CHIP-KNN: A Configurable and High-Performance K-Nearest Neighbors Accelerator on Cloud FPGAs

Alec Lu*, Zhenman Fang*, Nazanin Farahpour† and Lesley Shannon*
*Simon Fraser University, Burnaby, BC, Canada
{alec_lu, zhenman, lesley_shannon}@sfu.ca
†University of California, Los Angeles, USA; nazanin@cs.ucla.edu

***Abstract*—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—which optimizes the off-chip memory access on cloud FPGAs with multiple DRAM or HBM (high-bandwidth memory) banks. CHIP-KNN is configurable for all essential parameters used in the algorithm, including the size of the search dataset, the feature dimension of each data point, the distance metric, and the number of nearest neighbors - K. To optimize its performance, we build an analytical performance model to explore the design space and balance the computation and memory access performance. Given a user configuration of the KNN parameters, our tool can automatically generate the optimal accelerator design on the given FPGA platform. Our experimental results on the Nimbix cloud computing platform show that: Compared to a 16-thread CPU implementation, CHIP-KNN on the Xilinx Alveo U200 FPGA board with four DRAM banks and U280 FPGA board with HBM achieves an average of 7.5x and 19.8x performance speedup, and 6.1x and 16.0x performance/dollar improvement.

I. Introduction

The k-nearest neighbors (KNN) algorithm [1] is one of the top 10 most influential algorithms in the data mining research community [2]. It is widely used in many applications such as similarity search, image classification, and database query [3]–[5]. 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 [6], [7].

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 [8]–[10] 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 [8] achieves an equivalent performance as a 56-thread CPU implementation for large datasets. Compared with the GPU accelerator for KNN, the FPGA accelerator in [9] 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 [8]–[10]. First, most of them, except the latest design in [8], only support a fixed configuration of KNN with a small dataset, fixed feature dimension, and distance metric. 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, 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 V-A.

In this paper we design and implement CHIP-KNN, an open-source, HLS (high-level synthesis) C based, configurable, and high-performance KNN accelerator on cloud FPGAs. To better support large search datasets and scale to different cloud FPGA platforms, CHIP-KNN takes a scalable multi-PE (processing element) approach. For each PE, we optimize its computation—i.e., neighbor distance calculation and sorting—on a small tile of dataset buffered on chip by exploring the pipeline parallelism and fine-grained data parallelism enabled by a novel sorting algorithm. To optimize the data loading between off-chip memory and on-chip buffers for each PE, we carefully tune the data width of its memory access port and the size of its consecutive data access to better utilize off-chip memory bandwidth. Moreover, we scale the number of PEs to explore the coarse-grained parallelism and better utilize the available off-chip memory bandwidth of multiple DRAM or HBM banks on cloud FPGAs.

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. Given a user configuration of these KNN parameters and an FPGA platform, our tool can automatically generate the optimal accelerator design that reaches either 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, distance calculation, and distance sorting—to explore the design space and balance (overlap) the execution of these three stages.

We conduct our experiments on the Nimbix cloud [11] with the Xilinx Alveo U200 [12] and U280 [13] datacenter FPGA boards and various configurations of KNN parameters. On average, compared to a 16-thread CPU implementation, CHIP-KNN on Alveo U200 and U280 achieves 7.5x and 19.8x performance speedup; at the same time, CHIP-KNN also achieves 6.1x and 16.0x performance/dollar improvement.

II. KNN Algorithm and Cloud FPGAs

A. KNN Algorithm

In this paper we mainly focus on the exact KNN algorithm without any approximation for two reasons. First, approximate KNNs [14]–[16] 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-dimension feature vector. Common distance metrics include $Euclidean$ and $Manhattan$ distances between two feature vectors. Assume there are N points in the search space, the algorithmic complexity for this step is *O(N*D)*. 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. The data access complexity is also *O(N*D)*, making the memory access optimizations very important.
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. Note its algorithmic complexity is *O(N*K)* instead of *O(N*logN)*, since it only sorts the K smallest distances. 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*, and *distance metric*.

B. Cloud FPGAs

To meet the increasing demand in computing resource and off-chip bandwidth, modern cloud FPGAs typically consist of multiple FPGA dies and multiple DRAM or HBM banks. For example, in the Nimbix cloud [11], the Xilinx Alveo U200 [12] FPGA board has three SLRs (super logic regions, i.e., FPGA dies) and four DDR4 banks (64GB size), which can provide up to 76.8GB/s theoretical memory bandwidth. And the Alveo U280 [13] FPGA board has three SLRs and 32 HBM2 banks (8GB size), which can provide up to 460GB/s theoretical memory bandwidth. This provides great opportunity to accelerate the performance of the KNN algorithm.

