BiS-KM: Enabling Any-Precision K-Means on FPGAs

Zhenhao He, Zeke Wang, Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

ABSTRACT

K-Means is a popular clustering algorithm widely used and extensively studied in the literature. In this paper we explore the challenges and opportunities in using low precision input in conjunction with a standard K-Means algorithm as a way to improve the memory bandwidth utilization on hardware accelerators. Low precision input through quantization has become a standard technique in machine learning to reduce computational costs and memory traffic. When applied in FPGAs, several issues need to be addressed. First and foremost is the overhead of storing the data at different precision levels since, depending on the training objective, different levels of precision might be needed. Second, the FPGA design needs to accommodate varying precision without requiring reconfiguration. To address these concerns, we propose *Bit-Serial K-Means* (BiS-KM), a combination of a hybrid memory layout supporting data retrieval at any level of precision, a novel FPGA design based on bit-serial arithmetic, and a modified K-Means algorithm tailored to FPGAs. We have tested BiS-KM with various data sets and compared our design with a state-of-the-art FPGA accelerator. BiS-KM achieves an almost linear speedup as precision decreases, providing a more effective way to perform K-Means on FPGAs.

KEYWORDS

FPGA, K-Means, Bit-Serial Arithmetic, Low-Precision Clustering, Memory Layout

ACM Reference Format:
Zhenhao He, Zeke Wang, Gustavo Alonso. 2020. BiS-KM: Enabling Any-Precision K-Means on FPGAs. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375316

1 INTRODUCTION

K-Means is a popular clustering algorithm. K-Means aims to partition samples into a predefined number of clusters in a way that each sample belongs to the cluster with the minimum distance. Calculating distances requires massive computational efforts, especially when the number of clusters and the dimension of a sample are large. Due to its computational overhead, it has been a common target for optimization including offloading onto an FPGA to take advantage of the inherent parallelism on the chip [13, 16, 25, 36, 41].

A promising approach to accelerate K-Means is to use low precision, in terms of quantization (reducing normalized fixed-point or floating-point data to a few bits using statistically sound methods). Quantization has become a standard technique in machine learning [12, 14, 23, 49] but most existing K-Means designs focus on high-precision (e.g., 32-bit fixed-point or floating-point). Therefore, the question we address in this paper is whether we can use quantized data to compute K-Means and how to do it on an FPGA. Supporting quantized data is promising because: (1) the FPGA is usually bounded by the memory bandwidth so using lower precision (e.g., 8-bit instead of 32-bit) should reduce the total amount of data movement from the memory to the FPGA, thereby reducing the training time. (2) Quantized data requires a smaller logic footprint of the corresponding arithmetic units which should allow us to instantiate significantly more such units for the same area and power budget.

Despite these potential advantages, There are two important obstacles to leveraging low precision. The first is the overhead of quantization. One typical form of quantization involves storing the data in high-precision format and then doing the reduction (i.e., converting high-precision data to low-precision data) before processing the data on the FPGA. This approach causes significant computation overhead. Another way to do it is to maintain a separate copy of the data set for each precision level, resulting in 32 copies of the data (from 1-bit to 32-bit), which results in significant storage overhead. The second obstacle is the substantial hardware development effort. In order to fully take advantage of low precision, a hardware accelerator is needed for each precision level. Otherwise, the data in low precision (e.g., 5-bit) has to be padded with zeros to match the supported precision level (e.g., 8-bit) before computing K-Means.

To address these issues, we propose Bit-serial K-Means (BiS-KM), a novel system that employs an algorithm-software-hardware co-design methodology to enable K-Means to support any-precision clustering on the FPGA. BiS-KM consists of a variant of the K-Means algorithm (**C1**), a custom bit-serial memory layout (**C2**), and a bit-serial hardware accelerator (**C3**). The key idea of BiS-KM is to tightly integrate these three components to enable any-precision clustering while keeping the resource consumption reasonably low.
C1: A Variant of the K-Means Algorithm. Calculating the Euclidean distance does not work well with bit-serial arithmetic as a naive approach leads to significant resource consumption (Subsection 4.1), making it hard to fit into a mid-size FPGA. Accordingly, we propose a variant of the K-Means algorithm amenable to hardware implementation while still producing the same results as the original version.
C2: Custom Bit-serial Memory Layout. Inspired by existing bit-serial memory layouts [24, 43] supporting efficient data retrieval at different precision levels, we design a custom bit-serial memory layout tailored to the proposed variant of the K-Means algorithm. The new memory layout allows data to flow into the any-precision

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375316

Table 1: Notation used in the paper

Term	Definition
D	Number of dimensions of a sample
K	Number of clusters used in the training
N	Number of samples in the data set
$\vec{x}$	Sample vector
$\vec{\mu}$	Center vector
P_h	High-precision bitwidth
P_l	Low-precision bitwidth used at runtime
$a^{[i]}$	i-th bit of fixed-point value a
$Q_p(A)$	p-bit quantized value of high-precision value A
W	Number of bits of a memory transaction
$\#pipe$	Number of pipelines in the hardware
$DIFP$	Degree of inter-feature parallelism associated with any sample
$DISP$	Degree of inter-sample parallelism associated with a memory transaction

K-Means accelerator without any transposition overhead, which should lead to a lower resource consumption on the FPGA. (Subsection 5.2).

C3: Custom Bit-serial Hardware Accelerator. Based on the variant of the K-Means algorithm, we develop a custom bit-serial hardware accelerator supporting any-precision clustering on the FPGA, while keeping the resource consumption reasonably low. The key idea of the hardware accelerator is to use bit-serial arithmetic [1, 19, 45] to directly consume the data stream from the custom bit-serial memory layout (Subsection 5.1).

The experimental results show that 1) BiS-KM with low-precision data converges to the same loss as when using high precision data, and 2) BiS-KM achieves an almost linear performance improvement as the precision level decreases.

2 BACKGROUND

We now present the preliminaries for understanding BiS-KM. Table 1 contains the notation used throughout the paper.

2.1 K-Means Algorithm

The K-Means algorithm, also known as Lloyd's algorithm [26], is an unsupervised clustering algorithm that partitions N samples into K clusters $S = S_1, S_2 ...., S_K$ minimizing the squared error between the empirical mean of a cluster and the samples in the cluster (i.e., within-cluster sum of square errors):

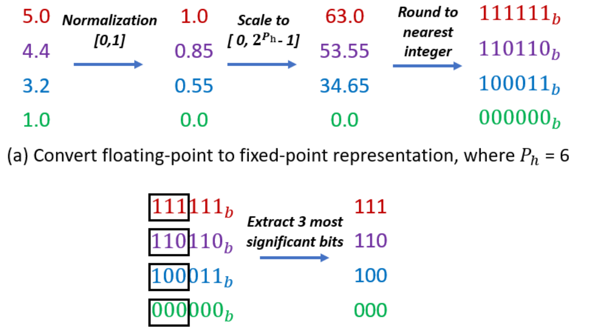
$$\underset{S}{\text{argmin}} \sum_{i=1}^{K} \sum_{\vec{x} \in S_i} ||\vec{x} - \vec{\mu}_i||^2, \tag{1}$$

where $\vec{x}$ is a sample vector of D dimensions and $\vec{\mu}_i$ is the mean of all samples in S_i.

The algorithms itself consists of three steps: (1) the initialization step, picking K random centers (centroids) for the clusters, (2) the assignment step, where each sample is assigned to its closest center by comparing the squared Euclidean distance ($||\vec{x} - \vec{\mu}_i||^2$), and (3) the update step, where the centers are recalculated as the mean of the samples assigned to them. The algorithm is executed either for a fixed number of iterations or until the center assignments no longer vary.

2.2 Float-to-Fixed Conversion and Fixed-Point Quantization

To explain how to quantize the raw data to low-precision fixed-point data, we assume that a data point consists of multiple dimensions, each of which is represented in a floating-point format.



(a) Convert floating-point to fixed-point representation, where P_h = 6

(b) Quantizing high-precision to low-precision, where P_h = 6 and P_l = 3

Figure 1: The processes of quantizing the original floating-point numbers to 3-bit low-precision fixed-point numbers

Float-to-Fixed Conversion. Figure 1(a) shows the conversion of a floating-point value to its corresponding fixed-point value. For each dimension, we normalize the original value g to the range [0, 1] and then scale it to a new value g' within range [0, $2^{P_h} - 1$] as follows:

$$g' = \frac{g - g_{min}}{g_{max} - g_{min}} \times (2^{P_h} - 1), \tag{2}$$

where P_h is the high-precision bitwidth of a fixed-point number, g_{min} and g_{max} are the minimum and maximum values for that dimension respectively. Afterwards, each floating-point g' is rounded to the nearest integer number a. The new fixed-point a can be represented as $\sum_{i=0}^{P_h-1}(a^{[i]} << i)$, where $a^{[i]}$ represents the i-th bit of a, with $a^{[P_h-1]}$ meaning the most significant bit.

