Received July 28, 2020, accepted August 13, 2020, date of publication August 18, 2020, date of current version August 28, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.3017552*

# Optimizing OpenCL Code for Performance on FPGA: k-Means Case Study With Integer Data Sets

**NUNO PAULINO[1,2], JOÃO CANAS FERREIRA[1,2], (Senior Member, IEEE), AND JOÃO M. P. CARDOSO[2], (Senior Member, IEEE)**
[1]INESC TEC, University of Porto, 4099-002 Porto, Portugal
[2]Faculty of Engineering, University of Porto, 4200-465 Porto, Portugal

Corresponding author: Nuno Paulino (nuno.m.paulino@inesctec.pt)

This work was supported in part by the Power Efficiency and Performance for Embedded and HPC Systems with Custom CGRAs (PEPCC) Project, financed by the Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) under Grant PTDC/EEI-HAC/30848/2017.

**ABSTRACT** High Level Synthesis (HLS) tools targeting Field Programmable Gate Arrays (FPGAs) aim to provide a method for programming these devices via high-level abstractions. Initially, HLS support for FPGAs focused on compiling C/C++ to hardware circuits. This raised the issue of determining the programming practices which resulted in the best performing circuits. Recently, to further increase the applicability of HLS approaches, renewed effort was placed on support for HLS of OpenCL code for FPGA, raising the same issues of coding practices and performance portability. This paper explores the performance of OpenCL code compiled for FPGAs for different coding techniques. We evaluate the use of task-kernels versus *NDRange* kernels, data vectorization, the use of on-chip local memories, and data transfer optimizations by exploiting burst access inference. We present this exploration via a case study of the k-means algorithm, and produce a total of 10 OpenCL implementations of the kernel. To determine the effects of different data set characteristics, and to determine the gains from specialization based on number of attributes, we generated a total of 12 integer data sets. The data sets vary regarding the number of instances, number of attributes (i.e., features), and number of clusters. We also vary the number of processing cores, and present the resulting required resources and operating frequencies. Finally, we execute the same OpenCL code on a 4 GHz Intel i7-6700K CPU, showing that the FPGA achieves speedups up to 1.54× for four cases, and energy savings up to 80 % in all cases.

**INDEX TERMS** OpenCL, k-means, clustering, FPGA, hardware accelerator, HLS.

## I. INTRODUCTION

Unlike devices such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs), the reconfigurability of Field Programmable Gate Array (FPGA) allows for very finely-tuned and application-specific implementations of circuits. For very demanding and yet very rigidly specified applications, specially those with energy consumption constraints, FPGAs are a popular choice [1]–[3]. However, the traditional programming flow for these devices requires describing a circuit's behaviour via Hardware Description Languages (HDLs), followed by lengthy and laborious simulation, verification, and integration. Also, despite the beneficial circuit specialization, the respective lack of programmability implies the same design effort for future revisions.

The associate editor coordinating the review of this manuscript and approving it for publication was Remigiusz Wisniewski.

In order make these devices more suited for general use, over a decade of development has focused on efficient generation of circuits via High Level Synthesis (HLS) of source code such as (subsets of) *C/C++* or *MATLAB* [4], [5]. The study of HLS techniques for FPGAs using these languages comprises a long body of work [6]–[9], focusing on optimizing the use of FPGAs as new computing platforms by studying appropriate coding techniques.

More recently, progress has been made towards extending this approach to other languages such as OpenCL. This makes FPGAs similar to GPUs, in that they can be deployed in the same fashion, and targeted by the same programming language. FPGA-based PCIe boards become appealing for servers workloads, since there is a less pronounced interference with the infrastructure, especially because development (and revision) of code continues to reside within a familiar ecosystem. Notable work in OpenCL compilation for FPGAs includes both academic research [10]–[12] and commercial tools that have become widely known [13], [14]. Just as with HLS of C/C++, research has emerged that studies the set of best practices when writing OpenCL code for FPGAs, how they differ from code targeting CPUs or GPUs, and if portability is or is not compromised.

In this paper, we evaluate the performance of OpenCL code on FPGA resulting from applying multiple coding techniques, including the use of single-task kernels versus *NDRange* kernels, combined with data vectorization and the use of local memories and burst accesses to local memory. We study these aspects via the popular k-means algorithm [15]. Starting with a sequential OpenCL implementation of the algorithm, we perform incremental modifications, and evaluate the performance and power consumption of all resulting code versions. To allow us to evaluate the effects of the number of instances (i.e., data points), number of features, and number of clusters, we randomly generated 12 data sets. The random generation produced instances with a given number of integer attributes, distributed according to the specified number of centroids, and ensured random correlation between attributes.

We focus on data sets with integer attributes since they are the best candidates for FPGA execution relying heavily on parallelism, due to the typically limited number of on-chip resources for floating-point calculations. Additionally, a considerable amount of domains rely on integer data sets for clustering and classification [16]. Examples include medical applications [17], [18], asset tracking [19], [20], and text processing [21], [22]. Given this, evaluating the use of FPGAs for this scenario is a relevant effort.

The contributions of this paper are the following:

- Demonstrating the changes required to a baseline sequential OpenCL implementation of a clustering algorithm to improve performance for FPGA;
- Presenting 10 OpenCL implementations for the case study of the k-means algorithm;
- Evaluation with 12 generated integer data sets, regarding execution time and resource requirements;
- A comparison between CPU and FPGA regarding execution time and energy consumption;
- Discussion about the effects of algorithm characteristics on code restructuring, and about the general applicability of the techniques employed.

This paper is organized as follows. Section II presents related work on acceleration via HLS of OpenCL to FPGAs, focusing on k-means or similar algorithms. Section III summarizes the k-means algorithm. Section IV shows the implemented code versions, explains the differences between them and the coding techniques used. Section V presents experimental setup and results, including a comparison between the several versions executing on FPGA, a comparison between CPU and FPGA execution, and a thorough discussion on the influence of the characteristics of the algorithm and of the OpenCL paradigm on the code techniques applied. Finally, Section VIII concludes the paper.

## II. RELATED WORK

Classification algorithms such as k-means, *k-nearest neighbours* (kNN), and mean-shift have been implemented frequently on FPGAs [23]. Although GPUs already enjoyed widespread use in data centers, to allow parallel processing of the large amounts of data required by these tasks, it was the relative energy efficiency of FPGAs that increased their appeal, along with the capability of fine tuning the hardware design. However, prior to the emergence of HLS, designs required hardware specification of the core via HDL, and integration with the host system. Synthesis of OpenCL to FPGA emerged to further address this.

One issue with the use of FPGAs as OpenCL devices is performance portability. Code optimized for CPUs or GPUs performs poorly when compiled for FPGAs, due to the different computing paradigms. As such, considerable refactoring is required to truly exploit the potential of FPGA computing. Existing work has thus focused on exploring the best methods for segmenting algorithms into inter-connected kernels, and on optimization of kernel code [24]–[26]. This section briefly summarizes some implementations of the k-means and similar algorithms on FPGA, resorting to either manual hardware design or HLS.

Hussain *et al.* [27] present an HDL based design of the k-means algorithm on a Virtex-4 based board. All the processing steps of the algorithm are offloaded to the FPGA. The design synthesis parameters include the bit width of the data, the number of attributes, the number of clusters, and the number of data points. The implementation resorts to fixed-point arithmetic. Data points reside in external memory, and are fetched by a streaming interface. Parallel distance calculations to each centroid are then performed, resorting to the Manhathan distance. The smallest distance is then added to an accumulator respective to the cluster, and finally processed by a pipelined divider. Comparisons are performed to an Intel Core 2 Duo CPU and an NVIDIA GeForce 9600M GT GPU. For two data sets of up to six million points with a single integer attribute, the FPGA implementation is up to 6.7× faster than the GPU.

Pu *et al.* present an OpenCL based implementation of the kNN algorithm [26]. The approach employs two kernels exchanging data through an on-chip local memory. The first kernel computes one point distance to one centroid per work-item. The second kernel computes a partial bubble to determine the nearest neighbours of each data point. The number of work-groups equals the number of data points, and each group contains $k$ work-items. The design was implemented on a Stratix IV based FPGA board. For comparison, an Intel i7-3770k CPU and an AMD Radeon HD7950 GPU were used. An integer data set with approximately 20 thousand data points and 64 attributes was used. For $k = 20$, the FPGA implementation achieves a speedup of 148× over a MATLAB implementation of kNN executing on the CPU. This is 2.7× slower but 3× more energy efficient than the GPU executing the same OpenCL.

Tang *et al.* provide this exploration for the k-means algorithm [28] via HLS of OpenCL code. A first implementation of the sequential baseline k-means algorithm leads to poor performance on the FPGA, due to poor use of memory bandwidth. The best performing solution relied on two kernels making use of local memories within the FPGA. The first is a multiple work-group kernel (i.e., *NDRange* kernel) for computation of distances, and the second a single-thread kernel (i.e., task kernel) for calculation of new centroids. Additional code variations to account for number of attributes were also studied, as well as fixed-point data types and the use of the Manhattan distance. For implementations on an Altera Stratix V FPGA, speedups of up to 21× are achieved in the best case versus an Intel Xeon hexa-core processor, and performance is comparable to an NVIDIA GTX280 GPU.

Muslim *et al.* study the kNN algorithm [29]. The algorithm calculates which $k$ nearest points in a training set are closest to each point $n$ of the input data set. Two implementations were tested on an AlphaData PCIe board with a Xilinx Virtex-7 family device. Firstly, a version where the distance calculation of each point is implemented as an *NDrange* kernel, and the remaining portions of the algorithm execute on the host machine. This implementation relied only on global data memory. The second version employs two kernels using local memories and inter-kernel communication channels. Performance is compared with an NVIDIA GTX960. The FPGA performs better for the second implementation, while the GPU performs better for the first implementation. Comparing the best cases, the FPGA achieves a speedup of 2.5×, and consumes between 8× and 9× less energy.

Canilho *et al.* [30] present a many-core design for k-means, implemented via HDL on a Xilinx Zynq-7000. The design relies on the Manhattan distance, and uses floating-point arithmetic for calculations. The configurable number of cores affects only the replication of the distance calculation module, which computes the distance of a single data point to a subset of the centroids. The closest centroid of each subset is then fed to a reduction module which performs the final selection via consecutive comparisons in a tree structure. The accumulation step uses distributed Block RAMs (BRAMs), storing each attribute of the accumulator in a separate memory, allowing for a parallel computation of the final division and centroid update, which is performed on the host ARM processor. For a data set of 1 million points with 32 attributes and 100 clusters, the hardware/software solution is up to 20× faster than software-only execution.

Raghavan *et al.* present an integrated approach implemented on a Virtex-6 FPGA [31]. A custom k-means module is attached to a host MicroBlaze central processor, and data are fetched via Direct Memory Access (DMA) from external DDR memory. The design parameters determine the replication of distance calculation sub-modules, meaning the k-means core processes multiple data points in parallel. The Squared Euclidean Distance is used as the distance metric. These sub-modules compute the distance of multiple points to a single cluster centroid only, meaning that distance calculations of each batch of points to each centroid is performed sequentially. For one data set with integer attributes from [16], the augmented solution resorting to 32 cores achieves speedups up to 368× against the MicroBlaze processor alone.

Shata *et al.* present a lengthy study of several generic code optimizations for OpenCL kernels [32]. These include avoiding the use of global variables (e.g., for reduce operations), studying the effect of suggested or enforced work-group size for different kernels, and employing inter-kernel channels. Regarding the use of channels between a producer-consumer pair, three cases were considered. Firstly, a *NDRange* producer kernel writing an arbitrary number of items in arbitrary order to one *NDRange* consumer via $N$ channels; secondly, use of only a single channel for a single consumer kernel; and finally, relying on an *NDRange* producer and $N$ task-kernel consumers with individual dedicated channels. These strategies were individually applied to several OpenCL use-cases to evaluate their effects. The k-means baseline is an OpenCL implementation in the form of two *NDRange* kernels, one for assignment, and the other for centroid recalculation. From the transformations studied, the use of channels led to the best speedup, which was 2.1× and 1.8× for a data dimensionality of 2 and 4, respectively. The speedup was independent of the number of clusters or points. A combination of optimizations led to a speedup of 3.2×. For the remaining benchmarks, speedups when relying on task kernels (i.e., single work-item work-groups) can range up to 140× when combined with manual loop unrolling.

