Accelerating kNN Search in High Dimensional Datasets on FPGA by Reducing External Memory Access

Xiaojia Song[a], Tao Xie[a], Stephen Fischer[b]

[a]San Diego State University, San Diego, 92182, CA, USA
[b]Samsung Semiconductor, Inc., San Jose, 95134, CA, USA

Abstract

Implementing an efficient k-Nearest Neighbors (kNN) algorithm on FPGA is becoming challenging due to the fact that both the size and dimensionality of datasets that kNN is working on have been rapidly growing, which makes external memory-access a performance bottleneck. To reduce the impact of the bottleneck, in this paper we implement two kNN kernels through high-level synthesis (HLS) on FPGA by employing two data access reduction methods: low-precision data representation (LPDR) and principal component analysis based filtering (PCAF). One kernel is called MBFS-kNN (Memory-efficient Brute-Force Searching kNN) and the other is called MPCAF-kNN (Memory-efficient PCAF kNN). The two kernels are adaptive to all key parameters. By comparing them with two state-of-the-art kNN implementations on a high-end CPU server, an existing BFS-kNN kernel on FPGA, and an existing BFS-kNN kernel on GPU, our experimental results show that the two kernels substantially improve the performance by greatly reducing external memory-accesses.

Keywords: k-Nearest Neighbors (kNN), FPGA, high-level synthesis, low-precision data representation, PCA-based filtering, memory-access bottleneck

1. Introduction

The k-Nearest Neighbors (kNN) search algorithm is one of the most popular machine learning algorithms due to its simplicity and accuracy [1][2][3][4][5][6]. It has been applied to a wide range of high-performance computing (HPC) applications such as image/video retrieval [7][8], big data analysis [1][3], machine learning [9], and computer vision [10]. In an image retrieval system, given a query image q, kNN searches an image database and then returns k images with features most similar to that of q [7].

kNN can be implemented on various computing platforms such as a conventional multi-core CPU server or a heterogeneous computing system using accelerators like GPU or FPGA. Among these platforms, an FPGA-based heterogeneous system is becoming increasingly attractive for a spectrum of applications thanks to FPGA's inherent parallelism, deeply-pipelined architecture, high energy-efficiency, and reconfigurability [11] [12][13]. Microsoft employed FPGAs to accelerate its Bing page ranking functions [11]. Baidu developed a software-defined accelerator for large-scale deep neural network (DNN) systems, which heavily rely on FPGA devices [12].

Existing work on FPGA-based kNN acceleration has demonstrated promising results [14][15][16][17]. However, a new challenge is emerging due to the fact that both the size and dimensionality of datasets that kNN is working on have been rapidly growing these days. Tineye, the first image search engine on the Web to use image identification technology, was launched in 2008. Since then, the number of images in Tineye's indexed image database has increased from 0.7 billion to 35 billion in 2019 [18]. At the same time, to obtain a more accurate representation of an image, the number of dimensions of each feature vector extracted by some neural network technology could be as large as 4,096 [19][20]. As a result, a kNN search in such a large database with a high dimensionality becomes both compute-intensive and memory-intensive [21]. On the other hand, a modern FPGA board normally can only provide a moderate bandwidth between an FPGA chip and its on-board external DRAM [22], which makes memory-access a performance bottleneck of a kNN kernel. For example, the maximal bandwidth between the FPGA chip and a single external on-board memory bank is only 512 bits per clock cycle in a VCU 1525 board [23] (see Section 4.1). Before the potential of the FPGA chip (i.e., its internal high level of parallelism and deep pipeline architecture) can be fully exploited, the impact of the external memory-access bottleneck on its performance needs to be effectively alleviated.

To reduce the impact of the external memory-access bottleneck, in this paper we implement two kNN kernels through HLS [24] on FPGA by employing two data access reduction methods: low-precision data representation (LPDR) [25] and principal component analysis based filtering (PCAF) [8]. The former has been successfully applied in various domains as it can improve hardware bandwidth utilization by lowering data precision, and thus, reducing the volume of data being read-/written [25][26]. The latter uses a data filtering mechanism to exclude those reference features that are not likely to be kNN features according to the PCA estimation [8]. One kernel is called MBFS-kNN (Memory-efficient Brute-Force Searching kNN) and the other is called MPCAF-kNN (Memory-efficient

PCAF kNN). While the former employs only the method of LPDR to reduce the number of memory accesses, the latter utilizes both methods to achieve the same goal. MBFS-kNN can be used to carry out an accurate kNN search, whereas MPCAF-kNN can perform an approximate kNN search. The motivation of developing MBFS-kNN is three-fold. First, it provides us with an understanding of the behavior of an accurate kNN search on FPGA in terms of performance and energy-efficiency. Second, it allows us to gain sufficient kNN design and implementation experience on FPGA, which lays a good foundation for us to develop a more advanced MPCAF-kNN kernel. Third, it enables one to easily apprehend the development of MPCAF-kNN. MPCAF-kNN is inspired by the PCAF algorithm, which was proposed in a recent study [8]. However, the original PCAF algorithm is not FPGA friendly. To solve this issue, we develop two new strategies (see Section 4.5) so that our optimized PCAF algorithm can fully exploit the characteristics of FPGA. Both kernels are adaptive to the number of dimensions (D), number of data points in a database (N), number of nearest neighbors (k), number of bits per feature (B), and number of principal components (d).

We evaluate the two kNN kernels in terms of performance and energy-efficiency by comparing them with two state-of-the-art kNN implementations on a high-end CPU server, an existing BFS-kNN kernel on FPGA, and an existing BFS-kNN kernel on GPU. The experimental results demonstrate that MBFS-kNN can achieve a performance equivalent to that of a 76-thread CPU server in the best case. It also outperforms the two existing BFS-kNN kernels in execution time and energy-efficiency by 5.5x and 1.97x, 7.44x and 22.29x, respectively. The MPCAF-kNN kernel achieves up to a performance equivalent to that of a 56-thread of CPU server. It also gains 21.75x energy-efficiency compared with the CPU server. Compared with the BFS-kNN, MPCAF-kNN reduces external memory accesses by 28~231x. This paper makes three contributions. First, the PCAF algorithm was originally proposed for a multi-core CPU server [8]. Directly transplanting it to FPGA, however, could not yield a high-performance kNN kernel (see Section 2). Two empirical optimization strategies (see Section 4.5) are developed so that the MPCAF-kNN kernel can effectively utilize the potential of FPGA. In addition, three optimization approaches (see Section 4.2) are proposed to enhance the performance of BFS-kNN. Second, to alleviate the external memory-access bottleneck problem, the low-precision data representation scheme is employed by the two proposed FPGA kernels. To the best of our knowledge, this work is the first research that utilizes a PCA-based data filtering mechanism to reduce memory accesses of a kNN kernel running on FPGA. Third, a comprehensive evaluation of the two kernels in performance and energy-efficiency is provided.

The rest of paper is organized as follows. Section 2 briefly summarizes the related work followed by the motivation of this research. Section 3 presents the background of this research including the algorithms of BFS-kNN and PCAF-kNN. Section 4 provides implementation and optimization details of the two kernels. Section 5 evaluates the two kernels. Finally, Section 6 concludes the paper.

2. Related Work and Motivation

Accelerating HPC applications by employing FPGA has been a hot research topic in the last decade. Modern HPC systems using FPGA have been developed to accomplish various tasks from deep learning models for image search to enhancing cloud services [11][27][28][12]. Traditional kNN implementations like [14][15] were all developed using an hardware description language (HDL). Hussain *et al.* proposed two adaptive FPGA architectures of the kNN classifier [14]. Along the same line, Manolakos *et al.* developed two hardware architectures described as soft parameterized IP cores in very high-speed integrated circuit hardware description language (VHDL) [15]. All hardware architectures proposed by [14][15] were implemented in an HDL, which cannot be directly used in an high-level synthesis (HLS) implementation. After the HLS technique became available, some developers switched their kNN implementations from HDL to HLS. For examle, Pu *et al.* employed a specific bubble sort algorithm to speed up the sorting phase of a BFS-kNN algorithm using OpenCL. Their kNN kernel outperforms a 4-thread CPU by 148 and 803 times in execution time and energy-efficiency, respectively [17]. Muslim *et al.* also accelerated a BFS-kNN search using FPGA under the OpenCL framework [16]. They found that an optimized FPGA-based kNN kernel could offer performance and energy-efficiency better than a GPU-based kNN implementation [16]. Unfortunately, [16][17] are not suitable for kNN search in a large dataset because they store temporary distances in on-chip memory whose size is usually very limited (e.g., only 345.9 Mb available on the FPGA chip XCVU9P-L2FSGD2104E, see Section 4.1).