Fixed-Point Quantization. A high-precision fixed-point data a is quantized to P_l-bit low-precision data $Q_P(a)$ by extracting the P_l most significant bits: $Q_P(a) = \sum_{i=P_l}^{P_h-1}(a^{[i]} << i)$. Figure 1(b) illustrates the process of quantizing each 6-bit high-precision value to a 3-bit low-precision value.

2.3 Bit-Serial Arithmetic

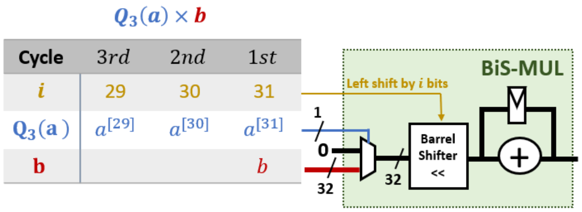
In this paper we heavily rely on bit-serial multipliers (BiS-MUL)[6, 35] and bit-serial dot product (BiS-DP) [1, 19]. Bit-serial arithmetic allows the flexible selection of the precision used in the computation, but results in high latency and, potentially, low throughput due to its inherently serial nature [45].

2.3.1 *Bit-Serial Multiplier (BiS-MUL).*

To illustrate how BiS-MUL works, we use an example multiplying a 3-bit quantized value $Q_3(a)$ by a 32-bit high-precision value b. $Q_3(a)$ is obtained by extracting the three most significant bits of a, represented by $\sum_{i=29}^{31}(a^{[i]} << i)$. Therefore, $Q_3(a) \times b$ can be rewritten to $b \times \sum_{i=29}^{31}(a^{[i]} << i)$, which is equivalent to $\sum_{i=29}^{31}(a^{[i]} \times b) << i$, as shown in Equation 3.

$$Q_3(a) \times b = b \times \sum_{i=29}^{31}(a^{[i]} << i) = \sum_{i=29}^{31}(a^{[i]} \times b) << i \tag{3}$$

A BiS-MUL unit has two inputs: a bit-serial input and a bit-parallel input. The bit-parallel input, i.e., the value b, arrives at the BiS-MUL unit every three cycles in Figure 2, while the bit-serial



Cycle	3rd	2nd	1st
i	29	30	31
$Q_3(a)$	$a^{[29]}$	$a^{[30]}$	$a^{[31]}$
b			b

Figure 2: Micro-architecture of BiS-MUL and an example of calculating $Q_3(a) \times b$, where $Q_3(a)$ is a quantized value with 3-bit precision, $a^{[i]}$ is the i-th bit of a, and b is a 32-bit high-precision value.

input $Q_3(a)$ arrives at the BiS-MUL unit one bit per cycle, starting from the most significant bit $a^{[31]}$ to $a^{[29]}$. Therefore, the product $Q_3(a) \times b$ is set to $(a^{[31]} \times b) << 31$ in the first cycle. $(a^{[30]} \times b) << 30$ is added to the product in the second cycle, while $(a^{[29]} \times b) << 29$ is added in the third cycle. Note that the computation of $a^{[i]} \times b$ can be easily implemented with a multiplexer, since $a^{[i]}$ is just a single bit. Therefore, the BiS-MUL unit can be realized with a multiplexer and a simple shift-and-add logic, requiring few resources.

2.3.2 Bit-Serial Dot Product (BiS-DP).

To illustrate BiS-DP, we use an example performing a dot product $\vec{A} \cdot \vec{B}$ of two vectors $\vec{A}$ and $\vec{B}$, each of which has D=4 features, where the d-th feature of $\vec{A}$ and $\vec{B}$ are represented by a_d and b_d, respectively, and d is from 0 to D-1=3. a_d is quantized to a 3-bit value $Q_3(a_d)$, while b_d remains in 32-bit precision.

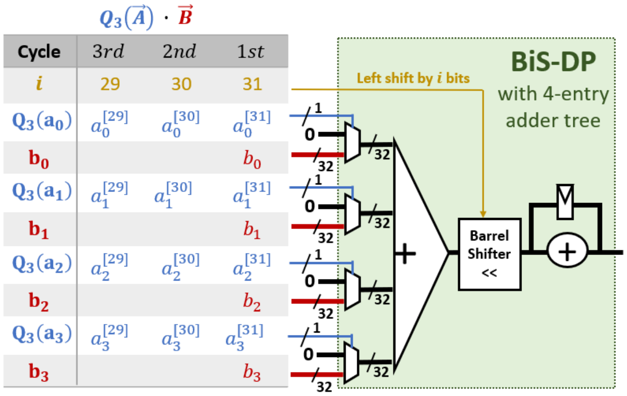
Naive approach. The simplest way to implement a fully pipelined $\vec{A} \cdot \vec{B}$ in Equation 4 is to use a BiS-MUL unit to perform a bit-serial multiplication on the d-th feature: $b_d \times \sum_{i=29}^{31}(a_d^{[i]} << i)$, followed by a D=4-*entry* adder tree able to immediately consume the multiplication results from four BiS-MULs. The adder tree is fully utilized only for 1-bit precision. In this example, its temporal utilization is 33.3% since inputs of the adder tree are valid every three cycles.

Efficient implementation. To increase the temporal hardware utilization, we can leverage the commutative property of addition to interchange the order of summations in Equation 4, yielding Equation 5. Now, we can use an adder tree to calculate the summation $\sum_{d=0}^{3}(a_d^{[i]} \times b_d)$, and then use a shifted accumulator to accumulate the sum from the adder tree. In such a way, the operation requires fewer resources to implement since it only needs one shifted accumulator that is on the output of the adder tree, instead of using the four shifted accumulators on the input side of the adder tree of the naive approach.

$$\sum_{d=0}^{3} Q_3(a_d) \times b_d = \sum_{d=0}^{3} b_d \times \sum_{i=29}^{31} (a_d^{[i]} << i) \tag{4}$$

$$= \sum_{i=29}^{31} (\sum_{d=0}^{3} a_d^{[i]} \times b_d) << i \tag{5}$$

Figure 3 shows the per-cycle inputs of a BiS-DP unit. In the first cycle, the most significant bit of the d-th feature of $\vec{A}$, $a_d^{[31]}$, is multiplied with b_d of $\vec{B}$, where d is from 0 to 3. Then, the four products enter the 4-entry adder tree which produces an accumulated product. The product is left shifted by 31, and then accumulated into the final result. Similarly, the second (or third) most significant bits of the four features of $\vec{A}$ enter the BiS-DP unit in the second (or third) cycle. However, we do not need to feed $\vec{B}$ again, as $\vec{B}$ stays the same. After three cycles, the accumulated output of $Q_3(\vec{A}) \cdot \vec{B}$ has been computed.



Cycle	3rd	2nd	1st
i	29	30	31
$Q_3(a_0)$	$a_0^{[29]}$	$a_0^{[30]}$	$a_0^{[31]}$
b_0			b_0
$Q_3(a_1)$	$a_1^{[29]}$	$a_1^{[30]}$	$a_1^{[31]}$
b_1			b_1
$Q_3(a_2)$	$a_2^{[29]}$	$a_2^{[30]}$	$a_2^{[31]}$
b_2			b_2
$Q_3(a_3)$	$a_3^{[29]}$	$a_3^{[30]}$	$a_3^{[31]}$
b_3			b_3

Figure 3: BiS-DP processing 4 features in parallel and an example calculating $Q_3(\vec{A}) \cdot \vec{B}$. The features of $\vec{A}$ are quantized to 3-bit precision and $a^{[i]}$ represents the i-th bit of a. The features of $\vec{B}$ are 32-bit high-precision.

3 SYSTEM OVERVIEW

The goal of BiS-KM is to achieve any-precision clustering while keeping resource consumption at a reasonable level. Using bit-serial multiplication when computing K-Means enables any-precision clustering, but also brings in a well-known challenge: its throughput is potentially low due to its inherent bit-serial nature, e.g., processing only one bit (instead of one element) per cycle. Intuitively, we need to instantiate more bit-serial computing units (requiring more FPGA resources) to achieve a similar throughput as that of existing bit-parallel K-Means designs.

