Hardware Realization of Nearest Neighbour Search Algorithm over an In-Memory Pre-Stored k-d Tree

Aleksandar Z. Kondić, *Student Member, IEEE*, and Vladimir M. Milovanović, *Senior Member, IEEE*

***Abstract*—Nearest neighbour search is a fundamental statistical classification algorithm with widespread use in artificial intelligence (AI) sub-fields such as machine learning, computer vision, and robotics. Considering the shift in host platforms running AI algorithms from general-purpose computers to specialized hardware implementations, a parameterizable design generator of special purpose hardware instances that perform nearest neighbour search is proposed, captured inside Chisel hardware construction language, and validated on an FPGA platform. Based on an algorithm of nearest neighbour search that traverses a k-dimensional tree pre-stored inside read-only memory (ROM), the generator provides parameters for configuring the structure and volume of the tree and the points stored within it.**

***Index Terms*—Nearest neighbour search, hardware implementation, Chisel hardware construction language, k-dimensional tree.**

I. Introduction

Nearest neighbour search is an algorithm which, for a given input point, finds a point closest to it among a set of points. It is useful for solving classification problems, which are especially prevalent in artificial intelligence (AI) subfields such as machine learning, computer vision [1], and robotics [2].

With artificial intelligence algorithms being increasingly shifted from general-purpose computers to dedicated hardware instances as a consequence of the need for increased computational power [3], various hardware implementations of classic AI algorithms targeting different platforms have appeared. The Nearest Neighbour Search (NNS), along with its variants, the Approximate Nearest Neighbour (ANN) and k-Nearest Neighbours (k-NN) algorithms, are no exceptions.

Considering field programmable gate arrays (FPGAs) as a hardware implementation platform of choice, there are various incarnations of the above mentioned algorithms. They are usually described and implemented either in the form of pure register-transfer level (RTL) [4], [5], high-level synthesis (HLS) [6], [7], or Open Computing Language (OpenCL) [8], [9] code. An alternative to these approaches is to write the behavioral code in an RTL-like form but utilizing a higher level hardware design language instead. One such language is Chisel [10], which is embedded in the Scala programming language, enabling its users to write RTL instance generators while providing benefits of both functional and object-oriented programming paradigms. This paper proposes an implementation of the nearest neighbour search algorithm in Chisel using an agile [11] digital design methodology.

Aleksandar Z. Kondić was with the Faculty of Engineering, University of Kragujevac, Sestre Janjić 6, Kragujevac, Serbia (e-mail: konda@uni.kg.ac.rs).

Vladimir M. Milovanović is with the Department of Electrical Engineering, Faculty of Engineering, University of Kragujevac, Sestre Janjić 6, 34000 Kragujevac, Serbia (e-mail: vlada@kg.ac.rs).

An effective implementation of the nearest neighbour search algorithm should presumably work for a large number of predefined points as potential output points for a given input point. For such an implementation to be efficient in terms of resource utilization for a target hardware platform such as FPGA, the points need to be stored inside a memory module. This naturally implies that the digital logic may have access only to a limited number of pre-stored points per clock cycle. Therefore, it is desirable to minimize the number of memory accesses for a given input point while obtaining the correct solution.

This is the same problem that a purely software implementation of a nearest neighbour search algorithm on a processor would have. To minimize the amount of time needed to process an input point, an efficient algorithm with a desirable run-time complexity needs to be chosen. While the simplest solution would be to run an exhaustive search of the entire memory containing pre-defined points to find a point with the minimal distance from the input point—yielding a linear run-time complexity—more efficient algorithms exist.

Similar problems were encountered in the field of computer graphics. In order to ensure the rendering of a scene in a timely manner, it was necessary to retrieve relevant spatial data of the scene efficiently. A technique named *binary space partitioning* (BSP) was developed to solve this problem, mainly implemented through a tree data structure [12]. The technique entails recursively subdividing space into two parts along a hyperplane. When a given point or polygon is queried, the search is performed only in the sub-spaces where it could possibly be located, thus reducing the search domain.

