ISSN No:-2456-2165

Optimizing Multi-Dimensional Data-Index Algorithms for Mic Architectures

Seid Mehammed[1], Demeke Getaneh[1], Md Nasre Alam[1], Getachew Worku[1], Tizazu bayih[2]
[1]Department of computer science, Woldia University
[2]Department of Information Technology, Woldia University

Abstract:- A data structure for geographical partitioning called multi-dimensional data-indexing enables effective CPU-based nearest-neighbor searches. Despite not being a natural match for Many-Integrated Core Architecture (MIC) implementation, depth-first search Multi-Dimensional Data-Indexing can nevertheless be successful with the right engineering choices.

We suggested a technique that minimizes data structure memory trace by limiting the maximum height of the DFS Multi-Dimensional Data-Indexing.

With tens of thousands to tens of millions of points in the MIC kernel code, we optimize the multi-core MIC NN search. In comparison to a single-core CPU of equivalent power, it is 20–40 times quicker. NN uses the knowledge obtained from improving MIC code to find ways to rewrite CPU code.

As a consequence, the initial level of CTA and engineering choices to make the Multi-Dimensional Data-Indexing search algorithm on CPU and MIC simpler account for the bulk of the parallel performance in this study. Threads inside each thread warp split onto several search pathways for the second level of CTA using Multi-Dimensional Data-Indexing.

Thread divergence removes the majority of the performance benefit of employing multiple threads per thread-block. Experiments in this article reveal that small thread block sizes produce the best results.

Keyword:- *Multi-Dimensional Data-Indexing, MIC, depth-first search, Thread-block size.*

I. INTRODUCTION

The NN issue is crucial in several fields of computer science, including computer graphics, machine learning, pattern recognition, statistics, and data mining, among others. It determines which point in a point cloud is nearest to a given query point. . [1].

There are a number of issues with NN search despite its significance and widespread use. n search points, S, and m query points, Q, as well as a distance measure in d dimensions, make up each NN search's input. The Euclidean distance between search point p ∈ S and query point q∈Q is dist (p, q) $= \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_d - p_d)^2}$. The output consists of the km nearest points, where k is the number of requested nearest points in the search set S for each query point in Q.

Finding the closest point (k=1) in the search set S for each query point in Q under the distance metric dist results in the output set R comprising m result points for the query nearest neighbor (QNN) search.

Generalize QNN to find the k closest points in the search set S for each query point in Q for the 'k' nearest neighbor (kNN) search (producing R containing km points). We assume that the query set Q and the search set S are the same (Q=S) for all nearest neighbor searches (NN and kNN). We suppose that Q=S introduces a new problem in which zero-distance results must be carefully excluded; otherwise, each query point would appear in the search results.

A few other NN searches can be supported by Multi-Dimensional Data-Indexing. Find all points from the search set S contained in each query region belonging to the query set QR using the range query closest neighbor (RNN) search (region). With this method, each region receives a different quantity of result points. [2].

II. RELATED WORK

In this section, we briefly discuss NN solutions, Multi-Dimensional Data Indexing s, and related NN work on the MIC.

- *NN Solutions*

A brute force QNN exploration could directly compare query point to all n point in search set.

For our NN search solution, focus on the Multi-Dimensional Data Indexing, generalized binary tree invented [4] and improved by several researcher in the year. According to [2] detail an efficient nearest neighbor (NN) algorithm using a depth-first search (DFS) balance Multi-Dimensional Data Indexing, a priority queue, and a trim optimization to avoid unproductive search paths. This approach result in $O(\log n)$ expected search times for each query point on well-distributed point sets. In [5] author implemented a fast and efficient version of NN search in his book.

ISSN No:-2456-2165

➢ *Multi-Dimensional Data Indexing Review*

Multi-Dimensional Data-Indexing is hierarchical spatial partitioning data structure that to organize objects in d-dimensional space. Multi-Dimensional Data-Indexing partitions points and more complicate objects into axis-align cells called nodes. This cutting plane partitions all points at each parent node into left and right child nodes.