An OpenCL implementation of a real world application for genomic workload is presented in [33]. The algorithm, *k-mers*, entails parallel comparisons of large amounts of genome data samples. A solution was achieved by segmenting the workload into producer and consumer kernels, with channel-based communication. The performance scalability was analysed by increasing the number of both types of kernels or the number of FPGAs. Due to the random memory access requirements of one of the kernels, a dual FPGA solution is required to outperform the baseline. This approach provides a performance improvement of 1.32× relative to execution on a dual 12-core Xeon CPU setup, and outperforms energy consumption of an NVIDIA GTX1080 Ti by 1.5×. The performance impact of bandwidth limitations of FPGA communication through PCIe channels is also characterized.

Song *et al.* also present an HLS implementation of the kNN algorithm on a PCIe board with a Xilinx UltraScale FPGA [34]. Focus is given on reducing data transfers between the FPGA and main memory, to minimize overhead. To achieve this, a combination of principal component analysis and data precision reduction is employed. A large data set consisting of one million 960-dimensional points is used to evaluate the implementation versus a dual Intel Xeon E5-2699 (2.2 GHz) server setup capable of executing up to 88 concurrent threads. As the number of bits used to represent data decreases, the performance of the FPGA implementation over the CPU setup noticeably increases. For a 4 bit data width, performance is comparable to 56 threads on the CPU, while operating at 260 MHz. Additionally, the amount of data transit is reduced by 28×, relative to conventional kNN.

Dias *et al.* also present an implementation for k-means through an HDL based custom design [35]. Design parameters include the bit-width of the fixed-point arithmetic employed, and the number of cores, i.e., replication of the pipelines for distance calculation and cluster assignment for a single data point. Hardware is also replicated as needed according to the number of attributes of the data. Regarding the distance calculation, multiple distance metrics can be chosen, including the Euclidean distance, for which a custom module was implemented. Experimental evaluation was carried out on a Virtex-6 FPGA. Synthetic data sets of normally distributed clusters were used to evaluate the effect of the chosen distance metric (Manhattan, Euclidean, or Squared Euclidean) on the resulting hardware requirements. Additional machine learning data sets [16], [36] were used to perform the same analysis for the number of attributes, the number of clusters, the parallelization level, and the data bit width. For four cores, up to 31 million data points can be processed per second for a data set where points possess 8 attributes.

## III. THE k-MEANS ALGORITHM

The k-means algorithm [15] is a process of data quantization that is widely used for data clustering and classification [37], [38], and is summarized in Algorithm 1.

Given a set of input data $N$, all of equal dimension $D$, the purpose of k-means clustering is to find a set of $K$ centroids $C_k = (c_1, c_2, \ldots, c_n)$, such that the sum of the squares of the distances of all points to their closest cluster is minimized. In every iteration, each data point is assigned to the cluster of the corresponding closest centroid, and new centroids are computed based on the assignments. The algorithm may have two termination conditions. It may end after a given number of iterations, or only when the position of the centroids in sequential iterations varies less than a given threshold. Implementations in this paper employ the later condition.

The $K$ parameter is given to the algorithm, and its value depends on some type of prior knowledge about the nature of the data to classify. Auxiliary algorithms, like *k-means++* [39] are used to determine initial centroids for k-means that reduce the chance of convergence to local minima.

Being heavily used in applications such as computer vision, machine learning, and market analytics, k-means is used to process very large amounts of data, often with many features. Parallelizing the algorithm is possible since the

**Algorithm 1** k-means Clustering

```
Data: Set of $N = X_1, X_2, \ldots, X_N$ input data, where
      $X_n = x_1, x_2, \ldots, x_d$, $threshold, K, D, N$
Result: Set of $K$ cluster centroids $C = C_1, C_2, \ldots, C_k$
        and assignments of each datum $X_n$ to a cluster $k$
while $error > threshold$ do
    set $old\_error = error$;
    set $error = 0$;
    forall the $X_n$ in $X$ do
        set $mindist = 0$;
        forall the $C_k$ in $C$ do
            Compute distance $dist$ of $X_n$ to $C_k$;
            if $dist < mindist$ then
                $mindist = dist$;
                assign $X_n$ to cluster $k$;
    $error = error + mindist$;
    forall the $C_k$ in $C$ do
        Compute new $C_k$ from points assigned to cluster
        $k$
```

computations of each datapoint's distance to each cluster are independent operations. The scalability of parallel computing solutions for k-means, and similar algorithms, is limited by aspects like the computing architecture (e.g., number of cores and operating frequency), and data transfer bandwidth.

This paper presents implementations that exploit parallelization, in the context of HLS of OpenCL code for FPGA targets. Although FPGAs operate at lower frequencies than other devices, their capability for implementing application-specific circuits leads to reductions in energy consumption, despite any possible increases in execution time. This makes FPGAs appealing for large-scale data centers where energy savings are paramount for cost-cutting reasons.

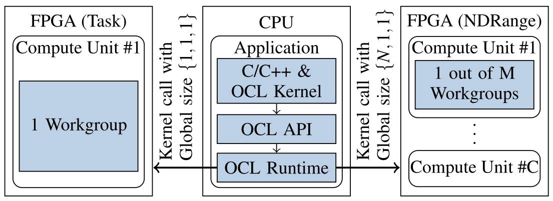


**FIGURE 1.** OpenCL Task-Kernel vs *NDRange* Kernel execution; for *NDRange*, workgroups have local size $\{1 < n < N, 1, 1\}$, where $N = total\ \#\ workitems$.

## IV. IMPLEMENTED CODE VERSIONS

We implemented versions of the k-means algorithm in OpenCL to evaluate the effects of different code styles when targeting FPGA devices via HLS. Figure 1 summarizes the execution models adopted on the FPGA via the overarching OpenCL model. The application is comprised of host-side code (C/C++) executing on the CPU, and OpenCL kernels which are invoked via the OpenCL API. Typically, the

OpenCL runtime compiles the kernel for the target device on-the-fly (e.g. OpenCL execution on the CPU). For FPGA execution, the runtime is responsible for reconfiguring the FPGA with the respective kernel circuit. For task-kernels, a single work-group with one work-item executes on one Compute Unit (CU). For *NDRange* kernels, multiple work-groups are scheduled onto the available CUs at runtime.

Regarding the optimization techniques used, some are recommended by the vendors of HLS tools for FPGAs [40]. These include optimizations to the host code and to the OpenCL code. Included in the later are the optimization of the data transfers with the FPGA, and the refactoring of code to express computation parallelism (e.g., loop pipelining and unrolling) and to exploit the use of multiple cores. We perform no optimization of the host code, but employ the listed techniques to optimize the FPGA-side OpenCL code.

**TABLE 1. k-means kernel versions.**

| Kernel | Description |
|---|---|
| v1 | Task-kernel; Baseline code |
| v2/v3 | Task-kernel; v1 + specialization for $D = 8$ or $D = 16$ |
| v4 | *NDRange*; Computation of new centroids by host |
| v5 | *NDRange*; v4 + specialization for $D = 8$ |
| v6 | *NDRange*; Only one point computed per work-group |
| v1b | v1 + burst access optimization |
| v5b2 | v5 + burst access optimization, specialized for $D = 2$ |
| v5b8 | v5 + burst access optimization, specialized for $D = 8$ |
| v5b16 | v5 + burst access optimization, specialized for $D = 16$ |

```
__kernel void s1kmeans1(global uint *data, int n, int m,
int k, int t, global uint *centr, global int *labels,
global uint *c1, global int *counts, global int *itcount)
{
  ulong old_error, error = INT_MAX;
  uint i = 0, j = 0; itcount[0] = 0;

  do {                                                      A
    old_error = error, error = 0; // save error
    for (i = 0; i < k; i++) {
      counts[i] = 0;  // clear tmp counts
      for (j = 0; j < m; j++) c1[i*m+j] = 0;
    }

    for (int h = 0; h < n; h++) {                         B

      uint mindist = INT_MAX;                           C
      for (i = 0; i < k; i++) {

        ulong dist = 0, diff = 0;                     D
        for (j = 0; j < m; j++) {
          diff = data[h*m+j] - centr[i*m+j];
          dist += diff*diff;
        }

        if((int)(dist/2) < (int)(mindist/2)) {
          labels[h] = i;
          mindist = dist;
        }
      }

      counts[labels[h]]++;
      for (j = 0; j < m; j++) // new aux sum
        c1[labels[h]*m+j] += data[h*m+j];

      error += mindist; // update error
    }

    itcount[0]++;
    for (i = 0; i < k; i++) // new centroids
      for (j = 0; (j < m) && (counts[i] > 0); j++)
         centr[i*m+j] = c1[i*m+j] / counts[i];

  } while(abs((error - old_error)) > t);
}
```

**LISTING 1. Version v1 (baseline).**

```
__kernel void s1kmeans2(
global uint8 *data, int n, int m, int k, float t,
global uint8 *centr, global int *labels, global uint *c1,
global int *counts, global int *itcount)
{
  ulong old_error, error = INT_MAX;
  uint i = 0, j = 0; itcount[0] = 0;

  do {                                                       A
    old_error = error, error = 0;
    for(i = 0; i < k; i++) {
      counts[i] = 0; c1[i] = (uint8)(0);
    }

    for(h = 0; h < n; h++) {                               B

      uint mindist = INT_MAX;                            C
      for(i = 0; i < k; i++) {

        uint8 d = data[h] - centr[i];                  D
        uint dist =
         d.s0 * d.s0 + d.s1 * d.s1 +
         d.s2 * d.s2 + d.s3 * d.s3 + d.s4 * d.s4 +
         d.s5 * d.s5 + d.s6 * d.s6 + d.s7 * d.s7;

        (...) // compare dist with mindist
      }

      // update tmp centroids
      c1[labels[h]] += data[h];
      counts[labels[h]]++;
      (...) // update error
    }

    itcount[0]++;
    (...) // update all centroids
  } while(abs((error - old_error)) > t);

}
```

**LISTING 2. Version v2/3 (v1 refactor for $D = \{8, 16\}$).**

```
__kernel void s1kmeans4(
global uint *data, int n, int m, int k, float t,
global uint *centr, global int *labels,
global uint *mindist)
{
  size_t gsz0 = get_global_size(0U);
  size_t gid0 = get_group_id(0U);
  int offset = gid0 * (n/gsz0);

  int h;                                                     B
  for (h = offset; h < offset + (n/gsz0); h++) {

    mindist[h] = INT_MAX;                                  C
    for (int i = 0; i < k; i++) {

      uint dist = 0;                                     D
      for (int j = 0; j < m; j++) {
        uint diff = data[h * m + j]
                  - centr[i * m + j];
        dist += diff*diff;
      }

      if ((int)(dist/2) < (int)(mindist[h]/2)) {
        labels[h] = i;
        mindist[h] = dist;
      }
    }

  }
}
```

**LISTING 3. Version v4 (multiple work groups).**

Table 1 summarizes the main aspects of each version, while Listings 1 to 5 present the most relevant excerpts of the code versions. Version *v1* serves as a baseline, and is similar to a plain *C* implementation of the algorithm, implementing only the keywords required to express it as an OpenCL kernel, e.g., *__kernel* and *global*. Versions *v1* to *v3*, are task-kernels, while versions *v4* to *v6* are *NDRange* kernels. The remaining cases are modifications which aim to maximize bandwidth utilization by directing the HLS tools to exploit burst accesses to the global off-chip data memory.

