CHIP-KNNv2: A Configurable and High-Performance K-Nearest Neighbors Accelerator on HBM-based FPGAs

KENNETH LIU*, ALEC LU*, KARTIK SAMTANI, and ZHENMAN FANG, School of Engineering Science, Simon Fraser University, Canada
LICHENG GUO, Computer Science Department, University of California, Los Angeles, United States

The k-nearest neighbors (KNN) algorithm is an essential algorithm in many applications, such as similarity search, image classification, and database query. With the rapid growth in the dataset size and the feature dimension of each data point, processing KNN becomes more compute and memory hungry. Most prior studies focus on accelerating the computation of KNN using the abundant parallel resource on FPGAs. However, they often overlook the memory access optimizations on FPGA platforms and only achieve a marginal speedup over a multi-thread CPU implementation for large datasets.

In this paper, we design and implement CHIP-KNN: an HLS-based, configurable, and high-performance KNN accelerator. CHIP-KNN optimizes the off-chip memory access on modern HBM-based FPGAs such as the AMD/Xilinx Alveo U280 FPGA board. CHIP-KNN is configurable for all essential parameters used in the algorithm, including the size of the search dataset, the feature dimension and data type representation of each data point, the distance metric, and the number of nearest neighbors - K. In terms of design architecture, we explore and discuss the trade-offs between two design versions: CHIP-KNNv1 (Ping-Pong buffer based) and CHIP-KNNv2 (streaming-based). Moreover, we investigate the routing congestion issue in our accelerator design, implement hierarchical structures to shorten critical paths, and integrate an open-source floorplanning optimization tool called TAPA/AutoBridge to eliminate the place-and-route issues. To explore the design space and balance the computation and memory access performance, we also build an analytical performance model. Given a user configuration of the KNN parameters, our tool can automatically generate TAPA HLS C code for the optimal accelerator design and the corresponding host code, on the HBM-based FPGA platform.

Our experimental results on the Alveo U280 show that, compared to a 48-thread CPU implementation, CHIP-KNNv2 achieves a geomean performance speedup of 15x, with a maximum speedup of 45x. Additionally, we show that CHIP-KNNv2 achieves up to 2.1x performance speedup over CHIP-KNNv1 while increasing configurability. Compared with the state-of-the-art Facebook AI Similarity Search (FAISS) [23] GPU implementation running on a Nvidia Tesla V100 GPU, CHIP-KNNv2 achieves an average latency reduction of 30.6x while requiring 34.3% of GPU power consumption.

CCS Concepts: • **Hardware → Hardware accelerators**; **Hardware-software codesign**; • **Computer systems organization → Reconfigurable computing**; **High-level language architectures**.

Additional Key Words and Phrases: K-Nearest Neighbors, HBM-based FPGA, High-Level Synthesis, Automation Framework

*Both authors contributed equally to this research.

Authors' addresses: Kenneth Liu, ksl24@sfu.ca; Alec Lu, alec_lu@sfu.ca; Kartik Samtani, kss24@sfu.ca; Zhenman Fang, zhenman@sfu.ca, School of Engineering Science, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A1S6; Licheng Guo, lcguo@ucla.edu, Computer Science Department, University of California, Los Angeles, 404 Westwood Plaza, Los Angeles, California, United States, 90095.

1936-7406/2023/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Kenneth Liu, Alec Lu, Kartik Samtani, Zhenman Fang, and Licheng Guo. 2023. CHIP-KNNv2: A Configurable and High-Performance K-Nearest Neighbors Accelerator on HBM-based FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2023), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The k-nearest neighbors (KNN) algorithm [2] is one of the top 10 most influential algorithms in the data mining research community [37]. It is widely used in many applications such as similarity search, image classification, and database query [17, 34, 41]. With the rapid growth in the size of the overall search dataset and the dimension of each data point's feature vector, there is an ever-increasing demand of computing resource and memory bandwidth to process the KNN algorithm [21, 24].

Previous works have investigated accelerating the KNN algorithm on CPUs and GPUs through parallel or distributed computing [3, 14, 15, 29, 30]. Considering the significant slowdown of CPU performance scaling and the high power consumption of GPUs, recently, accelerating the performance of KNN on FPGAs has gained increasing attention. Several prior studies [20, 32, 35] have achieved decent performance and/or energy efficiency improvements over the CPU and GPU implementations by exploring the massive fine-grained parallelism for the neighbor distance calculation and sorting in their FPGA-based KNN accelerator designs. For example, the latest FPGA accelerator for KNN in [35] achieves an equivalent performance as a 56-thread CPU implementation for large datasets while achieving a 324x higher energy-efficiency. Compared with the GPU accelerator for KNN, the FPGA accelerator in [32] achieves 3x better performance-per-Joule ratio for small datasets.

However, there are two major issues in most of these prior KNN accelerator designs on FPGAs [20, 32, 35]. First, most of them, except the latest design in [35], only support a fixed configuration of KNN with a small dataset, fixed feature dimension, distance metric, and data type representation. Second, most prior studies overlook the memory access optimizations, which limits the KNN accelerator performance on FPGAs, especially for large datasets that cannot fit into on-chip memory. Although modern datacenter FPGAs have equipped with multiple DRAM or HBM banks to boost the off-chip memory bandwidth—e.g., Xilinx Alveo U200 [38] and U280 [39] datacenter FPGA boards respectively provide up to 76.8GB/s and 460GB/s theoretical memory bandwidth using four DDR4 banks and 32 HBM2 banks—many existing KNN accelerator designs only utilize no more than 12% of the available off-chip bandwidth on their evaluated platforms as summarized in Section 5.1.

In this paper we design and implement CHIP-KNN: an open-source, HLS (high-level synthesis) C based, configurable, and high-performance KNN accelerator for HBM-based FPGAs, such as the AMD/Xilinx Alveo U280 FPGA board. We choose an HBM-equipped FPGA which provides high off-chip bandwidth, since the KNN algorithm is primarily bandwidth-bound. In terms of design architecture, we explore two different alternatives. Firstly, CHIP-KNNv1 is a Ping-Pong buffer based architecture, which uses multiple on-chip buffers to accelerate the on-chip computation and efficiently overlap off-chip memory transfers and the computation. Secondly, to further improve device resource utilization and acceleration performance, we design and implement CHIP-KNNv2, a streaming-based architecture that enables FIFO-based task-level dataflow. Both versions are well optimized for their off-chip memory access to utilize the bandwidth of multiple HBM banks and fully exploit the bandwidth of each HBM bank.

To better support large search datasets, CHIP-KNN takes a scalable multi-PE (processing element) approach. Our design relies on a template PE, where CHIP-KNNv1 loads data from off-chip memory into on-chip buffers for the the computation units - i.e., neighbour distance calculation and sorting,

and CHIP-KNNv2 streams data from off-chip memory into these computation units. These computation units are optimized by exploring the pipeline parallelism and fine-grained data parallelism enabled by a novel sorting algorithm. After the correct optimizations are identified, we scale up the number of PEs to take advantage of coarse-grained parallelism and to utilize multiple HBM memory banks with off-chip bandwidth optimizations. To merge the multiple copies of K nearest neighbours suggested by the multiple PEs, we utilize a hierarchical merge-tree design, which reduces congestion and improves the achievable kernel frequency. To further optimize the placement-and-route quality of our streaming-based design, CHIP-KNNv2, we also add support to use a coarse-grained floorplanning optimization tool named TAPA/AutoBridge [8, 18], for streaming-based HLS accelerator designs. Additionally, TAPA/AutoBridge also supports out-of-context synthesis to speed up the high-level synthesis process, and includes HBM-specific optimizations to reduce resource overhead from the HBM interface IPs compared with Vitis HLS.

CHIP-KNN is also configurable to all key parameters used in the KNN algorithm, which includes 1) the number of data points in the search space, N; 2) the dimension of each data point's feature vector, D; 3) the distance metric; and 4) the number of nearest neighbors, K. With CHIP-KNNv2, one can also configure 5) the data-type representation of search space points and distances. Given a user configuration of these KNN parameters and an FPGA platform, our tool can automatically generate the optimal accelerator design that could reach the off-chip memory bandwidth boundary or the FPGA computing resource boundary. To achieve this automation, we also build an analytical performance model for all the three major stages—data loading/streaming, distance calculation, and distance sorting—to explore the design space and balance the execution of these three stages.

We conduct our experiments on the AMD/Xilinx HBM-based Alveo U280 [39] datacenter FPGA, with various configurations of KNN parameters. CHIP-KNNv2 achieves a geomean 15.5x performance speedup over a 48-thread CPU implementation, with an 8.5x geomean speedup when only considering single-precision floating-point designs. We also show that CHIP-KNNv2 achieves up to a 2.1x performance speedup over CHIP-KNNv1. Compared against state-of-the-art FAISS [23] GPU implementation running on a Nvidia Tesla V100 GPU, our design achieves an average of 30.6x latency improvement, while using 34.3% of the power.

In summary, this paper makes the following contributions.

1. Exploration of both Ping-Pong buffer based and streaming-based architecture designs to accelerate the KNN algorithm on HBM-based datacenter FPGAs with off-chip memory optimizations and frequency optimizations.
2. An open-source tool, CHIP-KNN, to automatically generate an optimized KNN accelerator design on an HBM-based FPGA board for a wide set of KNN parameters, including the size of the search dataset, the feature dimension and data type representation of each data point, the distance metric, and the number of nearest neighbors - K.
3. Experimental results to demonstrate the superior performance gains of our CHIP-KNN designs on the FPGA.

The rest of this paper is organized as follows. Section 2 introduces background information about the KNN Algorithm and datacenter FPGAs. Section 3 discusses the two design alternatives of CHIP-KNN, including their computation customization, bandwidth optimizations, and frequency optimizations. Section 4 evaluates the CHIP-KNN design against a 48-thread CPU implementation, analyzes its performance across the configuration space, and analyzes its efficiency. Section 5 presents related work for KNN Acceleration on CPUs, GPUs, and FPGAs, including a quantitative comparison to state-of-the-art GPU accelerator. Finally, Section 6 concludes this paper.

2 KNN ALGORITHM AND DATACENTER FPGAS

2.1 KNN Algorithm

The KNN algorithm, specifically the brute-force search KNN, is widely used in numerous applications such as similarity searching, image classification, and database query search [17, 34, 41]. In this paper we mainly focus on the exact KNN algorithm without any approximation for two reasons. First, approximate KNNs [4, 5, 31] achieve lower accuracy. Second, even in approximate KNN methods, after the initial classification or filtering, they still have to apply the exact KNN in the final step. The exact KNN algorithm consists of two major tasks:

1. *Distance calculation.* For an input query, this step calculates its distance to every data point in the search space. Each point is represented by a D-dimensional feature vector. Common distance metrics include *Euclidean* and *Manhattan* distances between two feature vectors. Assuming there are N points in the search space, the algorithmic complexity for this step is:

$$DistCalcRuntime = O(N * D) \tag{1}$$

   There is abundant data parallelism in this function: 1) the distance calculation for each data point can be parallelized, and 2) to calculate a single distance, the computation between each feature dimension can be parallelized.
2. *Top K distances sorting.* This step sorts the N distances and returns the K nearest neighbors, where K is usually very small, e.g., $K = 10$. The algorithmic complexity of this step is:

$$TopKSortRuntime = O(N * K) \tag{2}$$

   This is because we only require the K smallest distances to be sorted. Task-level parallelism can be realized by sorting a subset of the search space in parallel.

To summarize, the essential parameters in the KNN algorithm include *N*, *D*, *K*, *distance metric*, as well as the *datatype* of each data point. The data access complexity is also $O(N * D)$, making the memory access optimizations very important.

