRAM (single port) with a request for a certain ID. The result two cycles later is the ADDR corresponding to said ID.

E. Visited BRAM

We created a third BRAM to keep track of visited points (using vertex id), as noted in the pseudocode shown in section II (system overview). We store a 0 or 1 at each vertex ID, where a 0 indicates the vertex has not been visited yet and a 1 indicates the vertex has been visited. This module works with the fetching framework – i.e. data for neighbors are only fetched from memory if the vertex has not been visited yet. We mark the first input vertex as visited at the beginning. Since we only send requests to this module if we want to visit a vertex, the resulting data at a given vertex ID will always be `1'b1`. As a result, given a request, we always write in `1'b1` to the BRAM and use the output (previous value in the BRAM) to determine if the vertex has been visited. This module takes 2 cycles to write the value to BRAM.

F. Graph Fetcher

This module fetches [Figure 5], for a given vertex address, all of its neighbors, and each neighbor's position vector of n dimensions, provided that neighbor was not previously visited. Figure 9 in Appendix section XI-A shows a block diagram of our graph fetcher system. As aforementioned, the vertex ID, the vertex location, and the neighbor list are sequentially placed in BRAM 2. Given a certain vertex address (as computed from some vertex ID), we can iterate through the addresses to fetch the position vector for each of neighbors.

This module starts by initializing the first neighbor at vertex `ADDR + DIM + 1` and accesses unvisited neighbors using BRAM 2. For each neighbor vertex address, we look up its associated vertex ID to index into the visited BRAM. If the neighbor is not visited, we continue by fetching its position data at addresses `RESULT(ADDR+DIM+1+N) + 1` to `RESULT(ADDR+DIM+1+N) + 1 + DIM` , where N is the current neighbor.

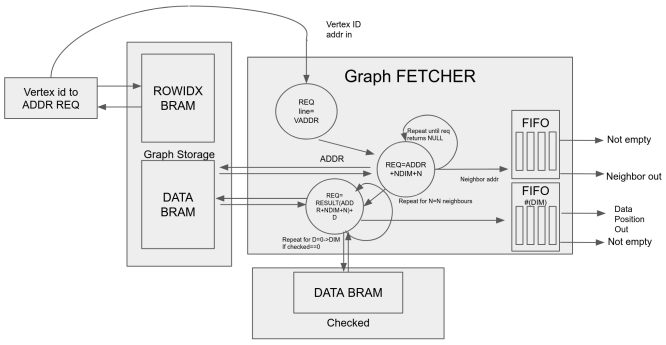


Fig. 5. Graph fetcher state machine.

We increment the request address until we get value 0 (NULL) in the BRAM for the neighbors. All outputs are buffered into two FIFOs (one for position and one for neighbors) to provide an easy access interface for other modules. We add an element to the neighbor FIFO after reading a 0 from the visited BRAM and add data to the position FIFO as we get valid data. Furthermore, the FIFOs allow caching of results when other modules are stalling/processing. The FIFO for the position vector is of size (`DIM` · depth of neighbors FIFO) to store data for all of the points in the neighbors FIFO.

Since the neighbor FIFO is fixed-size, elements are loaded only when space is available. If the FIFO is full before we read all neighbors, we stall the next neighbor BRAM request until an element has been dequeued from the neighbor FIFO. We keep the FIFO size around the average number of neighbors. The neighbor FIFO does not need to be emptied before starting to read the next vertex data since the order the neighbors will be visited remains the same. Note that while it is not depicted in the diagram, for simplicity, each read request includes a delay of 2 cycles for the BRAM to return, triggered by the ready bit on the memory module. FIFO reads, however, are single cycle. The neighbor and position vector FIFOs can dequeue/pop when data is ready/FIFO is not empty.