However, most prior studies [8]–[10] 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 [17], [18].

III. CHIP-KNN Design

To better accelerate KNN with large search datasets on different cloud FPGAs with multiple DRAM or HBM banks, CHIP-KNN takes a scalable multi-PE (processing element) approach. The overall architecture of our CHIP-KNN accelerator is shown in Figure 1. In Section III-A, we present the single-PE design with a novel sorting algorithm and common HLS optimizations such as buffering, pipelining, parallelization, and Ping-Pong buffer [19], [20]. In Section III-B, 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 DRAM or HBM banks. In Section III-C, we characterize the effective off-chip bandwidth of a single DRAM or HBM bank and optimize each PE's memory access. In Section III-D, we build an analytical performance model to guide the performance balancing of the computation and memory access stages. Finally, in Section III-E, we develop our automation tool to generate the optimal accelerator design for a given user configuration on a given cloud FPGA platform.

A. Single-PE Design with Hardware-Oriented KNN Algorithm

Algorithm 1 shows the pseudo code of our single-PE KNN algorithm. We support the following user-configured parameters as described in Section II-A: *N*, *D*, *K*, and *distance metric*. Currently, we only support the most widely used single-precision floating data type and Euclidean and Manhattan distances in CHIP-KNN, but it can be easily extended to support other data types and distance metrics. 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.

*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 (initial interval) of 1. The following two stages, *Dist_Calc* and *Top_K_Sort*, work on this local on-chip buffer. In Section III-C we will present more details on choosing the optimal memory access port width and tile size to maximize the off-chip memory bandwidth utilization of this stage.

*2) **Dist_Cal Stage (Lines 4-10)**:* This stage calculates the distance between the query point and each point in the buffered search space. Between two D-dimension data points, X and Y, the Manhattan distance and Euclidean distance are:

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

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

In this stage, we explore the following fine-grained data parallelism and pipeline parallelism. First, we fully parallelize

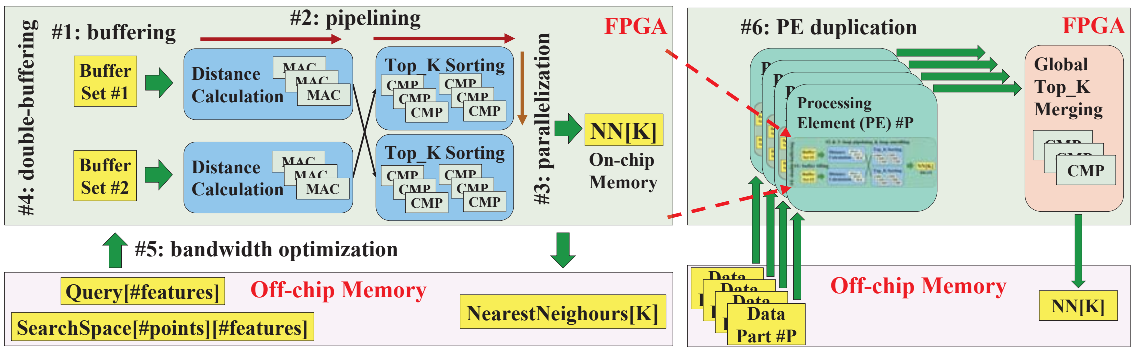

Fig. 1: Overall architecture of our CHIP-KNN accelerator

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

```
1:  function LOAD_BUF
2:      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:      input_query[D]
6:      local_search_space[B][D]
7:      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:     point_dist[B + K] //K dummy MAX dist
13:     point_id[B + K] //K dummy invalid ids
14:     k_nearest_dist[K+2] ////Initialized as MAX dist
15:     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
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 results in this function
```

(unroll) the calculation of all D dimensions in each distance calculation as shown in Equation 1 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 III-E, our automation tool will choose the optimal *dist_factor* and *dist_II* to balance the execution of the three stages in each PE.

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(3N).

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 can pipeline the i loop with an II of 3. Note that the ideal II of the i loop should be 2; however, when we scale the design to multiple PEs, Vivado HLS can only achieve 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 III-E, 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.

*4) **Ping-Pong Buffer**:* Finally, we use the Ping-Pong buffer technique [19], [20] to execute the *Load_Buf*, *Dist_Cal*, and *Top_K_Sort* stages in a coarse-grained pipeline. We call these three stages together as a single processing element (PE).

B. Multi-PE Scaling

To better scale the design to utilize the computing resource from multiple SLRs (FPGA dies) and off-chip bandwidth from multiple DRAM or 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 III-E.