## A. TASK-KERNEL (SINGLE-THREAD) VERSIONS

Listing 1 shows the code for the baseline kernel. Throughout this paper, we refer to the highlighted code segments, *A* to *D* where required, and omit actual code in favor of these identifiers in code shown later on, for clarity and to highlight only differences between versions.

```
__kernel void s1kmeans1b(
global uint16 *data, int n, int m, int k, int t,
global uint16 *centroids, global uint16 *labels,
global uint16 *counts, global int *itcount)
{
  ulong old_err, err = INT_MAX; int itcount1 = 0;

  uint tmplabels[MAXPTS], tmpcounts[MAXK],                  E1
  tmpsum[MAXM * MAXK], tmppts[TMPPTS * MAXM],
  tmpcent[MAXM * MAXK]
  __attribute__(xcl_array_partition(cyclic,16,1));

  // compute points per burst                               E2
  int ppb = BURSTSZ/(m * (sizeof(uint) * 8));
  int nbursts = k / ppb;
  if(nbursts == 0) nbursts = 1;

  // get initial centroids
  int stride = BURSTSZ/(sizeof(uint)*8);
  for(int i = 0, aux = 0; i < nbursts; i++)
    *(uint16 *)&(tmpcent[stride * i]) = centroids[i];

  do {                                                      A
    old_error = error, error = 0;
    for (i = 0; i < k; i++)
        tmpcounts[i] = 0;
    for (i = 0; i < k * m; i++)
        tmpsum[i] = 0;

    int ptctr = TMPPTS;                                     B
    for(int h = 0; h < n; h++) {

      if(ptctr == TMPPTS) {                                 E3
        ptctr = 0;
        int nbursts = TMPPTS / ppb;
        if(nbursts == 0) nbursts = 1;
        for(int i = 0; i < nbursts; i++)
          *(uint16 *) &(tmppts[stride * i])
             = data[(h/ppb) + i];
      }

      (...) // for every centroid                           C

        // use "tmppts" & "tmpcent"                         D
        // to compute distances

        (...) // compare dist with mindist

      (...) // update err; update "tmpsum"
      ptctr++; // next "tmppt"
    }
    (...) // update "tmpcent"
  } while(abs((err - old_err)) > t);

  for(int i = 0, j = 0; j < k; i += 16, j++)                E4
    centroids[j] = *(uint16*) &(tmpcent[i]);

  for(int i = 0, j = 0; i < k; i += 16, j++)
    counts[j] = *(uint16*) &(tmpcounts[i]);

  for(int i = 0, j = 0; i < n; i += 16, j++)
    labels[j] = *(uint16*) &(tmplabels[i]);
}
```

**LISTING 4.** Version v1b (v1 + burst access).

```
__kernel void s1kmeans5b8(
  global uint16 *data, int n, int m,
  int k, float t, global uint16 *centroids,
  global uint16 *labels, global uint16 *min_dist)
{
  size_t gsz0 = get_global_size(0U);
  size_t gid0 = get_group_id(0U);
  int npoints = n/gsz0;
  int offset = gid0 * npoints;

  uint tmplabels[MAXPTS], tmpdist[MAXPTS]                   E1
  __attribute__(xcl_array_partition(cyclic,16,1));
  uint8 tmppts[TMPPTS], tmpcentr[8 * MAXK]
  __attribute__(xcl_array_partition(cyclic,2,1));

  for(int i = 0; i < k/2; i++) {                            E2
    // All burst reads and writes
    // are performed with the maximum data
    // width, regardless of N, D, or K
    uint16 tmpread = centroids[i];
    tmpcentr[(i*2)+0] = tmpread.lo;
    tmpcentr[(i*2)+1] = tmpread.hi;
  }

  int ptctr = TMPPTS;                                       B
  for(int h = 0; h < npoints; h++) {

    if(ptctr == TMPPTS) {                                   E3
      ptctr = 0;
      for(int j = 0; j < TMPPTS/2; j++) {
        int idx = ((offset + h)/2) + j;
        uint16 tmpread = data[idx];
        tmppts[(j*2)+0] = tmpread.lo;
        tmppts[(j*2)+1] = tmpread.hi;
      }
    }

    (...) // for every centroid                             C

      // adapt D segment in kmeansv2/v3                     D
      // to resort to "tmppts" and
      // "tmpcntr" to compute distances

      (...) // compare dist with mindist

    ptctr++;
  }

  int i = 0, idx = (offset/16);                             E4
  for(i = 0, j = 0; i < npoints; i += 16, j++)
    labels[idx + j] = *(uint16 *) &(tmplabels[i]);

  for(i = 0, j = 0; i < npoints; i += 16, j++)
    min_dist[idx + j] = *(uint16 *) &(tmpdist[i]);

}
```

**LISTING 5.** Version v5b8 (v5 + to burst access).

The k-means kernel is implemented here by four loops: loop $A$ clears intermediate calculations and checks for the termination condition, loop $B$ iterates through every data point to assign to a cluster, loop $C$ computes the distance of each point to each candidate cluster, and loop $D$ iterates through each dimension of the data points. This version receives as inputs the integer parameters $n$, $m$, and $k$, which respectively represent the number of data points, the dimensionality of the data, and the number of clusters.

The distance metric used in section $D$ is the Squared Euclidean distance. All implementations avoid the use of the square root, since this operation is particularly costly in FPGAs for two reasons. Firstly, the square root is implemented as a function call which, when placed into an innermost loop, makes it difficult to pipeline the loop in hardware, and secondly because the square root requires its input argument to be a floating-point number. We chose to avoid floating-point operations due to the expected performance impact. Casting an integer to a floating-point value is also costly, and likewise prevents pipelining of the scope in which it is performed, as we observed in previous work [41]. We also needed to divide the new candidate distance, *dist*, and the current minimum distance by two, when performing the comparison in section $C$. Without doing so, comparing two values of type *ulong* directly, would lead to an incorrect behavior where the comparison would be performed as if values were signed long integers. To prevent this, we discard the most significant bit, and cast the values to signed integers.

Versions *v2* and *v3* are shown in Listing 2. The differences to *v1* are the data type of some input arrays (e.g., *uint8* for *v2*, as highlighted), and the replacement of the innermost loop in section $D$ with a vector operation to compute the distance over the $D = \{8, 16\}$ dimensions. This modification was tested to determine if explicit vectorization of the loop in $D$ is required for the HLS compiler to pipeline the loop in $C$.

## B. NDRange KERNEL (MULTI-THREAD) VERSIONS

While the former three code versions relied on a single work-group, i.e., thread, the versions shown in Listing 3 adopt a multiple work-group approach. This means multiple CUs can be instantiated to execute the kernel, and further parallelism can be achieved, whilst still allowing for loop pipelining to be applied to the innermost loops in *C* and *D*. Relative to *v4*, *v5* (omitted for clarity) merely replaces the loop in section *D* in the same fashion as the vectorization performed for *v2* shown in Listing 2, and likewise adjusts the data types of the input arguments *data* and *centr* to *uint8*.

The outermost loop is moved to the host side. This means that the kernel is called multiple times, while the termination condition is checked on the host, along with the calculation of the new centroids. Since each work-group handles only a batch of the total amount of data points, it would only be possible to compute partial sums, which would then be reduced on the host, and finally divided by the number of points in each cluster to compute the new centroids. Instead, we chose to compute the new centroids entirely on the host to reduce the data traffic. For *v4/5*, the data transferred are the new labels per point, and each point's distance to the centroid, in order to compute the new error on the host side also. The *v6* variant (omitted for clarity) is identical to *v4/v5*, except for the outermost loop, *B*, which was moved to the host side. That is, each work-group processes a single data point.

## C. OPTIMIZING MEMORY BANDWIDTH USAGE

Initial experimental results, shown later in Section V, lead us to refactor the code to optimize bandwidth use between the FPGA and the global memory, while also reducing the total number of accesses required. The kernel versions shown so far either do not vectorize their global input arguments (e.g., *v1*) to preserve flexibility, or relied on equating the vectorization width to the number of features in the input data (e.g., *v2/v3*). This allows for simpler code and is a straightforward transformation, but there is no advantage in not transmitting the most data per read/write to global memory. For the FPGA used (and its compilation environment), the maximum width of a data transfer is 512 bit, which corresponds to the vector datatype *uint16*. We re-implemented *v1* and multiple versions of *v5* (for $D = \{2, 8, 16\}$). The former functions as a secondary baseline, firstly to determine the gains of optimizing the data transfers by comparison with the primary baseline, and secondly to evaluate the improvement due to the multi-threading and vectorization adopted in *v5*, free from performance degradation due to poor usage of bandwidth.

Listing 4 shows the optimized version of *v1*. Segment *E1* declares local memories to store labels, the number of data points in each cluster, the sum of all data points in each cluster, a local batch of data points (whose amount is defined by *TMPPTS* constant), and the new centroids. All memories are given the attribute *xcl_array_partition*, with a cyclic partitioning strategy into 16 sub-ranges, with dimension 1. This means that each declared array is instantiated as a 16-port local memory, to which we can read/write 512 bit of data in a single clock cycle. This is necessary to allow for the burst writes/reads implemented in *E2*, *E3*, and *E4*. However, unlike the *v1* version, this introduces a limitation in the maximum supported values for *n*, *k*, and *m*, since the amount of on-chip BRAM is limited. However, this implementation still supports any combination of the three parameters (up to a given maximum for each). All omitted code in segments *A*, *B*, *C*, and *D* is the same as shown for *v1*, save for the fact that every operation is now performed over these local memories.

The burst reads are inferred by the compiler if the statement which assigns the global input argument to a local memory is placed within a loop amenable to pipelining, e.g., the loop within segment *E2*. This portion of code computes how many data points we are capable of retrieving per 512 bit transfer, which varies with the number of features, *m*. Then, it computes the number of bursts required to transmit all initial/current *k* centroids, and read one *uint16* datum into the local memory. We implement this via explicit pointer casting of the destination address, since we encountered inconsistent behavior and simulation errors when resorting to the standard OpenCL *vstore16* call. This happens likewise for writes to global memory, and in *E4* we resort to explicit pointer arithmetic and casts to send the computed centroids, counts and labels to the host, in place of calls to *vload16*. Like the read bursts, write bursts are inferred by the compiler if the write operation is expressed as shown, in an isolated loop with constant stride without other statements or loops.

Finally, the read loop within *E3* is similar to the read burst in *E2*, and is used to read *TMPPTS* data points at a time into the respective local memory. Without this loop, every iteration of the outermost loop of *B* would require an access to external memory to transmit a single data point. This periodic read every *TMPPTS* iterations of the loop reduces the total number of accesses, and increases the amount to data transmitted per access, reducing overall overhead.

We then applied the same strategy to *v5*. That is, we resorted to local memories and burst reads/writes, as shown in Listing 5. Segment *A* is once again moved to the host-side code, meaning that fewer local memories are required to hold new centroids and the number of points per cluster. Also, the version shown is specialized for $D = 8$, which simplifies the code further, particularly in the *E* segments, since it is easy to compute the number of points per burst, while avoiding the pointer arithmetic and casts required before. For versions *v5b2* and *v5b16* (specializations for $D = 2$ and $D = 16$), the partitioning for the *tmppts* and *tmpcentr* arrays must be modified, so that a single write/read corresponds to a complete datum. In the first case, the attribute is changed to *xcl_array_partition(cyclic, 4, 1)*, and in the latter, no partitioning is required. The loops in *E2* and *E3* must be changed accordingly. For instance, for $D = 16$, use of the *hi*/*lo* fields is not required, and a single assignment from the burst read to the *tmpcntr* array copies a complete datum.

If the number of features is a power of two, the OpenCL vector data types allow for this simplification. Otherwise, e.g., $D = 6$, we could either pad the two remaining *uint* elements in each *uint16* burst to zero, to keep data aligned, or opt to transmit partial packets. Either case requires additional code to handle the necessary data realignment.

Finally, unlike all other versions of the code, the partitioned local memories employed in *v1b* and *v5b*, which allow for pipelined read/write loops due to the multiple memory ports, are usable with the CPU, as the partitioning attribute is specific to the OpenCL compiler targeting the FPGA.

## V. EXPERIMENTAL RESULTS

### A. EXPERIMENTAL SETUP

The experimental setup consisted of a single desktop machine with an *Intel Core i7-6700K CPU (*4 GHz*)*, and an *Alpha Data ADM-PCIE-KU3* PCI-Express board with a Kintex-6 XCKU060 FPGA [42]. The host-side code allocated input and output arrays, read input data points from existing files, computed initial kernels through k-means++ [39], and performed the required setup to call the kernel. Data is exchanged between the CPU and the FPGA via the system memory, by resorting to traditional OpenCL API calls (specifically by Xilinx's OpenCL runtime). The same API is used to retrieve execution times of the kernel, while the total execution time is computed via system calls.

We executed each code version for several combinations of the amount of input data $N = \{32k, 64k\}$, number of clusters $K = \{8, 16\}$, and number of features $D = \{2, 8, 16\}$. Algorithm 2 shows the pseudo-code of the algorithm used to generate the twelve data sets processed in the experimental evaluation. Data sets can be generated with a given number of points, attributes, and clusters. Additionally, it ensures a specified upper bound to the value of each attribute, and a given minimum distance between centroids. The data points are distributed normally along each dimension, centered on one randomly generated centroid. The standard deviation along each dimension is randomized, as well as the correlation between dimensions. By changing the data types employed in the implementation, it is possible to generate data sets of integers of standard widths (e.g., 16 bit), floating-point numbers, or double-precision numbers.

Line 12 shows the new candidate centroid generation, where the distance of the new random candidate is compared to previously generated centroids until the condition is met. Afterwards, a vector holding each value for the *jth* dimension of all points in the new cluster is generated (Line 33). A component derived from the $(j-1)th$ dimension is applied via Cholesky decomposition to induce correlation (Line 36). Finally, the *NxD* matrix corresponding to each $cluster_i$ is constructed by appending each $j$ dimension (Line 38).