We tested our graph fetcher using a sample graph of 8 vertices with 4 dimensions each. Each vertex had between 2 and 4 neighbors, and most of the testing was completed with neighbor FIFO size greater than 4. For every neighbor, our system takes 2 cycles to find the address of the neighbor, 2 cycles to find the vertex ID, 2 cycles to check if the neighbor has been visited, and 2 cycles each to request and receive data (5 cycles total since data are requested on consecutive cycles) if the neighbor has not been visited. With a few additional cycles to update request variables and ready bits for requests, this process took 15 cycles for an unvisited neighbor. This value was 6 cycles if the neighbor had been visited. The module also took 4 additional cycles at the end to read 0 as the termination condition.

Our FIFOs were able to correctly dequeue the position vectors and neighbors in the order they were fetched. We also tested the module with neighbor FIFO size 2 on vertices with more than 2 neighbors. If the FIFO was full before all

neighbors could be fetched, the module successfully stalled the BRAM requests until 1 of the neighbor FIFO elements could be dequeued.

G. BFiS

We created a BFiS module to integrate all of the individual modules together. This module is structured based on the CPU implementation of our algorithm and implements the distance calculator, graph memory storage, visited BRAM, graph fetcher, and specialized priority queue modules. Similar to the two priority queues in our CPU implementation, we used two instantiations of the specialized priority queue module: one for priority queue `s` and one to keep track of the top L checked points. The checked priority queue must have a depth of L, while `s` ideally has a depth large enough to store all visited points that have not been checked (future work includes redesigning our module to work with a smaller, fixed-size `s`). We used one instantiation each of all other modules. The connections between the individual modules is shown in Figure 8 in Appendix section XI-A. We provided the vertex ID, query vector, and k as inputs to our module and output the top k vertex IDs sequentially on k different cycles.

```
module bfis #(parameter DIM = 2,
              parameter PQ_LENGTH = 8)(
  input wire clk_in,
  input wire rst_in,
  input wire [31:0] vertex_id_in,
  input wire valid_in,
  input wire [31:0] query_in [DIM-1:0],
  input wire [15:0] k_in,
  output logic [31:0] top_k_out,
  output logic valid_out,
  output logic [2:0] state
  );
```

We structured our BFiS implementation using a state machine with 8 states, as shown in Figure 6.

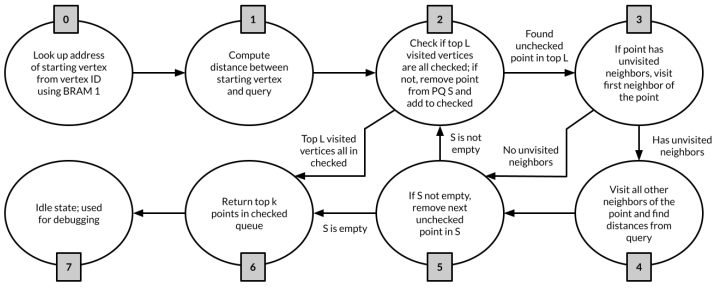


Fig. 6. BFiS finite state machine diagram

1) State 0: In state 0, we wait for the module to receive a valid input vertex ID. If the module receives a valid input, we initiate a lookup in BRAM 1 to find the address (ADDR) of the vertex in BRAM 2. Since our position data for the vertex is located from ADDR+1 to ADDR+DIM in BRAM 2, we can use this address to directly fetch the position data of the vertex. After initiating the lookup at ADDR+1, the module transitions to state 1.

2) State 1: We then input the fetched data one at the time into our distance calculator (section IV-A) to compute the distance between our starting point and query. After receiving a valid output from our distance module, we add our point to priority queue `s` and initiate a dequeue for use in state 2. Since we transition to state 2 whenever we dequeue an element from `s` during the algorithm and use the output from the priority queue in state 2, we chose to add and remove our starting point from `s` rather than just using the point directly.

3) State 2: This state is the start of the outer while loop, as specified by the pseudocode in section II. Once we receive a valid output (vertex v) from `s`, we determine if the top L visited points are all in our checked queue, similar to how we designed the corresponding part in the CPU code (Appendix section XI-B). If our checked queue has not yet reached capacity L, we add v to the queue and move to state 3. If our checked queue is full and v is closer to the query than the farthest point in the checked queue, we first remove the farthest point from the queue before adding v and transitioning to state 3. Otherwise, our L closest visited points are all checked, so we move to state 6 to end our algorithm. While determining which state to move to next, the module runs the graph fetcher on v in parallel.