Since each PE only produces the partial top_K result, we add a global top_K merger to merge the P copies of local top_K results to the global K-nearest neighbors. This global top_K merger only needs to execute once after all PEs finish processing all of their tiled buffers. It has an algorithmic complexity of O(P*K) and its execution time is negligible if the data access to all the local top_K results are on chip.

There are two major considerations in the multi-PE design.

1. *One big kernel vs. multiple small kernels.* In the one big kernel approach, all PEs can easily exchange data through on-chip buffer by default. However, it is usually quite challenging for the cross-SLR placement and routing. Therefore, we decide to use the multiple small kernels approach. Specifically, for each SLR, we implement one kernel that includes a maximum number of PEs that can fit into the SLR. And we also implement one kernel for the global top_K merger itself. As a result, we can bind each kernel to one SLR to avoid the cross-SLR placement and routing issue and bind the data access to multiple DDR or HBM banks to utilize more off-chip bandwidth.

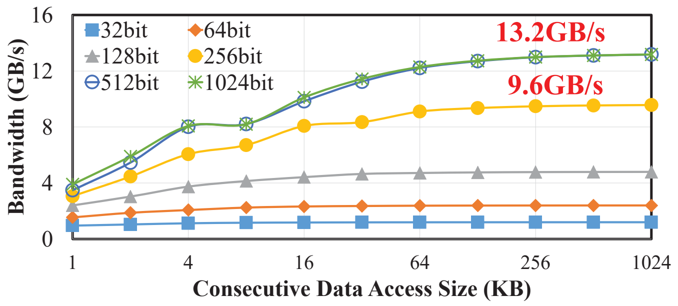

Fig. 2: Bandwidth of a single HBM bank on Alveo U280 FPGA, with different consecutive data access sizes and memory access port widths. Note the x-axis is plotted in log_2 scale.

2. *Kernel-to-kernel streaming.* The drawback of the multiple small kernels approach is that different kernels have to exchange data through off-chip memory by default. Specifically in our design, multiple kernels have to provide the local top_K results to the global top_K merger kernel. To address this issue, we exploit the new kernel-to-kernel streaming feature in Xilinx Vitis 2019.2 [21] to enable their data communication through the on-chip streaming.

C. 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 [18] to characterize the effective memory access bandwidth of a single HBM bank on the Alveo U280 FPGA board [13]. Shown in Figure 2, 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. We also characterize the effective bandwidth of a DDR4 bank on the Alveo U200 FPGA board [12] and find that it has a similar trend as the HBM bank, except that its peak effective bandwidth is at 17.94GB/s while the peak theoretical bandwidth is 19.2GB/s.

Based on these characterization results, we optimize the off-chip memory access bandwidth in CHIP-KNN (mainly in the *Load_Buf* stage) by carefully tuning 1) its memory access port width, denoted as $port\_width$, and 2) its consecutive data access size, i.e., the size of the *local_search_space* buffer, denoted as buf_size. Note that the best bandwidth configuration with $port\_width = 512bits$ and $buf\_size = 128KB$ does not necessary give the best overall performance of the multi-PE design of CHIP-KNN, because this configuration also consumes more resource and can limit the number of PEs and the placement and routing. In Section III-D and III-E, we will build an analytical performance model and automation tool to choose the optimal $port\_width$ and buf_size.

D. 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—including *Load_Buf*, *Dist_Cal*, and *Top_K_Sort* in Algorithm 1—that execute in a coarse-grained pipeline. The goal is to balance the execution of these three stages within each PE. Since each function is pipelined, its latency to execute a single tile is calculated as:

$$Latency = (pipe\_iterations - 1) * II + pipe\_depth \quad (2)$$

where $pipe\_iterations$ is the number of pipeline iterations, II is the initiation interval, and $pipe\_depth$ is the pipeline depth.

Load_Buf. The latency to load one tile is:

$$Load = [(buf\_size/port\_width - 1) + depth_{ld}] * \text{effective\_BW\_factor}(buf\_size,\ port\_width) \quad (3)$$

where the burst read achieves an II of 1, each load reads $port\_width$ size of data and 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, Vivado 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 III-C. 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*.

Dist_Calc. The latency to calculate distances for one tile is:

$$Dist\_Lat = (B/dist\_factor - 1) * dist\_II + depth_{dist} \quad (4)$$

where B is the number of buffered distances and can be derived as $B = buf\_size/D/sizeof(float)$. 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.

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

$$Sort\_Lat = ((B + K)/sort\_factor - 1) * 3 + depth_{sort} \quad (5)$$

which is similar to Equation 4, except that the II is fixed as 3. The *sort_factor* can be adjusted to tune the latency. The $depth_{sort}$ can be inferred from one-time HLS synthesis.