Space partitioning is a general method of subdividing space in a defined manner until a certain condition is satisfied. There are multiple implementations of this method in the form of different tree structures with specific criteria on how a space is divided into sub-spaces and under which conditions. Examples of some tree structures that perform space partitioning are k-d trees, quadtrees, and octrees. Concerning the nearest neighbour search problem, some of the appropriate data structures that can be used are R-trees and k-dimensional trees.

The principal data structure driving this particular implementation of the nearest neighbour search algorithm is the k-dimensional tree, or k-d tree for short. A k-d tree is essentially a binary search tree that contains multi-dimensional points and is traversed based on the value of one of the coordinates of the input point at each node in the tree.

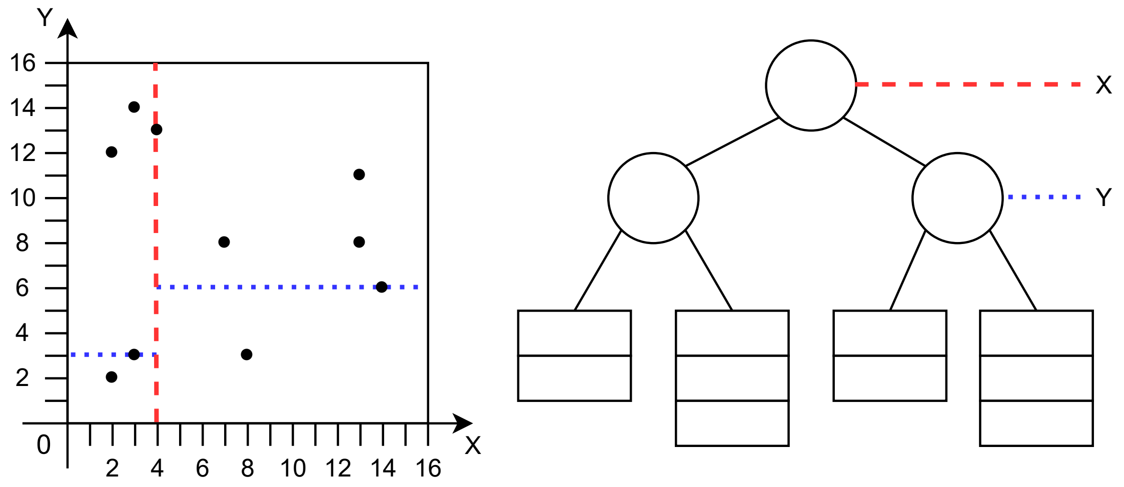

Fig. 1. An illustrative example of a k-dimensional tree (with $k = 2$ to simplify the drawing) and the two-dimensional space partitioning it performs.

Each node of the k-d tree contains a value by which the hyperspace it belongs to is split into two. The dimension in which the split is performed corresponds to the node's depth in the tree, which repetitively cycles from the last dimension to the first when the depth of the node becomes greater than the dimensionality of the points stored inside the tree. All child points that have the value of the coordinate in the corresponding dimension less than the node's stored value are part of the left sub-tree, while the child points with the corresponding coordinate's value greater than the value in the node are part of the right sub-tree. In the case of a child point having an equal corresponding coordinate value to the value of the node—due to the nature of the nearest neighbour search algorithm—it may belong to either of the sub-trees.

The average run-time complexity of the nearest neighbour search algorithm over a k-dimensional tree is $\mathcal{O}(m + \log_2 n)$, where n is the number of nodes in the tree and m is the average number of points contained in a leaf node.

II. A k-d Tree-Based Hardware Implementation

The primary purpose and the use scenario of the proposed implementation is to execute the nearest neighbour search algorithm over a k-dimensional tree. The tree structure, along with the points it contains is assumed to be constructed and stored beforehand inside some form of a read-only memory (ROM). In the case of an FPGA platform the ROM is in the form of a single-port block RAM and mimics the static RAM.