4) State 3: In this state, we check if v (element we last removed from `s`) has any unvisited neighbors using our result from the graph fetcher. If v has unvisited neighbors, we move to state 4 to visit them. Otherwise, we move to state 5.

5) State 4: This state runs the inner for loop, as specified by the pseudocode in section II. Using position information of the unvisited neighbors from the graph fetcher, for every unvisited neighbor, we find its distance from the query and add the neighbor to `s`. Once all of the distances have been computed and all of the unvisited neighbors have been added to `s`, indicating that they have been visited, the module transitions to state 5.

6) State 5: Once the inner loop has been completed, we want to dequeue the next element in `s`, if possible. If `s` does not have any elements left, we can move to state 6 and end the algorithm since all visited points have been checked. Otherwise, we dequeue the next element in `s` and return to state 2 to run our condition for adding the point to the checked queue.

7) State 6: This state indicates the end of the algorithm. We dequeue the k elements in checked with the smallest tag (our approximation of the k closest points to the query) one at a time. Since our specialized priority queue module takes L cycles to find the new closest point to the query after dequeuing an element, we stall subsequent dequeues by L cycles after receiving a valid checked output. In addition, since we want to return the vertex ID of the approximate k closest points, we look up the vertex ID of the point using BRAM 2 in parallel. After all k vertex IDs have been outputted, we move to state 7.

8) State 7: This state indicates that the module has completed running the algorithm and the approximate closest k points have been returned. State 7 was mostly used for

debugging on the FPGA to verify that the module was running through the states as expected.

We created a custom testbench for simulation and tested this module on a graph with 8 vertices of 4 dimensions each to verify that the algorithm was working as expected.

H. UART Input and Output

We use a stream over the UART module Manta to produce the appropriate inputs for initializing the BFiS system [3]. The input order is as follows:

1) each `query_in` vector component (DIM 32-bit values)
2) `k_in` value
3) starting vertex `v_id` value

Each element is padded by `32'hFFFFFFFF`, or -1. This value ensures that we can easily differentiate values in the stream. Once the appropriate number of values are recieved, BFiS is started.

The outputs are dumped over UART once BFiS completes. Each value of `top_k_out` is buffered into a FIFO and outputted when Manta is ready (since Manta does not poll every cycle). The cycle count from initialization is also outputted for evaluation.

V. Evaluation of Performance on FPGA

We tested our system on three of the same datasets we used to evaluate the CPU implementation:

1) Tester with 4 dimensional vectors
2) Tester with 8 dimensional vectors
3) Citeseer with 4 dimensional vectors

As mentioned in section III (CPU implementation), tester is a very small dataset with just 8 vertices, and citeseer is a large dataset with 3312 points. The position vectors are generated at random to encompass a space of 10000^{DIM} discrete points.

We evaluated performance using the number of cycles taken to complete a full BFiS run. Each cycle takes 10ns on this FPGA, which runs at 100 MHz. All logic on our FPGA was designed to meet this timing requirement to maintain cycle accuracy.

All results are in Table III.

TABLE III
Performance of BFiS on an FPGA

Dataset	Checked PQ Size	Cycles (10^{-8}s)	Speedup
tester_4	5	554	9.57
tester_8	5	614	9.12
citeseer_4	5	1541	7.33
citeseer_4	8	1946	5.96
citeseer_4	10	2536	4.42

Our results from the FPGA matched our results from the CPU implementation, verifying accuracy of our system. In addition, from these preliminary tests, our timing results were significantly better than the runtime of the CPU implementation. We do understand that there is overhead with Linux and C++ libraries that may add some base delay, but our results are promising.