In the following we describe the Bit-Serial K-Means (BiS-KM) and its implementation on an FPGA. We adopt an algorithm-software-hardware co-design methodology to enable K-Means to support any-precision clustering on the FPGA. First, we develop a variant of K-Means amenable to any-precision hardware implementation (Section 4). Second, based on this variant, we adopt a software-hardware co-design approach to efficiently implement the BiS-KM algorithm on FPGAs (Section 5).

4 BIT-SERIAL K-MEANS ALGORITHM

Using a conventional K-Means algorithm over quantized data can be done keeping the cluster means in high-precision format. The resulting algorithm can be expressed as:

$$\underset{i}{\text{argmin}} \|Q(\vec{x}) - \vec{\mu}_i\|^2, \tag{6}$$

where the $Q(\vec{x})$ represents the quantized sample. The calculation of the Euclidean distance of each sample to all the centers represents

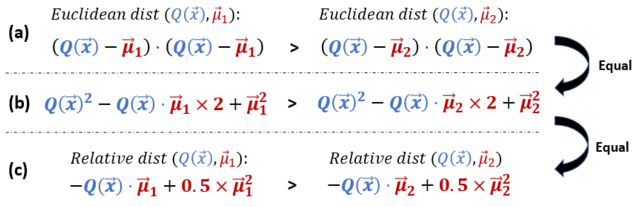

(a)
Euclidean dist $(Q(\vec{x}),\vec{\mu}_1)$:
$(Q(\vec{x})-\vec{\mu}_1)\cdot(Q(\vec{x})-\vec{\mu}_1)$
>
Euclidean dist $(Q(\vec{x}),\vec{\mu}_2)$:
$(Q(\vec{x})-\vec{\mu}_2)\cdot(Q(\vec{x})-\vec{\mu}_2)$
Equal
(b) $Q(\vec{x})^2-Q(\vec{x})\cdot\vec{\mu}_1\times 2+\vec{\mu}_1^2$ > $Q(\vec{x})^2-Q(\vec{x})\cdot\vec{\mu}_2\times 2+\vec{\mu}_2^2$
Equal
(c)
Relative dist $(Q(\vec{x}),\vec{\mu}_1)$:
$-Q(\vec{x})\cdot\vec{\mu}_1+0.5\times\vec{\mu}_1^2$
>
Relative dist $(Q(\vec{x}),\vec{\mu}_2)$
$-Q(\vec{x})\cdot\vec{\mu}_2+0.5\times\vec{\mu}_2^2$

Figure 4: Conversion from Euclidean distance (a) to Relative distance (c). Variables in blue are in bit-serial and red variables in bit-parallel format.

the majority of the computational load. As a first step, we use bit-serial arithmetic for the distance calculation and discuss the issues arising from the direct adoption of bit-serial arithmetic.

4.1 Bit-serial Euclidean distance

Using bit-serial arithmetic to implement the Euclidean distance computation can be done in two steps. First, we conduct the bit-serial subtraction between the bit-serial sample $Q(\vec{x})$ and the bit-parallel center $\vec{\mu}_i$. The corresponding output is bit-parallel. Second, we compute the square of the subtraction in a bit-parallel fashion.

This direct approach leads to significant resource consumption in terms of DSPs. It also leads to low utilization of these DSPs. In order to consume one W-bit memory transaction per cycle regardless of the precision, W bit-serial subtractions need to be instantiated, which do not consume too many FPGA resources. However, we then have to instantiate W bit-parallel $P_h \times P_h$ multipliers, used to compute the squares, to consume the data from the W bit-serial subtractions for one cluster, with one multiplier dedicated to each bit. As an example, to support the concurrent execution of $K = 8$ clusters and to saturate $W = 512$ memory bandwidth, a total of $8 \times 512 = 4,096$ bit-parallel multipliers would be required. Bit-parallel multipliers are implemented with DSPs in the FPGA. In our target Intel Arria 10 FPGA, there are 1,368 DSPs, each of which containing one 27×27 multiplier [17]. In the case of $P_h = 32$, one 32×32 multiplier can be implemented by either one DSP plus supporting soft logic or three DSPs working together. Either way, the required number of DSPs is far above the capacity of the Arria 10 board. Another issue worth mentioning is that the utilization of these bit-parallel multipliers is $1/P_l$, where P_l is the runtime precision. The reason is that each bit-serial subtraction can only produce a bit-parallel output every P_l cycles, while each bit-parallel multiplier is able to accept a bit-parallel multiplication each cycle.

From these calculations, we conclude that the using bit-serial arithmetic to calculate the Euclidean distance leads to a resource consumption larger than what is typically available in a mid-sized FPGA.

4.2 From Euclidean to Relative Distance

To avoid the problems described above, we transform the Euclidean distance into a *Relative* distance allowing us to fully exploit bit-serial computations on the FPGA. This Relative distance needs to preserve the same clustering results as the Euclidean distance. Figure 4 describes the transformation, using an example assigning one quantized sample $Q(\vec{x})$ to one of the two cluster centers $\vec{\mu}_1$ and $\vec{\mu}_2$. The transformation is done in two steps.

Algorithm 1: Bit-Serial K-Means (BiS-KM) Algorithm

```
   input  : $Q(\vec{x})$: quantized sample data,
            $K$: number of clusters,
            $t_{max}$: number of iterations
1  $t = 0$;
2  Randomly initialize $K$ centers $\vec{\mu}_1^0, \vec{\mu}_2^0, \ldots, \vec{\mu}_k^0$ at iteration 0;
   while $t < t_{max}$ do
3      $cnt_i, sum_i = 0$ for all $i = 1, \ldots, K$;
4      //Calculate the squared $L^2$ norm of each center
5      foreach $i = 1$ to $K$ do
6          $norm_i = \|\vec{\mu}_i^t\|^2 \times 0.5$
7      end
8      // Cluster Assignment Step
9      foreach $Q(\vec{x})$ do
10         $i^* = \text{argmin}_i \{norm_i - Q(\vec{x}) \cdot \vec{\mu}_i^t\}$;
11         $cnt_{i^*} = cnt_{i^*} + 1$;
12         $sum_{i^*} = sum_{i^*} + Q(\vec{x})$;
13     end
14     // Center Update Step
15     $t = t + 1$;
16     foreach $i = 1$ to $K$ do
17         $\vec{\mu}_i^t = \frac{sum_i}{cnt_i}$
18     end
19 end
```

In the first step, the square of the Euclidean distance is expanded. The result, shown in Figure 4(b), contains three parts: (1) the squared L^2 norm of the quantized sample $Q(\vec{x})^2$, (2) an inner product of the serialized sample and the center $Q(\vec{x}) \cdot \vec{\mu}$, and (3) the squared L^2 norm of the center $\vec{\mu}^2$.

In the second step, we transform the expanded Euclidean distance into a Relative distance (Figure 4(c)). The transformation is based on the observation that when comparing the Euclidean distances of one sample to different centers, $Q(\vec{x})^2$ (the first part) stays the same for both sides. Thus, $Q(\vec{x})^2$ can be removed from the comparison. The remaining two terms can be implemented using a reasonable amount of resources on FPGAs. In particular, the inner product $Q(\vec{x}) \cdot \vec{\mu}$ fits well into bit-serial multipliers, while $\vec{\mu}^2$ can be easily pre-computed at the beginning of each epoch using a small amount of resources as the number of clusters is not large.

4.3 BiS-KM Algorithm

Algorithm 1 illustrates the BiS-KM algorithm implementing the ideas just discussed. At the beginning of each iteration, the squared L^2 norm of each center is computed and then right shifted by one (Line 6). Then the dot products of each low-precision sample and all the high-precision centers are computed. The output of each dot product is a high-precision scalar value. As a next step, the distance between a sample and a cluster center is calculated by subtracting the dot product result from the squared L^2 norm. By comparing all the distances, the sample is assigned to the closest cluster (Line

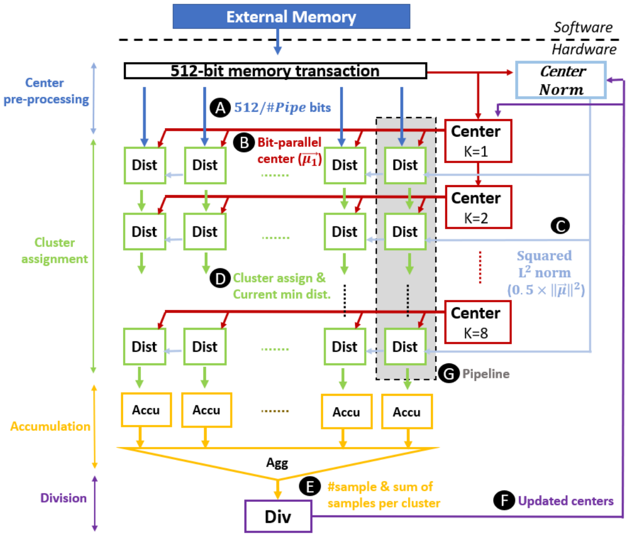


Figure 5: Overall hardware design architecture, where W = 512 and K = 8.