This implementation uses a variant of the k-dimensional tree in which the number of nodes in the tree is not necessarily equal to the number of points. The points are, after a proper traversal through the k-d tree, stored in the leaf nodes. A leaf node may contain more than one point. An example k-dimensional tree of this kind is shown in Fig. 1, along with an illustration of how the tree partitions a two-dimensional space (but in general it can be an arbitrary k-dimensional hyperspace).

The nearest neighbour search algorithm finds the closest point to the query point by first performing a traversal of the k-d tree until reaching a leaf node. When visiting a node, the value of the query point's coordinate in the dimension corresponding to the node's depth is checked against the value in the node. If the value is smaller, traversal proceeds to the left sub-tree. Otherwise, traversal proceeds to the right sub-tree. When reaching a leaf node, all of the points in the leaf node are checked, calculating the distances between them and the query point. The current closest point, along with its distance to the query point, are stored inside dedicated registers which are updated when a closer point is found.

After exhausting all of the points in a leaf node, the search algorithm traverses backwards, that is up the tree and checks if the hypersphere around the current closest point with the radius equal to its distance from the query point intersects the node's splitting hyperplane. If so, a closer point to the query point may exist on the other side of the splitting hyperplane, so the search algorithm proceeds by traversing down the sub-tree contained in the node's unvisited child, until reaching a leaf node again. This process is repeated until the algorithm terminates when it is guaranteed to yield a point stored within the k-dimensional tree with the minimal distance from the query point.

For the purposes of this work, the structure of the k-dimensional tree and the points it contains are stored in two separate memories (or two non-overlapping memory segments). The memory used to store information about the points contains coordinates of each point. The points inside this particular memory (segment) are arranged in such a way that the points belonging to the same leaf node of the k-dimensional tree occupy consecutive memory locations.

Memory containing the tree structure stores the properties of each node. The following properties are stored: an indicator bit of whether the node is a leaf node, the discriminating value stored inside the node for tree traversal (valid only for non-leaf nodes), the starting address in the points ROM and the