Variations on Multi-Dimensional Data-Indexing s differ in how cutting plane is pick. Multi-Dimensional Data-Indexing for search set S of n d-dimensional points takes O(d·n) storage and can be built in O($d.n \log n$)time. Build Multi-Dimensional Data-Indexing on the CPU and then transfer the kd-nodes onto the MIC.

To perform a nearest neighbor search in a Multi-Dimensional Data-Indexing, one can imagine traversing the entire tree, computing the distance of the query to the search points stored at each node while keeping track of the nearest neighbor point found thus far. Search queries (QNN, kNN , and RNN) that return t results have been show to take worst-case O$\left(d.n^{(1-1/d)}+t\right)$ time for all search point sets and expected O($\log_2(n)+t$)time for well-distributed search point sets.

For the 2D All-NN and All-*k*NN searches, multiply the theoretical cost of a single point query by the number of points (n) in our search set, giving O(n$\sqrt{n}$+tn) worst-case time and O($\log_2(n)+tn$) expected time using a balance Multi-Dimensional Data-Indexing implementation[3]. In addition to performing NN search, Multi-Dimensional Data-Indexing s can also solve point location, range search, and partial key retrieval problems [6].

➢ *Related NN work on the MIC*

The initial NN search solutions for MIC solutions were carried out by brute force, comparing each of the m points in Q to each of the n points in S. It takes O (n/p) time for each query point qi to compute the n query to search point distances using p threads. This is followed by a parallel reduction to determine the shortest distance for that query point, which also takes O (n/p) time. implemented a bucket sort to divide 3D points into fixed-size grid cells, followed by a brute force search in each query point's 333 cell vicinity. [6].

Implement a brute force NN algorithm in Intel VTune Amplifier XE for MIC with a 100+ to 1 speedup compared to the equivalent algorithm in MATLAB.

According to [8] built a breadth-first search MIC Multi-Dimensional Data-Indexing in Intel VTune Amplifier XE for MIC with splitting metric that combine empty space splitting and median splitting. SAH Multi-Dimensional Data-Indexing accelerate ray tracing, while VVH Multi-Dimensional Data-Indexing accelerated NN search.

NN search iterate using a range region search and by increasing fixed radius of the search region on each iteration. The Multi-Dimensional Data-Indexing built about 9-13 times faster than the CPU kd-tree. The MIC kNN search ran 7-10 times faster than the CPU kNN search. Developed a MIC ANN search based on the [2] approach with a kd-tree to assist in solving a 3D registration problem on the MIC[9]. The Multi-Dimensional Data-Indexing is built on the host CPU and then transfer to the MIC before running ANN. ANN search backtracks to candidate nodes using small fixed-length queue.

According to [6], MIC registration was 88 times faster than CPU registration. Inappropriately, performance comparison between MIC and CPU ANN search was not broken out from overall results.

➢ *The Multi-Dimensional Data-Indexing Data Structure*

Our NN search algorithm is adapted from [2]. It uses minimal Multi-Dimensional Data-Indexing, a search stack, and trims optimization. Demonstrate this solution for 2D points, although later in the paper, also do performance experiments on 3D and 4D points.

➢ *Multi-Dimensional Data-Indexing Search Concepts*

To help the reader understand the Multi-Dimensional Data-Indexing search, briefly enumerate the following six concepts: Each kd-node contains a search point ‹x, y, ..›. Best distance variable tracks the closest solution found so far

When doing a depth-first search (DFS), onside nodes are explored first, and overlapping offside nodes are stored for further inspection in a search stack. A kd-node index, onside/offside status, split axis, and split value are all attributes of each element kept on the search stack.

➢ *Multi-Dimensional Data-Indexing NN Search*

Our Multi-Dimensional Data-Indexing search algorithm works as follows. The root search element (root index, onside, x-axis) is push onto stack.

If the node is onside, the current kd-node loads from the node index. Next, if distance between query point and the current node search point is smaller than current best our method updates the best distance and best index.

The interval containing the query point is the onside node, and the remaining interval is the offside node. If kept, an offside search element is push onto the search stack. The onside search element is always push onto the search stack. When the search stack becomes empty, the best distance and best index indicate nearest neighbor.