10). For each cluster, we maintain a register to count the number of assignments (Line 11) and an accumulator that computes the coordinate-wise sum of all the assigned samples (Line 12). After all the samples are scanned, we perform the high-precision division of the accumulation value and the counter value to get the new centers (Line 17).

5 BIT-SERIAL K-MEANS SYSTEM

5.1 Overall Hardware Design of BiS-KM

The goal of the design is to consume one W-bit memory transaction per cycle. The design is also intended to a parameterizable number of clusters K and dimensions D at runtime. However, the maximum K and D are compile-time parameters, as they are limited by on-chip logic and memory respectively. The overall hardware design of BiS-KM, which consumes $W = 512$ bits at line rate and can concurrently accommodate at most $K = 8$ clusters, is shown in Figure 5. The BiS-KM hardware design consists of four modules:

(1) Center Pre-processing. At the beginning of the first iteration, this module loads the initial centroids for the cluster, which are stored in external memory in a bit-parallel fashion. Then, each initial cluster center is sent to the corresponding on-chip memory *Center*. The capacity of each *Center* decides the maximum number of dimensions. Meanwhile, these initial cluster centers are also forwarded to the *Center Norm* unit to calculate the squared L^2 norm of each center $||\vec{\mu}||^2 \times 0.5$. The *Center Norm* unit consists of bit-parallel multipliers (implemented with on-chip DSP blocks) and an adder tree to compute squared L^2 norm in parallel.

(2) Cluster Assignment. The goal of this module is to consume the bit-serial sample data in a bit-serial fashion from the external memory at line rate, e.g., consuming a 512-bit memory transaction per cycle. However, the sample data has to wait until all the centers are stored in *Center* and the squared L^2 norm calculation is done in the center pre-processing module. The cluster assignment module consists of #*Pipe Pipelines* Ⓖ. Within a *Pipeline*, eight Distance Processors (labelled as *Dist*) work as a systolic array to concurrently compute the distances of a sample to at most eight centers. Each *Dist* has two inputs. The first input is bit-serial $\frac{512}{\#Pipe}$ bits Ⓐ, each of which belongs to a different feature of $Q(\vec{x})$. The second input is bit-parallel $\frac{512}{\#Pipe}$ features of the center $\vec{\mu}$ Ⓑ. A *Dist* computes the dot product $Q(\vec{x}) \cdot \vec{\mu}$ with a bit-serial arithmetic, as shown in Figure 6.[1] A *Dist* reads the corresponding squared L^2 norm Ⓒ and then calculates the Relative distance by subtracting the squared L^2 norm and the dot product in the *Sub* unit. Afterwards, the Relative distance to the current center is compared with the Relative distance to the previous center in the *Cmp* unit. The smaller distance, together with its cluster assignment, is forwarded to the next *Dist* unit. Though the number of *Dists* in one *Pipeline* is fixed at compile time, the number of active clusters can be parameterized at runtime to a lower number by disabling unused *Dists*. Each *Pipeline* outputs the correct cluster assignment and the minimum distance Ⓓ to the accumulation module.

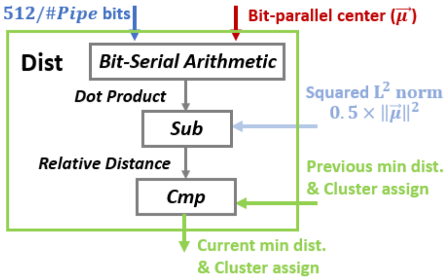

512/#Pipe bits
Bit-parallel center ($\vec{\mu}$)
Dist
Bit-Serial Arithmetic
Dot Product
Sub
Relative Distance
Cmp
Squared L² norm $0.5 \times ||\vec{\mu}||^2$
Previous min dist. & Cluster assign
Current min dist. & Cluster assign

Figure 6: Detailed architecture of Distance Processor (*Dist*)

(3) Accumulation. This module consists of two units: partial accumulation (*Accu*) and global aggregation (*Agg*). The *Accu* unit re-groups the bit-serial sample data into bit-parallel samples, and then computes the coordinate-wise sum of all the assigned samples Ⓔ. At the same time, it calculates the count of the assigned samples. Then, the *Agg* unit performs the global aggregation on the partial sums and counts from all the #*Pipe* pipelines via an adder tree.

(4) Division. The globally aggregated sums and counts are then forwarded to the *Div* unit that performs fixed-point division to generate the new centers. These new centers Ⓕ are routed to the *Center Norm* unit to start a new iteration. After the last iteration, the new centers are sent back to the software.

5.2 Bit-Serial Arithmetic and Bit-Serial Memory Layouts

Understanding of BiS-KM requires understanding the interplay between the bit-serial memory layout and the bit-serial hardware design (particularly bit-serial arithmetic). Essentially, the design choice of BiS-KM involves these two factors interacting in a tight manner to enable any-precision clustering.

Design Methodology for Bit-Serial Arithmetic. To maximize the exploration of the design space, we look into three types of bit-serial arithmetic designs: inter-sample parallelism (ISP), inter-feature parallelism (IFP), and hybrid. The performance requirement for all designs is to achieve line-rate throughput: W=512 bits per cycle, where W denotes the bitwidth of a memory transaction. We

[1] We discuss the design trade-off between bit-serial arithmetic and bit-serial memory layout in Section 5.2.

Table 2: Comparison of three designs, where W=512, K=8, P_h=32. The slot in red indicates the bottleneck of its design.

Hardware Metrics	ISP	Hybrid ($DIPF$ = 16)	IFP
Type of BS arithmetic unit	BiS-MUL	BiS-DP	BiS-DP
Number of BS arithmetic units	4,096	256	8
ALMs Num. of a BS arithmetic	130	379	8,306
ALMs Num. of all the BS arithmetic	532,480	97,024	66,448
Center bitwidth (bits)	32	512	16,384

analyze and then compare three designs, in terms of center bitwidth and resource consumption of bit-serial arithmetic (Table 2). Bit-serial arithmetic reads not only sample data in a bit-serial fashion, but also the corresponding on-chip center in a bit-parallel fashion. The data path between the on-chip center and bit-serial arithmetic can become a bottleneck. In particular, the center bitwidth can be too wide to fit in an FPGA.

Design Methodology for Bit-Serial Memory Layout. For each particular bit-serial arithmetic, the key idea is to design a custom bit-serial memory layout such that the data can flow into the bit-serial arithmetic without transposition overheads. Thus, for each of the three bit-serial arithmetic designs we propose a custom bit-serial memory layout. Such a software/hardware co-design approach significantly reduces FPGA resource consumption and increases energy efficiency. To illustrate the differences among the three bit-serial memory layouts, we use the sample table shown in Figure 7(a) as a running example throughout this subsection, and then transpose the table into any of three memory layouts in Figure 7(b), (c) and (d). This table consists of four samples, each of which has four features (D=4). Each feature has P_h=3 bits (high-precision). We assume the memory transaction size (W) is 4 bits.

5.2.1 *Inter-Sample Parallelism (ISP).*

The key idea of the ISP design is to fully exploit inter-sample parallelism, where each bit of the memory transaction comes from a different sample.

ISP Memory Layout. The ISP memory layout stores the first bits of the first feature of all the four samples (i.e., AaRr) in the first memory entry, as shown in Figure 7(b). The second memory entry consists of the second bits of the first feature (i.e., BbSs). After storing the first feature of the four samples (address from 0 to 2), we begin to store the second feature (address from 3 to 5), and so on. The advantage of the ISP memory layout is to enable any-precision feature retrieval. Take the memory accesses needed to retrieve the sample table at a precision of 2 bits as an example. We access the first bits of the first feature, followed by the second bits. Then, we access the first bits of the second feature, rather than the third bits of the first feature, avoiding unnecessary memory accesses.

ISP Bit-Serial Arithmetic. When using the ISP memory layout, the BiS-KM hardware design has to instantiate 512 BiS-MULs for a cluster to achieve line-rate throughput (512 bits per cycle), as shown in Figure 8(a), where each bit belongs to a different sample and the bit-serial arithmetic is a BiS-MUL (Subsection 2.3.1). In the case of supporting a maximum of eight clusters (K=8), we need 512 × 8 = 4, 096 BiS-MULs. Each of them requires 130 ALMs, resulting a massive amount of 532,480 ALMs in total, as shown in Table 2. In the ISP hardware design, all the 512 bits in a memory transaction come from the same feature of 512 samples, so we only need to read out one value of a cluster center within a cycle. It means that the bitwidth of on-chip centers is P_h=32 bits.