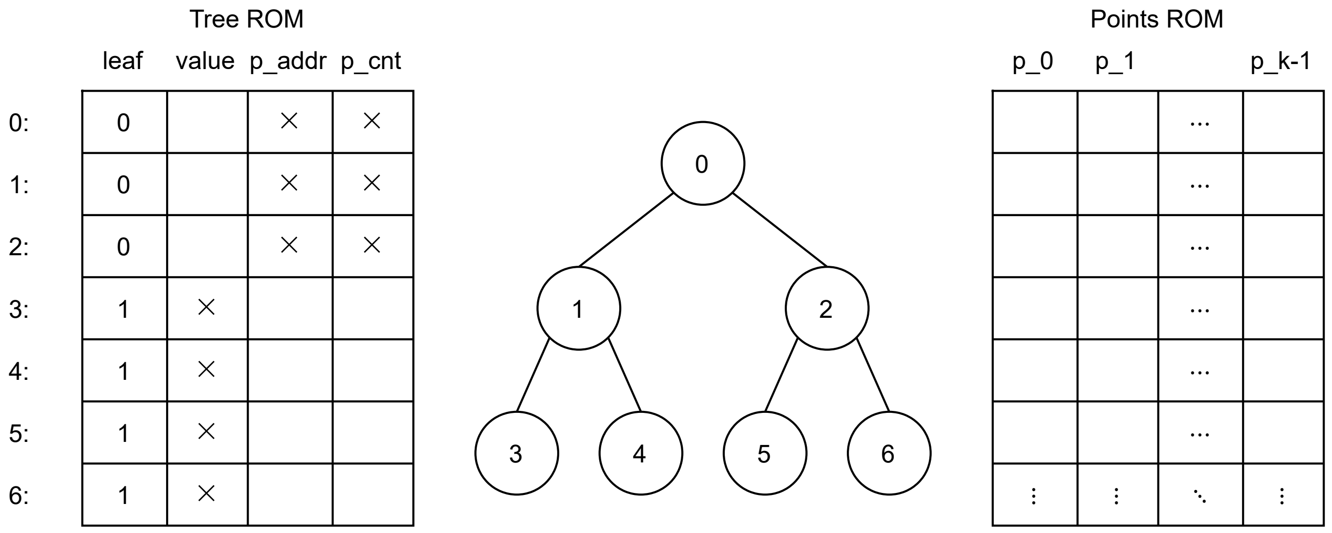

Fig. 2. Memory layout of a k-dimensional tree showing the associated Tree ROM and Points ROM structures over which the NNS realization operates.

number of points contained in the node (valid only for leaf nodes). A node at location n in the tree ROM has its left and right children at locations $2n + 1$ and $2n + 2$, respectively. With this it is assumed that the stored k-d tree is balanced. It is possible to construct a balanced k-d tree from an arbitrary set of points as long as the points with the same coordinate value as the discriminating value in the node may be stored in either of the node's child sub-trees. In particular use cases of interest this discriminating value is actually the median value of relevant coordinates of the points being considered during k-d tree construction. The described memory layout is illustrated in Fig. 2.

The tree traversal algorithm is recursive. Traversal is performed in a depth-first manner, that is similar to the depth-first search algorithm (DFS), which is also recursive. The DFS algorithm, starting at the root of the tree, visits its child nodes in a pre-defined order. When visiting one of the child nodes, another instance of the DFS algorithm is started on the node, running more instances of the DFS algorithm on its children if it has any. Once an instance of the DFS algorithm for one child node terminates, the same process is repeated for the other. Therefore, by the time DFS starts visiting the root node's second child, the entirety of the sub-tree rooted in its first child will have already been explored.

An example of the order of traversal of binary tree nodes in the depth-first search algorithm is shown in Fig. 3. Non-leaf tree nodes are each visited a total of three times in order to visit the subtree rooted in their second child after visiting the first, and to potentially traverse back up to the parent node.

Each non-leaf node's left child is first explored, followed by the right child. The primary characteristic of DFS is that after visiting the leaves, it traverses back up the tree in order to traverse down unvisited sub-trees, repeating this process until the entire tree is explored.

A k-d tree traversal is essentially a variation on DFS tree traversal. The difference with k-d tree traversal is that the order of the children visited depends on the query point for which the closest point is to be found. The left child is first visited if the query point is on the "left" side of the splitting hyperplane represented by the node, otherwise the right child is first visited. Also, if the first child node's closest point is at a distance shorter than or equal to the distance of the query point from the splitting hyperplane, the second child is not explored. Unlike depth-first search, with k-d tree search the entire tree may not necessarily be explored.

A tree traversal over an example k-d tree is illustrated in Fig. 4. The query point for which to find the closest point is $(5, 3)$. In this example, during the traversal three out of the four leaf nodes were visited. The metric used to calculate the distance between two points (or between the query point and a splitting hyperplane) is the squared Euclidean distance.

Software implementations of recursive algorithms may make use of recursive function calls, which are realized on a call stack, or allocate a stack structure specifically to store their data and implement the algorithm as a non-recursive function. In this case only the second option is viable, so the stack data structure is actually implemented as an array of registers. A separate dedicated register is used to store a pointer that keeps track of the position of the top of the stack in the array.

III. Design Generator of the k-d Tree-Based NNS

The previously described accelerator has been implemented as a parameterized RTL design generator in Chisel 3 hardware construction language. The generator has been extensively tested by following standard Chisel verification and implementation paths for FPGA design workflows. As a hardware library it is freely available for public use [13]. The next few paragraphs are elaborating on different generator parameters, as well as modes of operation of the module.