We performed four executions for each case to retrieve an average execution time. We executed all versions of the code both on the FPGA and on the CPU (save for *v1b* and *v5b* for the latter, as the code is not supported), to provide additional comparisons beyond the baseline code. We generated the data sets automatically based on the $N$, $K$, and $D$ parameters, assigning approximately the same amount of points to each cluster, and also ensuring random correlation between the data features. For all data sets, we confirmed that FPGA and

**Algorithm 2** Pseudo-Code for Data Set Generation

```
Data: Number of points $N$, Number of dimensions $D$
      Number of clusters $K$, $maxval$, $spread$
Result: Dataset $\{cluster_1, \dots, cluster_k\}$,
        $centroids = \{centroid_1, \dots, centroid_k\}$
1  Function $randn(m = 1, n = 1)$ is
2      return $A_{mxn}, \{a_{ij} = X \sim \mathcal{N}(0, 1)\}$
3  Function $rand(m = 1, n = 1)$ is
4      return $A_{mxn}, \{a_{ij} \in \mathbb{R} | 0 < a_{ij} < 1\}$
5  Function $lsq(a, b)$ is
6      return $\sum_{i=0}^{N} (a_i - b_i)^2$;
7  Function $randV()$ is
8      return $(rand(1, d) \times maxval) + spread$;
9  Function $newCentroid(centroids)$ is
10     $min \leftarrow 0$;
12     while $min > 2 \times spread^2$ do
13         $newc \leftarrow randV()$;
14         foreach $c \in centroids$ do
15             $lsq \leftarrow lsq(newc, c)$;
16             if $lsq < min$ then
17                 $min \leftarrow lsq$;
18     return $newc$;
19 Function $generateDataSet(N, D, K, maxval, spread)$ is
20     foreach $cluster_i$ do
21         if $i == 0$ then
22             $centroid_i \leftarrow randV()$;
23         else
24             $centroid_i \leftarrow newCentroid(centroids_{0,i-1})$;
25         $//Number\ of\ points\ in\ cluster$
26         $NP = floor((rand() + 1) \times N/(K \times 2))$;
27         $//Generate\ j_{th}\ coordinate\ for\ all\ points\ in\ cluster_i$
28         $variance = spread$;
29         foreach dimension $j \in D$ do
30             $variance \leftarrow randn() * variance$;
31             $corr \leftarrow rand() - 0.5$;
33             $coords_j \leftarrow randn(NP, 1) * variance$;
34             if $j > 1$ then
36                 $coords_j = coords_{j-1} \times corr + coords_j \times sqrt(1 - corr^2))$
38             $cluster_{ij} \leftarrow coords_j$;
39         $cluster_i \leftarrow cluster_i + centroid_i$;
```

CPU execution produced the same point clustering results and required the same number of iterations. We present the speedups of the several code versions relative to the *v1* baseline, followed by the improvements achieved by burst optimization, a comparison between FPGA execution and CPU execution, and finally summarize resource requirements and power and energy consumption.

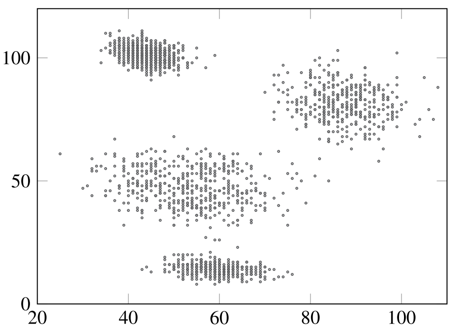


**FIGURE 2.** Example data set generated for $D = 2$, $K = 4$, and $N = 4096$.

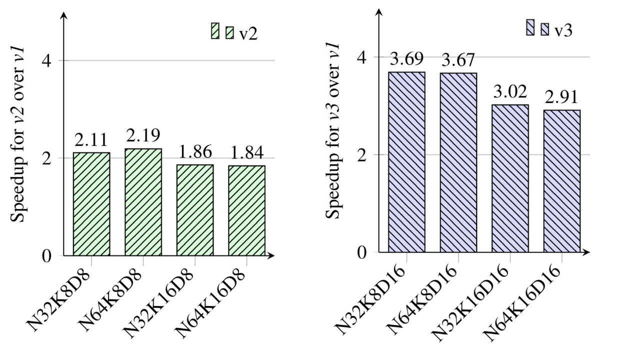


**FIGURE 3.** Speedup for *v2* and *v3* over baseline *v1* (one CU for all cases).

## B. SPEEDUPS FOR TASK-KERNEL (SINGLE-THREAD) IMPLEMENTATIONS

Figure 3 shows the speedups for *v2* and *v3* over the *v1* baseline, for the four specific cases that each version supports, i.e., where $D = 8$ or $D = 16$. All cases achieve speedups, and varying the data dimensionality results in the most significant change, as expected. However, there is some variation introduced by the $N$ and $K$ parameters, most noticeably the latter. Speedups for both *v2* and *v3* are greater for their runs where $K = 8$, than those where $K = 16$.

Since the data sets are different, we cannot draw direct conclusions based on the total run time. Instead, when computing the execution time required per iteration for all data sets, and comparing *v1*, *v2* and *v3*, we find that for all three the variation is in near direct proportion to $N$. However, the execution time for *v1* increases by approximately 25 % when $K$ is doubled. For the latter two cases, the increase is between 45 % and 55 %. We conclude that since the code in segment $D$, and the loop which computes the auxiliary sum *c1* have been accelerated, the impact of increasing $K$ on the first and last loops of segment $A$, which iterate over $k$, is greater.

We can verify this by analysing the data transfer profiles. Table 2 shows this data for the runs where $K = 8$. Data are shown, for both read and write transactions, for the total number of accesses, the transfer rate, the respective utilization of transfer bandwidth, and the average transfer size. The transfer rate is computed from the total amount of data transferred over the execution time. The bandwidth utilization is computed from the transfer rate over the maximum transfer rate of the FPGA board. Finally, the transfer efficiency in the last row is a combined metric for the read and write transactions. It is equal to the average number of bytes per transfer over 4 kB, which is the maximum burst size of the AXI4 bus which connects the CUs to the global memory.

**TABLE 2.** Effect of data vectorization on data transfer for task-kernel versions.

| Parameter | N64K8D8 | | | | N64K8D16 | | | |
|---|---|---|---|---|---|---|---|---|
| | v1 | | v2 | | v1 | | v3 | |
| | Read | Write | Read | Write | Read | Write | Read | Write |
| #Acc. ($10^6$) | 90.8 | 6.91 | 13.4 | 2.79 | 60.1 | 3.79 | 4.38 | 0.84 |
| Trf. Rate (MB/s) | 27.3 | 2.08 | 70.5 | 14.7 | 31.5 | 1.98 | 135 | 25.9 |
| BW Used (%) | 0.24 | 0.02 | 0.61 | 0.13 | 0.27 | 0.02 | 1.2 | 0.22 |
| Avg. Size (B) | 4 | 4 | 32 | 32 | 4 | 4 | 64 | 64 |
| Efficency (%) | | 0.10 | | 0.78 | | 0.10 | | 1.56 |

With respect to *v1*, we observe that *v2* achieves a read rate which is 2.58× (70.5 MB/s) greater, and a write rate which is 7.1× (14.7 MB/s) greater. For *v3*, the values are 4.3× and 13.1×, respectively. The average size of each data transfer increases by the width of the vectorization used, as expected. Note that although it is significant to compare data transfer rates, comparison of the total number of performed accesses across data sets (e.g., $N64K8D8$ vs $N64K8D16$) has no significance since the data sets converge differently and thus a different number of iterations are required.

However, when $K = 16$, the profiling reports for both *v2* and *v3* indicate that while the read transfer rate increases by 11 %, the write transfer rate decreases by 24 %, relative the respective cases where $K = 8$. The loop in segment $C$ for both *v2* and *v3* scales with $K \times D$, and performs data reads. The first loop of segment $A$ scales with $K$, and performs data writes. This is consistent with the observed transfer rates.

Despite this, both cases display a speedup derived from the vectorization and removal of the innermost loop, which is around 2× for *v2* and 4× for *v3*, which for both cases is 4× lower than what we expected to achieve, meaning that other factors are curtailing the benefits from wider data transfers.

Regarding pipelining, only the three innermost loops are pipelined for *v1*: the loop which clears the *c1* array, with an Initiation Interval (II) of 1 clock cycle, the loop in segment $D$, and the last loop which computes the new centroids, both with IIs of two clock cycles. This latter loop is the only one which is also pipelined for *v2* and *v3*, although with an II of 138 clock cycles. The loop in segment $C$ is not pipelined despite no longer containing any inner loops, and neither is the single loop which clears the *counts* and *c1* arrays. For the former case, the *if-else* statement likely prevents pipelining due to misprediction, and for the latter, the alternating access to two global arrays has the same effect.

## C. SPEEDUPS FOR NDRange KERNEL (MULTIPLE-THREAD) IMPLEMENTATIONS

Implementations *v4* and *v5* allow the instantiation of multiple CUs. Henceforth, we refer to number of cores of *NDRange* kernel implementations by suffixing the number of CUs to their identifiers, e.g., *v4#8* (8 CUs). Figure 4 shows the

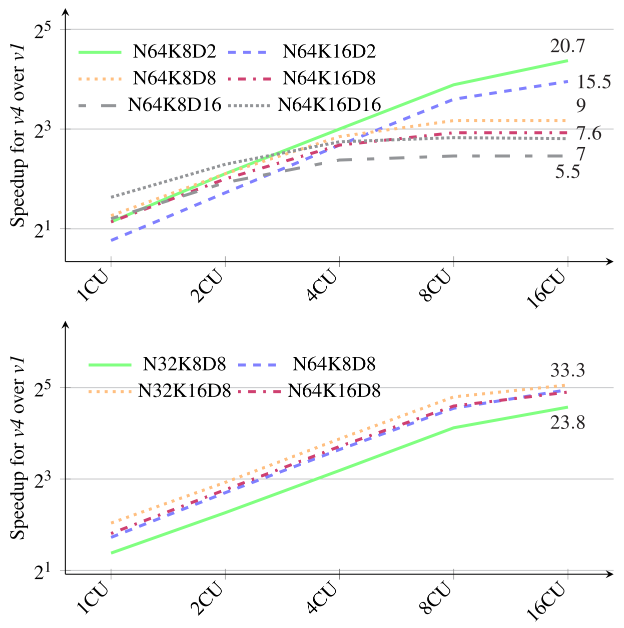


**FIGURE 4.** **Speedup for *v4* and *v5* over baseline *v1*, for several numbers of CUs.**

speedups for *v4* and *v5* over the baseline for several numbers of CUs (shown as incremental stacked bars). In every case, we set the number of work-groups to be equal to the number of CUs. We increased the number of CUs to the maximum of 16 permitted by the HLS flow, resulting in a noticeable effect on the speedup.

For *v4*, the four rightmost data sets omit the speedups attained for 16 CUs, since this number of cores for these cases does not alter the speedup, or results in a very small decrease, for reasons explained below. Analysing these cases must take into account that part of the k-means algorithm executes on the host. However, when comparing the time measurements retrieved from OpenCL calls, to the system timer, we conclude that the time spent on host execution accounts for only approximately 2 % of the total time.

For $D = 2$ and $D = 8$, the speedup decreases when $K$ is increased as we have shown in the previous section. Specifically, the execution time of *v4* varies in direct proportion to both $K$ and $N$. This can be explained by noting that the total number of iterations of the kernel is given by the number of points in the workgroup ($N \div \#CUs$) multiplied by $K$ and $D$. However, for $D = 16$, *v4* achieves a greater speedup for a higher value of $K$. Computing the execution time per iteration again, we observe that while it doubles for *v4* when doubling $K$, as explained, the total execution time is lower than that of *v1* for $N64K16D16$. This is because *v1* requires a total of six iterations to complete, while *v4* requires only three. Considering all data sets, we find that *v4* requires, on average, two thirds of the iterations required by *v1*, regardless of the number of CUs. This is unlike the behaviour for *v2/v3*, which has the same number of iterations as *v1* for all data sets.

Although the run time for *v4* increases with $D$ due to the loop in code segment $D$, we can see the mitigating effects from the modifications applied through *v5* on the right hand side of Figure 4, where we show the speedup for all cases supported by *v5*. The specialization for $D = 8$ results in analogous increases in performance to those of *v2* over *v1*, shown previously. We can directly compare $N64K8D8$ and $N64K16D8$ between both charts, and conclude that vectorization provides an improvement of approximately 3.9× for *v5* over *v4*, making this the most efficient implementation over the baseline, when resorting only to vectorization and multiple work-groups. We were able to increase the number of CUs up to the maximum permitted by the HLS flow, 16, without the loss of performance seen for *v4*. This indicates that all CUs perform concurrent work, and do not stall due to memory access contention, since the memory access time of each CU is reduced due to vectorization.