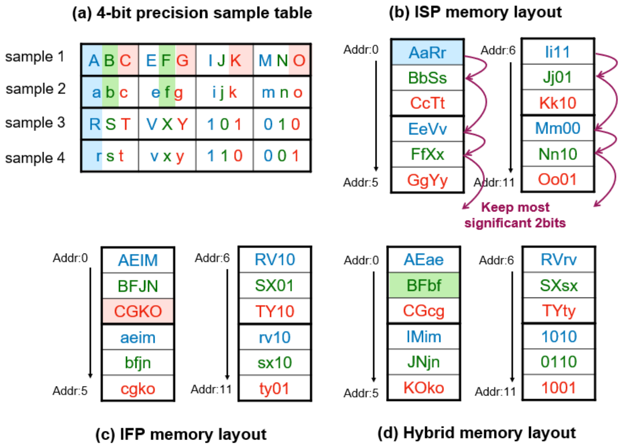


Figure 7: A running example of storing 4 high-precision samples in three memory layouts. Each symbol in the table is binary, representing either 0 or 1.

5.2.2 *Inter-Feature Parallelism (IFP).*

The key idea of the IFP design is to fully exploit inter-feature parallelism, where each bit of the memory transaction comes from a different feature within a sample.

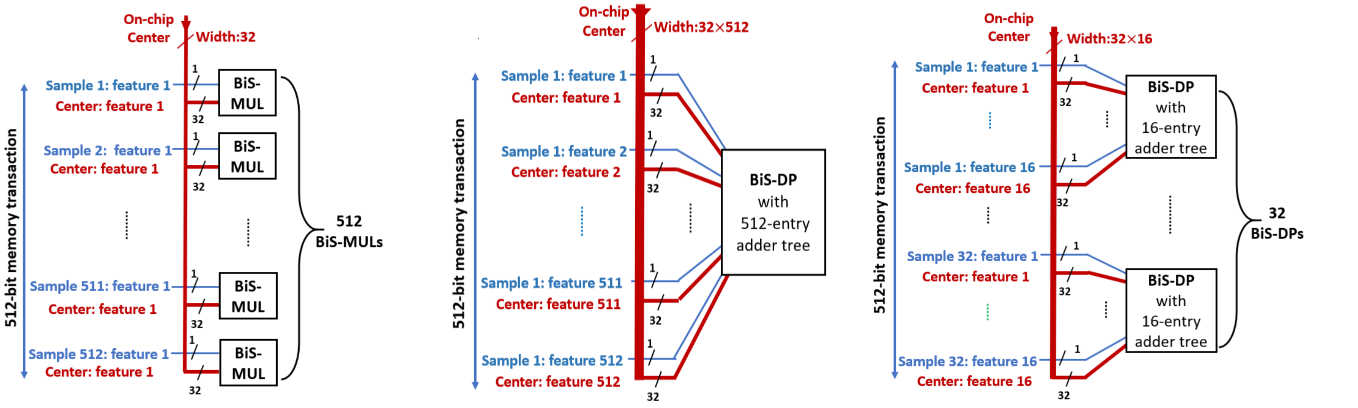
IFP Memory Layout. As shown in Figure 7(c), under the IFP memory layout, we store the first bits (i.e., AEIM) of all the four features of the first sample in the first memory entry, followed by the second bits (i.e., BFJN) of the first sample. After storing the first sample (address: 0 to 2), we begin to store the second sample (address: 3 to 5), and so on.

IFP Bit-Serial Arithmetic. Under the IFP memory layout, all the W=512 bits of a memory transaction come from different features within a sample. We can instantiate a 512-entry BiS-DP unit (Subsection 2.3.2) to process 512 bits in parallel for a center, as shown in Figure 8 (b). For concurrently supporting eight clusters, we only need to instantiate eight 512-entry BiS-DP units, which require 66,448 ALMs on the FPGA, where each BiS-DP unit requires 8,306 ALMs. Compared with the ISP hardware design, the resource consumption of the bit-serial arithmetic part is considerably less. However, one serious issue arises. In particular, the data path between the on-chip cluster center and the BiS-DP unit is extremely wide to support reading 512 center features within a cycle. It reaches $W \times P_h = 512 \times 32$ bits, as shown in Table 2. Such a wide data path makes the design infeasible on an FPGA.[2] Besides, the IFP hardware design imposes a very strict constraint on the number of dimensions of the data set since we have to pad the dimension to a multiple of 512, potentially wasting significant memory bandwidth and computing power if the dimension is not a multiple of 512.

5.2.3 *Hybrid.*

The key idea of the hybrid design is to exploit both inter-sample and inter-feature parallelism, where the degree of inter-sample

[2] The 16k-bit bitwidth needs 410 BRAMs to be implemented, as each BRAM typically provides at most 40-bit bitwidth on modern FPGAs. Accessing 410 BRAMs in lock step cannot be done at a reasonable frequency.



(a) Inter-Sample Parallelism (ISP)

(b) Inter-Feature Parallelism (IFP)

(c) Hybrid (*DIFP*=16)

Figure 8: Comparison of three bit-serial arithmetic designs that aim to achieve the same throughput: 512 bits per cycle.

parallelism (*DISP*) is defined to be the number of concurrent samples associated with a 512-bit memory transaction, and the degree of inter-feature parallelism (*DIFP*) is defined to be the number of concurrent features associated with any sample. *DISP* times *DIFP* is 512. We can observe that the ISP design is one extreme where *DIFP*=1, while the IFP design is the other extreme where *DIFP*=512. Since neither of the extreme designs successfully fits in the FPGA, we present a hybrid design between ISP and IFP.

Hybrid Memory Layout. As shown in Figure 7(d), the hybrid memory layout stores four bits in the first memory entry (address 0), where the first two bits are from the first bits of the first and second features of the first sample (i.e., AE), and the second two bits are from the first bits of the first and second features of the second sample (i.e., ae). Next, we store the second bits of first and second features of the first and second samples (i.e., BFbf) into the memory entry, and so on.

Hybrid Bit-Serial Arithmetic. Under the hybrid memory layout, the BiS-KM hardware design needs to instantiate *DISP* BiS-DP units, and each BiS-DP unit processes *DIFP* bits belonging to different features to achieve line rate: 512 bits per cycle. The benefit of the hybrid design is that by choosing the right *DIFP*, the bitwidth between the on-chip center and a BiS-DP unit drops down to *DIFP*× 32 bits, while keeping its FPGA resource consumption reasonably low. Thus, such a hybrid design allows the BiS-KM design to be successfully mapped to the FPGA. Figure 8(c) instantiates *DISP*=32 BiS-DP units[3] for one cluster. Therefore, the BiS-KM hardware design supporting eight clusters consists of $32 \times 8 = 256$ BiS-DP units and then requires 97,024 ALMs, as shown in Table 2. We observe that its resource consumption is much lower than the ISP hardware, while the datapath bitwidth between the on-chip center and the bit-serial arithmetic becomes 16×32 bits, with the resulting design fitting into the FPGA. In the following sections, when using BiS-KM design, we mean the hybrid design (*DIFP*=16) just described.

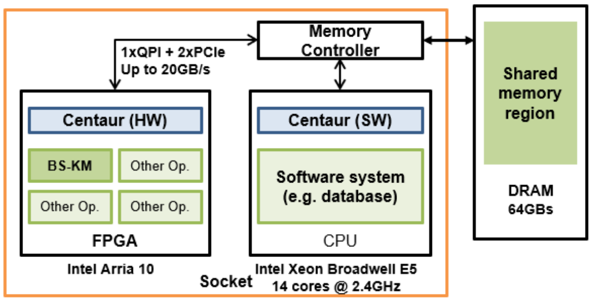


Figure 9: System architecture of the target platform

6 EMPIRICAL EVALUATION

6.1 Experimental Setup

System Architecture. We deploy BiS-KM on the second generation of the Intel Xeon+FPGA platform [28], consisting of an Arria 10 FPGA and a Broadwell 14-core E5 processor on the same socket (Figure 9). The FPGA has cache-coherent access to the CPU's main memory (64GB) through 1 QPI and 2 PCIe links, reaching an aggregated maximum throughput of 17GB/s. We use the open-source framework Centaur [30] for software-hardware integration. Centaur manages the data communication between the FPGA and the CPU. BiS-KM is instantiated within Centaur as a User-Defined-Function.