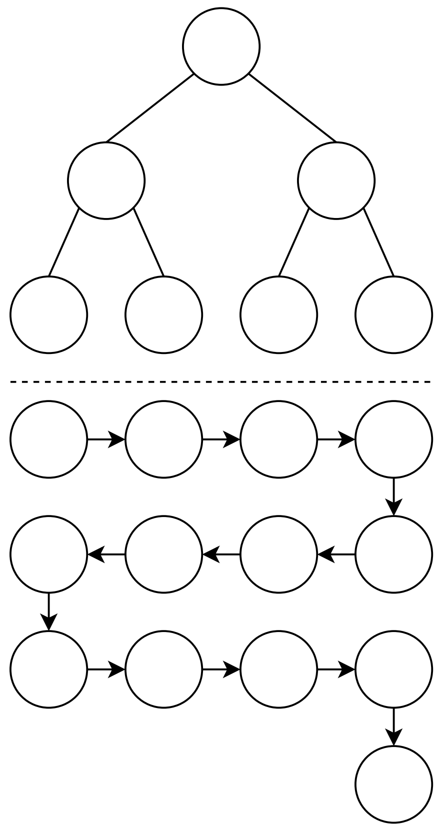

Fig. 3. An example of the order of nodes traversed in a binary tree using depth-first search.

A. Generator Parameters

Parameterizable properties of the design pertain mainly to the structure of the k-dimensional tree itself and its points.

The values of point coordinates are signed integers with a specified bit width, which is one of the parameters of the design generator. Another generator parameter is the number of dimensions of each point. The total size of the points ROM is inferred from the bit width of its unsigned integer addresses, which is specified as a yet another generator parameter. These three parameters make up the structure of the points ROM.

Concerning the structure of the tree ROM, each location contains one bit indicating whether the node is a leaf, a signed integer representing the discriminator value of a node (in our use cases referred to as the *median*), and two unsigned integers representing the location and count of points inside the points ROM. The bit width of the discriminator is the same as the bit width of the points' individual coordinates, while the bit width of the location and count of points is the same as the bit width of the addresses in the points ROM. The size of the tree ROM, along with the bit width of its addresses is inferred from a generator parameter specifying the maximum depth of the tree. The number of nodes in the tree may be arbitrary though, as the nearest neighbour search algorithm assumes that nodes marked as leaves in the tree ROM do not have children.

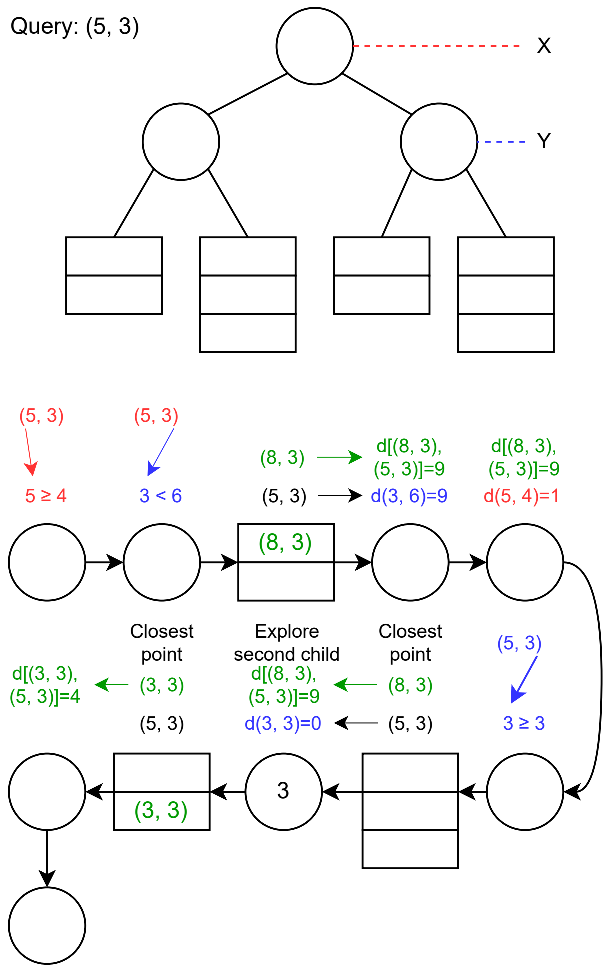

Fig. 4. An example of the order of nodes traversed in a k-d tree when finding the closest point to the query point $(5, 3)$.