To alleviate the problem of increasingly large data size and high dimensionality, several approximate kNN (AkNN) algorithms [29][30][31][32][33][8][34][35] have been proposed. Instead of returning the k actual nearest neighbors, they return k results that are highly likely to be the k nearest neighbors. Based on the strategy of finding approximate nearest neighbors, they can be generally divided into two camps: data-selection based (e.g., [29][32][35]) and data-filtering based (e.g., [31][8]). Algorithms in the first camp normally incur a large memory footprint. Besides, they usually exhibit a poor scalability due to the need of performing a large number of random memory accesses [31]. For example, the algorithm proposed in [32] utilized multiple randomized KD-trees (RKD-trees) to build its index structure. In the search stage, it traverses these trees so that promising candidate nodes can be stored in a queue for the subsequent distance calculation stage. Unfortunately, traversing multiple RKD-trees cannot be efficiently parallelized on FPGA. Based on our analysis, algorithms in the first camp are not good candidates for an FPGA implementation. This is mainly because they cannot exploit the abundant parallelism provided by FPGA.

Although AkNN algorithms in the second camp avoid the drawbacks mentioned above, we discover that not all of them are suitable for an FPGA implementation. Subspace clustering for filtering (SCF) [31], for example, is a state-of-the-art AkNN algorithm in the second camp. Its search precision depends on the nature of the reference features [31]. After a preliminary

investigation on algorithms in the second camp, we find that PCAF [8] is promising because it could be efficiently implemented on FPGA. It uses principal components analysis (PCA) to estimate the rank of distance between the query feature and the reference feature. Next, reference features that are not likely to be k-NN features according to the PCA estimation are excluded [8]. Compared with other data-filtering based AkNN algorithms, PCAF has a good scalability with a stable and high search precision on high dimensional datasets [8]. Thus, in this research we decide to implement a kNN kernel on FPGA based on the PCAF algorithm.

Implementing PCAF on FPGA, however, is not trivial. First, in the original PCAF algorithm the distance comparisons in the PCA space are executed sequentially, which cannot exploit the potential parallelism and deep pipelining of FPGA. As a result, the performance of PCAF on FPGA could be degraded. Second, even after the original data are mapped from a high dimensional space to a low dimensional space, the external memory-access bottleneck problem still exists as the memory bandwidth of FPGA is substantially less than that of CPU or GPU. To solve the first challenge, we propose two empirical optimization strategies (see Section 4.5) so that distance comparisons in the PCA space can be performed in a pipelined and parallel fashion. To address the second challenge, we optimize the proposed MPCAF-kNN kernel by employing the low-precision data representation (see Section 4.3). To the best of our knowledge, this work is the first attempt to accelerate AkNN search in a high dimensional dataset on FPGA.

3. Background

In this section, we first present the algorithms of existing BFS-kNN (Section 3.1) and PCAF-kNN (Section 3.2). Next, the implementation environment (i.e., the OpenCL framework and HLS tool chain) of our two kNN kernels is introduced (Section 3.3).

3.1. Algorithm of BFS-kNN

In general, a data point p can be defined as a D dimensional vector: $p = [d_1, d_2, ..., d_D]$. The database DB is defined as a set of N data points: $DB = \{p_1, p_2, ..., p_N\}$. Given a query data point q, kNN searches for k data points in a database that are most similar or related to q, where the similarity is often measured by Euclidean distance, Hamming distance, or learned distance metrics [36] [37]. In this paper, we use the Euclidean distance. The core of kNN consists of distance calculation (e.g., Euclidean distance) and top-k sorting [7].

3.2. Algorithm of PCAF-kNN

Principal component analysis (PCA) [38] is a popular algorithm for dimensionality reduction, and thus, is able to alleviate the curse of dimensionality in some contexts. When the size and dimension of a database DB become large, PCA can be used to map a DB to a low dimensional space [38]. It utilizes an orthogonal transformation to convert a set of data values of possibly correlated variables into a set of data values of linearly uncorrelated variables called *principal components* [38]. Algorithm 1 illustrates the algorithm of PACF-kNN, which performs an approximate kNN search.

Algorithm 1: PCAF-kNN

```
   Input  : q, DB, q', DB', and k
   Output: k nearest neighbors
1  Create and initialize a heap of size k with +∞;
2  Create and initialize a heap' of size k * m with +∞;
3  data_read(q' and q);
4  for i ← 0 to N − 1 do
5      data_read(p'_i);
6      δ' ← (q' − p'_i)^2;
7      if δ' < heap'.max then
8          data_read(p_i);
9          δ ← (q − p_i)^2;
10         if δ < heap.max then
11             heap'.insert(δ');
12             heap.insert(δ);
13         end
14     end
15 end
16 return final k nearest neighbors from heap;
```

PCAF first uses singular value decomposition (SVD) [39] to find the principal components of database DB with a dimensionality of D [8]. Next, DB and the query q are projected into a PCA space where their mappings are called *DB'* and *q'*, respectively. We use d to represent the dimensionality of *DB'*. Note that d is much smaller than D, which is the dimensionality of DB. The main idea of PCAF is to use the distance rank in a surrogate PCA space to filter out the data points that are unlikely to be in the kNN set [8]. Algorithm 1 shows the algorithm of PCAF-kNN. For each data point p_i in DB, PCAF-kNN first calculates the distance between the corresponding projected point (i.e., p'_i in *DB'*) and the projected query q' (see lines 5-6). If the distance δ' is smaller than the maximal distance in $heap'$, then the distance between p_i and q (i.e., δ) is calculated (see lines 7-9). Otherwise, PCAF-kNN processes next data point in *DB'*. If δ is smaller than the maximal distance in $heap$, then δ is inserted into $heap$ and δ' is inserted into $heap'$ (see lines 10-13). Note that the index of this data point is also stored in the two heaps. The final kNN data points will be in $heap$ after all the data points in DB have been processed.

The most expensive operations of BFS-kNN are data reading from external memory and distance calculation when D is large. The goal of PCAF is to reduce the number of executions of these two operations in the original space. Note that the m shown in line 2 of Algorithm 1 is used as a tunning parameter to improve the searching accuracy. Basically, it is an amplifier that enlarges the size of the temporary filter heap $heap'$ by m times so that k*m possible nearest neighbors will stay in $heap'$ (see Fig. 3), which increases the threshold used to filter out a data point. More details about a PCAF-kNN implementation on a multi-core CPU can be found from [8]

3.3. OpenCL framework and high-level synthesis

In the view of the OpenCL framework, a computing system consists of an array of compute devices, which might be central processing units (CPUs) or accelerators such as GPUs, FPGAs, DSPs (digital signal processors), or other processors attached to a host CPU [40]. The OpenCL framework defines a C-like language for writing programs. Functions executed on an OpenCL device are called *kernels*. A single compute device typically consists of several compute units, which in turn comprise multiple processing elements (PEs) [40]. A single kernel execution can run on all or many of the PEs in parallel [40].

In this paper, we employed a Xilinx VCU1525 FPGA board [23] as the accelerator. The OpenCL host code (i.e., a code that manages kernels' input/output data and coordinates their execution) and kernel program were developed using Xilinx SDAccel tool chain [41]. OpenCL C, C/C++, and HDLs (e.g., Verilog or VHDL) are all supported for a kernel development. In this paper, we design and implement the two kNN kernels in C++ and then convert them to a low level design through Xilinx HLS, which are then executed under the OpenCL framework. The two kernels have been highly optimized in C++ level, which will be elaborated in the next section.

4. Implementation and Optimization of the Two Kernels

In this section, we first introduce the FPGA hardware resources of VCU1525 [23]. Next, we elaborate the details of implementation and optimization of MBFS-kNN and MPCAF-kNN. Although the implementation and optimization of the two kernels are performed on Xilinx VCU1525 [23] acceleration board, they also can be applied to other FPGA devices by using the Xilinx HLS tool chain.

4.1. FPGA resources of VCU1525

On-board memory external to FPGA chip: The VCU 1525 board is populated with four DDR4-2400 SDRAM banks with a total capacity of 64 GB (i.e., each SDRAM bank is 16 GB). The four banks together serve as the global memory for data exchange between a host CPU and a kernel on the FPGA board. From the perspective of the FPGA chip, the global memory is its *external* memory where it can retrieve input data from and store output data to. This is the reason why the bottleneck to be addressed in this paper is called an *external* memory-access bottleneck. At any given time, a kernel is able to read data from or write data to one of the four banks. The maximal bandwidth to access an individual SDRAM bank is 512 bits per clock cycle, which means that a kernel can only read or write no more than 512 bits of data from or to a memory bank per clock cycle [23]. This limited bandwidth largely constrains the performance of a memory-intensive application.

On-chip resources: The resources on the FPGA chip (i.e., XCVU9P-L2FSGD2104E) include on-chip memory, digital signal processing (DSP) slices, registers, programmable logic such as flip-flops (i.e., FFs), and lookup tables (i.e., LUTs). On-chip memory consists of distributed RAM, block RAM (i.e., BRAM), and UltraRAM. Together, they can be used to build a customized high-speed storage to buffer data between tasks. The total capacity of on-chip memory is rather small (i.e., 345.9 Mb), which demands a careful design of a kernel that can effectively utilize these resources [42].