ISSN No:-2456-2165

III. OUR PROPOSED SOLUTION

❖ *MIC Hardware Considerations*

Consider the following seven concepts: memory hierarchy access speeds, floating-point, memory hierarchy capacities, memory alignment, coalescence, thread-block size, and divergent branching in our proposed approach.

➢ *Memory Hierarchy:*

Registers, shared memory, constant memory, and global memory are the quickest to slowest access speeds in the MIC memory hierarchy (RAM). Local variables should be kept in registers, simple indexed data structures should be kept in shared memory, and points and Multi-Dimensional Data-Indexing nodes should be kept in global memory for better performance. The number of data transfers from slower RAM to quicker shared memory has been reduced.

A single read per loop is used in the NN search code. In a 1 million search and query point QNN search, for example, each query visits roughly 40-80 kd-nodes (one read per node) to locate the exact response.

➢ *Floats:*

MIC architecture support both 32-bit and 64-bit floating-point data. Focus only on 32-bit floating-point data. In handful of queries, our MIC and CPU NN search return slightly different neighbors.

➢ *Memory Capacity:*

For 2D points, reduce kd-nodes from eight to two fields. To store query points, kd-nodes, and final search results, the 2D QNN search requires just seven 32-bit components per 2D point. The QNN search was able to process up to 36 million 2D points on the MIC using this method.

➢ *Memory Alignment:*

Data structures aligned on 4, 8, or 16-byte memory boundary perform faster than unaligned data.

➢ *Coalescence:*

Only sequential memory accesses are coalesced by MIC. All threads within a thread-warp start their searches at the root node with NN depth-first search (DFS) through Multi-Dimensional Data-Indexing, but shortly diverge to distinct unanticipated sub-trees within the Multi-Dimensional Data-Indexing and therefore different portions of memory. NN searches on DFS Multi-Dimensional Data Indexing don't always result in consecutive scans across the data warp, therefore coalescence isn't possible.

➢ *Latency:*

The MIC programmer hides latency via TLP by scheduling a massive grid of thread blocks, but block performance is still constrained by the slowest thread in each block. The grid can hold up to 65,536 thread blocks in any dimension. Each thread-block can hold a maximum of 5128 threads. For MIC, the thread manager distributes thread-blocks across 16 SMs, each with 16 SPs. Create a thread for each query point in our NN search. We may use padded access to pad our query up to the next multiple of the thread-block size by repeating the first query until it reaches that multiple. This method eliminates the range check contrast that would increase the gap between the first and second query results.

➢ *Divergent Branching:*

All threads in a thread block must follow both the "if" and "else" branch routes if at least two threads diverge at a conditional branch. Correct conduct, which accepts the performance impact brought on by divergence, necessitates the conditional logic that is still in place. Process the All-NN and All-kNN searches in a sequential Multi-Dimensional Data-Indexing order to improve the coherence of all threads in the thread block. As a result, the All-NN search outperforms the QNN search by 5–6%. As a trade-off, All-kNN search performs somewhat worse than kNN search.

Thread-Block Size (TBS): There are 16 KB of shared memory and 8 K of 32-bit registers that can be used by each MIC core. Our present NN search limits the maximum number of threads per SM to 2565 since temporary variables use roughly 24-32 registers. 192–240 bytes of shared memory are needed for QNN and All-NN searches on data structures like 20–28 element deep stacks. Depending on the search type and the size of the input data, our performance trials showed that an optimal thread-block size for our DFS Multi-Dimensional Data-Indexing NN search is between 8 and 16 threads per block.

➢ *Multi-Dimensional Data-Indexing Design Choice*

Based on the MIC hardware limits, sought to efficiently use MIC memory resource. Such a goal suggests bounding the Multi-Dimensional Data-Indexing height and reducing size of data structure in memory.

➢ *Bounding Multi-Dimensional Data-Indexing Height:*

Shared memory is target for our NN search stack Each MIC core only has 16KB of shared memory available to all threads. Have no more than 256 bytes available for all data if employing 64 threads per thread-block.

