A Memory-Access-Efficient Adaptive Implementation of kNN on FPGA through HLS

Xiaojia Song
Computational Science
San Diego State University
San Diego, CA, USA
xsong2@sdsu.edu

Tao Xie
Computer Science
San Diego State University
San Diego, CA, USA
txie@sdsu.edu

Stephen Fischer
Memory Solution Lab
Samsung Semiconductor, Inc.
San Jose, CA, USA
sg.fischer@samsung.com

***Abstract*—To reduce the impact of the memory-access constraint in k-Nearest Neighbors (kNN) problems, in this paper we implement one kNN kernel through high-level synthesis (HLS) on FPGA by employing two data access reduction methods: low-precision data representation and principal component analysis based filtering (PCAF). The kernel is called MPCAF-kNN (Memory-efficient PCAF kNN), which has been highly optimized to fully exploit the characteristics of FPGA. It is adaptive to all key parameters. We evaluate MPCAF-kNN by comparing it with a state-of-the-art kNN implementation on a high-end CPU server. Our results show that MPCAF-kNN achieves up to a performance equivalent to that of a 56-thread of CPU server while greatly reducing external memory-accesses.**

***Index Terms*—kNN, FPGA, High-level synthesis, Low-precision data representation, PCA-based filtering, Memory-access-efficient, Adaptive kernel.**

I. Introduction

Existing investigations on accelerating the search of k-Nearest Neighbors (kNN) using FPGA have presented some promising results [3] [6] [8]. 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 recently. For example, the number of images in Tineye's indexed image database has increased from 0.7 billion in 2008 to 35 billion in 2019 [10]. Meanwhile, the number of dimensions of each feature vector could be as large as 4,096 [9]. As a result, a kNN search in a large database with a high dimensionality becomes both compute-intensive and memory-intensive [5]. Unfortunately, a modern FPGA board normally can only provide a moderate throughput between the FPGA chip and its on-board external DRAM.

To reduce the impact of the memory access constraint, in this paper we implement one kNN kernel called MPCAF-kNN (Memory-efficient PCAF kNN) through high-level synthesis (HLS) [1] on FPGA by employing two data access reduction methods: low-precision data representation [4] and principal component analysis based filtering (PCAF) [2]. Low-precision data representation 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 [4]. PCAF, on the other hand, uses a data filtering mechanism to exclude those reference features that are not likely to be k-NN features according to the PCA estimation [2]. Although the idea of PCAF is borrowed from a recent research paper [2], this study is the first attempt to apply PCAF in a kNN kernel implementation on FPGA. The kernel has been highly optimized to fully exploit the characteristics of FPGA. Besides, it is adaptive to all key parameters including 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 MPCAF-kNN in terms of performance and energy-efficiency by comparing it with a state-of-the-art kNN implementations on a high-end CPU server. This paper makes the following contributions. First, to the best of our knowledge, this is the first research utilizes a PCA-based data filtering mechanism to reduce memory accesses of a kNN kernel running on FPGA. Second, two approaches are proposed to optimize MPCAF-kNN through HLS. (see Section III-C). The rest of paper is organized as follows. Section II briefly summarizes the related work. Section III provides implementation details of the kernel. Section IV evaluates the kernel. Finally, Section V concludes the paper.

II. Related work

Traditional kNN implementations like [3] were all developed using an HDL. Hussain *et al.* proposed two adaptive FPGA architectures of the kNN classifier using HDL [3]. After HLS became available, recent kNN implementations switched to HLS to fully exploit its advantages. For examle, Pu *et al.* employed a specific bubble sort algorithm to speed up the sorting phase of a BFS-kNN (Brute-Force Searching kNN) algorithm using OpenCL [8]. Muslim *et al.* also accelerated a BFS-kNN search using FPGA under the OpenCL framework [6]. They found that an optimized FPGA-based kNN kernel could offer performance and energy-efficiency better than a GPU-based kNN implementation [6]. Unfortunately, both [6] [8] are not suitable for kNN searching in a large dataset because they store all the temporary distance data in on-chip memory whose size is usually very limited.