2.2 Datacenter FPGAs

To meet the increasing demand in computing resource and off-chip bandwidth, modern Datacenter FPGAs typically consist of multiple FPGA dies and multiple DRAM or HBM banks. For example, the Xilinx Alveo U200 [38] FPGA board has three SLRs (super logic regions, i.e., FPGA dies) and four DDR4 banks (64GB total size), which can provide up to 76.8GB/s theoretical memory bandwidth. And the Alveo U280 [39] FPGA board has three SLRs and 32 HBM2 banks (8GB size), which can provide up to 460GB/s theoretical memory bandwidth. The U280 provides great opportunity to accelerate the performance of the KNN algorithm, due to the high memory bandwidth.

Most prior studies [20, 32, 35] on KNN acceleration overlook the memory access optimization on FPGA platforms and achieve sub-optimal performance. In fact, many of them only utilize no more than 12% of such available off-chip memory bandwidth. First, they do not explore the parallel access bandwidth of multiple DRAM or HBM banks. Second, even for accessing a single memory bank, they do not tune the access to achieve the maximum effective bandwidth, which may leave a bandwidth gap up to 16x according to [11, 43].

While taking advantage of these missed opportunities could further improve acceleration performance, the increasing logic complexity of highly parallel accelerator designs makes it very difficult for the placement and routing tools to meet the desired timing requirements, and in some cases, even fail to be mapped onto the FPGA. In this work, our goal is design a scalable and configurable KNN accelerator on modern HBM-based FPGAs, especially focusing on the off-chip memory access optimizations, frequency optimizations, configurability and automation support.

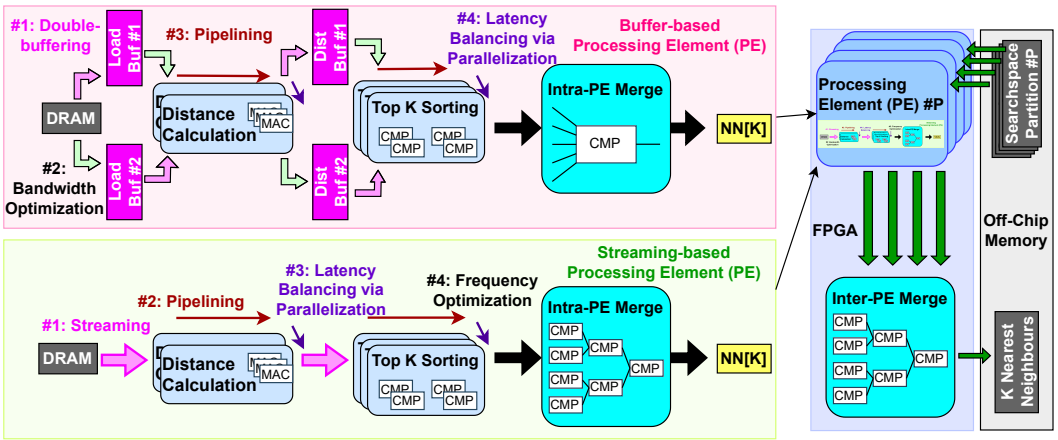


Fig. 1. Overall architecture of our CHIP-KNN accelerator. The top shows the Ping-Pong buffer based processing element (PE) design and the bottom shows the streaming-based PE design.

3 CHIP-KNN DESIGN

To better accelerate KNN with large search datasets, CHIP-KNN takes a multi-PE (processing element) approach with a scalable number of PE units. Figure 1 shows the overall design architecture. For the single-PE design, we explore and implement two design architectures: 1) a buffer-based implementation in Section 3.1, and 2) a streaming-based implementation in Section 3.2. For both architectures, we implement a novel sorting algorithm and explore common HLS optimizations such as pipelining and parallelization to accelerate the computation units. For the buffer-based architecture, we exploit the Ping-Pong buffer technique to overlap the off-chip memory access and computation using on-chip buffers; for the streaming-based architecture, we further reduce the on-chip resource usage for each PE by streamlining the data transfer. In Section 3.3, we present the multi-PE design with a global top K merger that explores the coarse-grained parallelism and the off-chip bandwidth of multiple HBM banks.

To improve the clock frequency of our multi-PE CHIP-KNN design, in Section 3.4, we first implement hierarchical tree-based merge modules to merge results from sorting modules within a PE and merge results from multiple PEs. Moreover, in Section 3.5, we also integrate state-of-the-art coarse-grained floorplanning and pipelining optimization tool TAPA/AutoBrdige [8, 18] to improve the floorplanning and thus the achievable frequency of CHIP-KNN. Since CHIP-KNN is bandwidth bound, in Section 3.6, we first characterize the effective off-chip memory of each HBM bank. Based on this characterization, in Section 3.7, we build an analytical performance model, which models the memory access performance and computation performance under different design parameters. Finally, in Section 3.8, we develop an automation framework to generate the final optimized CHIP-KNN accelerator design on FPGA based on the user-specified configurations, including the following parameters as described in Section 2.1: *N*, *D*, *K*, *distance metric*, and *datatype*.

3.1 Buffer-Based Single-PE KNN Design

Algorithm 1 shows the pseudo code of the buffer-based (CHIP-KNNv1) single-PE KNN algorithm. CHIP-KNNv1 supports the following user-configured parameters as described in Section 2.1: *N*, *D*, *K*, and *distance metric*. CHIP-KNNv1 supports Euclidean and Manhattan distances, and the most widely used single-precision floating-point data type. However, in Algorithm 1, we list *datatype*

Algorithm 1 Pseudo HLS-C code for single-PE KNN accelerator. User-configurable parameters are noted in *italic blue font*, including *N*, *D*, *K*, *distance metric*, and *datatype*, as described in Section 2.1. *B* denotes the number of buffered data points.

```
1:  function Load_Buf
2:      datatype local_search_space[B][D]
3:      memcpy (local_search_space ← a portion of N data points in search_space from off-chip memory)
        //pipeline II=1
4:  function Dist_Calc
5:      datatype input_query[D]
6:      datatype local_search_space[B][D]
7:      datatype point_dist[B]
8:      for i in 0 to B do
9:          //pipeline II=dist_II, unroll=dist_factor
10:         point_dist[i] = Manhattan or Euclidean distance
11: function Top_K_Sort
12:     datatype point_dist[B + K] //K dummy MAX dist
13:     int point_id[B + K] //K dummy invalid ids
14:     datatype k_nearest_dist[K+2] //Initialized as MAX dist
15:     int k_nearest_id[K+2] //Initialized as invalid ids
16:     for i in range 0 to B + K do
17:         //unroll=sort_factor, pipeline II=3 for CHIP-KNNv1 and II=2 for CHIP-KNNv2
18:         k_nearest_dist[0] = point_dist[i]
19:         k_nearest_id[0] = point_id[i]
20:         //Parallel compare-and-swap with items ahead
21:         for j in 1 to K; j+=2 do //fully unrolled
22:             if k_nearest_dist[j] < k_nearest_dist[j+1] then
23:                 swap (k_nearest_dist[j], k_nearest_dist[j+1])
24:                 swap (k_nearest_id[j], k_nearest_id[j+1])
25:         //Parallel compare-and-swap with items behind
26:         for j in 1 to K; j+=2 do //fully unrolled
27:             if k_nearest_dist[j] > k_nearest_dist[j-1] then
28:                 swap (k_nearest_dist[j], k_nearest_dist[j-1])
29:                 swap (k_nearest_id[j], k_nearest_id[j-1])
30:     //Optional step to merge local top K contenders in this function
```

as user-configurable, as CHIP-KNNv2 supports this additional configuration. For each tile of the dataset, the PE buffers it on chip and processes it in three major stages. Note all lines of code refer to Algorithm 1.

3.1.1 ***Load_Buf Stage (Lines 1-3).*** To improve the memory access performance, this stage reads a portion of the search space data points from off-chip memory and buffers them in the on-chip memory. This memory read uses burst access and achieves an II (initiation interval) of 1. The following two stages, *Dist_Calc* and *Top_K_Sort*, work on this local on-chip buffer. The off-chip memory access is also optimized by carefully tuning the memory ports' data width and consecutive data access size, as well as the operating frequency of the accelerator design, which will be further explained in Section 3.6.

3.1.2 ***Dist_Cal Stage (Lines 4-10).*** This stage calculates the distance between the query point and each point in the buffered search space. Currently, CHIP-KNN supports two types of distance metric calculations: Manhattan distance and Euclidean distance. One can easily extend CHIP-KNN with

other distance metrics. Between two D-dimension data points, X and Y, the Manhattan distance and Euclidean distance are:

$$M(X,Y) = \sum_{i=1}^{D} \left|X_i - Y_i\right|, E(X,Y) = \sum_{i=1}^{D} (X_i - Y_i)^2 \tag{3}$$

where we do not include the square root operation for Euclidean distance in either CPU, GPU, or FPGA implementations.

In this stage, we explore the following fine-grained data parallelism and pipeline parallelism. First, we fully parallelize (unroll) the calculation of all D dimensions in each distance calculation as shown in Equation 3 when necessary; for very high dimension D, we only perform partial unroll to balance different stages. Second, we further divide the buffered search space into *dist_factor* partitions and fully parallelize the distance calculation within each partition. Third, we pipeline the processing between multiple partitions with II of *dist_II*, as shown in lines 8-9. In Section 3.8, our automation tool will choose the optimal *dist_factor* and *dist_II* to balance the execution of the three stages in each PE.

3.1.3 ***Top_K_Sort Stage (Lines 11-30).*** This step sorts the top K nearest neighbors to the input query data point and returns the sorted top K distances and their corresponding data point IDs (lines 14-15). To improve the hardware efficiency, we propose the following novel top K sorting algorithm, which reduces the overall algorithmic complexity to O(2N).

1. To avoid the frequent off-chip memory write and read of the local top K results for each tile, we use the on-chip buffer *k_nearest_dist* to store the up-to-date top K results across all processed tiles within each PE. For each tile, this *k_nearest_dist* buffer is compared against all the B data points in the *point_dist* buffer (lines 12-13) to make sure it always keeps the K smallest distances. That is, we have the i loop that iterates the *point_dist* buffer as the outer loop (line 16), and the j loops that iterate the *k_nearest_dist* buffer as the inner loops (lines 21 and 26).
2. To enable fine-grained data parallelism and pipeline parallelism, inside each loop iteration i (line 16), we split the compare-and-swap loop into two j loops. For the first j loop (lines 20-24), it compares-and-swaps elements with their next neighbor. For the second j loop (lines 25-29), it compares-and-swaps elements with their previous neighbor. Both loops increment j by a step of two. As a result, we can fully parallelize (unroll) both j loops. Moreover, we should be able to pipeline the i loop with an II of 2; however, in CHIP-KNNv1, due to Vitis HLS tool limitation, it only achieved an II of 3.

To explore coarse-grained parallelism, we further divide the *point_dist* buffer into *sort_factor* partitions and all partitions sort their own top K results in parallel. After processing all tiles within the PE, a local merger within the *Top_K_Sort* function is used to merge *sort_factor* copies of top K results buffered on chip, which has an algorithmic complexity of O(*sort_factor**K). Since *sort_factor* is much smaller than N, the execution time of this local merger is negligible. In cases of high-dimensional feature vectors, this coarse-grained parallelism optimization is not needed since the *Top_K_Sort* stage runs much faster than the other two stages. In Section 3.8, our automation tool will decide whether the coarse-grained parallelism optimization and the corresponding local merger is needed, and if yes, it will choose the optimal *sort_factor* to balance the execution of the three stages in each PE.