Finally, the execution time per iteration for *v5* also varies in direct proportion with $N$, but is no longer affected by $K$. We can observe this comparing $N32K8D8$ to $N64K8D8$ and $N32K16D8$, where $K$ no longer decreases the speedup. However, the overall performance is still less than expected when instantiating 16 CUs, each performing data transfers of vector data of type *uint8*. We explain these two points by again analysing profiling data.

**TABLE 3.** **Effect of kernel parallelism on data transfer.**

| Parameter | N64K16D8 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | v1 | | v4#8 | | v2 | | v5#8 | |
| | Read | Write | Read | Write | Read | Write | Read | Write |
| #Acc. ($10^6$) | 536 | 23.6 | 267 | 7.70 | 74.9 | 10.3 | 51.5 | 7.70 |
| Trf. Rate (MB/s) | 39.0 | 1.72 | 148 | 4.27 | 80.0 | 11.0 | 723 | 108 |
| BW Used (%) | 0.34 | 0.02 | 1.29 | 0.04 | 0.69 | 0.10 | 6.28 | 0.94 |
| Avg. Size (B) | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Efficency (%) | 0.10 | | 0.10 | | 0.78 | | 0.78 | |

Table 3 shows this data for $N64K16D8$. We first compare *v1* to *v4#8*, which both support any value of $N$, $K$ or $D$, to determine the effects of multiple cores on the transfer rates. The number of reads and writes performed to global memory are 2× and 3× greater for *v1* than those performed for *v4*, respectively. These values hold for any number of cores used for *v4*. In contrast, the transfer rates increase in proportion to the number of cores. An implementation with 8 CUs provides read and write transfer rates that are 3.8× (148 MB/s) and 2.5× (4.27 MB/s) greater than those of *v1*, respectively. Increasing the number of cores to 16 provides no improvement as we have mentioned, meaning that further increases of the transfer rates are hampered by how often accesses are performed.

Also shown in Table 3 is a comparison between *v2* and *v5#8*, i.e., between the task-kernel implementation specialized for $D = 8$ and its equivalent multi-thread version. We can thus establish the improvement derived from parallelism alone. We find that the transfer efficiency is equal, regardless of the number of cores used for *v5*. This indicates that all performance improvements derive from straightforward parallelism, and that the cores do not perform concurrent accesses to memory. This confirms that the degradation in performance in *v4* when the number of CUs is increased from 8 to 16, is due to idle time introduced due to global memory access contention by each CU. The total number of accesses and average transfer size (4 B) are also constant, for any number of cores.

Version *v6* achieves speedups comparable to *v4*, except for the case when only one CU is used, since there are as many kernel calls as data points, leading to a large overhead. In this case, there is a slowdown of 1 % for *v6* relative to the baseline. Conversely, we note that the speedup for *v4* saturates for eight or more CUs, even under-performing slightly relative to *v6*. Version *v6* performs better for the CPU.

Finally, like *v1*, the loop in segment $D$ of *v4* is pipelined with a II of two clock cycles, while the outer loops in $B$ and $C$ are not, and likewise for *v5*. For *v6*, the only loop which remains in the kernel (segment $C$), is pipelined with an II of 145 clock cycles.

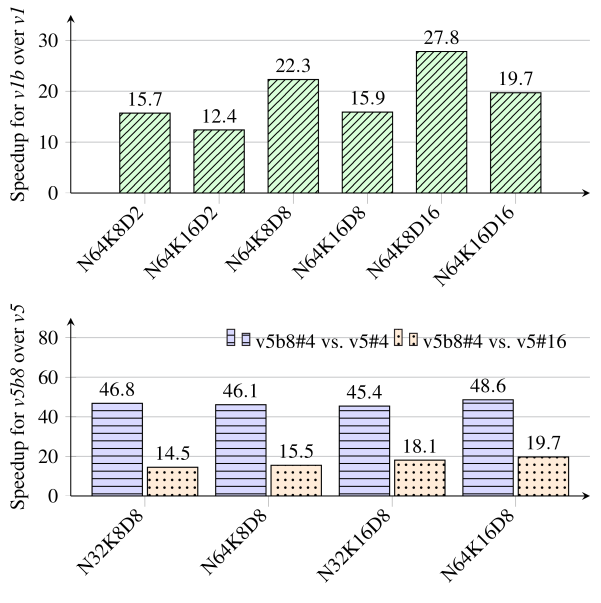


**FIGURE 5.** **Speedup for *v1b* and *v5b8* vs. the respective versions without burst optimization.**

### D. SPEEDUP FOR BURST OPTIMIZATION

Figure 5 shows the speedup for two burst optimized versions, *v1b* and *v5b#8*, over their respective baselines. As before, the number of work-groups are set to be equal to the number of CUs. For *v1b*, all eight innermost loops are successfully pipelined, with IIs between one and three clock cycles. These include all $E$ loops (see Listing 4), which are responsible for reads/writes to the global memory. Resorting to local memories allows for pipelining of other loops, such as the main loop in $D$ and the first two loops in $A$, and greatly decreases the amount of global memory accesses. Relative to *v1*, the improved version performs at most only 0.3 % and 0.1 % of reads and writes, respectively, for all data sets. We observe again the influence of increasing $K$ for any values of $N$ and $D$, and that for larger amounts of data with more features, the burst optimization shows the largest improvement.

On the right-hand side, we show two comparisons. Using the burst capable version, *v5b*, it is possible to instantiate at most four CUs due to resource requirements (specifically, insufficient BRAMs). Given this, we first compare *v5b8* (burst optimization with specialization for $D = 8$) with *v5* (specialization for $D = 8$) using only four CUs, to evaluate the gains of burst accesses alone; secondly, we evaluate *v5b8* using four cores, versus the non-burst version with 16 cores. In both cases, the burst optimization provides improvements upwards of 14×, and unlike previous cases, is not significantly affected by the number of clusters, $K$. When compared to *v1*, any specialization of *v5b* (i.e., for any tested value of $D$ = 2, 8, 16) achieves speedups between 414× and 752×, the latter being for $N64K16D16$. As the next section explains, this is the only code version that results in both energy savings and speedups over the CPU when running on the FPGA.

**TABLE 4.** **Effect of burst optimization on data transfers.**

| Parameter | N64K16D8 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | v1 | | v1b | | v5#1 | | v5b8#1 | |
| | Read | Write | Read | Write | Read | Write | Read | Write |
| #Acc. ($10^6$) | 536 | 23.6 | 0.95 | 0.004 | 51.5 | 7.70 | 0.03 | 0.12 |
| Trf. Rate (MB/s) | 38.5 | 1.72 | 17.6 | 0.08 | 105 | 15.7 | 98.3 | 24.6 |
| BW Used (%) | 0.39 | 0.02 | 0.15 | 0.001 | 0.91 | 0.14 | 0.85 | 0.21 |
| Avg. Size (B) | 4 | 4 | 64 | 64 | 32 | 32 | 1024 | 64 |
| Total (MB) | 2124 | 94.4 | 60.8 | 0.26 | 1649 | 246.4 | 34.5 | 7.9 |
| # Iterations | | | | 29 | | | | 15 |
| Efficency (%) | | 0.10 | | 1.56 | | 0.78 | | 6.25 |

Table 4 contains data transfer profiling information for the burst optimized versions for data set $N64K16D8$. On the left-hand side, the *v1* baseline is compared to its optimized version. Due to the use of the widest possible vector data type to perform reads and writes, the average transfer size for *v1b* is always 64 B, for any data set. However, a decrease of transfer rates is observed.

The use of local memories greatly reduces the accesses to global memory, and the use of burst transfers reduces the amount of time spent on data transactions. Since the transfer rate is computed given the amount of data to transfer, and the total run time of the kernel, this explains the reduced rates. For example, for the data set shown in Table 4, *v1b* reads 60 MB from global memory, while *v1* reads 2.1 GB. We also note that the amount of data written to global memory for *v1b* depends almost entirely on $N$ for all data sets, since the bulk of the data are the assignments of points to clusters per iteration. This means that write transfers do not vary with $K$ or $D$. In comparison to *v1*, the amount of data written to memory is between 60× and 350× smaller. In contrast, the amount of read data varies as a function of all three data set parameters, but is consistently 19× less for *v1b* when $K = 8$, and 35× lesser when $K = 16$. Only for $K = 8$ does *v1b* achieve a (marginally) higher read rate. Similarly to previous cases, when $K$ increases more time is spent on the loop in segment $C$, although the loops in *E3* still access data in bursts, and are independent of $K$, the additional run time in $C$ lowers the read rate. Regardless, for every data set the transfer efficiency of *v1b* over *v1* is 15.6× greater.

The right-hand side of Table 4 compares *v5* with its burst optimized equivalent. The most notable difference is the average read transfer size, which is 1024 B for *v5b8* (i.e. for specialization for $D = 8$) and *v5b16*, for any number of CUs. For *v5b2* the average is only 256 B. The values are as expected for these cases, as well as for *v1b*, and their differences are due to the achieved burst transfer length.

For any burst optimized version, the bulk of the read transactions is due to the loop in *E3*, which reads $TMPPTS = 32$ points at a time. For *v1b*, the trip count of this loop is data-dependant, which prevents the compiler from generating burst accesses. That is, the burst length is 1, although the maximum supported AXI transfer width of 64 B is used (i.e., a *utin16* vector data type). This is coherent with the number of read accesses per iteration of *v1b* for $N64K16D8$, which is $\sim 32k$ since 2 points out of the total of $64k$ are retrieved per iteration, and with the average transfer size of 64 B.

For any variant of *v5b*, the number of iterations of the loop is equal to $TMPPTS \div (16 \div D)$. For *v5b8*, since $D = 8$ and each iteration of the loop reads 64 B, the total amount of data read by the loop is 1 kB. This can be transferred in a single burst transaction, meaning that all iterations of the loop are processed at once. We did not specify a burst length for the inferred burst transfers, which lead the compiler to limit the length to 16. We observe this for *v5b16*, where the average transfer size is also 1 kB. This means that the 32 iterations of the loop in *E3* which fetch a total of 2 kB is implemented as 2 burst transactions, which is coherent with the number of measured read accesses. Despite this, the maximum transfer rates are achieved for *v5b16*. The read and write rate scale with the number of cores, with maximums of 524 MB/s and 65 MB/s for 4 CUs. The resulting transfer efficiency is 93.8× greater than that of the baseline.

Further performance improvements may be possible by increasing the *TMPPTS* parameter. This is only limited by the amount of the FPGA's on-chip memory. The *v5b* versions are also constrained by the memory limit through the values set for the *MAXK*, *MAXM*, and *MAXPTS* parameters, but achieve the best performance. In contrast, versions that do not resort to local memories, such as *v5*, support unbounded values for the data set parameters $K$, $N$, and $D$, and although they do not provide speedups over the CPU baseline, energy savings are still possible.

### *E. FPGA VS CPU: SPEEDUPS AND POWER CONSUMPTION*

While so far we have presented the performance gains of incremental optimizations to the OpenCL code for the FPGA over its sequential baseline, this section compares FPGA execution with CPU execution, by executing all (portable) code versions on the latter. The *Intel i7-6700K* used contains four cores, which are subdivided into a total of 8 OpenCL CUs.

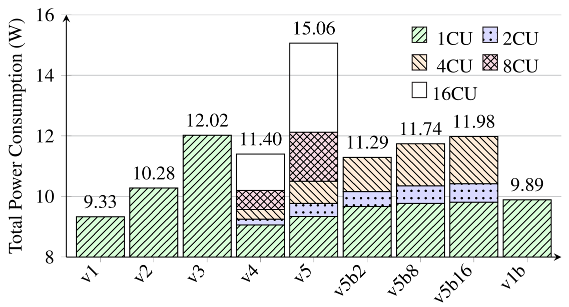


**FIGURE 6. Power consumption on FPGA for all cases and different numbers of CUs.**

Additionally, Intel's OpenCL runtime performs implicit vectorization of the kernel arguments [43]. In contrast, explicit vectorization is required when targeting the FPGA.