As explained in Section III-A and III-B, the latencies of the optional local top_K merger in the *Top_K_Sort* stage and the global top_K merger are negligible and thus not modeled. But their execution time is included in the final experiments.

E. 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, shown in Figure 3, to generate the optimal accelerator design that best utilizes the off-chip memory bandwidth under the available FPGA resource limit.

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-KNN design template. To explore the design space, we vary the buf_size of 64KB and 128KB and $port\_width$ of 256bits and 512bits. Based on our performance model, we generate the corresponding balanced *dist_factor*, *dist_II*, and *sort_factor* parameters of the three design stages.

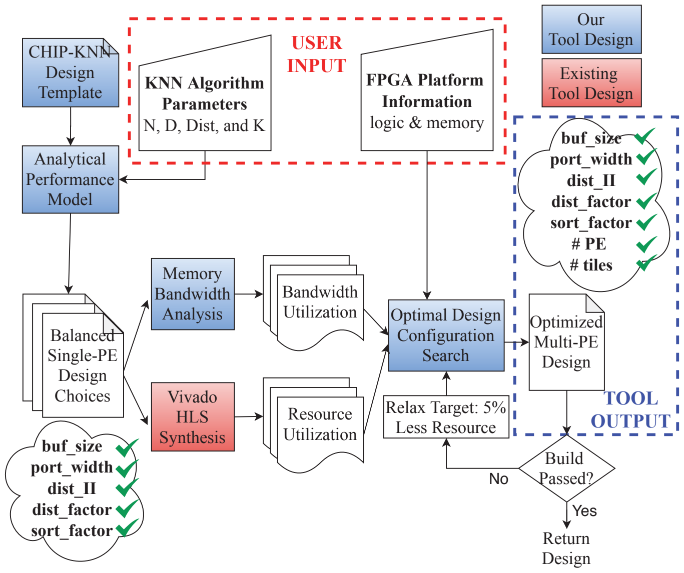

Fig. 3: Flowchart of design automation for CHIP-KNN

2. For each balanced single-PE design, our tool determines its resource utilization using Vivado HLS synthesis and its bandwidth utilization based on the memory bandwidth characterization in Section III-C.
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-KNN, we determine the maximum number of PEs that can fit onto the FPGA as:

$$\#PEs = min(\#off\_chip\_memory\_banks, \alpha * FPGA\_resource/PE\_resource) \quad (6)$$

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 [21]. 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.

IV. Results and Analysis

A. Experimental Setup

KNN configuration. We evaluate CHIP-KNN with a wide range of parameter configurations listed in Table I. The number of data points in the search space (N) ranges from 2M to 8M and the feature dimensions (D) ranges from 2 to 128. That is, the total size of the search space data, which is $N * D * sizeof(float)$ bytes, ranges from 16MB (when *N=2M* and *D=2*) to 4GB (when *N=8M* and *D=128*). 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.

TABLE I: Evaluated key KNN parameter configurations

KNN Parameters	Values
N: number of data points in search space	2M, 4M, 6M, 8M
D: feature dimension	2, 4, 8, 16, 32, 64, 128
K	5, 10, 15, 20
Distance metric	Manhattan, Euclidean

Hardware platform and software tool. We perform all our experiments on the Nimbix cloud [11]. To evaluate the CPU implementation, we use a computing instance with the 22nm 8-core (16-thread) Xeon E5-2640 v3 CPU and 128GB DRAM. The rental price for this CPU instance is $2.42/hour. The software program of KNN is parallelized using Pthread and compiled with gcc *-Ofast* optimization flag, which uses all 16 threads of the CPU instance and automatically explores the vectorization optimization. We evaluate our CHIP-KNN accelerator designs on both the 16nm Xilinx Alveo U200 (with four DDR4 banks) [12] and U280 (with 32 HBM2 banks) [13] datacenter FPGA boards as described in Section II-B. The rental prices for both FPGA instances are $3.00/hour. We build our CHIP-KNN designs using Xilinx Vitis 2019.2 [21]. For the FPGA version, we only include the kernel execution time. All speedup and performance/dollar improvement results are normalized to the 16-thread CPU implementation.