Give the following examples of balanced, static, and efficient array layouts:

➢ *Balanced Multi-Dimensional Data-Indexing:*

A balanced Multi-dimensional Data-Indexing of maximum height$\lceil(\log_2 n)\rceil$, with a difference of at most one level across all leaf nodes, is built by setting the cutting plane through median point of each sub-tree.

➢ *Reducing Memory Foot print:*

To increase the number of points in MIC memory, reduce the size of the Multi-Dimensional Data-Indexing data structure. kd-node fields include child pointers, parent pointers, split-axis, split-value, cell bounding-box, and stored-point.

➢ *Array Layout:*

Store the kd-nodes in an array as a left-balanced binary tree for faster indexing. The Kd-node is initially constructed as a left-balanced median. As part of the build process, Multi-Dimensional Data-Indexing converts to a left balance binary tree.

ISSN No:-2456-2165

➢ *D-Dimensionality:*

In these NN searches, use points with 2-4 dimensions x, y,... to decrease the amount of data saved on the MIC. Search can also be expanded to include more dimensions.

➢ *Eliminating fields:*

The parent pointer can be avoided by using the search stack in NN search to go backward. The split axis and split value are implied if a cyclic multi-dimensional data indexing tree splits x, y, x, y, etc. The split value is implied by the store d-dimensional-point. Cell bounding boxes are not required for the NN search.

➢ *Final Design:*

, (left panel), compute the minimum and maximum bound of search-points. The root of Multi-Dimensional Data-Indexing is conceptually associate with these min-max bound and sequence [1, n] of original point. Split-value is pick along one of dimensional axes. All points are partition into two smaller left and right boxes based on splitting value. Each child node has a bounding box and a partitioned sequence of points associated with it. Up until specific halting requirements are met, Multi-Dimensional Data-Indexing is iteratively refined by dividing each child sub-tree, associated box, and associated point sequences.

The quick median algorithm uses the same partition sub-routine as the quicksort method, which is utilized for sorting. (left-panel).

Multi-Dimensional Data-Indexing builds an algorithm from a list of search point (left-panel).

A single query point's perspective on the Multi-Dimensional Data-Indexing search method (right-panel).

❖ *Searching the Multi-Dimensional Data-Indexing*

Our NN search solutions are based on the Multi-Dimensional Data-Indexing search solution, already described in Section 2.1. This same search solution can be simplified and adapt to solve the Point Location problem as described in section 3.4.1. Give more details on a CPU Host function for the QNN and All-NN search solutions in section 3.4.2. kNN and All-kNN search solutions must track k closest points, so introduce more details on how to handle these k points in 3.4.3. The data structure use to track the k closest point also use shared memory.

As a 2D, 3D, and 4D static balance cyclic data structure, utilize our DFS Multi-Dimensional Data-Indexing data structure. For multi-dimensional data indexing, a single left-balance median point store is employed at each node. The nodes of Multi-Dimensional Data-Indexing are stored in a left-balanced binary tree array. In order to implement NN search, deep-first search with stack for backtracking is used. With this multidimensional data indexing design, predictable stack sizes are constrained by height. reduces the memory footprint of the Multi-Dimensional Data-Indexing and search nodes.

➢ *Building the Multi-Dimensional Data-Indexing*

As we show in

There are two stages to each iteration of a selection:

➢ *Pivot phase:*

An algorithm chooses a potential pivot value p by averaging three different methods.

➢ *Partition Phase:*

The pivot value is use to partition point into three data set. Left: points less than p, Middle: all points equal to the pivot value, and Right: all points greater than or equal to p. Construct the Multi-Dimensional Data-Indexing on the CPU and then transfer it to the MIC for the MIC NN search. A high-level overview is found in

➢ *Point Location Problem:*

May easily locate objects in a Multi-Dimensional Data-Indexing by moving down the tree until the interest cell is identified, and then searching for the point of interest inside that cell.