**TABLE 5. Best code version for CPU and FPGA, respective speedup, and power/energy consumption.**

| {N,K,D} | Best Version | | Speedup | Power (W) | | Energy (J) | |
|---|---|---|---|---|---|---|---|
| | CPU | FPGA | | CPU | FPGA | CPU | FPGA |
| {32,8,2} | v1#1 | v5b2#4 | 0.83 | ≈40 | 11.29 | 1.32 | 0.45 |
| {64,8,2} | v6#8 | | 0.86 | | | 0.69 | 0.23 |
| {32,16,2} | v6#2 | | 0.75 | | | 1.69 | 0.63 |
| {64,8,2} | v6#8 | | 0.75 | | | 2.28 | 0.86 |
| {32,8,8} | v2#1 | v5b8#4 | 0.76 | ≈40 | 11.74 | 0.65 | 0.25 |
| {64,8,8} | v2#1 | | 0.78 | | | 0.86 | 0.33 |
| {32,16,8} | v6#4 | | **1.54** | | | 1.06 | 0.20 |
| {64,16,8} | v2#1 | | **1.16** | | | 4.32 | 1.09 |
| {32,8,16} | v3#1 | v5b16#4 | 0.81 | ≈40 | 11.98 | 0.33 | 0.12 |
| {64,8,16} | v3#1 | | 0.76 | | | 0.55 | 0.22 |
| {32,16,16} | v3#1 | | **1.50** | | | 1.38 | 0.28 |
| {64,16,16} | v6#4 | | **1.44** | | | 1.46 | 0.30 |

Table 5 shows overall performance results as a function of the three parameters, $N$, $D$, and $K$, namely, which version of the code achieves the best performance for the CPU and FPGA, the speedup attained by FPGA execution over the CPU, and the respective power and energy consumption. For the CPU, the version which leads to the best result varies based on the input parameters, and is either *v2#1*, *v3#1*, or *v6#4*. The only exception is for input parameters $N32K8D2$ for which *v1* performs best, although only 1.03× better than *v6#16*. This contrasts with FPGA execution, for which *v6* performs the worst, due to the much higher number of kernel calls, transfer of data between the FPGA and the global memory, and the small workload of each kernel call itself.

As we had concluded before, versions of *v5b* specialized for the target data dimension $D$ always perform best by the FPGA. For most other cases, no speedup is attained over the CPU, except for the four instances highlighted in Table 5. Note that the speedup for, for instance, $N32K16D8$ is double that of the analogous case where $K = 8$. This holds for the other cases where speedups are achieved, which hints that the performance gains attained on the FPGA for *v5b* are derived from successfuly pipelining the main loop of segment $C$. Since this loop iterates up to the value of $K$, a sufficient number of iterations help in reaching a trade-off point where this strategy is more beneficial than the higher parallelism (i.e., more compute units) possible on the CPU. This behaviour does not occur for the four cases for which $D = 2$; we attribute this to the fact that the number of temporary points (*TMPPTS* in Listing 5) is constant (32) for all implementations of *v5b*. Since four data transfers are enough to read this amount of datapoints in segment *E3* when $D = 2$, this loop is executed much more often relative to the cases of $D = 8, 16$, leading to more overhead.

We retrieved power consumption values for the FPGA from post-route reports, and for the CPU using Intel's Running Average Power Limit (RAPL) interface [44] which queries processor condition in real-time. All versions of the code, when implemented on FPGA, dissipate approximately the same power, which scales mostly with the number of CUs instantiated. The power consumption ranges from 9.06 W (for *v4*) to 15.05 W (for *v5#16*), with 13 out of the 28 implemented circuits consuming between 9 W and 10 W. We retrieved power consumption metrics from the CPU for all code versions which were compatible with its OpenCL runtime (i.e., all except *v1b* and *v5b*), and concluded that the power consumption was very nearly equal for all cases, with an approximate value of 40 W. Finally, we evaluated the energy consumption for execution on the FPGA. Despite the increased execution time relative to the CPU, for most of the cases, a decrease between 60 % and 80 % in total energy consumed is possible for all cases. The greatest decrease occurs for *v5b#16*, when computing the centroids for the data set where $N = 32k$, $K = 16$, and $D = 8$.

**TABLE 6. Resource requirements and operating frequency for each version's maximum achievable number of CUs (LUTs and FFs are shown in thousands; percentages are relative to available kernel budget).**

| Ver.#CUs | LUTs | | FFs | | BRAM36 | | DSPs | | Freq. (MHz) |
|---|---|---|---|---|---|---|---|---|---|
| v1#1 | 10 | (4%) | 12 | (2%) | 1 | (<1%) | 31 | (1%) | 250 |
| v2#1 | 19 | (8%) | 27 | (5%) | 8 | (1%) | 36 | (1%) | 250 |
| v3#1 | 36 | (15%) | 51 | (9%) | 15 | (2%) | 60 | (2%) | 237 |
| v4#16 | 58 | (25%) | 82 | (15%) | 16 | (13%) | 384 | (14%) | 250 |
| v5#16 | 82 | (36%) | 15 | (27%) | 128 | (9%) | 624 | (23%) | 205 |
| v1b#1 | 18 | (8%) | 26 | (5%) | 87 | (59%) | 21 | (1%) | 245 |
| v5b2#4 | 31 | (13%) | 57 | (10%) | 572 | (59%) | 84 | (3%) | 242 |
| v5b8#4 | 35 | (15%) | 60 | (11%) | 572 | (59%) | 156 | (6%) | 250 |
| v5b16#4 | 36 | (16%) | 62 | (11%) | 572 | (59%) | 252 | (9%) | 182 |

## F. RESOURCE REQUIREMENTS

The resource requirements for the evaluated cases are shown in Table 6, comprising Lookup Tables (LUTs), Flip-Flop (FF), BRAMs, and Digital Signal Processors (DSPs) blocks. We show only each case's implementation with the highest number of CUs achievable, e.g., for the *v5b* cases, more than four CUs were not possible due to the resource requirements. For reference, the FPGA used contains 331 thousand LUTs, 663 thousand FFs, 1 thousand 36 bit BRAMs, and 2.7 thousand DSP blocks. The resources shown in Table 6 are for the kernel modules only, and do not include the hardware overhead of AXI buses. This overhead is fixed for all cases, representing 30 %, 17 %, and 10 % of all LUTs, FFs, and BRAMs, respectively. Thus the percentages shown in Table 6 are relative to the remaining resources available for use.

The baseline case *v1* requires the least amount of resources as it is the simplest implementation. Specifically, the number of required BRAMs is very low, since no local memories are required; the parameter *LUTMem* represents how many LUTs were used as local distributed memory. The number used is smaller than for all other cases for the same reason.

The effect of instantiating multiple CUs is noticeable for *v4/v5*, for which sixteen CUs were instantiated. The number of resources varies linearly with the number of CUs, so when comparing the cost of a single CU for *v4/v5* with *v1*, both cases require less resources overall. This is expected, since a considerable portion of the code was moved to the host, namely all loops in segment $A$ and the last loop in segment $B$, along with all arrays used to hold intermediate results.

On the other hand, each CU of any burst optimized version requires approximately the same number of LUTs and FFs as *v1*, with the exception of the number of BRAMs, which are used to hold all temporary variables introduced in the $E$ segments. In fact, it was the number of required BRAMs that limited the number of CUs to four in all these cases.

Regarding operating frequency, nearly all cases tested (i.e., any code version implemented by any number of CUs) operates at 250 MHz, except for *v5b2*, *v5b8*, or *v5b16*. For these cases, the lowest operating frequency is 182 MHz, and occurs for *v5b16#4*, while the remaining burst optimized instances achieve between 200 MHz and 250 MHz.

## G. COMPARISON TO C IMPLEMENTATION

In order to determine the impact of resorting to integer data types for intermediate calculations in the kernels, such as distances, we performed a comparison to pure $C$ execution on the CPU. The $C$ implementation of k-means used was the starting point for the *v1* implementation in OpenCL, and is in all aspects identical, except for the data types. All data types in the $C$ executions are *double* where appropriate (e.g., centroids and intermediate calculations.

We then retrieved the clustering results for all data sets for this implementation, and computed the error relative to execution of *v1* on the CPU. Note that, as we have stated before, we had previously verified that the clustering results obtained through OpenCL were identical between CPU and FPGA. The error relative to *double* precision is given by:

$$\sum_{k=0}^{K} \|d_k - c_k\| / \|d_k\| \tag{1}$$

where $d_k$ and $c_k$ are the centroids computed by the $C$ version and the centroids computed by the OpenCL implementations, respectively. All runs for the $C$ version used the same initial centroids as the OpenCL runs for each respective data set.

The resulting average error is 0.22 %, with a maximum error of 1 %. Additionally, we retrieved the execution times of the *C* version and compared them with the best case OpenCL per data set, as per Table 5 (columns 2 and 3). We find that the average speedup of OpenCL over *C* is 264× (81× — 461×) on the CPU, and 283× (67× — 667×) on the FPGA.

## VI. DISCUSSION AND OBSERVATIONS

In this set of experiments, we evaluated the performance of OpenCL kernels compiled for FPGAs by applying some of the possible implementation techniques on the k-means use case kernel. Specifically, we employed: explicit vectorization (and removal of innermost loops as consequence of vectorization), use of *NDRange* kernels (and consequent removal of outermost loops), and use of memories local to each work-group combined with burst memory access inference. Given the exploration presented, and the features explored, we present the following observations on the following aspects:

- type of kernel to use;
- effect of the number of CUs on the implementations;
- effect of the data set parameters on code structure and achievable number of CUs;
- use of loop pipelining and vectorization;
- possibility of optimization for energy consumption;
- use of local memories combined with burst accesses;
- algorithm workload performed by the host;
- cost-performance comparison of FPGA and CPU;
- generalized applicability of the developed code for other clustering/classification kernels, with two examples.
- unexplored strategies for kernel design

### *a: TASK-KERNELS VS. NDRange KERNELS*

For the tested implementations, *NDRange* kernels prove advantageous over task-kernels, as expected. Specifically, the *NDRange* kernels have global workgroup sizes of {>1, 1, 1}, and local sizes of {1, 1, 1}. Each work-group effectively behaves as a task-kernel which computes a given range of the trip count of the removed outermost loop. This approach exploits both the traditional OpenCL model of multiple identical work-groups with balanced loads, combined with the capability of compiling pipelined loops that is native to FPGAs. However, the application of this approach was only straightforward since there are no inter-iteration data dependencies in the outermost loop in *B*, which is a characteristic inherent to the target algorithm. Any such data dependencies would have to be addressed by transferring data between work-groups either through the host, by using global on-chip memory, or through channels. This would introduce the need for additional code (and therefore hardware) for synchronization between instances of the kernel (i.e., CUs).

### *b: EFFECTS DUE TO NUMBER OF COMPUTE UNITS*

Regarding the number of CUs employed, the resulting performance varies proportionally, as well as the amount of required resources. The achievable operating frequency is not directly affected by the number of cores (e.g., the operating frequency for *v4*, *v5*, and *v6* for any number of cores). Instead, the frequency only varies based on the type and amount of resources used by the kernel (e.g., BRAMs). Thus the number of CUs only indirectly decreases the frequency due to increased use of resources which affect routing (e.g., *v5b16* suffers a decrease for 4 CUs, relative to 2 CUs).

### *c: DATA SET PARAMETERS*

The $N$, $D$, and $K$ parameters of the data set do not restrict the number of CUs that can be instantiated. Instead, the parameters exert influence on code structure, depending on the nesting level of the loops they are directly associated with. The number of points $N$ controls the trip count of the loops in *B*, as seen in Listing 1. It cannot exert any influence on the possible code structure of the kernel unless it becomes the outermost loop, which is the approach adopted for *v4* and *v5*. This allows for the otherwise impossible partition of the total number of iterations into work groups, distributed among CUs. Also, for the solutions employing local memories (i.e., *v1b* and *v5b<D>*), there is an upper bound to the supported value of $N$, since these memories hold, for instance, the label assignments for all points in the data set.

The number of clusters, $K$, controls the trip count of the loop in *C*. This loop cannot be pipelined if the innermost loop $D$ has an unknown trip count. A restructuring based on this parameter would be possible if loops $B$ and $C$ were interchanged, with $C$ becoming the outer loop. Combined with moving loop $A$ to the host and therefore adopting an *NDRange* approach, separate ranges of iterations of $C$ could be divided among CUs. The disadvantage would be that each CU (i.e., workgroup) would need to access the entire range of $N$ input points. Although the total number of comparisons of each point to each centroid, and therefore memory accesses, is the same regardless of which loop is outermost, the implementations that resort to local memories benefit greatly from having the outermost loop access non-overlapping ranges of input data, and storing the much smaller amount of data required by the inner loop in local memories, which would not be possible if $C$ was the outermost loop.

The number of attributes $D$ exerts the greatest influence on code structure. Specialization through vector data types was based on the number of features, which also allowed for the removal of inner loops. Consequently, the structure of the code is influenced by supporting, or not, arbitrary run time values for $D$. Although $N$ and $K$ may assume arbitrary values for the kernel versions presented, arbitrary $D$ values are not supported for the best performing versions, i.e., *v5b<D>*. Depending on whether the number of features is smaller or greater than the vectorization width used, the loops responsible for burst transfers may require modifications. Also, the loop in $D$ (shown in Listing 1 and Listing 3) needs to be reintroduced if the number of features is greater than the vectorization width, and is conversely not required if the width is smaller. If the number of features is not a multiple or a divisor of the vectorization width, then further adjustments are required to account for alignment of the feature data. This would also require adapting the number of burst reads/writes required to transmit each complete feature vector.