B. Overall Speedup and Performance/Dollar Improvement

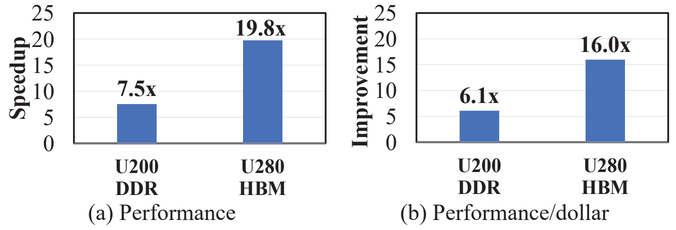

Fig. 4: Geometric mean of speedup and performance per dollar improvement over CPU across all configurations in Table I

Figure 4 summarizes the geometric mean of performance and performance/dollar improvement results of CHIP-KNN on Alveo U200 and U280 FPGAs over the 16-thread CPU version across all configurations in Table I. On average, CHIP-KNN achieves 7.5x speedup and 6.1x performance/dollar improvement on Alveo U200 FPGA, and 19.8x speedup and 16.0x performance/dollar improvement on Alveo U280 FPGA.

C. Speedup for Different KNN Configurations

Next, we present more detailed speedup results when we change one of the parameters in *N*, *D*, *K*, and *distance metric*. We omit the performance/dollar results which can be inferred from the speedup results and rental prices.

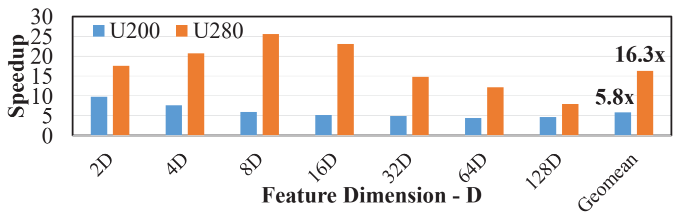

Fig. 5: Performance speedup for different feature dimensions with N=4M, K=10, and Euclidean distance

TABLE II: Execution time of CHIP-KNN for different feature dimensions with N=4M, K=10, and Euclidean distance

Execution Time (ms)	2D	4D	8D	16D	32D	64D	128D
CPU	5.93	7.39	11.11	20.04	38.55	70.89	142.54
U200	0.56	0.99	1.89	3.76	7.48	14.92	29.82
U280	0.32	0.36	0.44	0.85	2.48	5.45	17.33

1) Speedup for Different Feature Dimensions: Figure 5 presents the speedup of CHIP-KNN over the 16-thread CPU implementation at different feature dimensions (D=2, 4, ..., 128) with N=4M, K=10, and Euclidean distance. On the Alveo U200 FPGA, the speedup decreases when D increases. This is because the performance of CHIP-KNN on U200 FPGA is bound by the off-chip bandwidth; as shown in Table II, its execution time scales linearly with D that linearly increases the total dataset size. On the other hand, with a larger D, the CPU implementation can better utilize vectorization instructions and thus its execution time increases sub-linearly.

On the Alveo U280 FPGA that provides higher off-chip bandwidth with HBM banks, CHIP-KNN achieves much higher speedup, which is up to 25x for the 8D KNN design. The performance of CHIP-KNN on U280 FPGA is bound by the available computing resource and the placement and routing. The speedup fluctuation on U280 FPGA for different Ds is caused by the variation in the number of PEs and *port_width* that affect the final bandwidth utilization. We will analyze the accelerator efficiency in Section IV-D.

2) Speedup for Different Ks: Figure 6 shows the speedup of CHIP-KNN for K=5, 10, 15, and 20, with N=4M, D=64, and Euclidean distance. When K increases, CHIP-KNN achieves 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.

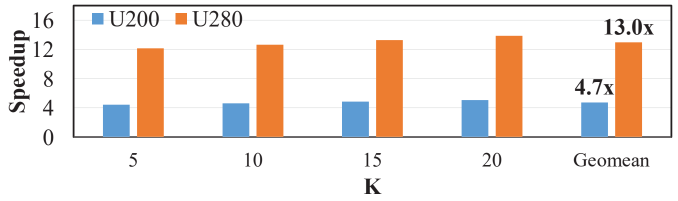

Fig. 6: Performance speedup for different Ks with N=4M, D=64, and Euclidean distance

3) Speedup for Different Distance Metrics: Figure 7 shows the speedup of CHIP-KNN for Manhattan and Euclidean distances with N=4M, D=64, and K=10. Using Manhattan distance, CHIP-KNN achieves about 10.9% higher speedup

TABLE III: Automation output, resource utilization, frequency, execution time, and bandwidth utilization of CHIP-KNN designs on Alveo U280 FPGA with D=2, 4, ..., 128, N=4M, K=10, and Euclidean distance