III. Implementation and Optimization

In this section, we first introduce the FPGA hardware resources of VCU1525 [1]. Next, we elaborate the details of implementation and optimization of MPCAF-kNN.

978-1-5386-6648-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICCD46524.2019.00030

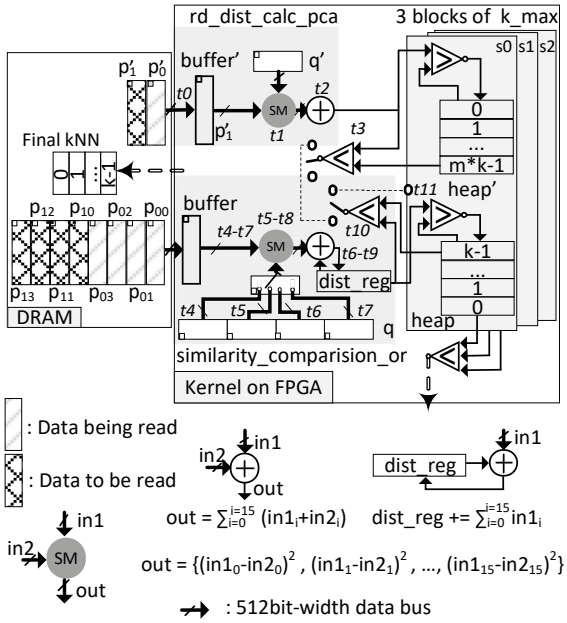

Figure 1: Implementation of PCAF-kNN on FPGA.

A. FPGA Resources of VCU1525

On-board memory external to FPGA chip: Totally, there are 4 SDRAM banks (64 GByte on-board DDR SDRAM). The maximal bandwidth to access an individual memory bank is 512 bits per clock cycle [1]. The limited bandwidth largely constrains the performance of a memory-intensive application.

B. PCA-based data filtering

The details of PCAF-kNN on CPU can be found in [2]. Our implementation of PCAF-kNN on FPGA is illustrated in Fig. 1. *k_max* block maintains a temporary heap called *heap'* whose size is *m*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 point in the original *DB* space and q.

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. 1). 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. 1).

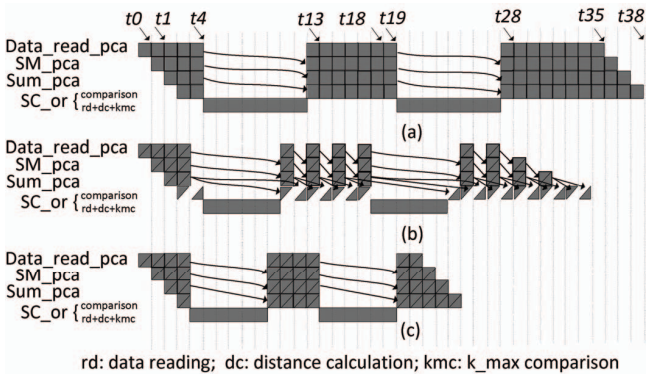

Figure 2: PCAF-kNN timelines: (a) 32-bit; (b) 16-bit; (c) 16-bit optimized.

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. 1).

C. Two optimizations of PCAF-kNN

Simply transplanting a PCAF-kNN algorithm designed for a CPU platform into an FPGA-based heterogeneous system could lead a poor performance. In this section, we propose two techniques that can optimize the performance of PCAF-kNN on FPGA.

*Optimization one: packaging tasks with feedback into one function.*If performing a stage is viewed as a task, a task dependency can be observed in line 5 of Algorithm-1 in work [2]. 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. This blocks the pipelining.

We package the task of data reading in the PCA space as a function called *data_read_pca()*. Similarly, the task of distance calculation in the PCA space (see line 4 of Algorithm-1 in work [2]) is packaged as *dist_calc_pca()*. Finally, tasks from line 5 to line 11 in Algorithm-1 in work [2] are packaged as a function named *similarity_com_or()*. Now, the three functions can be successfully pipelined as they do not have feedback among each other.