Proof of the Top_K_Sort algorithm. Finally, we prove the correctness of our novel hardware-friendly sorting algorithm. To get started, $k\_nearest\_dist[1:K]$ swaps in the first K distances from $point\_dist[0:K-1]$ after the first K iterations of the i loop. For any following loop iteration $i >= K$ (line 16), it compares $k\_nearest\_dist[1:K]$ (i.e., current top K distances) and $k\_nearest\_dist[0]$

(i.e., incoming $point\_dist[i]$) so that the largest distance is always swapped to $k\_nearest\_dist[0]$. This is guaranteed because:

1. If the largest distance is $k\_nearest\_dist[0]$ in iteration i, it is already there and does not need any swapping.
2. If the largest distance is newly introduced in iteration i', i.e., $i - K < i' < i$, then the furthest position this largest distance can go is $i - i'$. At the same time, in iteration i, it has already gone through $i - i'$ compares-and-swaps. Therefore, it is guaranteed to arrive at $k\_nearest\_dist[0]$.
3. If the largest distance was in $k\_nearest\_dist[1:K]$, in iteration $i - K$ or earlier, it has already gone through K compares-and-swaps to arrive at $k\_nearest\_dist[0]$.

In summary, $k\_nearest\_dist[1:K]$ always keeps the K smallest distances. The final extra K iterations (line 16) are used to ensure that the final $k\_nearest\_dist[1:K]$ are sorted from the largest to the smallest.

3.1.4 ***Ping-Pong Buffer.*** Finally, we use the Ping-Pong buffer technique [9, 10] to execute the *Load_Buf*, *Dist_Cal*, and *Top_K_Sort* stages in a coarse-grained pipeline. We call these three stages together a single processing element (PE).

3.2 Streaming-based Single-PE KNN Design

To further improve resource utilization and design frequency, we implement another design variant —CHIP-KNNv2—a streaming-based architecture which reuses most of the processing modules (except the intra-PE merge unit) from CHIP-KNNv1. The processing modules of CHIP-KNNv2 are constructed in a dataflow fashion by making the following changes, as shown in Figure 1.

1. In the distance compute module, we buffer the query point, and stream all search space points once to compute their distances from the query point;
2. In the sort module, we buffer the K nearest distance-id pairs, and stream new distance-id pairs from distance compute module through multiple parallel compare units to swap with the buffered K nearest distance-id pairs;
3. In the merge module (will be detailed in Section 3.4), we stream from the lowest distance value to the highest distance value elements from each *k_nearest* buffer for comparison, and merge the intermediate results into a single set of K nearest distance-id pairs.

CHIP-KNNv2 also provides an extra configuration to allow users to specify different *datatype*s of the search space points, including 8-bit, 16-bit, and 32-bit fixed-point, as well as 32-bit floating-point data types. Compared with the buffer-based CHIP-KNNv1 design, our CHIP-KNNv2 architecture eliminates the high on-chip buffer (BRAM and URAM) usage and their associated control logic overhead, and leverages the resource efficient FIFO channels to link the processing modules together.

3.3 Multi-PE Scaling

To better scale the design to utilize the computing resource from multiple SLRs (FPGA dies) and off-chip bandwidth from multiple HBM banks, CHIP-KNN further exploits the task-level parallelism by instantiating multiple PE instances to process different partitions of the search space in parallel. We denote the number of PEs as P, which will be generated by our automation tool in Section 3.8.

Since each PE only produces the partial top_K result, we add an inter-PE merge unit, to merge the P copies of top-K contenders to the global K-nearest neighbors. This inter-PE merger only needs to execute once after all PEs finish processing their section of the search space. Its execution time is negligible if the data access to all the local top_K results are on chip. This merge unit is discussed further in Section 3.4.

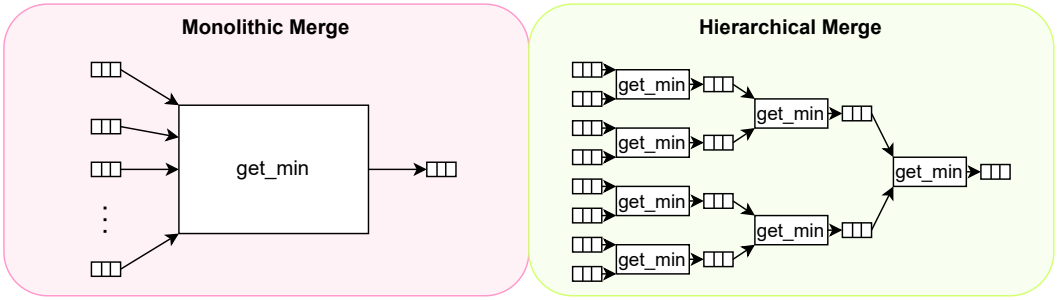


Fig. 2. Architecture diagrams of monolithic merging unit and hierarchical merging unit.

3.4 Hierarchical Tree-based Merge Module Design

To alleviate the routing congestion of our accelerator design, we investigate and rethink the design of our merge module which draws values from multiple input sources and funnels them through based on their comparison results. For our CHIP-KNN design, as input, our merge module takes multiple K nearest neighbour results from different sorting modules (i.e., intra-PE merging) or PE modules (i.e., inter-PE merging). The output is a single set of K nearest neighbours results.

Figure 2 shows architecture diagrams for two implementations: a monolithic merging unit, and a hierarchical merging unit. Here, we see that the monolithic merging unit features centralized logic, requiring many wires to route into a small area. On the other hand, the hierarchical merging unit features distributed pairwise merge nodes, connected in a tree-like fashion. These nodes could be distributed across the board, decreasing congestion in comparison to the monolithic merging unit.

To evaluate the end-to-end frequency gain from the hierarchical merge structure, we have collected and compared results across all of the dimensional configurations listed in Table 1 in Section 4.1, with K=10 and N=4M. The results show an average frequency improvement of 2.1%, and up to 5.2% frequency improvement. In fact, for one of the evaluated designs, the monolithic merge-based design failed to route, whereas the hierarchical merge-based design was successfully placed and routed, achieving the desired frequency of 225MHz.

3.5 Timing Closure and Floorplanning Optimization

Meeting timing constraints is a major challenge for many high-performance FPGA accelerator designs on multi-die datacenter FPGAs [12, 18, 44]. CHIP-KNN also suffers from this, as we have experienced in our CHIP-KNNv1 paper [27]. To address the timing closure issue, we have integrated the open-source TAPA/AutoBridge framework [8, 18] into our CHIP-KNN framework to build our streaming-based CHIP-KNNv2 multi-PE design.

TAPA/AutoBridge is a high-performance fast-compiling HLS framework that is fully compatible with the AMD/Xilinx Vitis/Vivado toolflow. It generally provides three main benefits. First, to improve the timing closure issue for dataflow programs where tasks communicate via FIFO streams, it automatically applies coarse-grained floorplanning optimizations: it constrains each task within a local FPGA region to improve local placement and routing, and inserts pipeline registers between local regions (i.e., dataflow tasks) to shorten global routing paths. This greatly improves the design build success rate and the final design frequency. With this integration, we are able to generate high-frequency CHIP-KNN accelerators. Second, it also reduces BRAM resource utilization for the standard Xilinx AXI interface by using a more lightweight resource-efficient streaming AXI interface to access off-chip memory. This is particularly helpful for the HBM-based U280 FPGA since the significant BRAM consumption on the bottom die of the FPGA (due to multiple AXI interfaces)

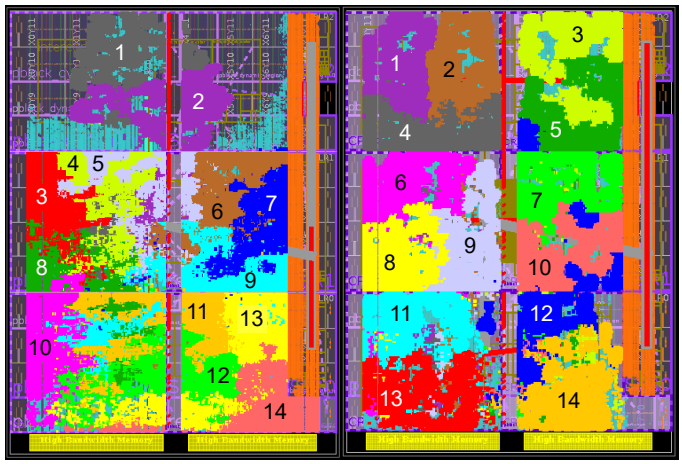
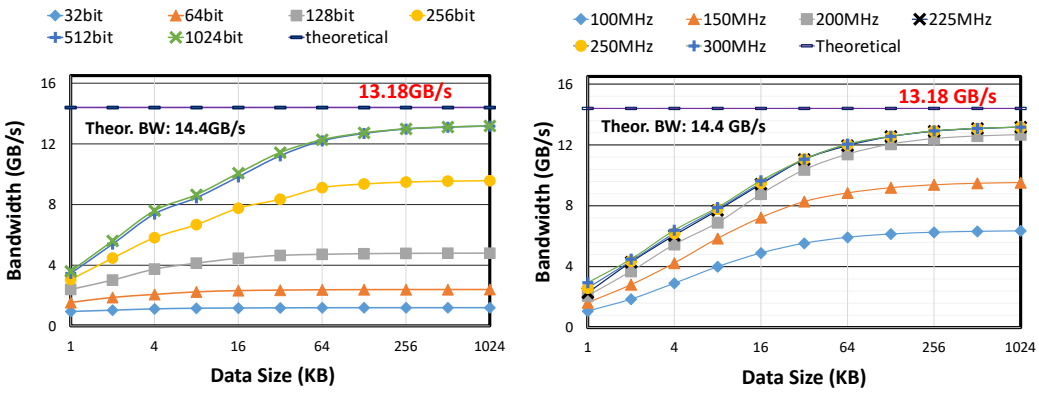


Fig. 3. Floorplanning results of CHIP-KNN designs: a) the CHIP-KNNv2 design without TAPA/AutoBridge at 104MHz, and b) the CHIP-KNNv2 design with TAPA/AutoBridge at 227MHz. The CHIP-KNNv1 design failed to place, even with only 12 PEs. Please note different colours represent different PEs. All designs use configuration parameters: N=4,194,304, K=10, D=2, Metric=Euclidean, with float32 data.

often causes timing violations and poor place-and-route results. This also saves resources for useful computations and alleviates timing constraints. Third, it provides a user-friendly programming model (called TAPA) for task-parallel dataflow programs in C++ by abstracting away the complex OpenCL API calls to support task parallelism. To leverage the benefits of TAPA/AutoBridge, as will be detailed in Section 3.8, CHIP-KNN automatically generates the TAPA HLS design and host code.

To illustrate the floorplanning differences, Figure 3 shows the same CHIP-KNNv2 design with and without Autobridge. The design configurations shown are as follows: $N = 4M$, $K = 10$, $D = 2$, Euclidean distance metric, with 32-bit single-precision floating-point data. The PEs are numbered in the figures, and are highlighted in different colors. The buffer-based CHIP-KNNv1 design is not included in Figure 3, because this design failed to place with even 12 PEs. This is due to the fact that the buffer-based CHIP-KNNv1 design uses considerably more resources than the streaming-based CHIP-KNNv2 design, discussed further in Section 4.4. Figure 3a shows the streaming-based CHIP-KNNv2 design without using AutoBridge, which achieved a kernel frequency of 104 MHz. Figure 3b shows the streaming-based CHIP-KNNv2 design with AutoBridge, which achieved a kernel frequency of 227 MHz. Comparing between these two figures visually, we see that AutoBridge organizes the PEs significantly better, reducing long delay paths, and grouping the logic together more tightly. For example, in Figure 3a, we see PEs 2, 5, 11, and 12 all straddling the central reserved column of the FPGA board, which causes additional logic delays. We also see that without coarse-grained floorplanning, Vivado decides to overlap the logic for different PEs, as seen in PEs 4, 5, and 8. Meanwhile, Figure 3b shows better clustering of logic. Only PE 12 seems to be significantly spread across the board, but the TAPA-inserted FIFOs enable this sort of floorplanning without a harsh frequency drop. We also notice better spreading of the logic - whereas Vivado alone clusters far more logic at the bottom of the board (close to the HBM banks) and leaves the top unused, Vivado with the guidance of AutoBridge more evenly distributes logic across the entire board.