D: feature dimension	buf size	port width	dist II	dist factor	sort factor	# PEs	#tiles /PE	Resource Utilization					freq (MHz)	time (ms)	bandwidth (GB/s)
								LUT	FF	BRAM	URAM	DSP			
2D	∼128KB	256bit	1	4	12	18	15	67	39	60	30	12	218	0.32	98
4D		512bit	1	4	12	16	32	68	41	58	27	20	228	0.36	173
8D		512bit	1	2	6	24	43	66	47	37	40	31	229	0.44	283
16D		512bit	1	1	3	28	74	63	45	32	47	36	205	0.85	296
32D		512bit	2	1	3	24	171	50	39	27	40	30	216	2.48	202
64D		512bit	3	1	1	16	512	45	37	21	27	28	260	5.45	183
128D		256bit	12	1	1	16	1024	47	38	18	14	15	259	17.33	115

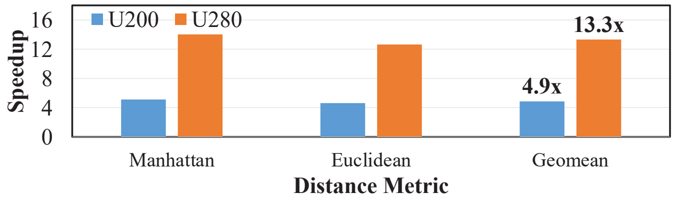

Fig. 7: Performance speedup for different distance metrics with N=4M, D=64, and K=10

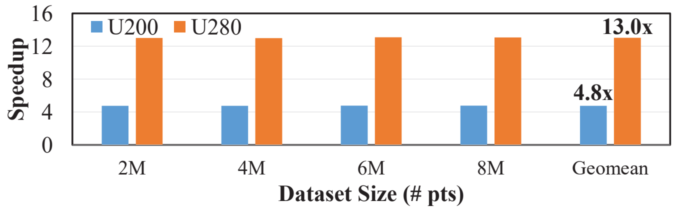

Fig. 8: Performance speedup for different dataset sizes with D=64, K=10, and Euclidean distance

than that using Euclidean distance. This is because CHIP-KNN achieves almost the same performance for the two distance metrics. However, in the CPU implementation, the version with Manhattan distance executes around 10.9% longer than that with Euclidean distance, due to the branch instruction overhead for the absolute math function.

4) Speedup for Different Dataset Sizes: Figure 8 shows the speedup of CHIP-KNN for N=2M, 4M, 6M, and 8M, with D=64, K=10, and Euclidean distance. The speedup results are pretty consistent for different dataset sizes.

D. Accelerator Efficiency Analysis

To evaluate the efficiency of CHIP-KNN in supporting both bandwidth-bound and resource-bound platforms, we analyze the designs on both Alveo U200 and U280 cloud FPGAs.

1. For the U200 FPGA, our CHIP-KNN designs are bound by its off-chip memory bandwidth. All the design points utilize less than 55% of all available FPGA resource and are thus not resource-bound. Based on the actual execution, our designs can utilize 67.1GB/s off-chip bandwidth, which is about 94% of the peak effective off-chip memory bandwidth (71.76GB/s) characterized in Section III-C.
2. For the U280 FPGA that has much higher off-chip bandwidth from HBM, our CHIP-KNN designs are limited by the FPGA resource and placement and routing constraints. Based on the actual execution, our designs can utilize up to 296GB/s off-chip bandwidth, which is only about 70% of the peak effective HBM bandwidth (422GB/s), and are thus not bandwidth-bound. Our designs either already utilize close to 70% of the FPGA resource or have to utilize less resource due to placement and routing failures.

TABLE IV: Cycles to execute one tile in CHIP-KNN on U280, with D=2, 4, ..., 128, N=4M, K=10, and Euclidean distance

Feature dimension	Load_Buf	Dist_Calc	Top_K_Sort
2D	4,315	4,230	4,257
4D	2,247	2,096	2,105
8D	2,300	2,172	2,147
16D	2,264	2,193	2,102
32D	2,264	2,313	1,043
64D	2,247	2,027	1,568
128D	4,215	4,259	2,951

To better illustrate the efficiency of our CHIP-KNN designs on the U280 FPGA and explain the speedup fluctuation results in Section IV-C1, we further analyze those designs with D=2, 4, ..., 128, N=4M, K=10, and Euclidean distance. Table III summarizes the PE configuration generated from the automation output, resource utilization, operating frequency, execution time and bandwidth performance of the designs.

1) Resource Utilization: To pass the placement and routing, in our design automation tool, we limit our designs to utilize less than 70% of any resource (i.e., $\alpha = 70\%$). As shown in Table III, 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%. Table III also lists the number of PEs and the frequency of each design, which is much lower than the target 300MHz. In future work, we plan to further investigate these designs and improve their placement and routing for better performance.