Optimization two: comparing distance values in PCA space in parallel. A timeline example of PCAF-kNN (see Fig. 1) on the dataset (d = 16, D = 64, B = 32, totally 20 data points) is shown in Fig. 2a. In the *rd_dist_calc_pca* block shown in Fig. 1, the data point p'_0 in the PCA space can be read from the DRAM into a kernel in one cycle at time *t0* and then it goes to *SM_pca* at time *t1* as shown in Fig. 2a, where 16 subtractions with the features of q' and square operations are

Algorithm 1: Optimized MPCAF-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, p'_{i+1}, ..., p'_{i+h−1});
       /* h: # of data points per read                    */
6      for j ← 0 to h − 1 do
7          #pragma HLS UNROLL
8          δ'_j ← (q' − p'_{i+j})^2;
9      end
10     for j ← 0 to h − 1 do
11         #pragma HLS UNROLL
12         (flag &= (δ'_j > heap'.max))
13     end
14     if flag==0 then
15         for j ← 0 to h − 1 do
16             data_read(p_{i+j});
17             δ_j ← (q − p_{i+j})^2;
18             if δ_j < heap.max then
19                 heap'.insert(δ'_j);
20                 heap.insert(δ_j);
21             end
22         end
23     end
24     i = i + h;
25 end
26 return final k nearest neighbors from heap;
```

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. 1. In the timeline example shown in Fig. 2a, 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. 1) is not invoked as a further examination in the original space for this data point becomes unnecessary. 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. 1. Since *similarity_comparison_or* conducts a similarity comparison in the original space, it needs 4 (i.e., *D*B*/512=64*32/512=4) cycles to read a data point from DDR. 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. 1) while avoiding to invoke the *similarity_comparison_or* block. Based on the results from [2], 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) is larger than 95%. This why PCAF can greatly accelerate the kNN search.

Now, we integrate low-precision data representation into PCAF-kNN to further reduce the impact of memory access constraint. Fig. 2b 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 DRAM, 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. 2a, 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. 2b consists of two triangles with each representing a data point in 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 at a given time multiple distance values can be generated, *SC_or* can only process them one-by-one due to the feedback between the processing of two adjacent data points (see line 5 to line 11 in Algorithm-1 in work [2]). This is the reason why we see two grey triangles are scheduled in *t3* and *t4*, respectively. This explains why a stall occurs repeatedly between two adjacent data readings after *t11* (see Fig. 2b). This issue will be even worse when using less bits (e.g. 8-bit or 4-bit) for data representation. Our goal is to eliminate these stalls.

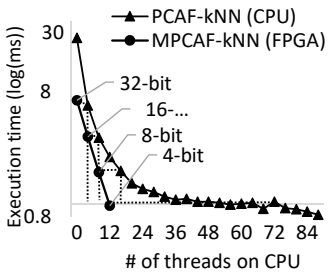

Figure 3: Performance comparison

One approach is to optimize Algorithm-1 in work [2]. We found that the performance bottleneck occurs in the *comparison* stage of *SC_or* (see Fig. 2b). Since in more than 95% [2] 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 in [2] can be revised to Algorithm 1. The main difference between them lies in line 5 of Algorithm-1 in work [2] and lines 10-13 of Algorithm 1, which show how these comparisons are carried out in parallel. Note that the *#paragma 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. A timeline of Algorithm 1 is shown in Fig. 2c, where all stalls in Fig. 2b disappear. We name the optimized algorithm of PCAF-kNN shown in Algorithm 1 MPCAF-kNN in this paper. Note that this optimization also applies to the kNN desgin on CPU or GPU.

Table I: Place and route synthesis results for MPCAF-kNN

(N=1000000, D=960, d=16, k=5, m=4, s=4, B=32/16/8/4)

B	32-bit	16-bit	8-bit	4-bit
Clock fr equency (Mhz)	300	300	297.7	263.7
Kernel Gmem Utilization	80.93%	79.45%	76.76%	71.39%