➢ *NN Search Remapping Issue:*

Directly translate the kd-tree NN search algorithm into code to tackle any NN search problem, including QNN.

```
procedure BuildKDTree(d, points, lbm kd-nodes)
// Initialize kd-tree nodes
n ← |points|
Allocate memory for n median kd-nodes
Allocate memory for n left balanced median
(lbm) kd-nodes
for all in points
  medianNodes[idx].xy ← points[idx].xy
  medianNodes[idx].pointIdx ← idx
  medianNodes[idx].nodeIdx ← INVALID
// Add root build item
top ← 0
build ← { [0,n-1], x-axis, 1 }
buildStack[top++] ← build;
// Build kd-Tree
while buildStack not empty do
  // Get current build item
  currItem ← buildStack[top--]
  [low,high] ← currItem.sequence
  currAxis ← currItem.splitAxis
  currIdx ← currItem.location
  N ← (high-low)+1
  M ← low + LBMpos(N)  // Left bal. Median
  L ← (low+M-1)/2
  R ← (M+1+high)/2
  left  ← 2*currIdx;
  right ← left+1;
  nextAxis = (currAxis+1) % d
  // Partition via Median Selection
  Partition(medianNodes, M) into sub-seqs.
    Left{low,M-1}, Median{M}, and
    Right{M+1,high}
  mNode ← medianNodes[M]
  lbmNode[currIdx] ←
    {mNode.xy,mNode.pointIdx,M}
  // Add right build item to stack
  rightItem ← {[low+M+1,high],nextAxis,right}
  buildStack[top++] ← rightItem
  // Add left build item to stack
  leftItem ← {[low,low+M-1],nextAxis,left}
  buildStack[top++] ← rightItem
end while
// Cleanup
Free memory associated with median kd-nodes
return lbm kd-nodes
```

```
procedure QNNsearch(d, qp, kd-nodes, remap)
// Initialize search
root ← kd-nodes[1]
bestIdx ← 1
bestDist ← Huge Value (Infinity)
// Add root search element
top ←0
rootElem ← {1, onside, x-axis, Infinity}
searchStack[top++] = rootElem;
// Find Nearest Neighbor
while searchStack not empty do
  currElem ← searchStack[top--];
  if currElem.state == offside,
    result ← trimtest(bestDist,
             currElem.splitValue )
    if result == rejected
      return to top of loop // while (search)
    end if
  end if
  // Update Best Distance
  currNode ← kd-nodes[currElem.nodeIdx]
  left ← 2 * currElem.nodeIdx
  right ← left+1
  currDist ← distance( qp.xy, currNode.xy )
  if currDist < bestDist
      bestDist ← currDist
      bestIdx ← nodeIdx
  end if
  currAxis ← currElem.splitAxis
  nextAxis ← (currAxis + 1) % d
  splitValue ← currElem.xy[currAxis];
  determine onside and offside nodes
    from qp, splitAxis, splitValue
  // Add offside node
  diff2 ← (qp.xy[currAxis] -
           currNode.xy[currAxis])^2
  result ← trimtest( bestDist, diff2 )
  if result == accepted
    offElem←{offIdx,offside,nextAxis,splitValue}
    searchStack[top++] ← offElem
  end if
  // Add onside node
  onElem ← {onIdx,onside,nextAxis,splitValue}
  searchStack[top++] ← onElem;
end while
bestIdx ← remap[bestIdx] // map node into pntIdx
return bestIdx and bestDist.
```

Fig 1: Build & Search Methods

❖ *Performance Results*

In this section, we compare parallel NN search performance on the MIC to serial NN search performance on the CPU. As we show below all performance tests, were conducts on Intel MIC using a desktop computer configures.

CPU Hardware: CPU = i7-920@2.67 GHz, RAM=12

MIC Hardware:2× MIC(30 SMs, 240 total SPs, 1.0 GB RAM, 159.0 GB/s peak throughput)

Software: MIC API= C++, IDE = Intel VTune Amplifier XE, OS =LINUX, Pointers= 64-bit

Data: Input size, n=[100 – 107], in increasing powers

➢ *NN Search Experiment Environment*

For each NN search type, TLP tests identify the optimal thread block size (TBS), and experiments to demonstrate the performance for growing input sizes (n) and growing search sizes are conducted as part of these performance trials (k).