2) Bandwidth Utilization: Table III summarizes the bandwidths of our designs, which is calculated as the total dataset size divided by the actual execution time. This bandwidth decides the final performance of our CHIP-KNN designs. It is affected by three factors: 1) buf_size, which is about 128KB in all designs; 2) $port\_width$ and 3) $\#PEs$, which vary in designs as shown in Table III. The design for D=16 achieves the highest bandwidth of 296GB/s, as it has 28 PEs and $port\_width = 512bits$. The designs for D=2 and D=128 achieve the lowest bandwidths of 98GB/s and 115GB/s, because they can only support 18 and 16 PEs, with $port\_width = 256bits$. For these two designs, we find that the design choice with $port\_width = 512bits$ significantly limit the number of PEs due to the placement and routing issue.

3) Performance Balancing: Table IV summaries the cycles of all three stages—*Load_Buf*, *Dist_Cal*, and *Top_K_Sort*—to execute a single tile ($buf\_size = 128KB$) in each PE on Alveo U280 FPGA, with D=2, 4, ..., 128, N=4M, K=10, and Euclidean distance. All the designs are pretty balanced. For the designs with larger Ds, the $Load\_Buf$ stage takes more cycles, thus we relax the $dist\_II$ for the $Dist\_Calc$ stage; and even though we set $sort\_factor$ to 1, the Top_K_Sort stage takes fewer cycles than the other two stages. For the designs with smaller Ds, the $Load\_Buf$ stage takes less cycles and thus we have to increase the $dist\_factor$ and $sort\_factor$. All the automation output results—including *buf_size*, *port_width*, *dist_II*, *dist_factor*, *sort_factor*, *#PEs*, and *#tiles per PE*—are also summarized in Table III.

V. Related Work

A. KNN Acceleration on FPGA

HLS-based KNN acceleration. In [8], 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 [22], 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 [9], 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 V, we are the first to optimize the memory access of KNN accelerators on cloud FPGAs and can utilize 87.4% of the Alveo U200 DRAM bandwidth and 64.3% of the Alveo U280 HBM bandwidth.

TABLE V: Bandwidth comparison of FPGA acceleration

Design BW (GB/s)	[8]	[22]	[9]	Ours on U200	Ours on U280
Achieved BW	9.1	1.8	1.4	67.1	296
Theoretical BW	76.8	25	12.8	76.8	460
BW utilization	11.9%	7%	11.1%	87.4%	64.3%

Acceleration for KNN-based classifier system. In [10], 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 [23], Vieira et al. proposed a flexible HDL-based streaming KNN classifier design for embedded FPGA-SoCs. However, they did not exploit the abundant parallelism in the distance calculation and sorting, nor did they fully optimize the off-chip memory access. In [24], Samiee et al. proposed a reduced-rank local distance metric for the KNN classifier mainly to improve the classification accuracy. In [25], Danopoulos et al. accelerated the vector indexing stage of an approximate KNN method used in the Facebook artificial intelligence similarity search framework. This is orthogonal to our work where we focus on the exact KNN algorithm.

B. KNN Acceleration on GPU

In [26], 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 [27], [28], Garcia et al. developed several CUDA implementations of the KNN kernel and recently 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. We have run their code on the Nimbix cloud instance with the 14nm Nvidia V100 GPU [29], with a rental price of \$3.05/hour similar to that of Alveo U200 and U280 FPGAs. Due to space constraints, we only present the geometric mean of the results across all configurations listed in Table I. As shown in Table VI, on average, CHIP-KNN on Alveo U280 FPGA achieves a 1,144x latency speedup over the GPU design when processing a single query. On the other hand, CHIP-KNN on Alveo U280 FPGA achieves a 28% throughput of the GPU design when processing a batch of 4,096 queries.

TABLE VI: Latency and throughput comparison for CHIP-KNN on U280 FPGA and GPU design [27], [28] on V100

Latency	1,144x faster than GPU	Throughput	28% of GPU

VI. 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 and a target cloud FPGA platform, our tool can automatically generate the optimal KNN accelerator design, which reaches either the off-chip memory bandwidth limit or the FPGA resource limit. To optimize the off-chip memory access, we have carefully tune the memory access port width, consecutive data access size, and concurrent number of memory banks. We also build an analytical performance model to guide our automation tool to find the optimal design with balanced execution of all stages. Finally, we have conducted a wide range of experiments on the Nimbix cloud. Compared to the 16-thread CPU version, on average, we achieve 7.5x and 19.8x performance speedup, and 6.1x and 16.0x performance/dollar improvement, on the Xilinx Alveo U200 and U280 FPGAs. In future work, we plan to investigate the place-and-route issue of CHIP-KNN on the Alveo U280 FPGA. This work is open-sourced at: https://github.com/SFU-HiAccel/CHIP-KNN.