Our design meets all timing requirements (i.e. pipelined appropriately to match that 10ns clock cycle). Lookup tables (LUT) and slices are the most used resources, according to our Vivado logs. These resources are used proportionally to the graph size. For instance, on our larger graph, we used 60% of Slice LUTs and 3.33% of Block RAM tiles. Our max delay is the graph memory bandwidth with a slack of 1.5ns. This is similar to modern machines, where memory bandwidth is a main limitation for performance. Still, the timing is well under the required value.

While we were able to fit our reference graphs into BRAM, much larger graphs are indeed 100s of megabytes and would require a different, unified, memory architecture, based on DRAM.

However, for the score of this project, our system meets all requirements for the resources provided on the Urbana boards with a Xilinx Spartan-7-50 chip.

VI. Conclusion

We were able to meet many of the requirements we set forth in our project checklist. Our system effectively implements the graph-based vector search algorithm Best First Search on a Xilinx FPGA for small graphs that can fit in BRAM. We also successfully tested and evaluated the FPGA implementation for accuracy and speed compared to a sequential CPU implementation. Our FPGA and CPU implementations achieved the same accuracy, as expected, while the FPGA implementation ran in similar time to the CPU implementation on a graph of 8 vertices using dimensions of 4 and 8, respectively, and 7.34x, 5.96x, and 4.42x faster than the CPU implementation on a graph of 3312 vertices using PQ capacities of 5, 8, and 10, respectively. We plan on improving this design in the future by adding a DRAM for larger graphs and further parallelizing to increase efficiency.

VII. Future Work

A. Large Graph Memory

Ideally, large graphs can be stored in DRAM, but we were unable to implement DRAM that as part of this project. Currently, we use graphs can fit in BRAM (less than a few kB) in size. Once we start working with larger graphs, we plan to have the BRAM used as a standard cache [Figure 7] with the same interface that we currently use for requesting data.

We have implemented the structure and general logic of a 4-way direct mapped cache that handles read requests and generates requests to a higher level of memory. We have decided to wait until we implement a DRAM controller to use it. Further work would need a larger board as our 'larger' graph already uses 60% of the board's resources. This cache is a BRAM with two ports – one for reading, and one for writing missed lines that are fetched from DRAM (which would store 32 byte words in 4 words per line segments). We do not deal with dirty lines because our memory is read only.

We did not focus on building a SPI or UART replacement for DRAM since the graphs that we wanted to test fit in BRAM without exhausting the FPGA resources.

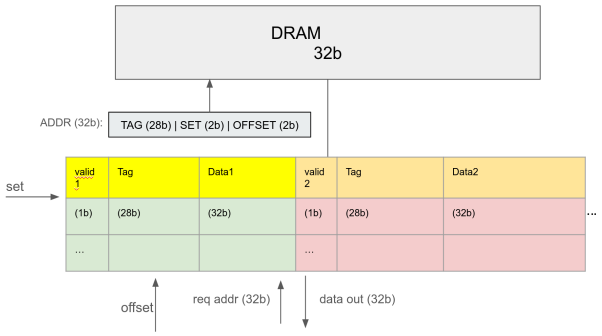


Fig. 7. DRAM based memory connected to a 2 port BRAM based cache. Only 2/4 ways are shown for space.

Some work that needs to be done for large graphs to work effectively include making a shared memory for vertex ids and graph CSR data (and possible checked status), since we only have a single DRAM bank. We can instantiate multiple caches to provide the same programming interface as we currently do. DRAM would be populated over UART or SPI.

B. Random Point Sampler

Currently, we choose the starting point as a value determined by the user (that can be chosen at random by the user python program if desired). We will ideally use a pseudo-random sampler to find a better point closer to the destination (hard coded value).

At the moment, we see this module selecting a small random subset of vertices (possibly using a linear-feedback shift register (LFSR)) from BRAM 1, computing the distances between each vertex and the query, and using the one with the smallest distance as the starting point. Choosing a starting point in this way would likely allow the algorithm to finish running in less iterations.

VIII. Retrospective

A. Redesigning modules

As we were integrating the system together, we realized that we could increase efficiency of our overall system by redesigning some of our modules.