(a) (b)

Fig. 4. Read bandwidth of a single HBM bank on Alveo U280 FPGA, with different consecutive data access sizes: a) shows the effective bandwidth at different AXI port widths ranging from 32-bit to 1024-bit; b) shows the effective bandwidth of a 512-bit AXI port at different kernel frequencies ranging from 100MHz to 300MHz [28]. Note the x-axis is plotted in log_2 scale.

3.6 Off-Chip Bandwidth Characterization and Optimization

To optimize the data access between off-chip memory and on-chip buffers for each PE, we follow the method described in [28] to characterize the effective memory access bandwidth of a single HBM bank on the Alveo U280 FPGA board [39]. Shown in Figure 4a, as the memory access port width increases and the consecutive data access size increases, the effective bandwidth increases. While the peak theoretical bandwidth of a single HBM bank is 14.4GB/s, the peak effective bandwidth is only 13.18GB/s and can only be achieved when the port width is no less than 512bits and the consecutive access size is no less than 128KB. Therefore, our CHIP-KNNv2 automation framework selects a 512-bit port width. Another important key bandwidth impacting factor is the kernel frequency as shown in Figure 4b. For a 512-bit port, the effective bandwidth increases as the kernel frequency increases until 225MHz, since the HBM stack on U280 operate at 450MHz, and the physical AXI port width is 256-bit. Therefore, our CHIP-KNN designs only need to achieve 225MHz clock frequency to exploit the effective off-chip bandwidth. When using all 32 HBM banks, the maximum achieved bandwidth is approximately 421.6 GB/s.

3.7 Analytical Performance Model

To guide our automation tool to select the optimal design points, we build an analytical performance model to calculate the latencies for all three stages— *Load_Buf*, *Dist_Cal*, and *Top_K_Sort* in Algorithm 1—that execute in a dataflow fashion. The goal is to balance the execution latencies of these three stages within each PE.

For any pipelined function, the total execution time is calculated as follows:

$$Latency = (pipe\_iterations - 1) * II + pipe\_depth \quad (4)$$

where $pipe\_iterations$ is the number of pipeline iterations, II is the initiation interval, and $pipe\_depth$ is the pipeline depth. Next, we apply the performance model from equation 4 to the buffer-based implementation. Note that the latencies of the merge units are negligible, as explained in Section 3.1 and 3.3. Therefore, we do not model them. However, their execution time is included in our experimental results in Section 4.

3.7.1 ***Buffer-based Architecture Performance Model.*** In order to balance the execution times of our three modules, we formulate an analytical performance model for each module.

Buffered_Load. The latency to load one tile is:

$$Load = [(buf\_size/port\_width - 1) + depth_{ld}] * \text{effective\_BW\_factor}(buf\_size,\ port\_width) \tag{5}$$

Here, the burst read achieves an II of 1, and each load reads $port\_width$ size of data. Therefore, it needs $buf\_size/port\_width$ number of loads. $depth_{ld}$ is the fixed initialization overhead that can be retrieved from one-time HLS synthesis. By default, Vitis HLS assumes an ideal linear bandwidth scaling with the $port\_width$ and does not consider the effective memory bandwidth that we have characterized in section 3.6. To make it more accurate, we introduce $effective\_BW\_factor = theoretic\_BW/effective\_BW$ to adjust the load latency based on buf_size and $port\_width$.

Buffered_Dist_Calc. In order to define the latency to calculate distances, we first need to determine the number of buffered distances, B, that we have to compute.

$$B = buf\_size/(D * bytes\_in\_datatype) \tag{6}$$

As D or $bytes\_in\_datatype$ grows, the number of distances that must be computed and buffered for each tile decreases. Now, the latency to calculate distances for one tile is:

$$Dist\_Lat = (B/dist\_factor - 1) * dist\_II + depth_{dist} \tag{7}$$

The $dist\_factor$ and $dist\_II$ can be adjusted to tune the latency of this function. The corresponding $depth\_dist$ can be inferred from the pipeline depth value when dist_factor=1 and dist_II=1, which can be retrieved from one-time HLS synthesis.

Buffered_Top_K_Sort. The latency to sort distances for one tile is:

$$Sort\_Lat = ((B + K)/sort\_factor - 1) * 3 + depth_{sort} \tag{8}$$

which is similar to Equation 7, except that the II is fixed as 3 (due to Vitis HLS limitation). The $sort\_factor$ can be adjusted to tune the latency. The $depth_{sort}$ can be inferred from one-time HLS synthesis.

3.7.2 ***Streaming-based Architecture Performance Model.*** For the streaming-based architecture, to achieve an overall II of 1 for the system, we analyze the II of each module, rather than the latency. This is because the overall system throughput is bottlenecked by the module with the largest II. For our design, we adjust the unroll factors of both the load module and distance calculation module to achieve an II of 1. Note that for the load module, our effective bandwidth factor no longer considers the buffer size or search space size, because the search space is typically large and is streamed in to have a large burst access size, which maximizes the memory bandwidth, as explained in Section 3.6. Additionally, following the characterization in Section 3.6, our port width is always chosen to be 512 bits. The streaming-based sorting module achieves an II of 2 as a result of the bottom-up synthesis, provided by our TAPA [8] integration. Therefore, we instantiate twice as many sorting modules, and stagger their computations to achieve an overall II of 1.

3.8 Automation Support for CHIP-KNN

To support flexible KNN designs and eliminate the laborious design space exploration, for a given user configuration of KNN parameters and a target FPGA platform, we develop a design automation tool. Figure 5 shows the automation toolflow for both design versions: CHIP-KNNv1 and CHIP-KNNv2, to generate the optimal accelerator design that best utilizes the off-chip memory bandwidth under the available FPGA resource limit.

3.8.1 ***CHIP-KNNv1 Automation Support.*** First we discuss the automation support for CHIP-KNNv1, which features the buffer-based architecture explained in Section 3.1. The automation support is illustrated in Figure 5.

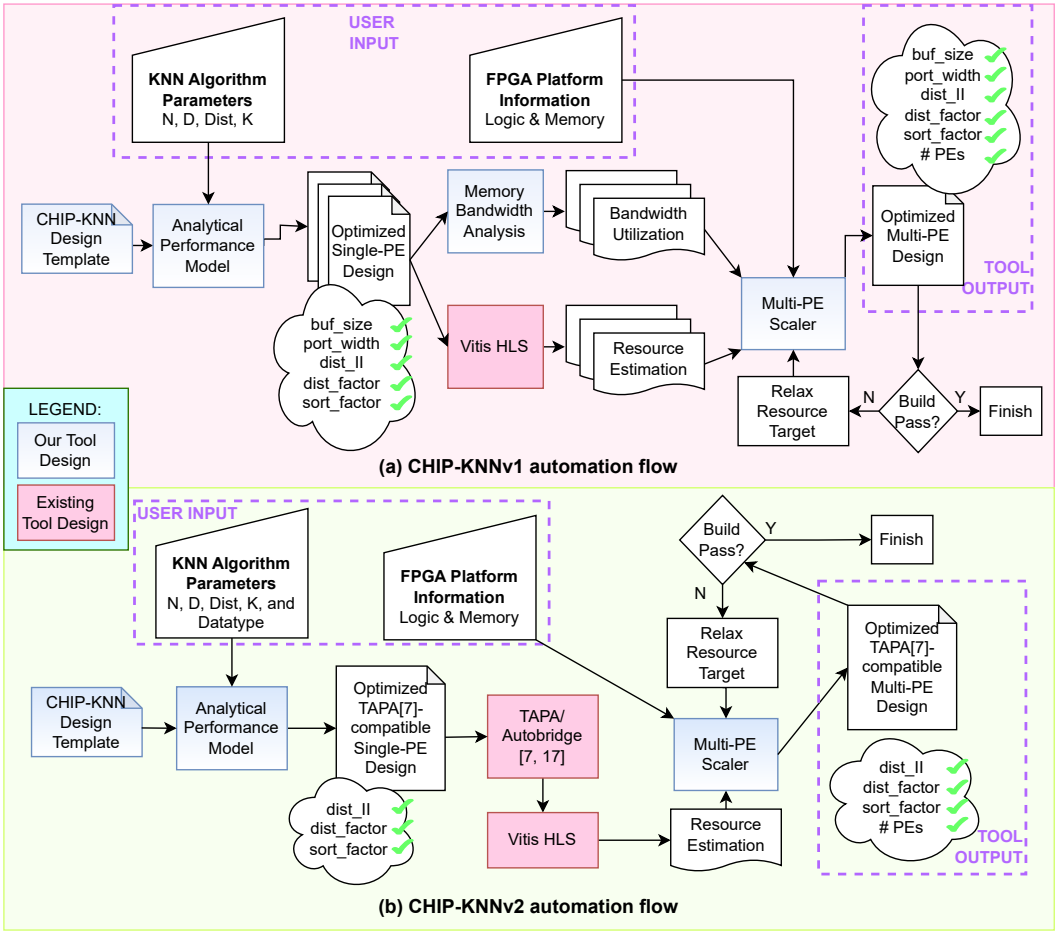


Fig. 5. Flowchart of design automation for (a) CHIP-KNNv1 and (b) CHIP-KNNv2.

1. Given a user configuration of *N*, *D*, *K*, and *distance metric* as input, our automation tool first generates a collection of balanced single-PE KNN designs based on our CHIP-KNNv1 design template. To explore the design space, we vary the *buf_size* of 64KB and 128KB and *port_width* of 256bits and 512bits. This is because the best performing single PE design that has balanced workloads and can achieve the best effective bandwidth of a single memory bank (i.e., *local_search_space* no less than 128KB and *port_width* no less than 512bit) sometimes cannot utilize the most off-chip bandwidth due to resource or routing constraint as the number of PEs scales. Based on our performance model from Section 3.7.1, we generate the corresponding balanced *dist_factor*, *dist_II*, and *sort_factor* parameters of the three design stages.
2. For each balanced single-PE design, our tool determines its resource utilization using Vitis HLS synthesis and its bandwidth utilization based on the memory bandwidth characterization in Section 3.6.
3. In the optimal design configuration search step, we first scale the number of PEs for each balanced design choice by taking the available logic and memory resource of the FPGA platform as input. Since each PE occupies one single off-chip memory bank in CHIP-KNNv1, we determine

the maximum number of PEs that can fit onto the FPGA as:

$$\#PEs = min(\#off\_chip\_memory\_banks, \alpha * FPGA\_resource/PE\_resource) \quad (9)$$

where α is a coefficient that is initially set as 70%, since a typical design that uses more than 70% of the FPGA resource is very hard to pass the placement and routing.

4. Based on the maximum number of PEs and the bandwidth utilization results for each PE, we can decide the total bandwidth utilization for each multi-PE design choice. Our tool chooses the design that achieves the highest bandwidth as the final optimal design point, and generates the final design with a set of parameters including *buf_size*, *port_width*, *dist_factor*, *dist_II*, *sort_factor*, *#PEs*, and *#tiles per PE* (i.e., number of buffered tiles per PE).
5. Finally, we build the generated optimal design with Xilinx Vitis 2019.2 tool [40]. If it is successfully built, it ends with the selected design. Otherwise if the design fails the placement and routing, we relax the current α by 5% (i.e., using 5% less resource) and repeat step 3 to 5 again until the design can be successfully built. Our experiments show that at most we need to take 5 iterations until $\alpha = 50\%$ to find the final optimal design that can be successfully built.