#### d: OPTIMIZATION FOR ENERGY CONSUMPTION

The implementations shown were iteratively designed by evaluating the resulting execution time, without direct consideration given to power consumption. However, the power measurements taken during FPGA execution demonstrate that no significant differences are observed between either the kernel version, or for different numbers of CUs for the same version. Thus, directly optimizing for power consumption through code modifications/optimizations seems unpromising. The energy efficiency of the FPGAs is largely controlled by the static power consumption of the device, which is determined by technology parameters.

#### e: LOOP PIPELINING AND VECTORIZATION

We did not make explicit use of loop pipelining directives. The compiler is capable of generating pipelined loops under certain conditions. Namely, the loop must not contain function calls with arbitrary runtime (such as *sqrt*), accesses to high latency memory (i.e. global memory), or inner loops that cannot be fully unrolled.

Regarding function calls and memory access, we avoided use of the square root function (i.e. employed squared Euclidean distance), and employed local partitioned memories. We did not resort to manual loop unrolling or unrolling pragmas. Instead, vectorization using vector data types in the innermost loop in $D$ effectively removes this loop, and thus use of vectorization not only better utilizes data transfer bandwidth but enables more loop pipelining opportunities. Combined with the use of local memories for the data points and centroids, loop $C$ achieves a body amenable to pipelining. Although always advantageous, vectorization must be applied explicitly when compiling for FPGAs. The *if* clause in $C$ did not prevent loop pipelining either, since it does not contain any of the forbidden elements mentioned earlier.

Finally, when compared to loop unrolling, removing the inner $D$ loop is limited by the maximum vectorization width, and how the width relates to the number of features. However, combining both methods is possible. For instance, replacing a loop without vectorization which iterates over a feature size of 32, with two unrolled iterations of an equivalent loop resorting to a vector data type of 16 elements.

#### f: LOCAL MEMORIES AND BURST ACCESSES TO GLOBAL MEMORY

The use of local memories leads to the greatest performance improvement. The only limitation on their use is the amount of available BRAMs. This limit can be reached due to the size of the local memories declared in the kernel code, or by instantiating more copies of the kernel, i.e., multiple CUs. In our solutions, we did not explore adjusting the size of the local memory used to hold the batches of points retrieved from the data set of $N$ points. All other local memories are large enough to hold a complete copy of the respective global data, e.g., the local copies of all centroid values, point labels, and current distances of each point to its assigned cluster. A potential improvement would be to replace these latter two local copies (i.e., *tmplabels* and *tmpdist* in *v5b*<*D*>) with smaller memories, and transmit label data to the host in a new loop analogous to *E3* which would execute after loop $C$. Since this loop would replace *E4*, the effect on run time would be minimal, but resource savings would be substantial. Also, this would allow support for unbounded values of $N$.

The use of local memory is only significantly beneficial however when combined with loops that result in inference of burst accesses to global memory. We achieve this with single-statement loops without conditional statements. The transfer rate is greater for wider vector data types. This means the available beat width (512 bit) is not automatically exploited when using narrower vector types by combining consecutive iterations of transfer loops into a single beat.

Regarding the maximization of transfer rate, an additional reader/writer kernel could be employed, which would perform burst accesses without interruption, and forward the data to other kernels via global on-chip memory or channels. However, this would impact on the load balance of the CUs, as discussed in the following section.

#### g: WORKLOAD PARTITIONING AND LOAD BALANCE

For implementations using multiple CUs (i.e., *v4* and *v5*), we structured the code to ensure a load balance between the cores. The bulk of the workload of the k-means kernel is the calculation of point distances to cluster centroids, which we distribute evenly among the CUs. The calculation of new centroids is done by the host.

Alternatively, the reduction operations of the centroid update step may be implemented in the OpenCL kernel as well, in three different ways: by writing work-item specific code, thus performing partial reductions on only one work-item per work-group, by writing work-item specific code where one work-item performs the full reduction for all work-groups, or by using multiple kernels. The first two methods would require considerably more control code, and result in additional hardware requirements. For instance, synchronization barriers would be required to idle the work-item responsible for reduction while the remaining items completed execution.

If resorting to an additional kernel for centroid update, either global memories or communication channels would be required. Also, only one CU would be in use during the centroid update, since there is no inherent parallelism in this step. At best, the sequential calculation of new centroids could be pipelined, but given the low volume of data to process, and that the FPGA operates at a lower frequency than the host code running on the CPU, we argue that this step is more efficient when executed on the CPU.

Instead, selectively executing some of the workload on the host allowed for kernel code resulting in uniform workload for all work-items, preventing CUs idleness and hardware overheads. The simpler code also facilitated the introduction of local memories and loops for inference of burst accesses. Work-item specific asymmetry was also avoided due to the local work-group sizes, as discussed in the next section.

Finally, another potential cause for work-group asymmetry are unbalanced *if-else* clauses. However, none are present in this case. Regardless, different execution times for work-items in different work-groups would only be relevant if there was exchange of data between groups.

### *h: LOCAL WORK-GROUP SIZES*

We did not define local work-group sizes, instead opting to bind the number of work-groups to the number of CUs. Except for *v6*, all local work-group sizes are $\{1, 1, 1\}$. Our *NDRange* kernel implementations (e.g., *v5*), effectively act as parallel task-kernels each processing a subset of the data. The number of points processed by each work-group is set implicitly by the iteration bounds of the loop in *B* (see Listing 5). Setting a local work-group size of $\{n > 1, 1, 1\}$ would considerably increase code complexity.

Specifically, the loops in *E1*, *E2*, and *E4* would have to be enclosed in conditional statements controlled by the local work-item *id*. That is, only one work-item (e.g. $id == 0$) would load data into local memories, and synchronization barriers would be needed to prevent the remaining work-items from proceeding until data was read in. Conversely, the work-item responsible for writing data in *E4* would be forced to halt until completion of the remaining items.

Additionally, for a local work-group size equal to $\{n > 1, 1, 1\}$ (where $n < N$), the main loop in *B* would process a different subset of size $n$. This means multiple instances of *E3* would be required per work-item, and therefore additional local memories to hold each non-overlapping batch of temporary points. Idle time for each work-item would also occur due to competing global memory accesses, and due to accesses to the local memory for the centroid values in *C*.

Lastly, besides increasing code complexity, the need for conditional statements, synchronization barriers, and more local memories, would increase hardware overhead. This is significant considering that for our best performing implementations, *v5b<D>*, the maximum CUs that could be instantiated, 4, was already limited by the available BRAMs.

### *i: GENERALIZED APPLICABILITY*

The best performing implementations can be adapted easily to other similar algorithms, for example, for the k-nearest neighbours (kNN) algorithm [45]. The pseudo-code for this example is shown in Algorithm 3. The batch of candidate centroids read in segment *E2* of Listing 5 can be replaced with a subset of the points to classify. Then, loop *B* iterates over the same *npoints* subset of points, loading batches to local memories as per *E3*. The code in *C* can compute the distance of each point to classify to each subset point, and store the

**Algorithm 3** Generalization of the Best Kernel Version for kNN Classfication (for Fixed Feature size)

```
Data: Size $N_s$ of subset of known points
      $S = S_0, \ldots, S_{N_s}$, Size $N_c$ of subset of points to
      classify $P = P_0, \ldots, P_{N_c}$, number of
      neighbours $K$
Result: One set $C_i = C_{i_0}, \ldots, C_{i_k}$ per point $P_i$ of its
        k-nearest points in the input subset $S$
declare $lP$;                          //local memory for $P$
declare $lC$;                          //local memory for $C$
declare $lS$;                          //local memory for $S$ batches
$lP \leftarrow P$;                     //Vectorized burst read loop
$c \leftarrow 0$;
for $i = 0$ to $Ns$ do
    if $lS$ batch consumed/empty then
        $c \leftarrow 0$;
        $lS \leftarrow$ Batch of $S$;  //Vectorized burst read loop
    forall the $lP_j$ in $lP$ do
        if distance($lP_j$ to $lS_c$) < maxDistance($C_j$) then
            Replace index of most distant point in set $C_j$
            with that of known point $S_i$ (i.e., $lS_c$)
    $c \leftarrow c + 1$;
$C \leftarrow lC$;                     //Vectorized burst write loop
```

indexes of its k-nearest points. Lastly, a final selection of the k-nearest neighbours can be performed on the host over a reduced set of candidates, whose number is equal to the number of work-groups times $k$.

Another example is the Mean-Shift clustering [46] algorithm. The pseudo-code of this example is shown in Algorithm 4. Loop *E3* can load an arbitrary number of search window centers into a partitioned local memory. Loop *B* can iterate over the same subset of data set points, compute a partial sum of all points within the search window distance, and the calculation of the new search window center could be performed on the host, as well as verification of convergence.

### *j: COST-PERFORMANCE COMPARISON WITH CPU*

The FPGA in the *Alpha Data ADM-PCIE-KU3* accelerator card used, a Xilinx Kintex UltraScale XCKU060-2, achieves notable performance in comparison to an *Intel Core i7-6700K CPU*, especially considering the release date of both devices, 2014 and Q2'2015, respectively, and that the CPU was a high-end device on release. The retail price of the FPGA card is 6× higher ($2700) than the CPU, but considering that the power consumption and possible speedups result in 4.8× better energy efficiency, the equivalent cost is only 1.25× higher, in exchange for 1.5× faster execution time. Also, the CPU was manufactured on a 14 nm node, and the FPGA on a 20 nm node. Considering this difference, and that the FPGA is a mid-range device for its family, these results indicate a potential for further performance gains for more recent high-end FPGAs manufactured at smaller nodes.

**TABLE 7.** Comparison and characteristics of related approaches.

| Work | Year | Algorithm | Platform | Approach | Baseline | Speedup |
|---|---|---|---|---|---|---|
| Hussain *et al.* [27] | 2011 | k-means | Virtex-4 ML402 Kit | HDL/Block Design | nVidia GeForce 9600M GT | Up to 6.7× |
| Pu *et al.* [26] | 2015 | kNN | Terasic DE4 (Stratix IV) | HLS of OpenCL | Intel i7-3770k | 148× |
| Tang *et al.* [28] | 2016 | k-means | Terasic DE5-Net (Stratix V A7) | HLS of OpenCL | Intel Xeon W3670 | Up to 21× |
| Muslim *et al.* [29] | 2016 | kNN | Alpha Data 7V3 (Virtex 7) | HLS of OpenCL | nVidia GTX960 | Up to 2.5× |
| Canilho *et al.* [30] | 2017 | k-means | Xilinx Zynq-7000 | HDL/Block Design | ARM Processor (Zynq-7000) | Up to 496× |
| Raghavan *et al.* [31] | 2017 | k-means | Virtex-6 ML605 Kit | HDL/Block Design | MicroBlaze Softcore CPU | Up to 368× |
| Shata *et al.* [32] | 2019 | k-means | Terasic DE5-Net (Stratix V A7) | HLS of OpenCL | 2 *NDRange* kernels on FPGA | Up to 3.2× |
| Cadenelli *et al.* [33] | 2019 | k-mers | Nallatech 510T (Arria 10) | HLS of OpenCL | Dual XeonE5-2680v3 | Up to 1.3× |
| Song *et al.* [34] | 2019 | kNN | Xilinx VCU1525 Kit (UltraScale+) | HLS of OpenCL | Intel Xeon E5-2699 | ≈ 56 CPU threads |
| Dias *et al.* [35] | 2020 | k-means | Virtex-6 ML605 Kit | HDL/Block Design | Raghavan *et al.* [32] | Up to 15573× |
| This work | 2020 | k-means | Alpha Data KU3 (Kintex 7) | HLS of OpenCL | Intel i7-6700K (OpenCL) | Up to 1.5× |

**Algorithm 4** Generalization of the Best Kernel Version for Mean-Shift Clustering (for Fixed Feature size)

```
Data: Size $N_s$ of subset of known points
      $S = S_0, \ldots, S_{N_s}$, number $N_k$ of search window
      centers $W = W_1, \ldots, W_{N_k}$, and window size $Ws$
Result: Partial sums $Wp = Wp_1, \ldots, Wp_{N_k}$ of new
        window centers
declare $lW$;                          //local memory for $W$
declare $lWp$; //local memory of window center partial
sums
declare $lS$;                   //local memory for $S$ batches
$lWs \leftarrow Ws$;                      //local copy of $Ws$
$lW \leftarrow W$;                          //Vectorized burst
read loop
$c \leftarrow 0$;
for $i = 0$ to $Ns$ do
    if $lS$ batch consumed/empty then
        $c \leftarrow 0$;
        $lS \leftarrow$ Batch of $S$;   //Vectorized burst read loop
    forall the $lW_j$ in $lW$ do
        if distance($lW_j$ to $lS_c$) < $lWs$ then
            $lWp_j = lWp_j + lS_c$;
    $c \leftarrow c + 1$;
$Wp \leftarrow lWp$                  //Vectorized burst write loop
```

### *k: UNEXPLORED DESIGN TECHNIQUES*

In this paper we did not explore the use of multiple heterogeneous kernels, and consequently did not employ variables declared in the global scope of the OpenCL source code files (visible to all kernels), and inter-kernel channels (i.e., OpenCL pipes). We note however some observations derived from preliminary exploration. Firstly, separating algorithm workload into multiple kernels is not a trivial task. It entails determining a segmentation point such that it not only separates two discrete processing steps in such a way that each can be optimized, but also then imposes an amount of data transfers between kernels which does not lead to prohibitive hardware overhead (e.g., global on-chip memories or channels). Secondly, designing two or more kernels to implement the totality of the algorithm workload may lead to load imbalance. Since kernels are heterogeneous, this may result in under-utilization of CUs and in overall performance degradation relative to a symmetric load implementation derived from a single-kernel approach. Lastly, since the FPGA OpenCL compilation flow allows determining the number of cores up to a fixed total, this introduces additional design parameters, namely, the number of cores per kernel.