As mentioned in section IV-A, we originally designed our distance calculator to use a recursive adder module but later realized that parallelizing the addition with the subtraction and multiplication was more efficient (and much simpler to implement).

In addition, we initially designed our graph fetcher to fetch the data and neighbors for a given vertex rather than the neighbors and data for the neighbors. However, we realized that whenever we needed to fetch the neighbors of a vertex, we also needed the data of the neighbors to calculate the distance between the neighbors and the query. As a result, our previous design would have required some redundant fetching and would have been harder to integrate with our overall system. We also redesigned our graph fetcher to interface with our visited BRAM module rather than following our initial plan of checking the visited condition directly in the BFiS module. This new design saved us additional cycles from fetching data of already visited points.

B. Simulation to synthesis on FPGA

While trying to synthesize our system on the FPGA, we realized that running in simulation alone was not adequate enough to test our system since physical hardware acts differently in edge cases. When we first ran our system on the FPGA, we had timing violations in our distance and specialized priority queue modules. Simulation does not measure timing violations despite using 10ns clock cycles. We had to pipeline more, separating large amounts of combinational logic over multiple cycles, to ensure timing constraints were kept after synthesizing on the FPGA.

After fixing the timing constraints, our FPGA no longer outputted the correct results. Since we had thoroughly tested each component in simulation, including integration of the whole system, we had to fully debug on the FPGA, which can be much harder than it looks. We ended up trying to isolate the problem by using only the LEDs on the FPGA and Manta to output signals for each component. After several days of redesigning modules and debugging, we narrowed down our incorrect results on the FPGA to blocking code and not all modules being reset when the FPGA is flashed.

Since we mixed a lot of blocking (combinational) and non-blocking (sequential) code, we ran into issues on the board where signals might have differed by less than a clock cycle. In some cases, we wanted values to be read same-cycle, hence the combinational logic, but we had to convert some of these signals to using sequential logic for the accuracy of our system. In addition, in some of our modules, we set variables to nonzero values at a system reset. However, the reset conditional didn't actually run for one of the modules, so we had to set the values outside of the reset conditional. This issue is definitely something to keep in mind for future FPGA-related projects.

C. Time management

Our time management was good for the most part, and the timeline we created in October helped us plan what to design next. We also aimed to finish reports and other deliverables ahead of the actual deadlines, allowing us enough time for revisions.

However, since we assumed that a working system on the FPGA would follow directly from a working simulation, we didn't dedicate any time for synthesis on our timeline and rushed to get our system working during the second-to-last week. We also did not account for time needed to redesign some of our modules.

IX. Code Repository

Our code temporarily can be found here.

Since this project is part of ongoing research, our repository is not public at this time but will likely be posted publicly for

use in the future. In the future, our code can be found at https://github.com/sanjayseshan/6.111_final_project.

X. Individual Contributions

A. Code

We both worked on code for the distance calculator, graph fetcher, specialized priority queue, and BFiS modules. Lasya wrote the CPU implementation of BFiS and the module for reading from and writing to the visited BRAM. Sanjay wrote the FIFO and graph memory modules.

B. Testing and Evaluation

Lasya tested the graph fetcher, specialized priority queue, and BFiS modules in simulation and evaluated the CPU implementation. Sanjay tested the FIFO module in simulation, fixed timing issues in our priority queue and distance calculator modules, and evaluated our design on the FPGA. Both of us tested the distance calculator in simulation and debugged issues going from simulation to synthesis on the FPGA.

Acknowledgment

We would like to thank Joe Steinmeyer and the rest of the 6.205 staff for their support and supervision during the course of this project's development. We would also like to thank Dr. Xuhao Chen and Prof. Arvind Mithal in the CSAIL Computation Structures Group who sponsored this project.