Kernel Gmem BW(MB/s)	9322.95	9152.18	8842.39	8224.57
FF/2,364,480	38446	38759	43033	49958
LUT/1,182,240	59273	66892	58496	95127
DSP/6,840	192	96	192	384
BRAM/2,160	208	178	182	190
Power (Watt)	22.42	22.07	22.43	22.83

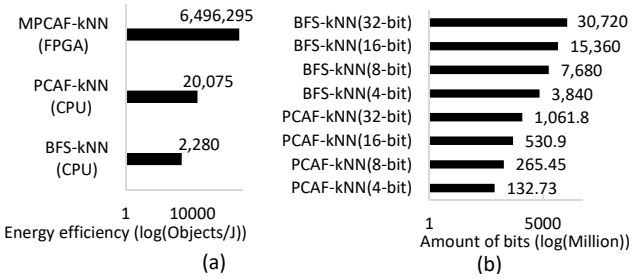

Figure 4: (a) Energy efficiency; (b) Memory access.

IV. Evaluation

A. Experimental Setup

1) Datasets: GIST1M is selected to evaluate all kNN implementations. It's 3.8 GB and contains one million 960-dimensional data points [7].

2) Baseline: ***CPU server:*** PowerEdge R730xd Rack Server has two Intel(R) Xeon(R) CPU E5-2699 @ 2.20GHz. Each CPU has 22 physical cores and 2 threads can run on each core (i.e., totally 88 threads). The server has 128 GB DDR4.

B. Evaluation of MPCAF-kNN

Theoretically, d can be chosen in the range of [1, D]. Two other parameters mentioned in Section III-B, 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. 1 where s is equal to 3). The *16-4-4* (d-m-s) 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. 3 compares the performance of a PCAF-kNN implementation on the CPU server and MPCAF-kNN on FPGA under the GIST1M dataset. 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. From Fig. 3 we can see that when both implementations use 32-bit for data representation the MPCAF-kNN kernel can achieve a performance equivalent to that of a 4-thread CPU server. 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

Table I summaries system clock frequency, memory access bandwidth, resource utilization, and system power after the placement and routing for MPCAF-kNN. Fig. 4a summaries the energy efficiency of four kNN implementations on the CPU server and FPGA. All kNN implementations use a single thread or kernel. Base on our results, we found that MPCAF-kNN achieves the highest energy-efficiency, which is 2,849x and 324x higher than that of BFS-kNN(CPU) and PCAF-kNN(CPU), respectively.

C. Evaluation of memory access

Fig.4b summarizes the amount of memory accesses of 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 28.9x compared to BFS-kNN (32-bit). Also, using a low-precision data (e.g., 4-bit) can reduce that number further to 231.46x .

V. Conclusions

In this paper, we design and implement one kNN kernel on FPGA through HLS. Two data access reduction methods are employed to reduce the external memory accesses. The kernel is adaptive to all key parameters. Further, we evaluate it under different settings. The experimental results show that our optimized MPCAF-kNN kernel outperforms existing ones in both execution time and energy-efficiency. We plan to release the source code of the two kernels to benefit the community.

VI. 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] Xilinx virtex ultrascale+ fpga vcu1525 acceleration development kit. https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html, 2018.

[2] H. Feng, D. Eyers, S. Mills, Y. Wu, and Z. Huang. Principal component analysis based filtering for scalable, high precision k-nn search. *IEEE Transactions on Computers*, 67(2):252–267, 2018.

[3] 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 AHS*, pages 205–212. IEEE.

[4] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. Fpga-accelerated dense linear machine learning: A precision-convergence trade-off. In *FCCM*, pages 160–167. IEEE, 2017.

[5] 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 *IPDPS*, pages 896–907. IEEE, 2018.

[6] F. B. Muslim, A. Demian, L. Ma, L. Lavagno, and A. Qamar. Energy-efficient fpga implementation of the k-nearest neighbors algorithm using opencl. In *FedCSIS Position Papers*, pages 141–145, 2016.

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

[8] Y. Pu, J. Peng, L. Huang, and J. Chen. An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl. In *FCCM*, pages 167–170. IEEE, 2015.

[9] A. Shah, R. Naseem, S. Iqbal, M. A. Shah, et al. Improving cbir accuracy using convolutional neural network for feature extraction. In *ICET*, pages 1–5. IEEE, 2017.

[10] W. Zhou, H. Li, and Q. Tian. Recent advance in content-based image retrieval: A literature survey. *arXiv preprint arXiv:1706.06064*, 2017.