B. Modes of Operation

To keep track of the nodes visited for the purposes of traversing back up the tree after visiting a leaf node, a stack structure is implemented as an array of registers. Three distinct values are pushed onto the top of the stack to aid with the execution of k-d tree search:

- *Address of the node in tree ROM* – this is the main piece of data used to keep track which tree node to visit next.
- *The child of the node to visit next* – a single bit that determines whether to visit the first or the second child of the node during tree traversal. A value of 0 corresponds to visiting the first child, while a value of 1 corresponds to visiting the second child. When the value is 1, first the distance of the current closest point to the query point is compared to the distance of the query point to the splitting hyperplane of the node (the median value of the node in this case). If the distance of the closest point to the query point is not greater, the second child is not visited.
  Since after visiting the second child of a node there is nothing left to process, the current node visited is popped from the stack before the second child node is pushed

onto the stack. More precisely, the value at the top of the stack, which currently contains data about the current node being visited, is simply replaced with the data of its second child. In case the second child node is not to be visited, the current node is popped from the stack.

- *The depth of the node in the tree* – This information is used for calculating the distance of the current closest point from the splitting hyperplane. The value of the depth directly maps to which coordinate of the current closest point to compare to the stored node median and is also used to determine the coordinate of the query point to compare to the median value during tree traversal.
  While the depth of the node can be calculated from its index (memory address) in the tree ROM, it is simpler to just push onto the stack the current depth value of the node incremented by one when pushing child nodes. The depth value does not exceed the dimensionality of the points in the tree as it cycles between 0 and $k - 1$.

Since the first node to process when a new query point is given is always the root of the tree (address 0 in tree ROM), whose depth is 0, and the next node to process is always its first child, the stack is always initialized to contain these values as the sole element of the stack before processing a new point.

At each clock cycle, the values at the top of the stack are retrieved, which mostly determine the mode of operation of the module. The relevant modes of operation are as follows:

- *Leaf processing* – this mode is active when the indicator for whether the current node is a leaf has the value 1. The values at the top of the stack are not used in this case. At each clock cycle, a counter indicating how many points were visited in the points ROM is incremented until reaching the value in tree ROM that indicates how many points a leaf node has. After that, the counter is reset to 0 and the current node is popped off the stack.
  The value of the counter is added to the starting address of the node's points in the points ROM, yielding an address of each point to be retrieved from the points ROM. The distance of each point is compared to the current minimal distance from the query point. In case it is smaller, both registers containing the closest point and its distance from the query point are updated accordingly.
  Since there is a delay of a few clock cycles due to memory access in the points ROM and distance calculations using registers to decrease the length of logic paths, once the points counter is set to 0, a "delay" counter is also initialized to 2. Each clock cycle the value of this counter is decremented until it reaches 0, regardless of the mode of operation. The module may not produce a valid result on its output while the value of this counter is greater than 0, even if there are no remaining nodes left in the tree to process for a given query point.
- *Tree traversal, first child node being next* – This mode is active when the current node is not a leaf (explained above) and the value of the child indicator at the top of the stack is 0. The appropriate query point coordinate is compared to the median value of the node to determine which of the left and right children is the first child to be visited. After that, the corresponding first child node is pushed onto the stack.
- *Tree traversal, second child node potentially being next* – This mode is active when the current node is not a leaf and the value of the child indicator is 1. The stored current minimal distance from the query point is compared to the distance of the query point's appropriate coordinate from the median value of the node. If the current minimal distance is not greater, the second child of the node will not be visited, and the current node is popped off the stack. Otherwise, values at the top of the stack are replaced with the values corresponding to the second child node.
  Since the median distance calculation also uses a register in order to decrease the length of logic paths, the result of this calculation is available in the next clock cycle. Therefore, a special one-bit register is set to signify that this mode of operation is still in progress. In the next clock cycle this register is reset, and the rest of the operations are performed as described.
- *Final phase of the algorithm* – active when the node stack is empty. While the value of the previously described "delay" counter is greater than 0, no additional operations are performed. Once the value of the counter reaches 0, the output valid signal of the module is set to 1, while the output point is simply a set of wires connected to the register storing the current closest point to the input point. The initial values of the stack are pushed onto the empty stack to prepare the processing of the next input point.