➢ *Multidimensional Data Indexing Construction on the CPU.*

As we, show **Error! Reference source not found.** shows the CPU cost of building the Multi-Dimensional Data-Indexing for different numbers of 2D-points.

Table 1: CPU Build Performance

***n*, # of points**	**1**	**10**	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**	**10^7**
Build Time (in ms)	0.019	0.045	0.151	2.43	22.74	192.52	2,165.31	24,491.28
Time/Pnt (ms/pnt)	0.014	0.0043	0.00165	0.00165	0.00214	0.00163	0.00179	0.00202

ISSN No:-2456-2165

Amortize time per point to build Multi-Dimensional Data-Indexing initially decreases and then surprisingly levels off after 100 points. Expect time per point to increase, matching theoretical O(n·log n) performance. Our best guess for this surprising result is that some CPU caching effects came into play.

❖ *Finding optimal thread-block size*

For each of our NN searches, manually try thread-block sizes containing between 1 and 80 threads to discover the best thread-block size on Intel MIC. Table 1 (panels a-c) shows the results of 2D QNN, All-NN, kNN, and All-kNN for data sets with 1 million and 10 million search points, respectively.

Optimal thread-block 10x1 with a speedup of 46.4 for QNN 1 million search points. It was 7x1 with a speedup of 43.6 for 10 million points. The ideal thread block for All-NN of 1 million points was 10x1 with a speedup of for 10 million points; it was 10x1 with a speedup of 36.8. The best thread-block for kNN with 1 million search points and k=32 was 4x1, with a speedup of 18.1. The optimum thread-block for an All-kNN search with 1 million points and k = 32 was 4x1, with a speedup of 15.7.

As we show Fig depict a) The graph plots MIC or CPU speedup for 2D QNN, All-NN search for increasing thread block-size with fixed-size search and query data set of one million point. b) Graph is the same but for 2D kNN and All-kNN searches. c) The graph plots the 2D QNN, All-NN speedup for 10 million points. d) Graph track 2D kNN and All-kNN speedups for increasing values of n. e) Graph tracks 2D QNN and All-NN speedups for increasing values of n. f) The graph tracks 2D kNN and All-kNN speedups for increasing values of k from 1-32.

In panels, d-f in Fig, increase n, total number of search point across MoreThan few order of magnitude using optimal thread-block size for each type of NN search.