Hardware Configuration. The hardware implementation in our experiment consists of $DISP = 32$ pipelines, each of which accommodates a sample. Each pipeline is equipped with 8 Distance Processors to support a maximum of 8 clusters and each Distance Processor contains a BiS-DP unit to process 16 bits from 16 features ($DIFP = 16$) per cycle. The maximum number of dimensions supported is 1024. The clocking frequency is 200MHz.

Workloads. We run our experiments with four real-world data sets: OpenStreetMap [18], Forest [2], Gas [39] and Epileptic [10], as shown in Table 3. The data sets cover a wide range of dimensions and are representative for clustering tasks. Because the original data set size of Gas, Epileptic and OpenStreetMap is small, we duplicate the original data (8 times, 8 times and 64 times respectively) in order

[3] The bit-serial arithmetic in this design is the BiS-DP unit with 16-entry adder tree.

Table 3: Evaluated data sets.

Data sets	Features	Samples	Clusters
OpenStreetMap [18]	28	674,944	6
Forest [2]	54	581,012	7
Gas [39]	128	111,280	6
Epileptic [10]	178	92,000	5

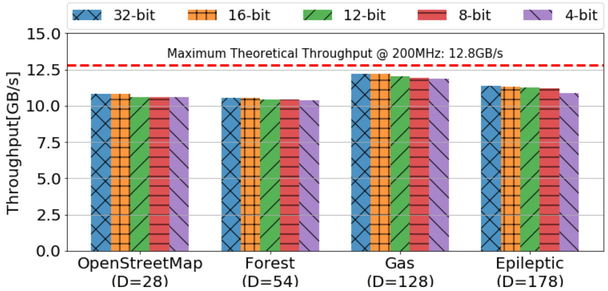


Figure 10: Throughput of different data sets running with various precision levels using BiS-KM

to amortize the communication overhead between the CPU and the FPGA. Since the K-Means algorithm itself is sensitive to the initial centers, we use the same initial centers in all the experiments for each data set.

Hardware Baseline. To evaluate the effectiveness of our BiS-KM design, we choose the state-of-the-art flexible K-Means accelerator (Flex-KM) [13] as our baseline.[4]

6.2 Hardware Efficiency: Throughput

In this subsection, we examine the hardware efficiency of the BiS-KM design in terms of throughput. The throughput is calculated by the data set size divided by the elapsed time required by an iteration. “x-bit" means the BiS-KM design with an x-bit precision level, where x varies from 1 to 32.

Effect of Dimensionality on Throughput. We examine the effect of the number of dimensions on the achievable throughput. Under the BiS-KM memory layout, if the dimension of a data set is not a multiple of *DIFP*, we have to use zero padding to align it to *DIFP*, potentially wasting a certain amount of memory bandwidth due to the padding. Figure 10 shows the throughput of BiS-KM on the four data sets for a varying number of dimensions.

The throughput of BiS-KM varies only slightly with different dimensions. This is because the padding overhead is relatively small over the memory traffic between the FPGA and the host memory.

The throughput of BiS-KM roughly reaches the theoretical memory bandwidth when the dimension of a data set is a multiple of *DIFP*=16. Take the data set Gas (D = 128) as an example, BiS-KM can roughly saturate the FPGA’s memory bandwidth, with its throughput close to the theoretical maximum bandwidth (512 bits at 200MHz is 12.8GB/s). However, there is still a small gap from the theoretical maximum bandwidth due to the fact that the computing pipelines are stalled in the global aggregation and division stages.

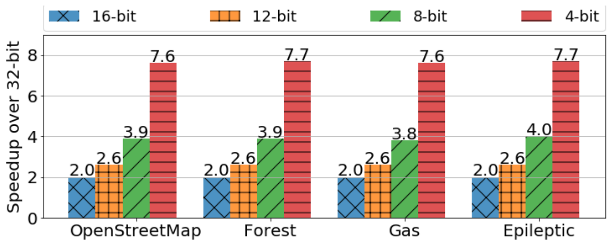
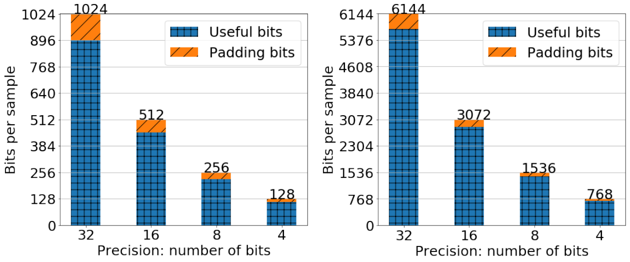


Figure 11: Speedup of runtime per iteration of various low-precision over 32-bit precision computation



(a) OpenStreetMap (28 features)

(b) Epileptic (178 features)

Figure 12: Memory traffic (bits) per sample as the precision varies

Effect of Precision Level. Figure 11 depicts the runtime speedups of different low-precision levels of BiS-KM over the 32-bit precision Flex-KM for four data sets. We make three observations.

First, BiS-KM achieves roughly linear speedup as the precision decreases, due to the linear reduction of memory traffic (Figure 12). Thus, we conclude that the performance of BiS-KM is mainly bounded by the memory bandwidth between the CPU main memory and the FPGA.

Second, the slightly sub-linear speedup observed at 4-bit precision level (Figure 11), is due to strided memory access, particularly when accessing the most significant four bits of every 32 bits. The DRAM’s row buffer hit rate is about 4/32=12.5%, affecting the achievable memory throughput.[5]

Third, the actual throughput roughly stays the same with varying precision levels, as depicted in Figure 10, demonstrating that BiS-KM allows us to take full advantage of low precision.

We conclude that BiS-KM is able to efficiently support any-precision clustering on the FPGA.

6.3 Statistical Efficiency: Loss vs. Iterations

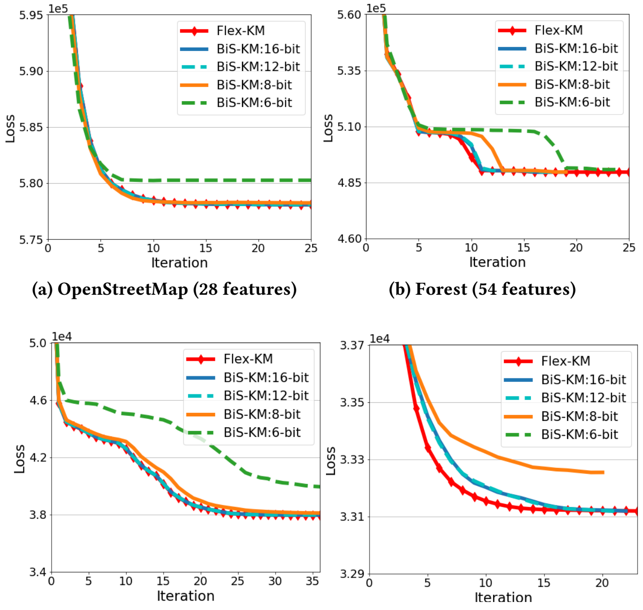
We now examine the statistical efficiency of BiS-KM with different precision levels, in terms of loss (i.e., within-cluster sum of square error) vs. iterations (Figure 13). We use the 32-bit precision Flex-KM as our baseline. We make four observations.

First, low precision levels do converge to the same loss as 32-bit precision. Figure 13 illustrates that a 12-bit precision level is adequate to converge to the same loss as the 32-bit precision does, demonstrating the great advantage of leveraging low precision.

Second, a different data set can require a different minimum precision level to converge. Figure 13 illustrates that the minimum precision level required by the OpenStreetMap, the Forest, the Gas

[4] Actually, we re-implement Flex-KM on our Arria 10 FPGA according to the paper [13]. Our implementation can run at line rate (512 bits per cycle) with the same frequency of 200MHz.

[5] The problem becomes worse at lower precision, e.g., a 2-bit precision, since the row buffer hit rate becomes even lower. Therefore, below 4-bits precision, the gains in hardware efficiency cannot amortize the losses in statistical efficiency.