Acknowledgements

We acknowledge the support from NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020, COHESA (NETGP485577-15), CWSE PDF (470957), and RGPIN341516; 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] N. S. Altman, "An introduction to kernel and nearest-neighbor non-parametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[2] X. Wu, V. Kumar, R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. Mclachlan, S. K. A. Ng, B. Liu, P. Yu, Z.-H. Zhou, M. Steinbach, D. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, pp. 1–37, 12 2007.

[3] T. Seidl and H.-P. Kriegel, "Optimal multi-step k-nearest neighbor search," *SIGMOD Rec.*, vol. 27, no. 2, p. 154–165, Jun. 1998.

[4] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "Knn model-based approach in classification," in *International Conference on Ontologies, Databases and Applications of Semantics (ODBASE 2003)*. Switzerland: Springer, 2003, pp. 986–996.

[5] B. Yao, F. Li, and P. Kumar, "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)*, 2010, pp. 4–15.

[6] V. Hyvonen, T. Pitkanen, S. Tasoulis, E. Jaasaari, R. Tuomainen, L. Wang, J. Corander, and T. Roos, "Fast nearest neighbor search through sparse random projections and voting," *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016.

[7] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, and M. Oskin, "Application codesign of near-data processing for similarity search," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 896–907.

[8] X. Song, T. Xie, and S. Fischer, "A memory-access-efficient adaptive implementation of knn on fpga through hls," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, Nov 2019, pp. 177–180.

[9] Y. Pu, J. Peng, L. Huang, and J. Chen, "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*, May 2015, pp. 167–170.

[10] H. M. Hussain, K. Benkrid, and H. Seker, "An adaptive implementation of a dynamically reconfigurable k-nearest neighbour classifier on fpga," in *2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2012, pp. 205–212.

[11] Nimbix, "Accelerate your workflows with xilinx alveo accelerator cards in the cloud," 2020, last accessed July 28, 2020. [Online]. Available: https://www.nimbix.net/alveo

[12] Xilinx, "Alveo u200 and u250 data center accelerator cards data sheet," 2020, last accessed July 28, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf

[13] ——, "Alveo u280 data center accelerator cards data sheet," 2020, last accessed July 28, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf

[14] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *J. ACM*, vol. 45, no. 6, p. 891–923, Nov. 1998.

[15] S. Arya and D. M. Mount, "Ann: library for approximate nearest neighbor searching," in *Proceedings of IEEE CGC Workshop on Computational Geometry*, 1998, pp. 33–40.

[16] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *In VISAPP International Conference on Computer Vision Theory and Applications*, 2009, pp. 331–340.

[17] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for hls," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017.

[18] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2019.

[19] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "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, 2018, pp. 93–96.

[20] J. Cong, Z. Fang, Y. Hao, P. Wei, C. H. Yu, C. Zhang, and P. Zhou, "Best-effort FPGA programming: A few steps can go a long way," *CoRR*, vol. abs/1807.01340, 2018.

[21] Xilinx, "Vitis unified software platform," 2020, last accessed July 28, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development

[22] L. Liu, "Acceleration of k-Nearest Neighbor and SRAD Algorithms Using Intel FPGA SDK for OpenCL," Master's thesis, University of Windsor, 2018.

[23] J. Vieira, R. P. Duarte, and H. C. Neto, "knn-stuff: knn streaming unit for fpgas," *IEEE Access*, vol. 7, pp. 170 864–170 877, 2019.

[24] A. Samiee, Y. Huang, and Y. Bai, "Frldm: Empowering k-nearest neighbor (knn) through fpga-based reduced-rank local distance metric," in *2018 IEEE International Conference on Big Data (Big Data)*, Dec 2018, pp. 4742–4746.

[25] D. Danopoulos, C. Kachris, and D. Soudris, "Fpga acceleration of approximate knn indexing on high- dimensional vectors," in *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, July 2019, pp. 59–65.

[26] T. Matsumoto and M. L. Yiu, "Accelerating exact similarity search on cpu-gpu systems," in *2015 IEEE International Conference on Data Mining*, Nov 2015, pp. 320–329.

[27] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, June 2008, pp. 1–6.

[28] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching," in *2010 IEEE International Conference on Image Processing*, Sep. 2010, pp. 3757–3760.

[29] Nvidia, "Nvidia v100 tensor core gpu," 2020, last accessed July 28, 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/