3.8.2 ***CHIP-KNNv2 Automation Support.*** With CHIP-KNNv2, which uses the streaming-based single-PE design explained in Section 3.2, the automation support is simplified, faster, and more configurable. This is shown in Figure 5. In this subsection, we discuss the key differences between the automation support in CHIP-KNNv2 and CHIP-KNNv1.

1. Increased configurability: end-users can now select a target datatype, which can be either 32-bit single-precision floating-point, 32-bit fixed-point, 16-bit fixed-point, or 8-bit fixed-point.
2. Faster design space exploration: rather than synthesizing several pareto-optimal designs with different combinations of buffer-size/port-width, we simplify the automation to target a single design with the optimal port width of 512 bits for the Alveo U280 FPGA, as identified in [28]. Furthermore, the buffer size is no longer a consideration, as we have moved to a streaming-based architecture.
3. Improved placement and routing: our automation tool now generates the kernel and host code in TAPA HLS C [8], which can leverage the coarse-grained floorplanning and pipelining optimizations by AutoBridge [18] to improve the timing closure. As explained in Section 3.5, with this integration, the achievable frequency increases significantly.

4 EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Experimental Setup

KNN configuration. We evaluate CHIP-KNN with a wide range of parameter configurations listed in Table 1. The size of the search space (N) ranges from 2 million to 8 million data points, the number of dimensions (D) ranges from 2 to 128, and the datatype includes 8-bit fixed-point, 16-bit fixed-point, 32-bit fixed-point, and single-precision floating-point. Therefore, the total size of the search space data, which is $N * D * bytes\_in\_datatype$ bytes, ranges from 4MB (when *N=2M*, *D=2*, and datatype is 8-bit fixed-point) to 4GB (when *N=8M*, *D=128*, and datatype is 32 bits wide). The K value we evaluate ranges from 5 to 20, since typically small *K*s are used in real-world applications. For distance metrics, both Manhattan and Euclidean distances are evaluated. All the results are presented for a single input query.

Hardware platform and software tool. To evaluate the CPU implementation, we use a dual-socket 14nm Xeon Silver 4214 CPU (with a total of 24 cores and 48 hyper-threads) and 196GB DRAM. Our CPU system has a maximum memory bandwidth of 160 GB/s, measured using Intel's Memory Latency Checker tool [22]. Our baseline CPU implementation of KNN was adapted from the Rodinia benchmark suite [7]. The query data are stored using vector arrays and the processing

Table 1. Evaluated key KNN parameter configurations. *Note that Datatype is configurable in CHIP-KNNv2, and CHIP-KNNv1 only supports float32.

KNN Parameters	Values
N: Number of data points in the searchspace	2M, 4M, 6M, 8M
D: feature dimension	2, 4, 8, 16, 32, 64, 128
K	5, 10, 15, 20
Distance metric	Manhattan, Euclidean
Datatype*	float32, 32-fixed, 16-fixed, 8-fixed

is parallelized using 48 threads (i.e., all available hardware threads), and compiled with gcc *-Ofast* optimization flag, which automatically explores the vectorization optimization. To benchmark our CPU design, we compare it against state-of-the-art FAISS CPU implementation [23] across all design configurations described in Table 1. Our results show that, in terms of latency (end-to-end time per query), our CPU implementation outperforms FAISS from 3.3x to 100x (depending on the dimension). This is because FAISS prioritizes throughput rather than latency. On the other hand, in terms of throughput (queries-per-second), our CPU implementation is competitive with FAISS, achieving 3.3x the throughput at low dimensions, and 0.76x at high dimensions.

For the GPU implementation, we use the state-of-the-art FAISS GPU [23] implementation, and run experiments on an NVIDIA Tesla V100 with 16GB DRAM. We evaluate our CHIP-KNN accelerator designs on the 16nm AMD/Xilinx Alveo U280 datacenter FPGA board (with 32 HBM2 banks) [39] as described in Section 2.2. Both CHIP-KNNv1 and CHIP-KNNv2 accelerators are entirely developed using Vitis HLS. We build our CHIP-KNNv1 designs using Xilinx Vitis 2019.2, and CHIP-KNNv2 designs using the recent version of TAPA (Ver.0.0.20220807.1), which requires Xilinx Vitis 2020.2 [40].

4.2 Overall CHIP-KNN Speedup Over CPU

Figure 6 summarizes the geometric mean of performance improvement results of CHIP-KNN accelerated on an Alveo U280 FPGA over the 48-thread CPU version across all configurations in Table 1. On average, for supporting the 32-bit single-precision floating-point data representation, our buffer-based CHIP-KNNv1 design achieves 5.23x speedup while streaming-based CHIP-KNNv2 design achieves 8.52x speedup. Taking into account all of the supported data representations in the CHIP-KNNv2 design, it achieves 15.49x speedup over the CPU implementation.

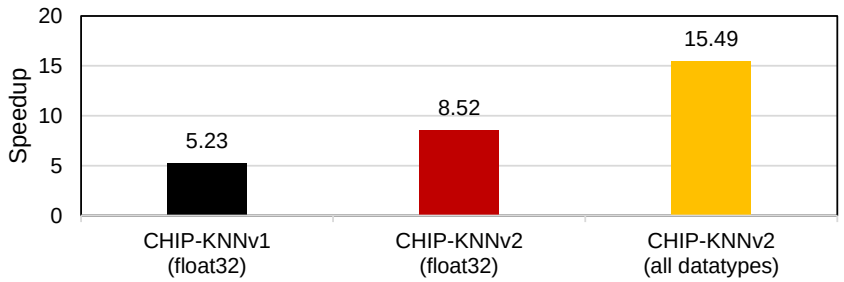


Fig. 6. Geometric mean of speedup over CPU across all configurations in Table 1.

Figures 7 and 8 showcase the latency and throughput comparison between CHIP-KNNv2 and the 48-thread CPU implementation, with floating-point data type. As shown, CHIP-KNNv2 outperforms

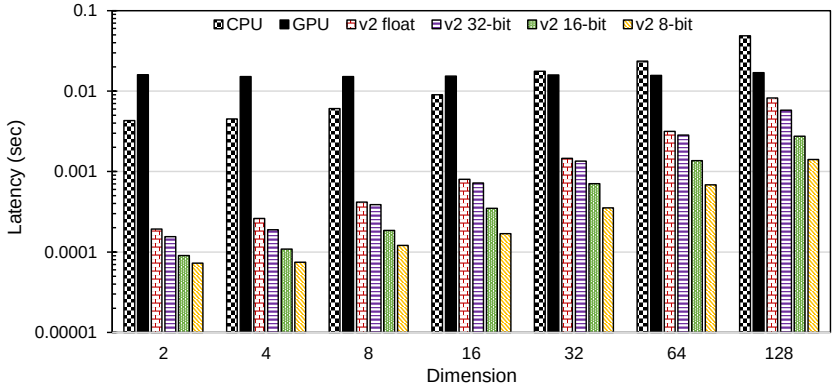


Fig. 7. Single-query latency comparison between CHIP-KNNv2 , CPU, and FAISS-GPU, with N=4M, K=10, and Euclidean distance. CPU and GPU implementations use floating-point data.

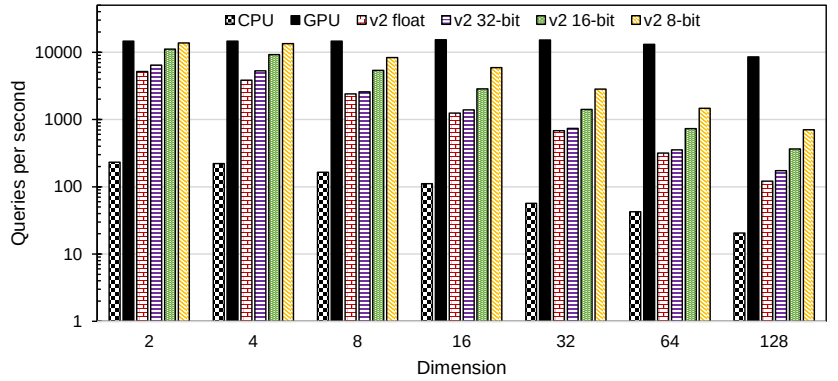


Fig. 8. Throughput comparison between CHIP-KNNv2 , CPU, and FAISS-GPU, with N=4M, K=10, and Euclidean distance. CPU and GPU implementations use floating-point data.

the CPU in both latency and throughput at each evaluated configuration, from 22.2x improvement with 2D data, to 5.9x improvement with 128D data.

In fact, our accelerator also achieves a lower power usage than the CPU implementation at each configuration. Figure 9 shows the throughput-per-watt comparison. Our results show that our accelerator, using floating-point data, is 43.8x more energy efficient with 2D data, and 11.8x more energy efficient with 128D data. On average, the power usage of CHIP-KNNv2 is 51.2% of that on the CPU, due to the efficient custom design on FPGA.

4.3 Comparison Between CHIP-KNN and GPU

First, our FPGA design outperforms FAISS-GPU implementation in latency, as shown in Figure 7. This is because our FPGA accelerator design optimizes for latency by exploiting data parallelism in both distance calculation and distance sorting, as well as the pipeline parallelism between each stage. In contrast, the FAISS' GPU implementation optimizes for the overall throughput by concurrently processing multiple queries in batches, with the searchspace data loaded once and shared for each batch. In this way, the GPU outperforms CHIP-KNNv2, shown in Figure 8. Figure 9 presents the throughput-per-watt comparison. For lower-dimension designs, CHIP-KNNv2 is competitive to FAISS' GPU implementation in throughput-per-watt. This is because the GPU is less able to

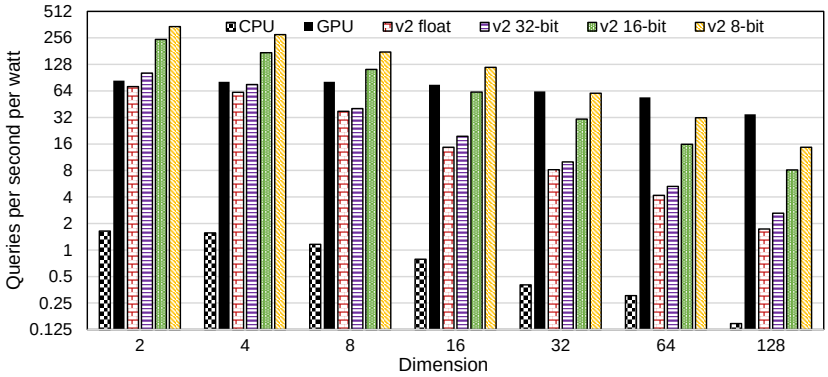


Fig. 9. Throughput-per-watt comparison between CHIP-KNNv2, CPU, and FAISS-GPU, with N=4M, K=10, and Euclidean distance. CPU and GPU implementations use floating-point data.

exploit high parallelism in the low-dimension computation. However, as the dimensions increase, CHIP-KNNv2 is outperformed by FAISS-GPU.

More specifically, our FPGA design is outperformed by FAISS-GPU in throughput for the following reasons. Firstly, the V100 GPU has more computation resources running at a higher frequency, and thus has more parallel computation power. Secondly, the FAISS design re-uses loaded searchspace points across a batch of queries, which allows for more computation-per-byte. This design is not feasible for the FPGA, due to the relatively limited computing resources. Lastly, the V100 GPU has a theoretical memory bandwidth of 900 GB/s, which is almost twice that of the U280 FPGA at 460 GB/s.