Other resources: The configurable logic block (CLB) is the main resource for implementing a general-purpose combinatorial and sequential circuit. CLBs usually are used as LUTs and FFs. They can be easily connected to each other to create a larger function. A DSP slice can be configured to a complex arithmetic function such as a multiplier or an accumulator. Detailed information of resources on the VCU 1525 board can be found at [23].

4.2. Optimization of BFS-kNN

Implementing a BFS-kNN on FPGA has two challenges [16][17]. First, after distance calculation N distance values have been generated for a *DB* with N data points. When N becomes large there will not be enough on-chip memory to store these values. While most efficient kNN algorithms employ an indexing data structure to prune the search, BFS-kNN has to calculate all N distance values, which is time-consuming. Second, performing distance calculation and sorting separately increases the latency because sorting can only start after all distance values are available. To solve these challenges, we propose the following three optimization approaches:

1. Using a *k_max heap* to store k distance values rather than a huge buffer to store all distance values (see Fig. 1).
2. Pipelining data_reading, distance calculation, and *k_max* comparison functions.
3. Aligning the throughput of distance calculation and *k_max* comparison to external memory-access bandwidth (i.e., 512 bits per cycle).

Now we use the KDD-CUP 2004 quantum physics dataset [43] as an example to show step-by-step how our optimized BFS-kNN works. This dataset (i.e., *DB*) consists of 50,000 data points (i.e., $N = 50,000$) with each data point having 64 features (i.e., $D = 64$). Each feature is a 32-bit data (i.e., $B = 32$). To simplify our demonstration, we will only conduct a kNN search with the first two data points shown in Fig. 1.

Since the maximal external memory bandwidth for a kernel is 512 bits per clock cycle and each data point in *DB* totally has $64 \times 32 = 2{,}048$ bits, 4 clock cycles (i.e., 2,048/512 = 4) are needed to move one data point into the kernel on FPGA. Also, each data point is split into 4 data blocks tagged as $p_{i0}, p_{i1}, p_{i2}, p_{i3}$ where i indicates the *i-th* data point in *DB*. To fully exploit the maximal external memory bandwidth, two designs of BFS-kNN are explored (see Fig. 1a and Fig. 1b). In the design shown in Fig.1a (hereafter, BFSa-kNN), a data buffer is created on the FPGA chip to store one data point (see *buffer* in Fig. 1a). Note that reading q into a kernel is a one-time task. We assume that q has been read into the kernel before time *t0* (see *query data* in Fig. 1a). At time instance *t0*, the first 512-bit data block (i.e., p_{00}) is read into the *buffer*, after which p_{01}, p_{02}, p_{03} are fetched into *buffer* at *t1, t2, t3*, respectively. After the entire first data point p_0 is available in the kernel, the 64 features in

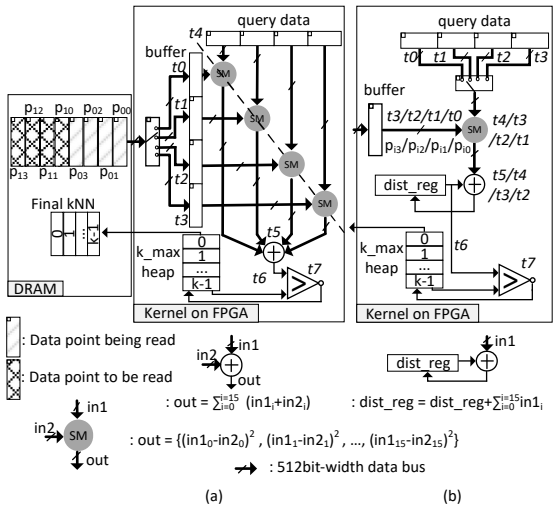

Figure 1: (a) BFSa-kNN; (b) BFSb-kNN.

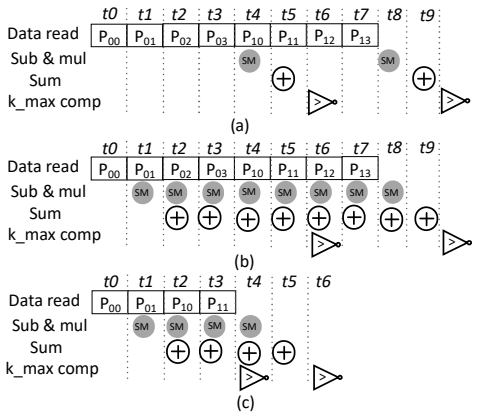

Figure 2: (a) BFSa-kNN 32-bit; (b) BFSb-kNN 32-bit; (c) BFSb-kNN 16-bit.

buffer and the *query data* will be used for a Euclidean distance calculation in parallel at *t4* and *t5*. Next, at *t6* the new distance value obtained at *t5* will be compared with the current maximal value in *k_max heap*. If it is smaller than the maximal value, the latter will be first kicked out, after which the new distance value will be inserted into *k_max heap*. Otherwise, the new distance value is simply discarded. Thanks to FPGA's deeply-pipelined architecture, p_{10} can be fetched into *buffer* immediately after the distance calculation of data point p_0 starts at *t4*. Similarly, the *SM* array is ready for a distance calculation on the next data point p_1 at *t5* instead of after *t7*. The timeline of BFSa-kNN is shown in Fig. 2a.

BFSa-kNN works well when the dimensionality (D) of DB is low. For a high dimensional DB, however, it will have some issues. First, the size of the on-chip *buffer* is determined by the value of $D \times B$. More on-chip memory will be consumed when D becomes larger. Unfortunately, on-chip memory of an FPGA board is usually very limited. Second, a large value of D requires a huge fanout from the DDR read logic to *buffer*, which reduces the routing quality. This in turn may cause a timing issue and scale down system clock frequency. Third, the parallel multiplication in the *SM* array will consume a large number of DSP slices (see *Other resources* in Section 4.1). A very high FPGA resource utilization will also cause the timing issue and scale down the system clock frequency. To solve these issues, we carefully examined the design of BFSa-kNN (see Fig. 1a) and its timeline (see Fig. 2a). We discovered that memory access on an external on-board SDRAM bank is a performance bottleneck, which makes a high degree of parallelism for a distance calculation unnecessary. This is because anyway there is always a stall (i.e., *t7* in Fig. 2a) between processing two consecutive data points. The implication is that one SM is sufficient. As a result, FPGA resources can be saved. Also, the performance of distance calculation and *k_max* comparison (i.e., *k_max comp* in Fig.2a) should be aligned to the external on-board memory-access bandwidth, which is a performance bottleneck of a BFSa-kNN kernel. A refined design of BFSa-kNN is called BFSb-kNN and it is illustrated in Fig. 1b.

The main difference between BFSa-kNN and BFSb-kNN is that the latter only employs one SM in its distance calculation phase. It perfectly pipelines data reading and distance calculation, which leads to a higher hardware utilization compared with BFSa-kNN shown in Fig. 1a. Although BFSb-kNN (see Fig. 1b) just uses 25% DSP slices of that of BFSa-kNN (see Fig. 1a), they have the same latency (see Fig. 2a and Fig. 2b). BFSb-kNN also uses less routing resources, which leads to a higher placement and routing quality as well as a higher clock frequency. Therefore, our proposed MBFS-kNN will be built on top of BFSb-kNN. One serious concern on BFS-kNN is that it needs to literally go through all the data points in a dataset in order to accomplish a kNN search. When the size and dimensionality of the dataset greatly increase BFS-kNN becomes notoriously memory-intensive. To alleviate this problem, we employ two data access reduction methods, which will be explained in the next two subsections.

4.3. Low-precision data representation

The first method is called low-precision data representation, which utilizes less bits to represent each feature. Consequently, in each 512-bit reading from the external DRAM, a kernel can obtain more data points. Let us still use the aforementioned dataset as an example. If each feature only uses 16 bits instead of 32 bits (i.e., $B = 16$), then after each reading the kernel will receive 32 features (i.e., 512/16 =32) with each being 16 bits. Now, moving 64 features of one data point into the kernel only needs 2 instead of 4 clock cycles. The timeline of BFSb-kNN with each feature being 16 bits is shown in Fig. 2c, which demonstrates that performance is improved by around 2 times compared with BFSb-kNN with each feature being 32 bits. We call a BFSb-kNN algorithm that employs low-precision data representation MBFS-kNN. In our experiments, we cast each 32-bit floating data to a 16-bit floating data.