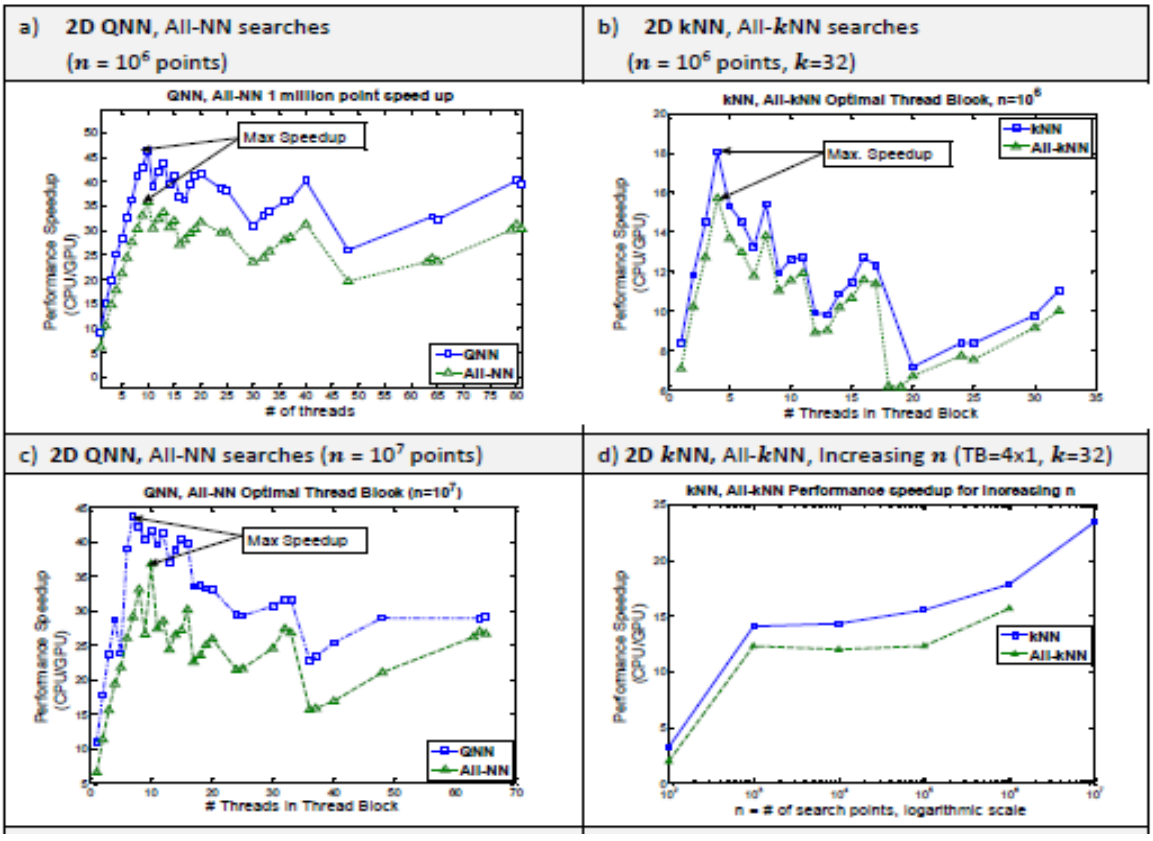
➢ *Increasing n:*

The largest speed-up is achieved for n=10 million for 2D QNN; speed-ups range from [20 - 41.5]. See comparable outcomes for All-NN: the increases in the range of [20 - 36.8]; maximum again at 10 million points. For both search, if n≤100 points, it is better to use brute force solution.

For 2D kNN and All-kNN, set k=32. For kNN, see speedups in the range [14-18] with the maximum at 1 million points. There is enough memory to run a query with 10 million points, but then have to decrease k = 8 for both the search stack and closest heap to fit into shared memory. When decrease, see speedup of 23.4. For All-kNN, see speed-up in the range [12-15.7] with the maximum again at 1 million points.

➢ *Increasing k:*

Set n=106 for the 2D searches, and change k between 1-32. The speed-up appears to follow a shallow inverse quadratic curve in both instances. . All the speedups for the kNN search are in the range [17.9 - 22.7] with k=6 as the maximum. The outcomes are comparable for the All-kNN search, with speedups in [15.7 - 18.4] with the maximum at k=3.