In Table 2, we present the average performance across all the floating-point configurations listed in Table 1. As shown, CHIP-KNNv2 achieves an average latency improvement of 30.6x over the GPU design, for processing a single query. On the other hand, CHIP-KNNv2 on Alveo U280 FPGA achieves 13.5% of the throughput that the GPU design achieves when processing a batch of 4,096 queries. In terms of power usage, CHIP-KNNv2 uses 34.3% of the power that the GPU implementation requires.

Table 2. Average latency, throughput, and power usage comparison for CHIP-KNNv2 on U280 FPGA and GPU design [14, 15] on V100, all using floating-point data type.

Latency	Throughput	Power Usage
30.6x faster than GPU	13.5% of GPU	34.3% of GPU

4.4 Comparison Between CHIP-KNNv1 and CHIP-KNNv2

To better understand the performance and resource differences between the two design variants, CHIP-KNNv1 and CHIP-KNNv2, we first present the single-PE resource utilization comparison results, where both designs achieve the same performance. Figure 10 compares the resource usages—the LUT, FF, BRAM, URAM, and DSP utilization of a single PE—for one configuration with D=4, N=4M, K=10, and using Euclidean distance metric on floating-point data points. Our experiments show that other configurations have similar results. For a fair comparison, we generated both designs using Vitis v2019.2, rather than generating CHIP-KNNv1 on Vitis v2019.2, and CHIP-KNNv2 on Vitis v2020.2. Note that lower bar indicates improved resource efficiency. Across all resources except for DSP (since both versions have the same processing capability), CHIP-KNNv2 design requires fewer resources than CHIP-KNNv1 design. Especially for the on-chip memory (i.e., BRAM

and URAM), it is almost eliminated in CHIP-KNNv2 due to its streaming-based architecture; the remaining small amount of BRAM usage is mainly consumed by the AXI interfaces connecting the HBM banks. The resource reduction in LUT and FF mainly come from removing the associated control logic for the on-chip memory components.

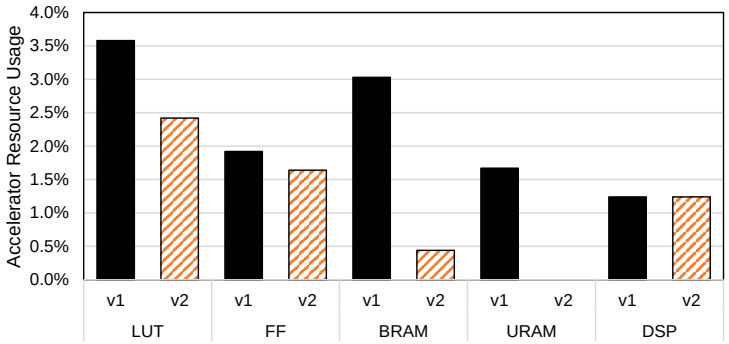


Fig. 10. Resource usage comparison of a single-PE accelerator design between CHIP-KNNv1 and CHIP-KNNv2, with D=4, N=4M, K=10, float32 data, and Euclidean metric. Both versions achieve the same performance.

For the overall multi-PE performance comparison, Figure 11 shows throughput results of the two design variants; the throughput is measured as effective off-chip bandwidth (GB/s) as both CHIP-KNN designs are bandwidth bound. To better illustrate the performance difference, we sweep through different feature dimensions ranging from 2D to 128D with N=4M, K=10, and using Euclidean distance metric on floating-point data points. For low feature dimensions (i.e., 16D and lower), CHIP-KNNv2 perform better than CHIP-KNNv1 designs mainly because of the improved resource utilization from the streaming-based design. For high feature dimension (i.e., 32D and higher) designs, CHIP-KNNv1 designs become bounded by the available computing resource and severely suffer from the poor placement and routing quality; whereas CHIP-KNNv2 designs integrated with TAPA/AutoBridge are effective to mitigate the design timing closure issue and achieve an average throughput improvement of 1.83x over CHIP-KNNv1.

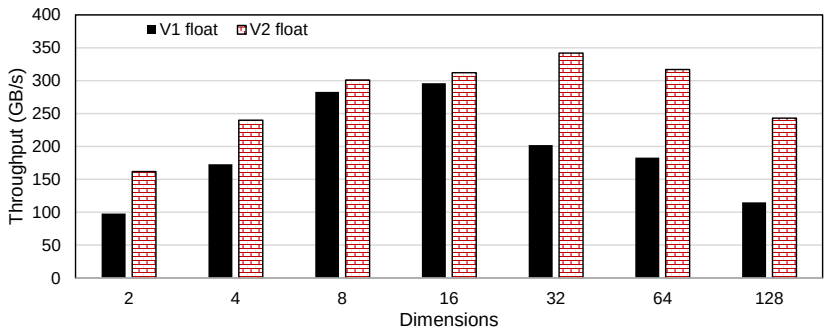


Fig. 11. Throughput for different feature dimensions with N=4M, K=10, and Euclidean distance.

4.5 Speedup for Different KNN Configurations

To provide further details on the performance of CHIP-KNN against the optimized CPU implementation, we present speedup results for different data representations when varying each of the essential parameters in D, K, distance metric, and N.

4.5.1 Speedup for Different Feature Dimensions. Figure 12 presents the speedup of CHIP-KNN designs over the 48-thread CPU implementation at different feature dimensions (D=2, 4, ..., 128) with N=4M, K=10, and Euclidean distance. For designs with 32-bit data types (i.e., float32 and 32-bit fixed-point), the speedup generally decreases as the feature dimension increases. This is because the CHIP-KNN designs are bound by the off-chip bandwidth, and the placement and routing. On the other hand, for higher feature dimensions, the CPU implementation can better utilize vectorization instructions and thus its execution time increases sub-linearly. For low-precision data types (i.e., 16-bit and 8-bit fixed-point), the achieved speedups are much higher. While the performance of CHIP-KNN designs follows a similar trend as that of the 32-bit data type designs, the CPU architecture is less optimized than our FPGA designs for processing the 8-bit and 16-bit data points. The speedup fluctuation for different feature dimension is partially caused by variation in the achievable number of PEs that affects the final bandwidth utilization, and partially due to inherent noise in the CPU runtimes. We will analyze the accelerator efficiency in Section 4.6. We also notice an anomalous speedup increase at the 32-dimensional configuration. This is due to the CPU runtime increasing significantly over the 16 dimensional implementation.

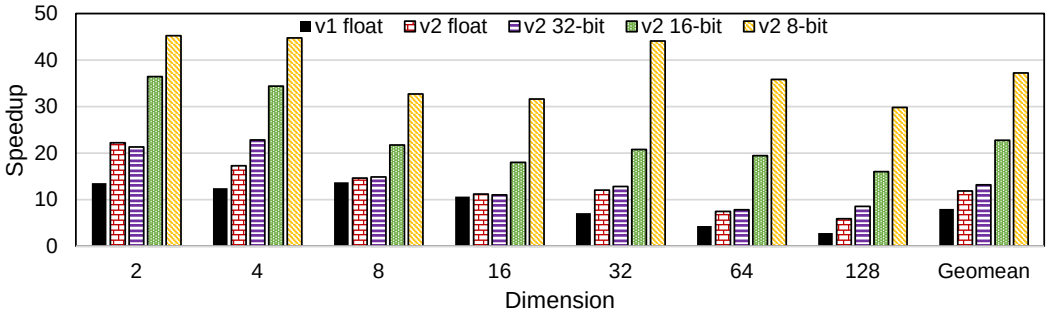


Fig. 12. Performance speedup for different feature dimensions with N=4M, K=10, and Euclidean distance.

4.5.2 Speedup for Different Ks. Figure 13 shows the speedup of CHIP-KNN designs for K=5, 10, 15, and 20, with N=4M, D=64, and Euclidean distance. When K increases, CHIP-KNN designs for 32-bit floating-point and fixed-point data types achieve slightly higher speedup. This is because, with a larger K, the FPGA accelerator performance remains almost the same according to Algorithm 1, while the CPU execution time slightly increases in the top_K_sort stage. The speedup slightly decreases for 16-bit fixed-point and varies for 8-bit fixed-point CHIP-KNNv2 designs due to the heuristic nature of TAPA/AutoBridge's floorplanning optimizations.

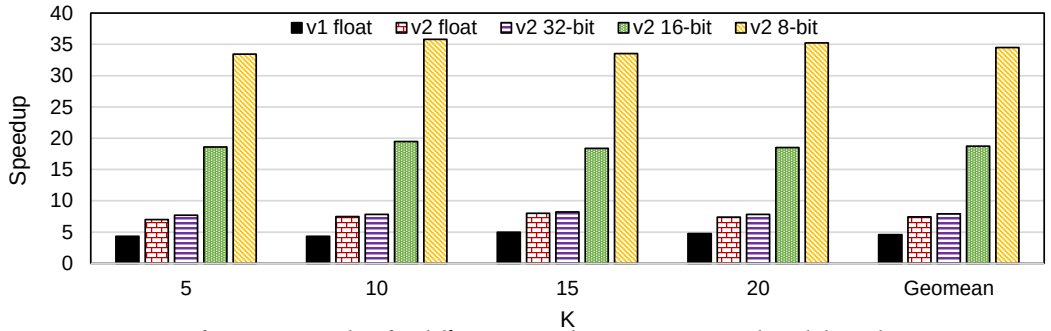


Fig. 13. Performance speedup for different Ks with N=4M, D=64, and Euclidean distance.

4.5.3 Speedup for Different Distance Metrics. Figure 14 shows the speedup of CHIP-KNN designs for Manhattan and Euclidean distances with N=4M, D=64, and K=10. Across all data types, CHIP-KNN designs achieve nearly identical execution time for both Manhattan and Euclidean distance metrics. However, for the CPU implementation, the version with Manhattan distance metric takes longer to execute, due to overhead in the branch instruction (i.e. branch prediction) from the absolute-value math function. This effect is exaggerated in graph, due to the nature of the speedup computation: $speedup = CPU\_runtime/FPGA\_runtime$. Since the 8-bit design runs very quickly, the denominator is small. Therefore, an increase in the CPU runtime is most noticeable in the speedup value for the 8-bit design.

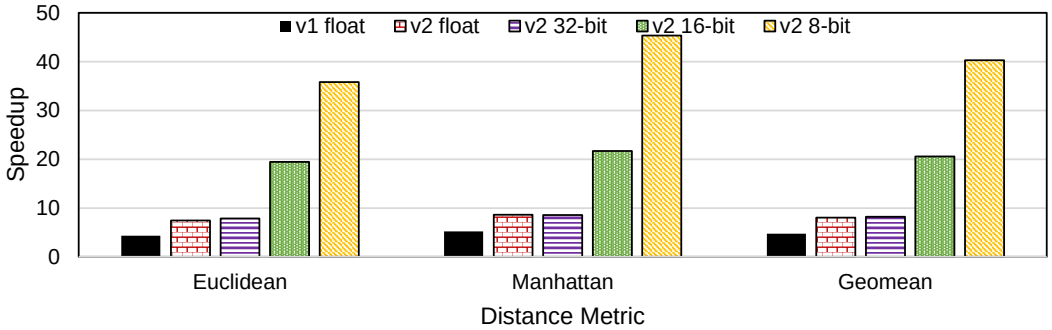


Fig. 14. Performance speedup for different distance metrics with N=4M, K=10, and D=64.