The block diagram in Fig. 5 depicts the generator's design. The inputs and outputs of this design adhere to the Ready/Valid handshaking protocol. Apart from the *ready* and *valid* signals, both the input and the output consist of a single k-dimensional point represented as a series of k signed integers depicting their respective coordinate values. The dimensionality of these points, along with the structure of the point and tree ROMs, depend on the generator's parameters.

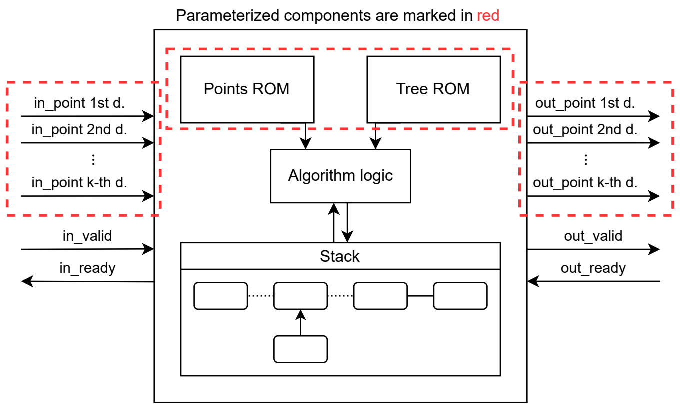

Fig. 5. Interface of the implemented Chisel design showing its input, output, and internals.

IV. Implementation and Testing Results

Testing of the generator is also performed using testing facilities provided by Chisel, i.e. *ChiselTest*. All generator parameters are randomized during testing, and the ROMs are also populated by appropriate randomly generated k-dimensional trees. The output of generated instances for random inputs is compared against the output of a k-d tree golden model written in Scala. The distances of the outputs of Chisel instances are compared with the distances of the outputs produced by the respective Scala golden model class instances.

The Scala golden model of the k-dimensional tree has also undergone rigorous testing. A list of random points is generated after selecting a random number of dimensions for the points. From the list of points and the desired number of tree nodes an instance of the golden model class is created.

This “golden model” instance is then supplied with random points as input. The output point’s distance from the input point is compared to the distance of the closest point to the input point from the list of points in the tree, which is obtained by applying a simple brute-force exhaustive search algorithm. Due to the order of nodes and points traversed not being the same for the k-d tree model and the brute-force algorithm, only distances of the respective closest points to the input point are compared.

For additional testing and real in-hardware validation, various instances obtained from the design generator have been synthesized and implemented onto a commercially available FPGA development board. The board in question is Digilent’s Arty A7 with Xilinx’s Artix-7 FPGA family. All instances have been synthesized for a 100 MHz target clock frequency.

Resource utilization for the different generated instances is shown in Table I. Slice LUT utilization is most influenced by the bit width of the coordinates and k, the dimensionality of the points. A more minor effect on slice LUT utilization is exerted by the sizes of the point and tree ROMs. The number of slice registers seems to be mostly influenced by the bit width of the coordinates, followed by the dimensinoality of the points. Number of dimensions k has an influence on both Block RAM Tile and DSP multiplier counts, although the greatest influence on the number of DSP multipliers is exerted by point coordinate data bit width.