Another advantage of using low-precision data representation is that it makes hardware operators smaller and faster. For example, a basic multiplication unit in an FPGA is a DSP48E macro, which provides a multiplier of 27-bit × 18-bit. Using a 32-bit data type would require four DSP48E macros, whereas using a 16-bit data type only demands one such resource. As a result, more data can be processed using the same resources

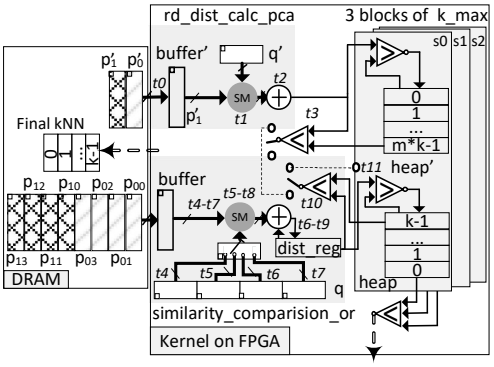

Figure 3: Implementation of PCAF-kNN on FPGA.

in the FPGA chip. However, both the query q and database (*DB*) need to be carefully pre-processed or generated when the method of low-precision data representation is applied. For an existing dataset, a statistical analysis should be used to obtain maximal effective bits for all features. In case that generating new database (*DB*) is needed, quantization can be employed in the data generator to reduce the number of bits that represent each feature. Take the CBIR (Content-Based Image Retrial) system as an example. When one builds a feature vector database for CBIR a quantized neural network can be used to generate a low-precision dataset [44]. It is worth noting that employing an improper or a too low data precision usually results in a poor accuracy, which makes a kNN search meaningless. Care must be taken when choosing the number of bits used for each feature. Another method to break the memory wall is called PCAF, which will be elaborated next.

4.4. PCA-based data filtering

The implementation of PCAF-kNN on FPGA is illustrated in Fig. 3. We still use the dataset in the last section to describe how PCAF-kNN performs a search. The dimensionality (d) of the PCA space are set to 16. How we choose the dimensionality of the PCA space will be explained in Section5.

A PCAF-kNN kernel can finish reading a p'_x (i.e., a data point in the PCA space) in one cycle. Major components in a PCAF-kNN kernel include a *rd_dist_calc_pca* functional block, a *similarity_comparison_or* functional block, and three identical *k_max* functional blocks. Each *k_max* block maintains a temporary heap called *heap'* whose size is $m \times k$ and a k-sized heap called *heap*. The *rd_dist_calc_pca* block is in charge of reading each data point in the PCA space and then calculating its distance from q'. The *similarity_comparison_or* block performs a similarity comparison between a data point in the original *DB* space and q.

Since reading q and q' (i.e., the projection of q in the PCA space) into the kernel is a one-time task, we assume that both have been fetched into the kernel before time *t0*. At time *t0*, p'_0 (i.e., a PCA space projection of data point p_{0x}) is read into *buffer*. At time *t1*, *SM* starts the subtraction and multiplication for the 16 data pairs from p'_0 and q'. After time *t2*, the calculation of a distance value will be finished, and then, it will be compared with the maximal value of *heap'* at time *t3* (see Fig. 3). If the new distance value is larger, this data point is not a potential nearest neighbor. Therefore, PCAF-kNN simply discards it and then immediately starts to process the second data point in the PCA space. Otherwise, the *similarity_comarision_or* block will be invoked to read the first data point p_0 of the original space *DB* between time *t4* and *t7*. Next, PCAF-kNN calculates the distance between p_0 and query q by accumulating intermediate distance values in *dist_reg*. The final distance value will become available at time *t10* and it will be compared with the maximal value of *heap* at time *t11* (see Fig. 3).

If the new distance value is larger, then the *rd_dist_calc_pca* block continues to compare the second distance value with the maximal value of *heap'*. Otherwise, PCAF-kNN confirms that the data point p_0 is a potential nearest neighbor for query q. Thus, the two distance values obtained from the two functional blocks and the index of p_0 will be inserted to *heap'* and *heap*, respectively. After all data points in the PCA space have been processed, the final k nearest neighbors are output from *heap* on FPGA chip to a buffer called *Final kNN* on the external DRAM (see Fig. 3). Note that the size of the temporary filter *heap'* is deliberately augmented from k to $k \times m$ where m is an integer. The number of *k_max* functional blocks (i.e., s) can also be increased from 1 to n. These two parameters (i.e., m and s) ensure that more data points in the PCA space can be kept for a further verification, which increases the probability of finding the k-nearest-neighbors in the original space. Obviously, the benefit of using these two parameters is at the cost of sacrificing the filtering rate [8].

4.5. Two empirical optimization strategies of PCAF-kNN

Simply transplanting a PCAF-kNN algorithm designed for a CPU platform into an FPGA-based heterogeneous system without any adaption and optimization could lead to a poor performance. In this section, we propose two new empirical optimization strategies to optimize the performance of PCAF-kNN on FPGA.

Optimization Strategy One (*Packaging Tasks with Feedback into One Function*): In BFS-kNN, the three stages (i.e., data reading, distance calculation, and *k_max* comparison) of one data point are independent of that of its neighbor data points. Hence, they can be perfectly pipelined as shown in Fig. 2. However, this property does not exist in PCAF-kNN. The reason is that the similarity comparison of the current data point might depend on the outcome of its prior data point's similarity comparison. If an execution on a stage is viewed as a task, a task dependency (i.e., also called feedback in Xilinx HLS tool chain) can be observed in line 7 of Algorithm 1. In other words, the task of the current data point's similarity comparison has to wait till the processing of its prior data point is finished, which leads to pipeline stalls.

Without the pipelining strategy, however, the performance of PCAF-kNN will be greatly degraded. To solve this dilemma, we package the task of data reading in the PCA space (see line 5 of Algorithm 1) as a function called *data_read_pca()*. Similarly, the task of distance calculation in the PCA space (see line 6 of Algorithm 1) is packaged as *dist_calc_pca()*. Finally, tasks from line 7 to line 14 in Algorithm 1 are packaged as a

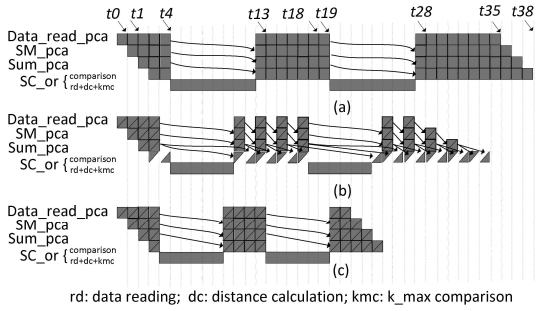

Figure 4: PCAF-kNN timelines: (a) 32-bit; (b) 16-bit; (c) 16-bit optimized.

function named *similarity_com_or()*. While the first two functions are carried out by the functional block *rd_dist_calc_pca* shown in Fig. 3, the last function is executed by the *similarity_comparison_or* functional block shown in Fig. 3. Now, the three functions can be successfully pipelined as they do not have any feedback among each other.

Optimization Strategy Two (*Comparing Distance Values in PCA Space in Parallel*): A timeline example of PCAF-kNN (see Fig. 3) on the dataset (d = 16, D = 64, B = 32, totally 20 data points) used in Section 4.4 is shown in Fig. 4a. In the *rd_dist_calc_pca* block shown in Fig. 3, the data point p'_0 in the PCA space can be read from the external SDRAM into a kernel in one cycle at time *t0* and then it goes to *SM_pca* at time *t1* as shown in Fig. 4a, where 16 subtractions with the features of q' and square operations are executed in parallel. Next, *Sum_pca* accumulates the 16 values from *SM_pca* to generate a distance value in the PCA space. *SC_or* stands for *similarity_comparison_or* shown in Fig. 3. In the timeline example shown in Fig. 4a, we assume that the distance value of data point p'_0 is larger than the maximal value of *heap'*. Thus, *SC_or* (i.e., *similarity_comparison_or* shown in Fig. 3) is not invoked as a further examination in the original space for this data point becomes unwanted. The distance value is simply discarded. At the same time, the distance value of the data point read in time *t1* (i.e., data point p'_1) is ready, which is assumed to be smaller than the maximal value of *heap'*. As a result, it will be further processed in *similarity_comparison_or* shown in Fig. 3. Since *similarity_comparison_or* conducts a similarity comparison in the original space, it needs 4 cycles (i.e., (D×B)/512 = (64×32)/512 = 4) to read a data point from the external memory. Thus, the data processing path of *SC_or* becomes longer. The goal of PCAF is to keep more data processing in the *rd_dist_calc_pca* block (see Fig. 3) while avoiding to invoke the *similarity_comparison_or* block. Based on the results from [8], in most cases the filtering rate F (i.e., the ratio between the number of data points processed only in the PCA space and the total number of data points in the original space) is larger than 95%. This the reason why PCAF can greatly accelerate a kNN search.

Now, we integrate low-precision data representation (see Section 4.3) into PCAF-kNN to further reduce the impact of memory access constraint. Fig. 4b shows the timeline when 16 bits instead of 32 bits are used to represent each feature in a data point. Since using 16 bits for each feature saves 50% cycles for reading a data point from the external on-board memory, each cycle a kernel can read 32 (i.e., 512 bits /16 bits = 32) features, which covers two data points in the PCA space. In Fig. 4a, each grey square represents one data point in the PCA space as each feature is 32-bit. However, when each feature is only 16-bit a grey square shown in Fig. 4b consists of two triangles with each representing a data point in the PCA space. To fully exploit the increased data reading speed (i.e., the number of data points can be read per cycle), *SM_pca* needs to double its resources to processing more incoming data points. Two distance values from the PCA space will be generated concurrently. Although multiple distance values can be generated at a given time, *SC_or* can only process them one-by-one due to the feedback between the processing of two adjacent data points (see lines 7-14 in Algorithm 1). This is the reason why we see two grey triangles are scheduled in *t3* and *t4*, respectively. The throughput mismatch between a producer (i.e., *Data_read_pca* + *SM_pca* + *Sum_pca*) and a consumer (i.e., *SC_or*) blocks burst data reading in the PCA space, which explains why a stall occurs repeatedly between two adjacent data readings after *t11* (see Fig. 4b). This issue will become even worse when using less bits (e.g., 8-bit or 4-bit) for data representation. To eliminate these stalls, we propose our second empirical optimization approach (i.e., Optimization Two) as below.