4.5.4 Speedup for Different Dataset Sizes. Figure 15 shows the speedup of CHIP-KNN designs for N=2M, 4M, 6M, and 8M, with D=64, K=10, and Euclidean distance. Our CHIP-KNN designs achieve a consistent speedup, due to the fact that the runtime scales linearly with input size, on both CPU and FPGA.

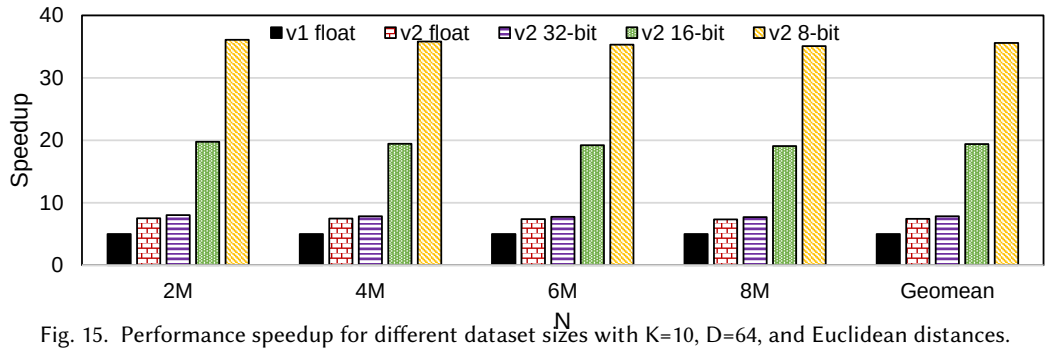


Fig. 15. Performance speedup for different dataset sizes with K=10, D=64, and Euclidean distances.

4.6 Accelerator Efficiency Analysis

To better illustrate the efficiency of our CHIP-KNN designs on the U280 FPGA and explain the speedup fluctuation results in Section 4.5, we further analyze those designs with D=2, 4, ..., 128, N=4M, K=10, and Euclidean distance. Table 3 summarizes the resource utilization, execution time, throughput, and bandwidth utilization of the designs generated using our automation framework.

Dimension	Version	Resource Usage (%)					#HBMs Used	#PEs	Throughput (GB/s)	Bandwidth utilization (%)	Runtime (ms)	Freq. (MHz)
		LUT	FF	BRAM	URAM	DSP						
2	v1	67	39	60	30	12	9*	18*	98	21.3	0.32	218
	v2	64	36	10	0	22	14	14	162	35.2	0.19	227
4	v1	68	41	58	27	20	16	16	173	37.6	0.36	228
	v2	61	37	10	0	30	21	21	240	52.2	0.26	227
8	v1	66	47	37	40	31	24	24	283	61.5	0.44	229
	v2	52	31	10	0	33	25	25	301	65.4	0.42	225
16	v1	63	45	32	47	36	28	28	296	64.4	0.85	205
	v2	47	29	10	0	35	27	27	312	67.8	0.8	218
32	v1	50	39	27	40	30	24	24	202	43.9	2.48	216
	v2	49	34	10	0	39	30	30	342	74.3	1.46	221
64	v1	45	37	21	27	28	16	16	183	39.8	5.45	260
	v2	48	35	10	0	33	26	26	317	68.9	3.15	225
128	v1	47	38	18	14	15	8*	16*	115	25.0	17.33	259
	v2	46	36	10	0	26	20	20	243	52.8	8.24	225

Table 3. Resource utilization and throughput of CHIP-KNN designs on Alveo U280 FPGA with different dimensions, at N=4M, K=10, Euclidean distance, and single-precision floating-point data. Number of PEs, number of HBM banks used, executioin time, and clock frequency are also included. *For the two corner cases when D=2 and D=128 in CHIP-KNNv1, our automation tool decides that the best performance could be achieved when each PE uses 256-bit AXI port and two PEs share one HBM bank in CHIP-KNNv1. Bandwidth Utilization is calculated by achieved throughput divided by total HBM bandwidth.

4.6.1 Resource Utilization and Design Frequency. To pass the placement and routing, in our design automation tool, we limit our CHIP-KNNv1 designs to utilize less than 70% of any resource (i.e., $\alpha = 70\%$). As shown in Table 3, all these designs are limited by the LUT resource. For the designs with D=2, 4, and 8, their LUT utilization is fairly close to 70%. For the designs with higher dimensions, they are limited by the placement and routing issue and we have to relax the target resource utilization, i.e., the α value. For the design with D=16, we have to relax α to 65%; for the designs with D=32, 64, and 128, we have to relax α to 50%. Note the constant 10% BRAM usage in CHIP-KNNv2 is due to the Vitis shell. For most of the designs, especially CHIP-KNNv2 designs, we have achieved 225MHz or higher frequency; in occasional cases where the design did not achieve 225MHz, their frequency is also very close to 225MHz.

Table 3 also lists the number of PEs and the number of HBM banks used by the PEs in the design. In most cases, each PE uses one HBM bank with 512-bit AXI port, except two corner cases when D=2 and D=128 in CHIP-KNNv1. In those two cases, our automation tool decides that the best performance could be achieved when each PE uses 256-bit AXI port and two PEs share one HBM bank in CHIP-KNNv1. For our CHIP-KNNv2 designs, with the more resource efficient single-PE design as discussed in Section 4.4 and integration with TAPA/AutoBridge as discussed in Section 3.5, they instantiate more PEs and utilize more HBM banks in parallel, especially for designs with D=32, 64, and 128.

4.6.2 Bandwidth Utilization. To analyze the off-chip bandwidth utilization, we compute the throughput as the total amount of input data, divided by the kernel execution time. The achieved throughput is the key to CHIP-KNN performance, and is optimized according to the analytical performance model from 3.7. As described in Section 3.6, the maximum practically achievable throughput is 421.6 GB/s. We see that the CHIP-KNNv2 floating-point designs are capable of achieving a throughput of up to 342 GB/s, when using 30 HBM banks. This is improved from CHIP-KNNv1, which achieves a maximum throughput of 296 GB/s, when using 28 HBM banks. The lowest throughput is achieved at designs with D=2 and D=128, and in the worst of these cases CHIP-KNNv2 is able to achieve 162 GB/s throughput when using 14 HBM banks, while CHIP-KNNv1 achieves 98 GB/s throughput with 9 HBM banks. These designs achieve lower throughput

because each PE requires more resources, and therefore we are unable to utilize as many HBM banks.

4.6.3 Performance Balancing. Table 4 summarizes the automation tool's performance model outputs, alongside the cycle latencies of each stage - $Load\_Buf$, $Dist\_Cal$, and Top_K_Sort - for each PE in our designs. As shown, the design's latencies are very well-balanced in most cases. For designs with higher dimensions, the $Load\_Buf$ stage takes more cycles, thus we relax the $dist\_II$ for the $Dist\_Calc$ stage. For the designs with lower dimensions, the $Load\_Buf$ stage takes fewer cycles and thus we have to increase the $dist\_factor$ and $sort\_factor$. Notice that the Top_K_Sort stage sometimes takes fewer cycles than the other two stages, especially when D is 16 and higher; this is because the $Sort\_Factor$ is constrained (i.e., its smallest value is 1), and we cannot reduce the resource usage of the sorting unit to slow it down further.

Comparing between the CHIP-KNNv1 and CHIP-KNNv2 implementations, we see that the outputs of the performance model—specifically, $Sort\_Factor$—are different. This is alluded to the difference in the achieved II of the sorting unit, shown in Equations 8 and **??** in Section 3.7. In this table, we see the ramifications - the sort factor for CHIP-KNNv1 is higher, which is the result of the achieved II being 3 instead of 2. Each sort unit requires a few more cycles to execute, and therefore we instantiate more sorting units by increasing $Sort\_Factor$. The distance factor is different only in the 2-D configurations, and this is because the CHIP-KNNv1 design used a port width of 256 bits. The $dist\_II$ also diverges slightly at the 64-D and 128-D designs. This is once again due to the sort unit's II - in order to balance the latency between top-K sort and distance calculation, the required $dist\_II$ diverges between the two versions.

Dimension	Version	Dist II	Dist Factor	Sort Factor	Load_Buf Cycle latency	Dist_Calc Cycle latency	Top_K_Sort Cycle latency
2	v1	1	4	12	64,725	63,450	63,855
	v2	1	8	16	37,462	37,491	37,505
4	v1	1	4	12	71,904	67,072	67,360
	v2	1	4	8	49,942	49,987	49,985
8	v1	1	2	6	98,900	93,396	92,321
	v2	1	2	4	83,893	83,952	83,935
16	v1	1	1	3	167,536	162,282	155,548
	v2	1	1	2	155,351	155,458	155,393
32	v1	2	1	3	387,144	395,523	178,353
	v2	2	1	1	279,628	279,764	279,669
64	v1	3	1	1	1,150,464	1,037,824	802,816
	v2	4	1	1	645,288	645,468	645,347
128	v1	12	1	1	4,316,160	4,361,216	3,021,824
	v2	8	1	1	1,677,740	1,678,008	1,258,383

Table 4. Automation tool outputs and execution cycles for CHIP-KNN designs on Alveo U280 FPGA with different dimensions, at N=4M, K=10, Euclidean distance, and single-precision floating-point data.

5 RELATED WORK

5.1 KNN Acceleration on FPGA

HLS-based KNN acceleration. In [35], Song et al. presented an HLS-C based KNN accelerator that is adaptive to all key KNN parameters. It supports low-precision data representation and PCA-based approximate KNN algorithm. However, their design is not fully optimized and only

uses 11.9% of the available off-chip bandwidth. In [26], Liu implemented an OpenCL-based KNN accelerator, which uses bitonic sort to sort the nearest neighbors. However, they only tested their design under small datasets and utilized 7% of the off-chip bandwidth. In [32], Pu et al. implemented an OpenCL-based KNN accelerator, which features a high-speed parallel sorting algorithm based on bubble sort. However, their design only supports a fixed KNN configuration and utilizes 11.1% of the off-chip bandwidth. As summarized in Table 5, we are the first to optimize the memory access of KNN accelerators on datacenter FPGAs and can utilize up to 81.3% of the off-chip bandwidth of the Xilinx Alveo U280 FPGA.

Design BW (GB/s)	[35]	[26]	[32]	CHIP-KNNv1	CHIP-KNNv2 (float)	CHIP-KNNv2 (all datatypes)
Max. Achieved BW	9.1	1.8	1.4	296	342	374
Theoretical BW	76.8	25	12.8	460	460	460
BW utilization	11.9%	7%	11.1%	64.3%	74.3%	81.3%

Table 5. Bandwidth comparison of FPGA acceleration.

Acceleration for KNN-based classifier system. In [20], Hussain et al. developed an HDL-based KNN classifier to speed up the ensemble classification on FPGA through dynamic partial reconfiguration that achieves 5x speedup. However, their design only supports small datasets that can be stored on chip using FIFOs. In [36], Vieira et al. proposed a flexible HDL-based streaming KNN classifier design for embedded FPGA-SoCs and compared performance results with a single-core ARM Cortex-A9 processor. However, they did not exploit the abundant parallelism in the distance calculation and sorting, nor did they fully optimize the off-chip memory access. Lastly, while their framework generates user configured designs, the performance cannot be guaranteed to be optimal for the device. For accelerating KNN on embedded FPGA SoC platforms, Gorgin et al. proposed kNN-MSDF [16], a HDL-based design which uses binary signed digit representation to minimize the hardware cost for implementing the distance computing logic and performs early termination to save energy consumption. Different from our work, their accelerator is only evaluated on rather small datasets (i.e., average dataset size less than 16k and data dimension less than 20). Besides, it does not provide the automatic design generation and the comprehensive user configurable parameters as CHIP-KNN. In [33], Samiee et al. proposed a reduced-rank local distance metric for the KNN classifier mainly to improve the classification accuracy. Their work mainly focused on the final classification accuracy using the proposed distance metric. The author presented performance result for a fixed benchmark without detailed explanation.