ISSN No:-2456-2165

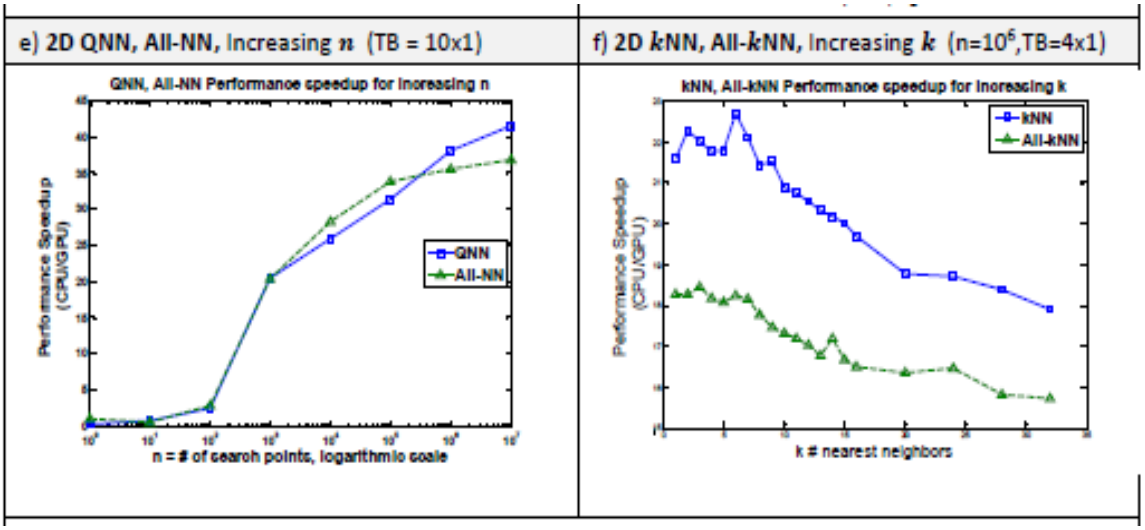


Fig 2: Multi-Dimensional Data-Indexing Search Results

IV. CONCLUSION

QNNs, kNNs, All-NN, and All-kNN search algorithms are validated using DFS minimal Multi-Dimensional Data Indexing. Each node in the static, balanced, cyclical Minimal Multi-Dimensional Data-Indexing architecture stores a single point that corresponds to the left-balance median split along the current axis. Our Multi-Dimensional Data-Indexing architecture can handle more points with better speed because of the efficient memory use.

Serial NN search on the CPU takes longer than parallel NN search on the MIC, which can handle up to 38 million 2D points. Multi-core MIC QNN searches are 25–40 times quicker than single-core CPU QNN searches. In comparison to the All-NN search on the CPU, the All-NN search on the MIC is 10–40 times quicker. The kNN search is 14–20 times quicker on the MIC than on the CPU. The MIC all-kNN search is 9–18 times quicker than the CPU all-kNN search.

The CPU serial NN search process is considerably slower than the MIC parallel NN search method, which can handle up to 22 million 3D points. Ten to thirty times quicker than a single-core CPU QNN search is a multi-core MIC QNN search. In comparison to the All-NN search on the CPU, the All-NN search on the MIC is 12–30 times quicker. The speed of the MIC kNN search is 8–18 times quicker than the CPU kNN search.

MIC CPU ALL-kNN searches are 8–16 times quicker than GPU ALL-kNN searches. MIC Parallel NN searches are quicker than serial NN searches on the CPU and can handle up to 22 million 4D points. 8 to 22 times quicker than single-core CPU QNN searches are multi-core MIC QNN searches. MIC In comparison to the GPU All-NN search, the CPU All-NN search is 11–21 times quicker. The CPU-based kNN search is 6–14 times slower than the MIC kNN search. MIC Compared to the GPU All-kNN search, the CPU ALL-kNN search is 6–13 times quicker.

REFERENCES

[1]. G. Shakhnarovich, T. Darrell, and P. Indyk, "*Nearest-neighbor methods in learning and vision*", in Neural Information Processing, 2005.
[2]. S. Arya, G. D. Da Fonseca, and D. M. Mount, "*A unified approach to approximate proximity searching*", in European Symposium on Algorithms, 2010: Springer, pp. 374-385.
[3]. H. Samet. "*Foundations of multidimensional and metric data structures*". Morgan Kaufmann, 2006.
[4]. J. L. Bentley, "*Multidimensional binary search trees used for associative searching*", Communications of the ACM, vol. 18, no. 9, pp. 509-517, 1975.
[5]. M. Jensen, "*Value maximization, stakeholder theory, and the corporate objective function*", European financial management, vol. 7, no. 3, pp. 297-317, 2001.
[6]. D. Qiu, S. May, and A. Nüchter, "*GPU-accelerated nearest neighbor search for 3D registration*", in International conference on computer vision systems, 2009: Springer, pp. 194-203.
[7]. T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "*Photon mapping on programmable graphics hardware*," in ACM SIGGRAPH 2005 Courses, pp. 258-es, 2005.
[8]. K. Zhou, Q. Hou, R. Wang, and B. Guo, "*Real-time kd-tree construction on graphics hardware*", ACM Transactions on Graphics (TOG), vol. 27, no. 5, pp. 1-11, 2008.
[9]. S. S. Skiena. "*The algorithm design manual*". Springer International Publishing, 2020.