MPCAF-kNN: We found that the performance bottleneck occurs in the *comparison* stage of *SC_or* (see Fig. 4b). Since in more than 95% [8] cases the stages of *rd+dc+kmc* in *SC_or* will not be executed, which implies that most of the distance values obtained in the PCA space are larger than the current maximal value of *heap'* (i.e., *heap'_max*). This observation suggests that multiple comparisons between distance values in the PCA space and the same *heap'_max* can be performed concurrently. Thus, Algorithm 1 can be revised to Algorithm 2. The main difference between them lies in lines 7-14 of Algorithm 1 and lines 10-23 of Algorithm 2, which show how these comparisons are carried out in parallel. Note that the *#pragma HLS UNROLL* directive informs the HLS compiler to flat the specified for-loop. By doing so, multiple distance values from the PCA space can be compared with *heap'_max* in one cycle. If all of them are smaller than *heap'_max*, which is true in more than 95% cases, the subsequent sequential comparisons (line 14-23 of Algorithm 2) can be skipped. A timeline example of Algorithm 2 is shown in Fig. 4c, where all stalls in Fig. 4b disappear. This is because in vast majority cases the comparisons between two data points in the PCA space and *heap'_max* can now be performed in one cycle. That is why the two grey triangles are now reunited into one grey square in Fig. 4c. Without these stalls, the performance of a PCAF-kNN kernel becomes scalable when different low-precision data representations are employed. We name the revised algorithm of PCAF-kNN shown in Algorithm 2 MPCAF-kNN in this paper. Note that Optimization Two can also be applied to a kNN desgin on a CPU or GPU based platform. The reason is that it improves the original PCAF algorithm itself.

Algorithm 2: MPCAF-kNN

Input : q, DB, q', DB', and k
Output: k nearest neighbors

Create and initialize a *heap* of size k with $+\infty$;
Create and initialize a $heap'$ of size $k * m$ with $+\infty$;
data_read(q' and q);
for $i \leftarrow 0$ **to** $N - 1$ **do**
    data_read($p'_i, p'_{i+1}, ..., p'_{i+h-1}$);
    `/* h: # of data points per read */`
    for $j \leftarrow 0$ **to** $h - 1$ **do**
        #pragma HLS UNROLL
        $\delta'_j \leftarrow (q' - p'_{i+j})^2$;
    end
    for $j \leftarrow 0$ **to** $h - 1$ **do**
        #pragma HLS UNROLL
        $(flag \ \& = (\delta'_j > heap'.max))$
    end
    if *flag==0* **then**
        for $j \leftarrow 0$ **to** $h - 1$ **do**
            data_read(p_{i+j});
            $\delta_j \leftarrow (q - p_{i+j})^2$;
            if $\delta_j < heap.max$ **then**
                $heap'.insert(\delta'_j)$;
                $heap.insert(\delta_j)$;
            end
        end
    end
    $i = i + h$;
end
return final k nearest neighbors from *heap*;

5. Evaluation

In this section, we evaluate two optimized kNN kernels: MBFS-kNN (see the first paragraph of Section 4.3) and MPCAF-kNN (see the last paragraph of Section 4.5) in terms of execution time, FPGA resource utilization, and energy-efficiency. A high-end PowerEdge R730xd Rack server [45] is chosen as a baseline platform where BFS-kNN and PCAF-kNN are running. Also, we compare MBFS-kNN with two existing BFS-kNN implementations (on FPGA and GPU) provided by [17]. Note that some of the CPU/GPUs support 16 bit, 8-bit, and even 4-bit computing (Turing). Exhausting all low-precision data representations on CPUs and GPUs requires different platforms and optimizations, which is beyond the scope of this paper. Therefore, we only use a general case (i.e., 32-bit data representation) for kNN running on CPU and GPU.

5.1. Experimental setup

Two datasets, KDD-CUP quantum physics [43] and GIST1M images [46], are selected to evaluate all kNN implementations. The KDD-CUP dataset consists of 50,000 data points with each point having 64 features [43]. Its size is around 63 MB [43]. This dataset stores physical features and a class label of each particle. GIST1M contains one million 960-dimensional data points extracted from a variety of images by using global color GIST descriptors [47]. Its size is 3.8 GB.

Table 1: Place and Route Synthesis Results for MBFS-kNN

(N=1,000,000, D=960, k=5, B=32/16/8/4)

B	32-bit	16-bit	8-bit	4-bit
Freq* (MHz)	303.1	303.2	252.3	210.3
KGU*	100%	100%	100%	100%
KGB* (MB/s)	17,916.6	17,916.5	18,233.9	18,435.7
FF/2,364,480	18,312 (0.8%)	18,277 (0.8%)	109,869 (0.5%)	99,818 (0.4%)
LUT/1,182,240	17,758 (1.5%)	24,054 (2.0%)	198,796 (16.8%)	234,218 (19.8%)
DSP/6,840	128 (1.9%)	64 (0.9%)	196 (2.9%)	512 (7.5%)
BRAM/2,160	88 (4.1%)	172 (8.0%)	532 (24.6%)	1172 (54.3%)
on-chip Power (watt)	17.52	19.02	21.08	20.58

Freq* : Clock frequency;
KGU*: Kernel Gmem Utilization;
KGB*: Kernel Gmem BW;

The CPU server used in our experiments is a PowerEdge R730xd Rack Server, which has two Intel(R) Xeon(R) CPU E5-2699 @ 2.20GHz. Each CPU has 22 physical cores with each core supporting 2 threads. Thus, totally 88 threads can run in parallel in the server. The server has 128 GB DDR4. The FPGA platform that we used is a VCU1525 board [23] whose resource information can be found in Section 4.1. We only used one memory bank in our experiments and its maximal bandwidth is 18 GB/s based on our measurement. The FPGA platform used in [17] is a Terasic DE4 board with a Stratix IV 4SGX530 FPGA and two DDR2 memory banks. The maximal bandwidth of each of the two DDR2 memory banks is 12.75 GB/s. The GPU platform used in [17] is an AMD Radeon HD7950 with 28 compute units and a maximal working frequency of 900 MHz. The board consists of a 3 GB GDDR5 memory with a bandwidth of 240 GB/s. Note that all experimental results of the GPU and Terasic FPGA board presented in this section are provided by [17].

5.2. Evaluation of MBFS-kNN

In this section, we first evaluate MBFS-kNN using a large dataset GIST1M [46]. Next, an evaluation of MBFS-kNN on a small dataset KDD-CUP [43] will be provided. We employ the FLANN library [48] for the implementation of BFS-kNN on CPU in order to improve the baseline performance of BFS-kNN on CPU. FLANN [48] is a library for performing nearest neighbor searches in a high dimensional space. It has been highly optimized by supporting multi-threading. We implemented both single-threaded and multi-threaded BFS-kNN on CPU. The number of threads varies from 1 to 88.

Fig. 5 shows performance comparisons between BFS-kNN running on the PowerEdge CPU server (i.e., "BFS-kNN (CPU)")

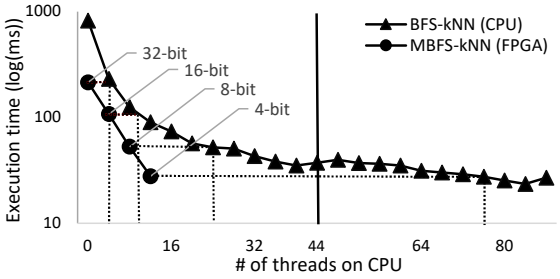

Figure 5: Performance comparisons under GIST1M: BFS-kNN vs MBFS-kNN; note that the scales on the X-axis only apply to BFS-kNN (CPU).