Acceleration for approximated nearest neighbour search. Previous work [1, 13, 25] have explored hardware acceleration of the algorithms involved in approximated nearest-neighbour search (ANNS). In [6], Tavakoli et al. proposed an OpenCL-based FPGA accelerator for the KNN algorithm that primarily focuses on scaling down high-dimensional sample data using random projection and providing users a way to trade accuracy for performance. In [13], Danopoulos et al. accelerated the vector indexing stage of an approximate KNN method used in the Facebook Faiss [23] framework. In [1], Abdelhadi et al. designed a specialized FPGA-based accelerator that exploits the low latency on-chip memory for accelerating Product-Quantization (PQ) based ANNS. Lastly, in [25], Lee et al. proposed a specialized architecture for accelerating compression-based ANNS algorithms used in Facebook Faiss [23] and Google ScaNN [19], based on an in-depth analysis on the inefficiency of executing the ANNS on CPU/GPU. These ANNS-based acceleration work are orthogonal to our work where we focus on the exact KNN algorithm.

5.2 KNN Acceleration on CPU

In [42], Yu et al. accelerated the KNN kernel on a x86 CPU by combining the distance calculation and sorting into a single operation to better utilize the memory bandwidth. As a result, they achieved over 4x performance speedup compared to other existing methods of the time. In [4, 5], Arya et al. proposed an optimized algorithm and developed a library to approximate KNN searching on a CPU.

5.3 KNN Acceleration on GPU

Besides the FAISS GPU implementation [23] that we have compared against in Section 4.3, there are a couple more GPU acceleration studies. In [29], Matsumoto et al. used the GPU to accelerate the distance calculation and the CPU to perform the sorting. Thus, their performance is limited by the sorting stage. In [14, 15], Garcia et al. developed several CUDA implementations of the KNN kernel and updated their GitHub source code in 2018 (https://github.com/vincentfpgarcia/kNN-CUDA). Their implementation assumes a batch of input queries are processed concurrently. For the distance calculation, they exploit the abundant parallelism among different queries, search space points, and feature dimensions. For distance sorting, they only exploit the parallelism among queries; the sorting remains sequential within the processing for each query.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have designed and implemented a configurable and high-performance KNN accelerator called CHIP-KNN. Given a user configuration of key KNN parameters, our tool can automatically generate the optimal KNN accelerator design, which best utilizes the available resources and off-chip bandwidth. We present two designs: CHIP-KNNv1, a buffer-based design; and CHIP-KNNv2, a streaming-based design. Both designs are capable of consuming data as fast as it can load data, through load-balancing techniques. To take advantage of this, we optimize the off-chip memory access by identifying an optimal memory access port width and target frequency. We also build an analytical performance model to guide our automation tool to find the optimal design with balanced execution of all stages. CHIP-KNNv2 improves upon CHIP-KNNv1 in architectural design, user configurability, and floorplanning. With CHIP-KNNv2, we have added support for TAPA/AutoBridge [8, 18] to improve our achievable frequency. We also added support for user-configurable datatype. Finally, we have conducted a wide range of experiments. Compared to a 48-thread CPU version, we achieve between 6x and 45x performance speedup across different configurations. CHIP-KNNv1 is open-sourced at: https://github.com/SFU-HiAccel/CHIP-KNN, and we plan to open-source CHIP-KNNv2 soon.

ACKNOWLEDGEMENTS

We acknowledge the partial support from NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; Canada Foundation for Innovation John R. Evans Leaders Fund and British Columbia Knowledge Development Fund; Simon Fraser University New Faculty Start-up Grant; Huawei, Xilinx, and Nvidia.

REFERENCES

[1] Ameer M.S. Abdelhadi, Christos-Savvas Bouganis, and George A. Constantinides. 2019. Accelerated Approximate Nearest Neighbors Search Through Hierarchical Product Quantization. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 90–98. https://doi.org/10.1109/ICFPT47387.2019.00019

[2] N. S. Altman. 1992. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician* 46, 3 (1992), 175–185.

[3] G. Aparício, I. Blanquer, and V. Hernández. 2007. A Parallel Implementation of the K Nearest Neighbours Classifier in Three Levels: Threads, MPI Processes and the Grid. In *High Performance Computing for Computational Science -*

VECPAR 2006, Michel Daydé, José M. L. M. Palma, Álvaro L. G. A. Coutinho, Esther Pacitti, and João Correia Lopes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 225–235.

[4] Sunil Arya and David M. Mount. 1998. ANN: library for approximate nearest neighbor searching. In *Proceedings of IEEE CGC Workshop on Computational Geometry*. 33–40.

[5] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. 1998. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *J. ACM* 45, 6 (Nov. 1998), 891–923.

[6] Erfan Bank Tavakoli, Amir Beygi, and Xuebin Yao. 2022. RPkNN: An OpenCL-Based FPGA Implementation of the Dimensionality-Reduced kNN Algorithm Using Random Projection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, 4 (2022), 549–552. https://doi.org/10.1109/TVLSI.2022.3147743

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, USA, 44–54. http://rodinia.cs.virginia.edu/doku.php?id=start

[8] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 204–213.

[9] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. 2018. Best-Effort FPGA Programming: A Few Steps Can Go a Long Way. *CoRR* abs/1807.01340 (2018).

[10] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding Performance Differences of FPGAs and GPUs. In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. IEEE Computer Society, 93–96.

[11] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2017. Bandwidth Optimization Through On-Chip Memory Restructuring for HLS. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 43, 6 pages.

[12] Intel Corporation. 2021. AN 584: Timing Closure Methodology for Advanced FPGA Designs. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an584.pdf

[13] D. Danopoulos, C. Kachris, and D. Soudris. 2019. FPGA Acceleration of Approximate KNN Indexing on High-Dimensional Vectors. In *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. 59–65. https://doi.org/10.1109/ReCoSoC48741.2019.9034938

[14] V. Garcia, E. Debreuve, and M. Barlaud. 2008. Fast k nearest neighbor search using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. 1–6. https://doi.org/10.1109/CVPRW.2008.4563100

[15] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. 2010. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*. 3757–3760. https://doi.org/10.1109/ICIP.2010.5654017

[16] Saeid Gorgin, MohammadHosein Gholamrezaei, Danial Javaheri, and Jeong-A Lee. 2022. kNN-MSDF: A Hardware Accelerator for k-Nearest Neighbors Using Most Significant Digit First Computation. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 1–6. https://doi.org/10.1109/SOCC56010.2022.9908102

[17] Gongde Guo, Hui Wang, David Bell, Yaxin Bi, and Kieran Greer. 2003. KNN Model-Based Approach in Classification. In *International Conference on Ontologies, Databases and Applications of Semantics (ODBASE 2003)*. Springer, Switzerland, 986–996.

[18] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 81–92.

[19] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *International Conference on Machine Learning*. Article 364, 10 pages. https://arxiv.org/abs/1908.10396

[20] H. M. Hussain, K. Benkrid, and H. Seker. 2012. An adaptive implementation of a dynamically reconfigurable K-nearest neighbour classifier on FPGA. In *2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. 205–212. https://doi.org/10.1109/AHS.2012.6268651

[21] Ville Hyvonen, Teemu Pitkanen, Sotiris Tasoulis, Elias Jaasaari, Risto Tuomainen, Liang Wang, Jukka Corander, and Teemu Roos. 2016. Fast nearest neighbor search through sparse random projections and voting. *2016 IEEE International Conference on Big Data (Big Data)* (Dec 2016), 881–888.

[22] Intel. 2022. Intel (R) Memory Latency Checker. https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html Last accessed July 3, 2022.

[23] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[24] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, and M. Oskin. 2018. Application Codesign of Near-Data Processing for Similarity Search. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 896–907. https://doi.org/10.1109/IPDPS.2018.00099

[25] Yejin Lee, Hyunji Choi, Sunhong Min, Hyunseung Lee, Sangwon Beak, Dawoon Jeong, Jae W. Lee, and Tae Jun Ham. 2022. ANNA: Specialized Architecture for Approximate Nearest Neighbor Search. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 169–183. https://doi.org/10.1109/HPCA53966.2022.00021

[26] Liyuan Liu. 2018. *Acceleration of k-Nearest Neighbor and SRAD Algorithms Using Intel FPGA SDK for OpenCL*. Master's thesis. University of Windsor.

[27] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. 2020. CHIP-KNN: A Configurable and High-Performance K-Nearest Neighbors Accelerator on Cloud FPGAs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 139–147.

[28] Alec Lu, Zhenman Fang, and Lesley Shannon. 2022. Demystifying the soft and hardened memory systems of modern fpgas for software programmers through Microbenchmarking. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4 (2022), 1–33. https://doi.org/10.1145/3517131

[29] T. Matsumoto and M. L. Yiu. 2015. Accelerating Exact Similarity Search on CPU-GPU Systems. In *2015 IEEE International Conference on Data Mining*. 320–329. https://doi.org/10.1109/ICDM.2015.125

[30] Ming Ouyang. 2016. KNN in the Jaccard space. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2016.7761587

[31] Marius Muja and David G. Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*. 331–340.

[32] Y. Pu, J. Peng, L. Huang, and J. Chen. 2015. An Efficient KNN Algorithm Implemented on FPGA Based Heterogeneous Computing System Using OpenCL. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 167–170. https://doi.org/10.1109/FCCM.2015.7

[33] A. Samiee, Y. Huang, and Y. Bai. 2018. FRLDM: Empowering K-nearest Neighbor (KNN) through FPGA-based Reduced-rank Local Distance Metric. In *2018 IEEE International Conference on Big Data (Big Data)*. 4742–4746. https://doi.org/10.1109/BigData.2018.8622087

[34] Thomas Seidl and Hans-Peter Kriegel. 1998. Optimal Multi-Step k-Nearest Neighbor Search. *SIGMOD Rec.* 27, 2 (June 1998), 154–165.

[35] X. Song, T. Xie, and S. Fischer. 2019. A Memory-Access-Efficient Adaptive Implementation of kNN on FPGA through HLS. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. 177–180. https://doi.org/10.1109/ICCD46524.2019.00030

[36] J. Vieira, R. P. Duarte, and H. C. Neto. 2019. kNN-STUFF: kNN STreaming Unit for Fpgas. *IEEE Access* 7 (2019), 170864–170877. https://doi.org/10.1109/ACCESS.2019.2955864

[37] Xindong Wu, Vipin Kumar, Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, G. Mclachlan, Shu Kay Angus Ng, Bing Liu, Philip Yu, Zhi-Hua Zhou, Michael Steinbach, David Hand, and Dan Steinberg. 2007. Top 10 algorithms in data mining. *Knowledge and Information Systems* 14 (12 2007), 1–37. https://doi.org/10.1007/s10115-007-0114-2

[38] Xilinx. 2020. Alveo U200 and U250 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf Last accessed July 28, 2020.

[39] Xilinx. 2020. Alveo U280 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf Last accessed July 28, 2020.

[40] Xilinx. 2020. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development Last accessed July 28, 2020.

[41] Bin Yao, Feifei Li, and Piyush Kumar. 2010. K nearest neighbor queries and kNN-Joins in large relational databases (almost) for free. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 4–15.

[42] C. D. Yu, J. Huang, W. Austin, B. Xiao, and G. Biros. 2015. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807601

[43] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2019. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2019), 2072–2085.

[44] Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. 2014. Fast and effective placement and routing directed high-level synthesis for fpgas. *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays* (2014), 1–10. https://doi.org/10.1145/2554688.2554775