References

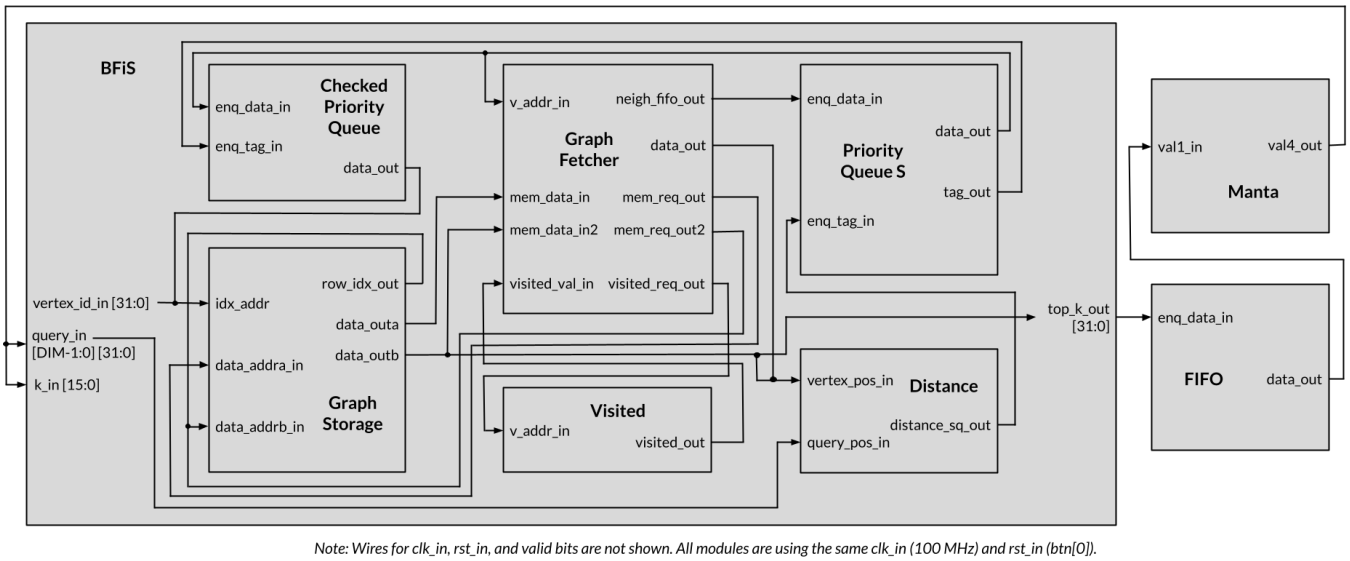
[1] W. Jiang, S. Li, Y. Zhu, J. d. F. Licht, Z. He, R. Shi, C. Renggli, S. Zhang, T. Rekatsinas, T. Hoefler *et al.*, "Co-design hardware and algorithm for vector search," *arXiv preprint arXiv:2306.11182*, 2023.

[2] Z. Peng, M. Zhang, K. Li, R. Jin, and B. Ren, "iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 313–328.

[3] F. Moseley, "Manta: An in-situ debugging tool for programmable hardware," M. Eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2023.