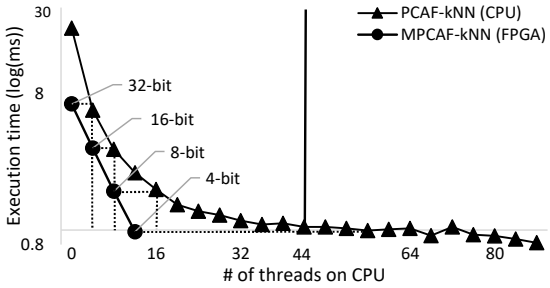

Figure 6: Performance comparisons under GIST1M: PCAF-kNN vs MPCAF-kNN; note that the scales on the X-axis only apply to PCAF-kNN (CPU).

and MBFS-kNN running on the VCU1525 [23] FPGA board (i.e., "MBFS-kNN (FPGA)") under the GIST1M dataset. From Fig. 5, we can see that when both implementations use 32-bit (i.e., B=32) for data representation MBFS-kNN can achieve a performance similar to that of a 4-threaded CPU. In particular, the execution time of a 32-bit MBFS-kNN (FPGA) kernel is 214.1 *ms*, whereas the execution time of a 4-threaded 32-bit BFS-kNN (CPU) is 230.3 *ms*. When using a smaller number of bits to represent each feature the MBFS-kNN kernel can save more clock cycles to fetch each data point from the external DRAM for a similarity comparison. The number of threads on the CPU server needed for an equivalent performance of MBFS-kNN with B being 16, 8, and 4 are 10, 24, and 76, respectively. The main purpose of Fig. 5 is to demonstrate the impact of using a low-precision data representation on the performance of MBFS-kNN. In terms of search accuracy, we observed that when using a 16-bit data representation MBFS-kNN (FPGA) did not lose any search accuracy under the GIST1M dataset. The same observation held when MBFS-kNN (FPGA) used a 16-bit data representation under the KDD-CUP dataset. However, we found that some fake nearest neighbors would be returned if the number of bits for each data point was further reduced to 8 or 4. Thus, as we mentioned in Section 4.3, a proper low-precision data representation should be carefully chosen to guarantee an acceptable search accuracy. From Fig. 5 we can see that an 8-threaded BFS-kNN (CPU) outperforms a 32-bit MBFS-kNN (FPGA) kernel. We also observe that after the number of threads exceeds the number of physical cores in the CPU server (i.e., 44) BFS-kNN (CPU) only slightly improves its performance (see the vertical line at 44 threads in Fig. 5) due to hyperthreading (i.e., more than one thread are running on a physical core).

Table 1 presents a summary of place and route synthesis results targeting the VCU1525 FPGA board [23]. Since a kernel can read more data points at a given time when a smaller value of B is used, to process them in time a higher degree of parallelism is needed on FPGA. As a result, more on-chip resources (e.g., BRAM and DSPs) are needed. This is the reason why we see that the utilizations of LUT, DSP, and BRAM increase when B decreases (see Table 1). From Table 1, we can see that compiling the kernel targeting the FPGA with different configurations of B, the DDR bandwidth (Kernel Gmem Utilization) can always achieve a 100% utilization. The implication is that the maximal data processing bandwidth of the MBFS-kNN kernel is constrained by the DDR bandwidth. Thus, one can speculate that if the data arrival rate of a data stream clustering workload (i.e., a non-stop incoming data stream) is higher than the DDR bandwidth the MBFS-kNN kernel will be overwhelmed. Since the kernel with the configuration of B equal to 4 costs the most hardware resources (see Table 1), the placement and routing on this kernel will be more complex than the other three kernels (i.e., 32-bit, 16-bit, and 8-bit). As long as the SDAccel tool chain [41] cannot meet the timing requirement, it will scale the clock frequency down to the maximum one that it can. In general, a higher resource utilization makes it harder to meet the timing requirement. That is why our results in Table 1 show that the kernel with 32-bit obtains the highest clock frequency (i.e., 300 Mhz). The kernels with 8 and 4 bits just achieve 252.3 and 210.3 Mhz, respectively.

Table 2: Comparisons with the Results from [17]

(N=20,480, D=64, k=20, B=32)

	MBFS-kNN	FPGA[17]	GPU[17]
Execution time (ms)	0.63	3.46	1.24
Frequency (MHz)	300	131.42	900
Objects/second	1,587.30	289.34	804.96
Power (watt)	17.70	24	200
Feature size (nm)	16	40	28
Objects/joule	89.69	12.056	4.024

We also compare the implementation of our MBFS-kNN kernel with two implementations of BFS-kNN on a FPGA and GPU from [17]. Note that the two implementations of BFS-kNN on FPGA and GPU provided by [17] have been optimized by fully exploiting the parallelism of FPGA and GPU, respectively. Since the authors of [17] only used a subset of the KDD-CUP [43] dataset (see Section 5.1) (i.e., 20,480 64-dimensional data points) to conduct their experiments, our MBFS-kNN kernel used the same subset in order to make the comparison fair. Similar to [17], we chose to compute 20 query objects in the experiments of MBFS-kNN and then present the average case for one query in Table 2. An object could be the information of a person (e.g., an image of a person) or a particle (e.g., physical features of a particle). Each object is represented by a data point, which is normally multi-dimensional. Results of the other two implementations shown in this table come from [17]. From the results shown in Table 2, we can see that in

terms of execution time MBFS-kNN outperforms BFS-kNN on FPGA and BFS-kNN on GPU by 5.5x and 1.97x, respectively. Note that the execution time (i.e., called runtime in [17]) of the two implementations provided by [17] was divided by 20 and then presented in Table 2 because that runtime was measured for 20 queries. Energy-efficiency is defined as the number of objects that an kNN kernel can process per joule of energy. Table 2 shows that the energy-efficiency of MBFS-kNN is 7.44x and 22.29x of that of the two existing BFS-kNN implementations from [17]. Although the experiments of the three kNN implementations used the same dataset, the comparisons among them are not completely fair because MBFS-kNN employed an FPGA board different from the one used in [17]. In particular, the clock frequency of the VCU 1525 board used in this paper is 300 MHz, which is 2.28x of that of the Terasic DE4 board used in [17]. However, we can see that the performance of MBFS-kNN kernel is 5.5x of that of the BFS-kNN kernel on FPGA in [17]. The obvious discrepancy between the FPGA hardware performance gap of the two boards (i.e., 2.28x) and the performance gap of the two kNN kernels (i.e., 5.5x) suggests that at least part of performance improvement of MBFS-kNN stems from its optimized algorithm.

5.3. Evaluation of MPCAF-kNN

In the last section, we see how low-precision data representation saves memory bandwidth and improves the performance of an MBFS-kNN kernel (see Fig. 5). Another way to reduce the impact of memory-access bottleneck on the performance of a kNN search is to reduce the number of dimensions of a dataset. PCAF, which is explained in Section 3.2, provides a good solution to do that. The basic idea of PCAF-kNN is to conduct distance calculations in a low dimensional PCA space first. Next, data points that are unlikely to be kNN are filtered out. The first step of PCAF is to select the dimensionality of the PCA space (i.e., d). Theoretically, d can be chosen in the range of [1, D]. A higher value of d implies that more information of the original dataset can be reserved in the PCA space. The value of d has a huge impact on filtering rate and search accuracy. Two other parameters mentioned in Section 4.4, which are also crucial to filtering rate and search accuracy, are heap scaling factor (i.e., m) and the number of *k_max* functional blocks (i.e., s) (see Fig. 3 where s is equal to 3). The main purpose of these two parameters is to provide more *k_max* functional blocks (i.e., $s \times m \times k$) with each having a deeper *heap'* (i.e., $m \times k$). Consequently, more data points in the PCA space can be kept for a further verification, which increases the probability of finding the k nearest neighbors in the original space. Obviously, the benefit of using these two parameters is at the cost of sacrificing the filtering rate. More details about m and s can be found in [8].

Fig. 7 manifests the relationship between search accuracy and filtering rate with different settings of *d-m-s* under GIST1M. Each value of a search accuracy is an average of 1,000-query testings to the GIST1M dataset. While the round dot in the legend of Fig. 7 represents the scenarios where d is equal to 16, the triangle and square stand for d=32 and d=64, respectively. Apparently, one can see from this figure that a larger value of

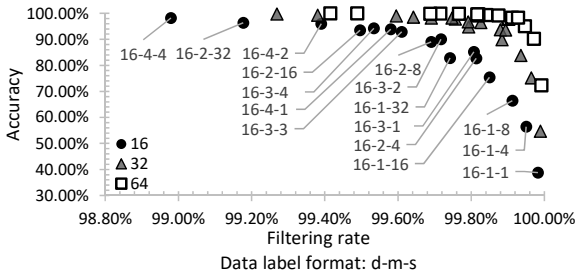

Figure 7: Accuracy vs filtering rate in PCAF under GIST1M.

Table 3: Place and Route Synthesis Results for MPCAF-kNN

(N=1,000,000, D=960, d=16, k=5, m=4, s=4, B=32/16/8/4)

B	32-bit	16-bit	8-bit	4-bit
Freq* (MHz)	300	300	297.7	263.7
KGU*	80.93%	79.45%	76.76%	71.39%
KGB* (MB/s)	9322.95	9152.18	8842.39	8224.57
FF 2,364,480	38446 (1.6%)	38759 (1.6%)	43033 (1.8%)	49958 (2.1%)
LUT 1,182,240	59273 (5.0%)	66892 (5.7%)	58496 (4.9%)	95127 (8.0%)
DSP 6,840	192 (2.8%)	96 (1.4%)	192 (2.8%)	384 (5.6%)
BRAM 2,160	208 (9.6%)	178 (8.2%)	182 (8.4%)	190 (8.7%)
Power (watt)	22.42	22.07	22.43	22.83

Freq* : Clock frequency;
KGU*: Kernel Gmem Utilization;
KGB*: Kernel Gmem BW;

d leads to a higher search accuracy. When d is a fixed number (i.e., 16) a larger value of m or s also improves search accuracy and reduces filtering rate, which can be observed by comparing *16-4-4* vs *16-4-1* and *16-1-1* vs *16-1-16*.