TABLE I
FPGA Resource Utilization for Generated Design Instances

Generator Instance Parameters				FPGA Resources			
Data Width	Nodes	Points	k	Slice LUTs	Slice Regs	BRAM Tiles	DSP muls
8 bits	31	100	3	728	272	1	3
16 bits	31	100	3	383	288	1.5	7
24 bits	31	100	3	528	444	1.5	11
32 bits	31	100	3	706	412	1	19
16 bits	7	100	3	334	287	1.5	7
16 bits	15	100	3	328	288	1.5	7
16 bits	63	100	3	339	288	1.5	7
16 bits	31	50	3	326	271	1.5	7
16 bits	31	200	3	372	311	1.5	6
16 bits	31	100	2	339	232	1	6
16 bits	31	100	4	381	337	1.5	8
16 bits	31	100	5	453	393	2	9

V. Conclusion

An approach to implementing a nearest neighbour search algorithm on an FPGA hardware platform has been explored. One of the key characteristics in this approach is in using a pre-stored k-dimensional tree to perform nearest neighbour search operations. Another is in using the Chisel hardware design language to create a generator of instances that can accommodate k-d trees with different structure parameters.

A variety of instances have undergone testing and additional verification by implementing them on a commercial FPGA development board. Apart from testing and verification, some consideration has also been given to their utilization of resources. This paper proves on an example case of the nearest neighbour search algorithm that parameterizable design generators can be used to produce instances of AI and machine learning hardware modules as an alternative to using CPU-based implementations.

References

[1] O. Boiman, E. Shechtman, and M. Irani, “In defense of nearest-neighbor based image classification,” in *2008 IEEE conference on computer vision and pattern recognition*. IEEE, 2008, pp. 1–8.

[2] A. Bewley and B. Upcroft, “Advantages of exploiting projection structure for segmenting dense 3d point clouds,” in *Australian Conference on Robotics and Automation*, vol. 2, 2013.

[3] M. A. Talib, S. Majzoub, Q. Nasir, and D. Jamal, “A systematic literature review on hardware implementation of artificial intelligence algorithms,” *The Journal of Supercomputing*, vol. 77, no. 2, pp. 1897–1938, 2021.

[4] M. A. Mohsin and D. G. Perera, “An fpga-based hardware accelerator for k-nearest neighbor classification for machine learning on mobile devices,” in *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2018, pp. 1–7.

[5] T. Ito, Y. Itotani, S. Wakabayashi, S. Nagayama, and M. Inagi, “A nearest neighbor search engine using distance-based hashing,” in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 150–157.

[6] Z.-H. Li, J.-F. Jin, X.-G. Zhou, and Z.-H. Feng, “K-nearest neighbor algorithm implementation on fpga using high level synthesis,” in *2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*. IEEE, 2016, pp. 600–602.

[7] A. Lu, Z. Fang, N. Farahpour, and L. Shannon, “Chip-knn: A configurable and high-performance k-nearest neighbors accelerator on cloud fpgas,” in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 139–147.

[8] J. Zhang, S. Khoram, and J. Li, “Efficient large-scale approximate nearest neighbor search on opencl fpga,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4924–4932.

[9] F. B. Muslim, A. Demian, L. Ma, L. Lavagno, and A. Qamar, “Energy-efficient fpga implementation of the k-nearest neighbors algorithm using opencl.” in *FedCSIS (Position Papers)*, 2016, pp. 141–145.

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: Constructing hardware in a scala embedded language,” in *DAC Design automation conference 2012*. IEEE, 2012, pp. 1212–1221.

[11] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic *et al.*, “An agile approach to building risc-v microprocessors,” *ieee Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[12] H. Fuchs, Z. M. Kedem, and B. F. Naylor, “On visible surface generation by a priori tree structures,” in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, 1980, pp. 124–133.

[13] A. Kondić and V. Milovanović, “Hardware realization of nearest neighbour search algorithm over an in-memory pre-stored k-d tree,” www.github.com/milovanovic/nns, accessed: April 15, 2022.