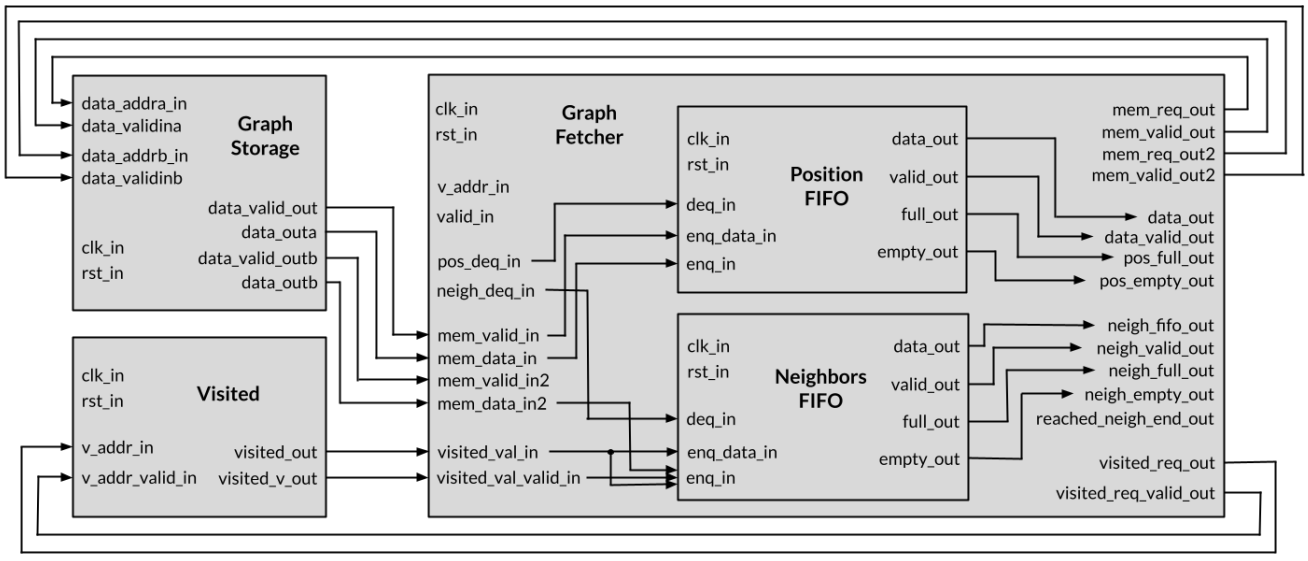
XI. APPENDIX

A. Block Diagrams of Module Connections



Note: Wires for clk_in, rst_in, and valid bits are not shown. All modules are using the same clk_in (100 MHz) and rst_in (btn[0]).

Fig. 8. Higher level block diagram for entire system



Note: Wires for clk_in and rst_in are not shown. All modules are using the same clk_in (100 MHz) and rst_in (btn[0]).

Fig. 9. Block diagram for graph fetcher

B. C++ Code for CPU Implementing BFiS

```cpp
#include <tuple>
#include <queue>
#include "ann.h"
#include "utils.hh"
#include "common.hpp"

void kNN_search(int K, int qsize, int dim, size_t dsize,
                const float *queries, const float *points, vid_t *results, Graph &g) {

  // find K closest points for each query
  for (int query_id=0; query_id<qsize; query_id++) {

      const float* query = queries + query_id * dim; // define query
      int L = 10*K; // queue capacity L
      int visited [dsize] = { 0 }; // create visited list


      // priority queues
      std::priority_queue<tuple<float,int>, vector<tuple<float, int>>, greater<tuple<float, int>>> S;
      std::priority_queue<tuple<float,int>> checked;


      // index of starting point
      int index = 0;

      // distance calculation
      float dist = 0.0;
      for(int i=0; i< dim; i++){
        dist = dist + pow((points[i]-query[i]), 2);
      }


      // add starting point to S
      S.push(make_tuple(dist, index));
      visited[index] = 1;


      while (!S.empty()) {
        // first unchecked point
        vidType v = get<1>(S.top());

        // marked v_i as checked if less than L points checked or dist less than max checked dist
        if (checked.size() < L || get<0>(S.top()) < get<0>(checked.top())) {
          checked.push(S.top());

          // resize based on L
          if (checked.size()>L) checked.pop();
        }

        // otherwise, if size is L, exit
        else {
          if(checked.size()==L) break;
        }
        S.pop();


        // iterate through neighbors
        for(vidType u:g.N(v)) {
          if (visited[u] == 0) {
            visited[u] = 1;


            // distance calculation
            const float* point = points + u*dim;
            dist = 0.0;
            for(int i=0; i< dim; i++){
              dist += pow(point[i]-query[i], 2);
            }


            // add point to S
            S.push(make_tuple(dist, u));
```

```
            }
          }
        }

        // remove elements not in top K
        while (checked.size() > K) {
          checked.pop();
        }

        // insert values in results
        int checked_length = checked.size();
        for (int i=0; (i<K && i<checked_length); i++) {
          results[query_id * K + (K-i-1)] = get<1>(checked.top());
          checked.pop();
        }
      }
}
```