Choosing an appropriate setting of *d-m-s* is important to the performance of PCAF-kNN on FPGA. A larger d will result in a higher search accuracy but reading each data point from DARM will need more clock cycles. So, a smaller d is preferred. We noticed that all filtering rates shown in Fig. 7 are higher than 98%, which means for vast majority of data points their distance calculations only occur in the PCA space. The *16-4-4* setting is chosen for our MPCAF-kNN on the GIST1M dataset as it achieves a high search accuracy (i.e., 98.18%) and a high filtering rate (i.e., 98.98%).

Fig. 6 compares the performance of a PCAF-kNN implementation on the CPU server and MPCAF-kNN on FPGA under the GIST1M dataset. We run the code of PCAF-kNN provided by [8] on the PowerEdge CPU server to obtain the results represented by the PCAF-kNN(CPU) curve shown in Fig. 6. MPCAF-kNN(FPGA) shows the results obtained by our proposed MPCAF-kNN on the VCU1525 FPGA board with *d-m-s* being *16-4-4* and B varying from 4 to 32. Note that the value of B is fixed to 32 for PCAF-kNN(CPU) [8]. Also, the number of threads varies from 1 to 88 for PCAF-kNN(CPU). From Fig. 6 we can see that when both implementations use 32-bit for data

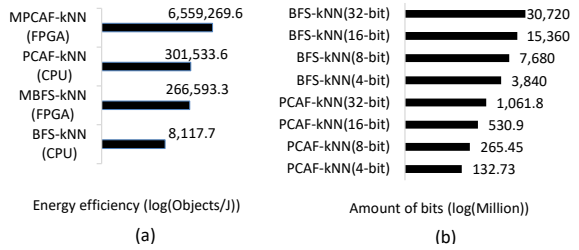

Figure 8: (a) Energy efficiency; (b) memory access.

representation the MPCAF-kNN kernel can achieve a performance similar to that of a 4-thread CPU server. In particular, the execution time of a 32-bit MPCAF-kNN (FPGA) kernel is 6.8 *ms*, whereas the execution time of a 4-threaded 32-bit PCAF-kNN (CPU) is 6.2 *ms*. When a lower data-precision is employed the performance of MPCAF-kNN quickly improves. MPCAF-kNN with a 16-bit, 8-bit, and 4-bit data-precision achieves a performance equivalent to that of a 8-thread, 16-thread, and 56-thread CPU server, respectively. The improvements come from a higher degree of parallelism and pipelining in the FPGA kernel. Also, using a low-precision data representation reduces the impact of memory-access bottleneck. Similar to Fig. 5, we observe that after the number of threads exceeds the number of physical cores in the CPU server PCAF-kNN (CPU) only marginally improves its performance (see the vertical line at 44 threads) due to hyperthreading.

Table 3 summaries system clock frequency, memory access bandwidth, resource utilization, and system power after the placement and routing for MPCAF-kNN. We can see that clock frequency of 32-bit and 16-bit can both achieve 300 MHz. Similar to MBFS-kNN, using a lower precision for data representation will improve the data point reading speed. To match this improvement, more on-chip resources (e.g., DSP) are needed. As we explained before, a higher resource utilization may reduce the quality of the placement and routing, which scales down the final system clock frequency. Thus, clock frequencies of kernels using 8-bit and 4-bit cannot achieve 300 MHz.

Now we evaluate the energy-efficiency of the four kNN implementations. We first removed all unrelated components (e.g. Ethernet card, unused storage, etc.) from the server. Likewise, we stripped the FPGA platform down to its essentials. The goal is to obtain a minimal system that can support the platform to run a kNN implementation. Next, we used a power meter to measure the power consumption of the CPU server and the FPGA platform. We found that the power consumption of the CPU server is 534.9 *watts* when a single-threaded BFS-kNN (CPU) or a single-threaded PCAF-kNN (CPU) is running on the CPU server. It is a reasonable speculation that a 4-threaded BFS-kNN (CPU) or a 4-threaded PCAF-kNN (CPU) would demand a power consumption more than 534.9 *watts* as four CPU cores are working in parallel. Still, let us use 534.9 *watts* as the power consumption for a 4-threaded kNN (CPU) implementation, which gives it some advantages. To make the comparisons fair, we compare the energy-efficiency of a 32-bit kNN FPGA kernel with a 4-threaded kNN CPU implementation because they deliver a similar level of performance (see Fig. 5 and Fig. 6). We argue that an energy-efficiency comparison between two kNN implementations is meaningful only when they offer a similar performance level. Energy-efficiency is defined in the last paragraph of Section 5.2. The energy-efficiency of a 4-threaded 32-bit BFS-kNN (CPU) is 8,117.7 object/joule (i.e., 1,000,000 objects / (230.3 *ms* × 534.9 *watt*) = 8,117.7 object/joule). Note that the GIST1M dataset contains one million data points (see the first paragraph of Section 5.1) and the execution time of a 4-threaded 32-bit BFS-kNN (CPU) is 230.3 *ms* (see Fig. 5). Similarly, the energy-efficiency of a 4-threaded 32-bit PCAF-kNN (CPU) is 301,533.6 object/joule (i.e., 1,000,000 / (6.2 *ms* × 534.9 *watt*) = 301,533.6 object/joule). The power consumption of MBFS-kNN and MPCAF-kNN using 32-bit on FPGA can be found in Table 1 (i.e., 17.52 *watts*) and Table 3 (i.e., 22.42 *watts*). The energy-efficiency of a 32-bit MBFS-kNN (FPGA) kernel is equal to 271,553.2 object/joule (i.e., 1,000,000 / (214.1 *ms* × 17.52 *watt*) = 266,593.3 object/joule). The energy-efficiency of a 32-bit MPCAF-kNN (FPGA) kernel is 6,559,269.6 object/joule (i.e., 1,000,000 / (6.8 *ms* × 22.42 *watt*) = 6,559,269.6 object/joule). Thus, the energy-efficiency of a 32-bit BFS-kNN (FPGA) kernel is 32.84x (i.e., 266,593.3 / 8,117.7 = 32.84) of that of a 4-threaded 32-bit BFS-kNN (CPU). The energy-efficiency of a 32-bit PCAF-kNN (FPGA) kernel is 21.75x (i.e., 6,559,269.6 / 301,533.6 = 21.75) of that of a 4-threaded 32-bit PCAF-kNN (CPU). Fig. 8a summaries the energy-efficiency of the four kNN implementations. All results shown in Fig. 8 are obtained under GIST1M. In Fig. 8, while all bars are plotted on a base-10 logarithmic scale, all numbers are real measurements from our experiments without using any logarithmic function.

5.4. Evaluation of Memory Access

In this section, we present a quantitative analysis of memory access for both MBFS-kNN and MPCAF-kNN in different data-precision representations under the GIST1M dataset. For MBFS-kNN, it needs to literally read each data point and then conduct a similarity comparison with the query q. Thus, the total number of bits to access is $N \times D \times B$. For example, when MBFS-kNN uses a 32-bit data representation it needs to access 1,000,000 × 960 × 32 = 30,720 million bits of data from the external on-board memory. For the MPCAF-kNN kernel, the total amount of data accessed can be calculated by $N\times(F\times d+(1\text{-}F)\times D)\times B$, where F is the filtering rate. For example, a MPCAF-kNN kernel using a 32-bit data representation with a *16-4-4* setting and achieving a 98.18% filtering rate needs to access 1,000,000×(0.9818×16+(1-0.9818)×960)×32 = 1,061.8 million bits of data from the external on-board memory.

Fig.8b summarizes the number of memory accesses of MBFS-kNN and MPCAF-kNN with different precisions of data representation. We can see that the amount of data accessed by MPCAF-kNN (32-bit) is reduced by 28.93x (i.e., 30,720 millions/1,061.8 million = 28.93) compared to BFS-kNN (32-bit) although the latter can perform an accurate kNN search. Using a low-precision data (e.g., 4-bit) can reduce that number further to 231.45x (i.e., 30,720 million/132.73 million = 231.45).

Fig.8b demonstrates that reducing the number of memory accesses becomes crucial to the performance of a kernel on FPGA.

6. Conclusions

In this paper, we design and implement two kNN kernels on FPGA through HLS. Two data access reduction methods (i.e., LPDR and PCAF) are employed to reduce the number of external memory accesses. The experimental results show that our optimized kNN kernels outperform existing ones in both execution time and energy-efficiency. The main contribution of this work lies in developing an array of empirical optimization techniques (e.g., three optimization approaches shown in Section 4.2 and two optimization strategies presented in Section 4.5). These optimization techniques provide insights into how an HPC algorithm originally designed for a CPU platform can be efficiently implemented on FPGA.