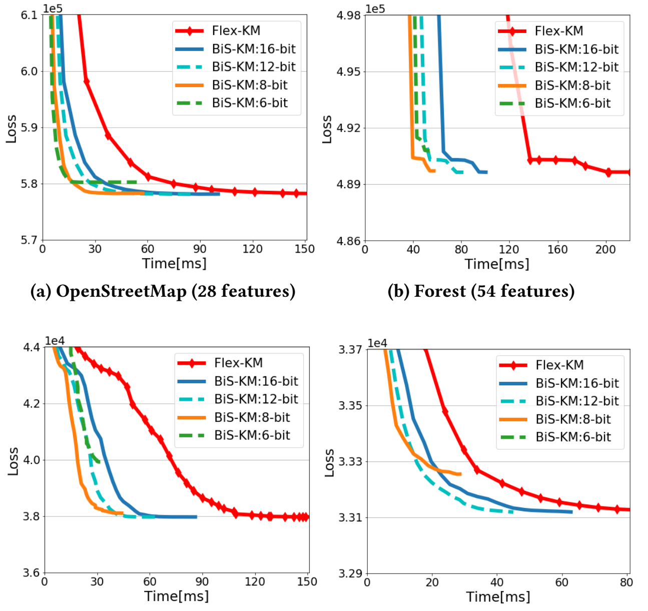
(a) OpenStreetMap (28 features)

(b) Forest (54 features)

(c) Gas (128 features)

(d) Epileptic (178 features)

Figure 13: Convergence comparison: training loss vs. iterations under various precision levels. In (d), the curve of 6-bit precision is out of the range of y-axis.



(a) OpenStreetMap (28 features)

(b) Forest (54 features)

(c) Gas (128 features)

(d) Epileptic (178 features)

Figure 14: End-to-end comparison: training loss vs. runtime under various precision levels. In (d), the curve for 6-bit precision is out of the range of y-axis.

and the Epileptic to converge to the same loss as 32-bit precision are 8 bits, 6 bits, 8 bits and 12 bits, respectively. This observation motivates our BiS-KM design allowing any-precision clustering with only one hardware implementation.

Third, a low precision level is able to successfully enter a smaller local minimum as the 32-bit precision does. Figure 13 (b) illustrates that the BiS-KM design with a low-precision level is capable of following the transfer from a local minimum to a smaller local minimum for the data set Forest, indicating that the statistical efficiency can be preserved when using low precision data.

Fourth, BiS-KM typically requires a similar number of iterations to converge to the same loss compared with the 32-bit precision Flex-KM. Figure 13 shows that BiS-KM requires roughly the same number of iterations to converge as Flex-KM does for the data sets OpenStreetMap, Gas and Epiletptic.

We conclude that the low-precision clustering enabled by BiS-KM can preserve the statistical efficiency.

6.4 End-to-End Comparison: Loss vs. Time

In this subsection, we validate that BiS-KM with the low-precision dataset outperforms the 32-bit precision Flex-KM, in terms of end-to-end convergence rate. Figure 14 shows the convergence trends, loss vs. runtime, with various precision levels for four data sets. We observe that low precision leads to a significantly faster convergence rate. For the data sets OpenStreetMap, Forest and Gas, BiS-KM can achieve about 4X speedup to reach the same loss as the 32-bit precision Flex-KM does. However, BiS-KM can only achieve roughly 2.5X speedup for the data set Epileptic, which requires a 12-bit precision to converge to the same training loss as Flex-KM does.

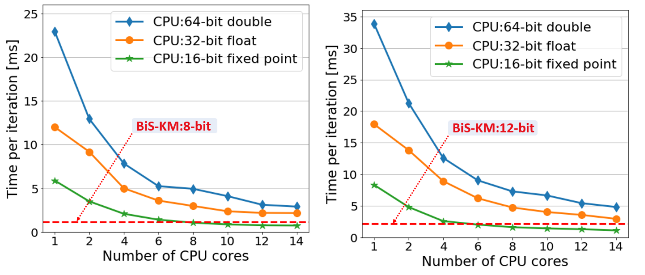
6.5 Comparison with CPU Implementations

CPU Baselines. We choose a highly optimized multi-core AVX2-enhanced CPU implementation as our software baseline [3]. The software baseline is originally implemented with AVX2 64-bit double-precision instructions, labelled as “CPU:64-bit double”. Actually, we try to achieve more data parallelism using two smaller vector types: *vector float* and *vector short*.[6] Accordingly, we produce two more CPU baselines: “CPU:32-bit float” and “CPU:16-bit fixed point”, to improve the performance of K-Means on CPUs.

Comparison Methodology. Since all the K-Means implementations on CPUs have roughly the same statistical efficiency as BiS-KM running at a reasonable precision level, the hardware efficiency comparison is the main metric showing the efficiency of BiS-KM.

Comparison of Hardware Efficiency. Figure 15 illustrates the runtime-per-iteration comparison between the three software implementations and BiS-KM with the lowest precision level that leads to the same loss as 32-bit precision does, for the Gas and the Epileptic data set. The CPU implementation with a smaller vector datatype leads to higher performance, since a smaller vector datatype yields more data-level parallelism using SIMD and induces less memory traffic. BiS-KM is faster than “CPU:64-bit double” and “CPU:32-bit float”, even though the 14-core CPU has 60GB/s memory bandwidth while our FPGA has only roughly 15GB/s. This is because

[6] Multiplication-related AVX2 instruction does not support *vector char* type. Even when the dataset is in 8-bit precision, we cannot achieve more parallelism, since we have to pad to a 16-bit boundary for further computation.



(a) Gas

(b) Epileptic

Figure 15: Runtime comparison between three CPU implementations with increasing number of cores and BiS-KM with the lowest precision level that is able to converge.

BiS-KM takes advantage of low precision, e.g., using 6-bit precision. BiS-KM has roughly the same performance as "CPU:16-bit fixed point" with 6 (or 8) cores, since the K-Means algorithm is able to take full advantage of task-level (e.g., multi-core) and data-level (e.g., 32-way SIMD) parallelism on the CPU. Note, if we implement BiS-KM on a larger FPGA, e.g., VCU118, which has more FPGA resources and higher memory bandwidth, BiS-KM's performance would improve. Nevertheless, the fact that the FPGA can compete with 14 cores demonstrates the feasibility and advantages of the proposed approach even in its current configuration.

6.6 Resource Consumption Breakdown

Table 4 shows the resource consumption breakdown of four modules in the BiS-KM hardware design. ALMs and BRAMs (i.e., "M20Ks" in the Table) are mostly used the cluster assignment and accumulation modules, while the DSP utilization is low since it is mainly used to calculate the squared L^2 norm in the center pre-processing module. Table 4 also shows the resource consumption of the major components, e.g., Accu and Agg. We observe that each component requires a very small amount of FPGA resources. For example, each *Dist* consumes about 0.1% ALMs, allowing us to instantiate a massive amount of *Dists* to process multiple cluster centers concurrently on the FPGA.

Table 4: Resource consumption breakdown of the BiS-KM hardware design with *DIFP*=16 and *#pipe*=32

Resources	ALMs	M20Ks	DSPs
Center Norm	786 (0.18%)	0 (0%)	48 (3.16%)
Dist	452 (0.11%)	0 (0%)	0 (0%)
Accu	1,789 (0.42%)	29 (3.75%)	0 (0%)
Agg	219 (0.05%)	3 (0.70%)	1 (0.07%)
Div	846 (0.20%)	1 (0.03%)	0 (0%)
Center pre-processing	1,357 (0.32%)	26 (0.78%)	49 (3.22%)
Cluster assignment	115,522 (27.10%)	208 (6.21%)	0 (0%)
Accumulation	57,466 (13.45%)	931 (27.79%)	1 (0.07%)
Division	1,674 (0.39%)	14 (0.42%)	0 (0%)
BiS-KM	176,019 (41.26%)	1,179 (35.19%)	50 (3.29%)

7 RELATED WORK

To our knowledge, BiS-KM is the first novel solution that incorporates algorithm, software and hardware designs to enable any-precision K-Means. We contrast closely related work with BiS-KM on 1) FPGA-accelerated K-Means, 2) fast bulk bit-wise operations and 3) low-precision DNN and ML.

FPGA-Accelerated K-Means. There is a wide range of research on accelerating the K-Means with the FPGA for various applications. However, most of the existing approaches focus on high-precision input data [5, 7, 11, 13, 15, 16, 25, 32, 36, 40–42]. Among these, there is very few work that has considered the low-precision K-Means. Estlick et al. [7] run the K-Means algorithm on the CPU over the truncated datasets, whose B least significant bits are truncated, where B is 4, 6, or 8. In contrast, BiS-KM enables any-precision K-Means clustering using a single FPGA design.

Fast Bulk Bit-wise Operations. A broad range of applications, such as database scans [8, 9, 24, 33, 44, 46] and low-precision machine learning and neural networks [37, 38, 43] use fast bulk bit-wise operations to improve their performance. Closest to BiS-KM is the work by Wang et al. [43] that proposes a customized ML-Weaving memory layout to facilitate the hardware design of low-precision generalized linear model training.