Although the two proposed kNN kernels are scalable to all key parameters, they are still "single-threaded" in the sense that each of them has to sequentially process a large dataset. As a result, their turnaround times could be long. To solve this issue, in future work we plan to use a divide-and-conquer approach to splitting a large dataset into multiple data partitions, and then, create multiple kNN kernels. This way multiple kNN kernels can work on distinct data partitions in parallel, similar to a "multi-threaded" execution mode. In this scenario, we need to design a "master" kernel that is able to schedule the workload and merge the outputs from all "salve" kernels in order to obtain the final output. The number of kernels should be configurable. We speculate that the scalability of the two proposed kNN kernels could be substantially improved when they work in the conjectured "multi-threaded" mode.

7. ACKNOWLEDGMENT

This research was supported by Samsung Memory Solution Laboratory (MSL). We thank our colleagues from MSL who provided insight and expertise that greatly assisted the research. This work is also partially supported by the US National Science Foundation under grant CNS-1813485.

References

[1] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, et al., Top 10 algorithms in data mining, Knowledge and Information Systems 14 (1) (2008) 1–37.

[2] J. Gou, H. Ma, W. Ou, S. Zeng, Y. Rao, H. Yang, A generalized mean distance-based k-nearest neighbor classifier, Expert Systems with Applications 115 (2019) 356–372.

[3] J. Gou, L. Sun, L. Du, H. Ma, T. Xiong, W. Ou, Y. Zhan, A representation coefficient-based k-nearest centroid neighbor classifier, Expert Systems with Applications (2022) 116529.

[4] H. Arslan, H. Arslan, A new COVID-19 detection method from human genome sequences using CpG island features and KNN classifier, Engineering Science and Technology, an International Journal 24 (4) (2021) 839–847.

[5] F. Shamrat, S. Chakraborty, M. Imran, J. N. Muna, M. M. Billah, P. Das, O. Rahman, Sentiment analysis on twitter tweets about COVID-19 vaccines using NLP and supervised KNN classification algorithm, Indones. J. Electr. Eng. Comput. Sci 23 (1).

[6] G. Lin, A. Lin, J. Cao, Multidimensional KNN algorithm based on EEMD and complexity measures in financial time series forecasting, Expert Systems with Applications 168 (2021) 114443.

[7] M. Alkhawlani, M. Elmogy, H. El Bakry, Text-based, content-based, and semantic-based image retrievals: A survey, Int. J. Comput. Inf. Technol 4 (01).

[8] H. Feng, D. Eyers, S. Mills, Y. Wu, Z. Huang, Principal Component Analysis Based Filtering for Scalable, High Precision k-NN Search, IEEE Transactions on Computers 67 (2) (2018) 252–267.

[9] A. Torralba, R. Fergus, W. T. Freeman, 80 million tiny images: A large data set for nonparametric object and scene recognition, IEEE transactions on pattern analysis and machine intelligence 30 (11) (2008) 1958–1970.

[10] Z. Wu, Q. Ke, M. Isard, J. Sun, Bundling features for large scale partial-duplicate web image search, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 25–32, 2009.

[11] Microsoft Extends FPGA Reach from Bing to Deep Learning, https://www.nextplatform.com/2015/08/27/microsoft-extends-fpga-reach-from-bing-to-deep-learning/ 2015.

[12] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, S. Jiang, SDA: Software-defined accelerator for large-scale DNN systems, in: 2014 IEEE Hot Chips 26 Symposium (HCS), IEEE, 1–23, 2014.

[13] X. Song, T. Xie, S. Fischer, Two Reconfigurable NDP Servers: Understanding the Impact of Near-Data Processing on Data Center Applications, ACM Transactions on Storage (TOS) 17 (4) (2021) 1–27.

[14] H. M. Hussain, K. Benkrid, 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), IEEE, 205–212, 2012.

[15] E. S. Manolakos, I. Stamoulias, Flexible IP cores for the k-NN classification problem and their FPGA implementation, in: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), IEEE, 1–4, 2010.

[16] F. B. Muslim, A. Demian, L. Ma, L. Lavagno, A. Qamar, Energy-efficient FPGA Implementation of the k-Nearest Neighbors Algorithm Using OpenCL, in: FedCSIS Position Papers, 141–145, 2016.

[17] Y. Pu, J. Peng, L. Huang, 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, IEEE, 167–170, 2015.

[18] W. Zhou, H. Li, Q. Tian, Recent advance in content-based image retrieval: A literature survey, arXiv preprint arXiv:1706.06064 .

[19] G. Amato, F. Falchi, C. Gennaro, F. Rabitti, YFCC100M hybridnet fc6 deep features for content-based image retrieval, in: Proceedings of the 2016 ACM Workshop on Multimedia COMMONS, ACM, 11–18, 2016.

[20] A. Shah, R. Naseem, S. Iqbal, M. A. Shah, et al., Improving CBIR accuracy using convolutional neural network for feature extraction, in: 2017 13th International Conference on Emerging Technologies (ICET), IEEE, 1–5, 2017.

[21] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, M. Oskin, Application Codesign of Near-Data Processing for Similarity Search, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 896–907, 2018.

[22] J. Zhang, S. Khoram, J. Li, Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 207–216, 2017.

[23] Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit, https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html, 2018.

[24] High-level synthesis, https://en.wikipedia.org/wiki/High-level_synthesis, 2016.

[25] J. Eilert, A. Ehliar, D. Liu, Using low precision floating point numbers to reduce memory cost for MP3 decoding, in: IEEE 6th Workshop on Multimedia Signal Processing, 2004., IEEE, 119–122, 2004.

[26] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, C. Zhang, FPGA-

accelerated dense linear machine learning: A precision-convergence trade-off, in: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 160–167, 2017.

[27] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, H. Yang, Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37 (1) (2017) 35–47.

[28] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hannikainen, T. D. Hamalainen, A parallel MPEG-4 encoder for FPGA based multiprocessor SoC, in: International Conference on Field Programmable Logic and Applications, 2005., IEEE, 380–385, 2005.

[29] M. Al Hasan, H. Yildirim, A. Chakraborty, Sonnet: Efficient approximate nearest neighbor using multi-core, in: 2010 IEEE International Conference on Data Mining, IEEE, 719–724, 2010.

[30] A. Beygelzimer, S. Kakade, J. Langford, Cover trees for nearest neighbor, in: Proceedings of the 23rd international conference on Machine learning, ACM, 97–104, 2006.

[31] X. Tang, Z. Huang, D. Eyers, S. Mills, M. Guo, Scalable multicore k-NN search via subspace clustering for filtering, IEEE Transactions on Parallel and Distributed Systems 26 (12) (2014) 3449–3460.

[32] M. Muja, D. G. Lowe, Fast approximate nearest neighbors with automatic algorithm configuration., VISAPP (1) 2 (331-340) (2009) 2.

[33] L. Cayton, Accelerating nearest neighbor search on manycore systems, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE, 402–413, 2012.

[34] S. O'Hara, B. A. Draper, Are you using the right approximate nearest neighbor algorithm?, in: 2013 IEEE Workshop on Applications of Computer Vision (WACV), IEEE, 9–14, 2013.

[35] Y. Tao, K. Yi, C. Sheng, P. Kalnis, Quality and efficiency in high dimensional nearest neighbor search, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, ACM, 563–576, 2009.

[36] Y. Gong, S. Lazebnik, A. Gordo, F. Perronnin, Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval, IEEE transactions on pattern analysis and machine intelligence 35 (12) (2013) 2916–2929.

[37] E. P. Xing, M. I. Jordan, S. J. Russell, A. Y. Ng, Distance metric learning with application to clustering with side-information, in: Advances in neural information processing systems, 521–528, 2003.

[38] I. Jolliffe, Principal component analysis, Springer, 2011.

[39] G. H. Golub, C. Reinsch, Singular value decomposition and least squares solutions, in: Linear Algebra, Springer, 134–151, 1971.

[40] J. E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, Computing in science & engineering 12 (3) (2010) 66.

[41] SDAccel Development Environment, `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, 2018.

[42] UltraScale Architecture Configurable Logic Block, `https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf`, 2017.

[43] KDD-CUP 2004 quantum physics data set, `https://www.kdd.org/kdd-cup/view/kdd-cup-2004/Data`, 2004.

[44] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Quantized neural networks: Training neural networks with low precision weights and activations, The Journal of Machine Learning Research 18 (1) (2017) 6869–6898.

[45] PowerEdge R730xd Rack Server, `https://www.dell.com/en-us/work/shop/povw/poweredge-r730xd`, 2018.

[46] H. Jegou, M. Douze, C. Schmid, Product quantization for nearest neighbor search, IEEE transactions on pattern analysis and machine intelligence 33 (1) (2011) 117–128.

[47] A. Oliva, A. Torralba, Modeling the shape of the scene: A holistic representation of the spatial envelope, International journal of computer vision 42 (3) (2001) 145–175.

[48] M. Muja, D. Lowe, Flann-fast library for approximate nearest neighbors user manual, Computer Science Department, University of British Columbia, Vancouver, BC, Canada .