Low-Precision DNN and ML. Hardware acceleration of deep neural networks [27, 31, 47, 48, 50] and machine learning algorithms [21, 22, 29] has been a common topic for many years. Recently, researchers focus shifts to use low-precision hardware to further accelerate these workloads because the statistical efficiency of these algorithms can be well preserved in low precision. Plenty of low-precision designs [4, 20, 37, 49] focus on using a fixed quantization of data and a fixed-bitwidth accelerator to accelerate DNN and ML workloads, while other research work [19, 34, 38] focuses on exploiting the bit-level precision variability of hardware arithmetic for interference. In contrast, BiS-KM focuses on any-precision K-Means clustering.

8 CONCLUSION

BiS-KM is an innovative system designed for flexible computation of K-Means over low precision data. The design incorporates a new K-Means algorithm, a novel memory layout tailored to K-Means computation, and an efficient mapping to an FPGA using bit-serial arithmetic. BiS-KM is capable of retrieving any-precision data from a compact memory storage and supports any-precision clustering in a single design. Compared to a state-of-the-art hardware 32-bit precision solution, BiS-KM achieves an almost linear speedup with lower precision and its performance favourably compares to that of K-Means running on multi-core CPUs.

ACKNOWLEDGMENT

We would like to thank Intel for their generous donation of the Xeon+FPGA prototypes. Some experiments in the paper were obtained through the Intel Hardware Accelerator Research Program (HARP2) at the Paderborn Center for Parallel Computing (PC^2).

REFERENCES

[1] Jorge Albericio, Alberto Delmás, et al. 2017. Bit-pragmatic Deep Neural Network Computing. In *MICRO*.

[2] Jock A. Blackard and D. J. Dean. 1999. Comparative Accuracies of Artificial Neural Networks and Discriminant Analysis in Predicting Forest Cover Types From Cartographic Variables. In *Computers and Electronics in Agriculture*.

[3] Christian Böhm, Martin Perdacher, et al. 2017. Multi-core K-means. In *SIAM*.

[4] Ruizhe Cai, Ao Ren, et al. 2018. VIBNN: Hardware Acceleration of Bayesian Neural Networks. In *ASPLOS*.

[5] Y. M. Choi and H. K. H. So. 2014. Map-reduce Processing of K-Means Algorithm with FPGA-accelerated Computer Cluster. In *ASAP*.

[6] Alberto Delmas, Sayeh Sharify, et al. 2017. Tartan: Accelerating Fully-Connected and Convolutional Layers in Deep Learning Networks by Exploiting Numerical Precision Variability. *CoRR* (2017).

[7] Mike Estlick, Miriam Leeser, et al. 2001. Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware. In *FPGA*.

[8] Z. Feng and E. Lo. 2015. Accelerating Aggregation Using Intra-cycle Parallelism. In *ICDE*.

[9] Ziqiang Feng, Eric Lo, et al. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *SIGMOD*.

[10] Ralph G. Andrzejak, Klaus Lehnertz, et al. 2002. Indications of Nonlinear Deterministic and Finite-dimensional Structures in Time Series of Brain Electrical Activity: Dependence on Recording Region and Brain State. *Physical review. E, Statistical, nonlinear, and soft matter physics* (2002).

[11] Maya Gokhale, Jan Frigo, et al. 2003. Experience with a Hybrid Processor: K-Means Clustering. *The Journal of Supercomputing* (2003).

[12] Suyog Gupta, Ankur Agrawal, et al. 2015. Deep Learning with Limited Numerical Precision. In *ICML*.

[13] Zhenhao He, David Sidler, Zsolt Istvan, et al. 2018. A Flexible K-Means Operator for Hybrid Databases. In *FPL*.

[14] Itay Hubara, Matthieu Courbariaux, et al. 2017. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *J. Mach. Learn. Res.* (2017).

[15] H. M. Hussain, K. Benkrid, et al. 2011. FPGA Implementation of K-Means Algorithm for Bioinformatics Application: An Accelerated Approach to Clustering Microarray Data. In *NASA/ESA Conference on Adaptive Hardware and Systems*.

[16] Hanaa M. Hussain, Khaled Benkrid, et al. 2012. Novel Dynamic Partial Reconfiguration Implementation of K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs. *Int. J. Reconfig. Comput.* (2012).

[17] Intel. 2018. *Intel Arria 10 Device Overview*.

[18] B. A. Johnson and K. Iizuka. 2016. Integrating OpenStreetMap Crowdsourced Data and Landsat Time-series Imagery for Rapid Land Use/Land Cover (LULC) Mapping: Case Study of the Laguna de Bay Area of the Philippines. *Applied Geography* (2016).

[19] P. Judd, J. Albericio, et al. 2016. Stripes: Bit-serial Deep Neural Network Computing. In *MICRO*.

[20] K. Kara, D. Alistarh, G. Alonso, et al. 2017. FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off. In *FCCM*.

[21] Kaan Kara, Ken Eguro, et al. 2018. ColumnML: Column-Store Machine Learning with On-The-Fly Data Transformation. In *VLDB*.

[22] Kaan Kara, Zeke Wang, et al. 2019. DoppioDB 2.0: Hardware Techniques for Improved Integration of Machine Learning into Databases. In *VLDB*.

[23] Bernd Lesser, Manfred Mücke, and Wilfried N. Gansterer. 2011. Effects of Reduced Precision on Floating-Point SVM Classification Accuracy. In *ICCS*.

[24] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*.

[25] Z. Lin, C. Lo, and P. Chow. 2012. K-means Implementation on FPGA for High-dimensional Data Using Triangle Inequality. In *FPL*.

[26] S. Lloyd. 1982. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory* (1982).

[27] L. Lu, Y. Liang, et al. 2017. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. In *FCCM*.

[28] N. Oliver, R.R. Sharma, S. Chang, et al. 2011. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *ReConFig*.

[29] Muhsen Owaida and Gustavo Alonso. 2018. Application Partitioning on FPGA Clusters: Inference over Decision Tree Ensembles. In *FPL*.

[30] Muhsen Owaida, David Sidler, et al. 2017. Centaur: A Framework for Hybrid CPU-FPGA Databases. In *FCCM*.

[31] Jiantao Qiu, Jie Wang, et al. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *FPGA*.

[32] Takashi Saegusa and Tsutomu Maruyama. 2006. An FPGA Implementation of K-means Clustering for Color Images Based on Kd-tree. In *FPL*.

[33] Vivek Seshadri, Donghyuk Lee, et al. 2017. Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*.

[34] H. Sharma, J. Park, N. Suda, et al. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks. In *ISCA*.

[35] A. Sinha and A. P. Chandrakasan. 1999. Energy Efficient Filtering Using Adaptive Precision and Variable Voltage. In *IEEE International ASIC/SOC Conference*.

[36] Qing Y Tang and Mohammed AS Khalid. 2016. Acceleration of K-means Algorithm Using Altera SDK for OpenCL. *ACM TRETS* (2016).

[37] Yaman Umuroglu, Nicholas J. Fraser, et al. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *FPGA*.

[38] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. 2018. BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing. *CoRR* (2018).

[39] Alexander Vergara, Shankar Vembu, et al. 2012. Chemical Gas Sensor Drift Compensation Using Classifier Ensembles. *Sensors and Actuators B: Chemical* (2012).

[40] Xiaojun Wang and Miriam Leeser. 2007. K-Means Clustering for Multispectral Images Using Floating-point Divide. In *FCCM*.

[41] Zeke Wang et al. 2016. Melia: A MapReduce Framework on OpenCL-based FPGAs. *IEEE TPDS* (2016).

[42] Z. Wang, B. He, W. Zhang, and S. Jiang. 2016. A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs. In *HPCA*.

[43] Zeke Wang, Kaan Kara, Hantian Zhang, et al. 2019. Accelerating Generalized Linear Models with MLWeaving: A One-size-fits-all System for Any-precision Learning. *VLDB* (2019).

[44] Z. Wang, K. Zhang, H. Zhou, X. Liu, and B. He. 2018. Hebe: An Order-Oblivious and High-Performance Execution Scheme for Conjunctive Predicates. In *ICDE*.

[45] S. A. White. 1989. Applications of Distributed Arithmetic to Digital Signal Processing: a Tutorial Review. *IEEE ASSP Magazine* (1989).

[46] Thomas Willhalm, Ismail Oukid, Ingo MÃijller, et al. 2013. Vectorizing Database Column Scans with Complex Predicates. In *VLDB*.

[47] Chen Zhang, Peng Li, et al. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*.

[48] Chen Zhang, Di Wu, et al. 2016. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *ISLPED*.

[49] Hantian Zhang, Jerry Li, and Kaan Kara others. 2017. ZipML: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning. In *ICML*.

[50] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *FPGA*.