## VII. COMPARISON WITH STATE-OF-THE-ART

Table 7 compares several aspects of the approaches summarized in Section II. Upper bounds for speedups are given since averages are not available for all cases, and also because some approaches report the arithmetic mean while others report the geometric mean. It is difficult to draw a straightforward comparison due to the diversity of baselines, data sets, and target devices. We find that the approaches most similar to our own, considering the baseline, approach, and platform, are Tang *et al.* [28] and Muslim *et al.* [29].

Regarding Tang *et al.* [28], the k-means algorithm was also implemented via OpenCL on a comparable FPGA. The approach relied on multiple kernels exchanging data via local memories, while our approach resorted to a single-kernel, either task-kernel or *NDRange*. The k-means implementation that ran on the Xeon W3670 was OpenMP based, making use of this CPU's six cores. From the achieved 21× speedup we may extrapolate a speedup of 126× if a single-core alone had been used. Our implementation achieved an average speedup of 283× for the FPGA over the CPU executing sequential C code (single-thread). We believe the greater performance is due to greater synthesis optimization by consolidating the entire kernel in a single loop. This also allows for loop pipelining of work-items, as opposed to handling each point as a single work-item, which requires greater OpenCL runtime overhead for workload scheduling.

Muslim *et al.* [29] evaluate performance for the kNN algorithm, which shares considerable similarities to k-means. The evaluation platform used contains a Virtex-7 FPGA, versus the Kintex-7 on our own board. While the achievable operating frequencies and process nodes are comparable, the Virtex-7 family contains significantly more digital signal processing blocks. The best performing implementation of kNN on the FPGA relied on two kernels (distance calculation and assignment) sharing a global on-chip memory loaded by burst accesses to the global system memory, similar to our approach. Both our approach and that of Muslim *et al.* use OpenCL execution on another device as a baseline to OpenCL execution on the FPGA. The code executing on the devices is either identical or very similar. Although we employ an Intel i7-6700K and Muslim *et al.* employ an NVIDIA GTX960 GPU, both devices were released in the same year (2015), and both contain 8 OpenCL cores. Given this, the resulting speedups between the approaches are consistent given the coding techniques employed.

From the reviewed work, we find that adoption of OpenCL has led to more complex and flexible designs. One advantage of FPGA-based accelerator boards programmed via OpenCL over HDL implementations is that, for algorithms like k-means or kNN, it is significantly more straightforward to support arbitrary values for parameters such as the number of clusters or number of attributes. When designing circuits at low-level, the hardware details may greatly depend on high-level algorithm details. This leads to specialization of circuits based on adopting fixed values for one or more algorithm parameters.

## VIII. CONCLUSION

This paper evaluated the performance of OpenCL code compiled for FPGAs, by applying multiple transformations to a baseline implementation of the case study of the k-means clustering algorithm. The resulting kernel version were compiled by commercial HLS tools. The resulting circuits were evaluated in terms of execution time for several data set sizes, number of features, and number of clusters. We compared the FPGA implementations amongst themselves, relative to a non-parallel oriented implementation of the algorithm serving as a baseline, in order to determine the effects of different coding techniques. We also compared the execution time and power consumption of the FPGA with a desktop CPU.

We find that the code that yields the best results for the FPGA is not portable to other OpenCL capable devices, due to the necessity of using vendor specific extensions for efficient exploitation of FPGA characteristics. Specifically, to achieve speedups over the CPU baseline, significant transformations were required to make use of local memories, by combining global memory accesses into well defined loops that the HLS compiler was capable of synthesizing into efficient burst accesses. Relative to the sequential baseline implementation of the k-means algorithm as a task-kernel on the FPGA, these optimizations improved the efficiency of global memory access by 15.6×. When combined with and *NDRange* implementation and data vectorization, this increases to 93.8×. The resulting speedups over the OpenCL baseline executing on the FPGA reach 725×.

Kernel implementations with these features resulted in the best FPGA performance for any combination of data dimensionality, number of clusters, and number of points. For four out of the twelve combinations parameters, the FPGA achieves speedups over the CPU of approximately 1.5×, and reduces energy consumption for all cases by 60 % to 80 %. Considering that the operating frequency for the FPGA is around 250 MHz for almost all cases (and only 182 MHz for the best case), and that the CPU operates at 4 GHz, the potential for performance improvement by per-application specialization via HLS for FPGA is well illustrated. Additionally, the FPGA we used lies on the low-end of its product range, indicating potentially higher gains for higher-end devices, especially if more on-chip memory is available.

## REFERENCES

[1] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–10.

[2] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Comput. Appl.*, vol. 32, no. 4, pp. 1109–1139, Feb. 2020.

[3] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached FPGAs for data center applications," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2016, pp. 36–43.

[4] Xilinx. (2017). *Vivado High-Level Synthesis*. Accessed: Apr. 11, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[5] MathWorks. (2017). *HDL Coder–MATLAB & Simulink*. Accessed: Apr. 20, 2020. [Online]. Available: https://www.mathworks.com/products/hdl-coder.html

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[7] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[8] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in *Advances in Systems Science*, J. Świątek, A. Grzech, P. Świątek, and J. M. Tomczak, Eds. Cham, Switzerland: Springer, 2014, pp. 483–492.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2, pp. 1–27, Sep. 2013.

[10] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on FPGAS," in *Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2012, pp. 531–534.

[11] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown, "OpenCL for FPGAs: Prototyping a compiler," in *Proc. Int. Conf. Reconfigurable Syst. Algorithm (ERSA)*, 2012, pp. 3–12.

[12] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from OpenCL programs," in *Proc. IEEE 19th Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2011, pp. 186–193.

[13] Xilinx. (2020). *SDAccel Development Environment*. Accessed: Apr. 11, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[14] Intel. (2020). *Intel FPGA SDK for OpenCL*. Accessed: Apr. 11, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html

[15] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982.

[16] D. Dua and C. Graff. (2017). *UCI Machine Learning Repository*. Accessed: May 25, 2020. [Online]. Available: http://archive.ics.uci.edu/ml

[17] T. D. Hocking, P. Goerner-Potvin, A. Morin, X. Shao, T. Pastinen, and G. Bourque, "Optimizing ChIP-seq peak detectors using visual labels and supervised machine learning," *Bioinformatics*, vol. 33, Oct. 2016, Art. no. btw672.

[18] M. Patrício, J. Pereira, J. Crisóstomo, P. Matafome, M. Gomes, R. Seiça, and F. Caramelo, "Using resistin, glucose, age and BMI to predict the presence of breast cancer," *BMC Cancer*, vol. 18, no. 1, p. 29, Dec. 2018.

[19] M. Mohammadi and A. Al-Fuqaha, "Enabling cognitive smart cities using big data and machine learning: Approaches and challenges," *IEEE Commun. Mag.*, vol. 56, no. 2, pp. 94–101, Feb. 2018.

[20] A. Metzger, P. Leitner, D. Ivanovic, E. Schmieders, R. Franklin, M. Carro, S. Dustdar, and K. Pohl, "Comparing and combining predictive business process monitoring techniques," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 45, no. 2, pp. 276–290, Feb. 2015.

[21] H. Soleimani and D. J. Miller, "Semi-supervised multi-label topic models for document classification and sentence labeling," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.* New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 105–114, doi: 10.1145/2983323.2983752.

[22] I. Guyon, S. Gunn, A. B. Hur, and G. Dror, "Result analysis of the NIPS 2003 feature selection challenge," in *Proc. 17th Int. Conf. Neural Inf. Process. Syst. (NIPS)*. Cambridge, MA, USA: MIT Press, 2004, pp. 545–552.

[23] D. G. Bailey, "Image processing using FPGAs," *J. Imag.*, vol. 5, no. 53, 2019.

[24] Z. Wang, B. He, and W. Zhang, "A study of data partitioning on OpenCL-based FPGAs," in *Proc. 25th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2015, pp. 1–8.

[25] L. Di Tucci, K. O'Brien, M. Blott, and M. D. Santambrogio, "Architectural optimizations for high performance and energy efficient smith-waterman implementation on FPGAs using OpenCL," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 716–721.

[26] Y. Pu, J. Peng, L. Huang, and J. Chen, "An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, pp. 167–170.

[27] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly parameterized K-means clustering on FPGAs: Comparative results with GPPs and GPUs," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Nov. 2011, pp. 475–480.

[28] Q. Y. Tang and M. A. S. Khalid, "Acceleration of K-means algorithm using altera SDK for OpenCL," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 1, pp. 1–19, Dec. 2016.

[29] F. Muslim, A. Demian, L. Ma, L. Lavagno, and A. Qamar, "Energy-efficient FPGA implementation of the K-nearest neighbors algorithm using OpenCL," in *Proc. Position Papers Federated Conf. Comput. Sci. Inf. Syst.*, Oct. 2016, pp. 141–145.

[30] J. Canilho, M. Véstias, and H. Neto, "Multi-core for K-means clustering on FPGA," in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, Lausanne, Switzerland, 2016, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/7577313, doi: 10.1109/FPL.2016.7577313.

[31] R. Raghavan and D. G. Perera, "A fast and scalable FPGA-based parallel processing architecture for K-means clustering for big data analysis," in *Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal Process. (PACRIM)*, Aug. 2017, pp. 1–8.

[32] K. Shata, M. K. Elteir, and A. A. EL-Zoghabi, "Optimized implementation of OpenCL kernels on FPGAs," *J. Syst. Archit.*, vol. 97, pp. 491–505, Aug. 2019.

[33] N. Cadenelli, Z. Jaksić, J. Polo, and D. Carrera, "Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads," *Future Gener. Comput. Syst.*, vol. 94, pp. 148–159, May 2019.

[34] X. Song, T. Xie, and S. Fischer, "A Memory-Access-Efficient adaptive implementation of kNN on FPGA through HLS," in *Proc. IEEE 37th Int. Conf. Comput. Design (ICCD)*, Nov. 2019, pp. 177–180.

[35] L. A. Dias, J. C. Ferreira, and M. A. C. Fernandes, "Parallel implementation of K-means algorithm on FPGA," *IEEE Access*, vol. 8, pp. 41071–41084, 2020.

[36] P. Fränti and S. Sieranoja, "K-means properties on six clustering benchmark datasets," *Appl. Intell.*, vol. 48, pp. 4743–4759, Jul. 2018. [Online]. Available: http://cs.uef.fi/sipu/datasets/

[37] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Ann. Data Sci.*, vol. 2, no. 2, pp. 165–193, Jun. 2015.

[38] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowl. Inf. Syst.*, vol. 14, no. 1, pp. 1–37, 2008.

[39] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proc. 18th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2007, pp. 1027–1035.

[40] Xilinx. (2019). *UG1207–SDAccel Development Environment Methodology Guide-Performance Optimization*. Accessed: Apr. 11, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1207-sdaccel-optimization-guide.pdf

[41] N. Paulino, L. Reis, and J. M. P. Cardoso, "On coding techniques for targeting FPGAs via OpenCL," in *Proc. Int. Conf. Parallel Comput. Parallel Comput. Everywhere (ParCo)*, Sep. 2017, pp. 652–663.

[42] Alpha Data. (2018). *ADM-PCIE-KU3 User Manual 1.13*. Accessed: Apr. 20, 2020. [Online]. Available: http://www.alpha-data.com/pdfs/adm-pcie-ku3 usermanual.pdf

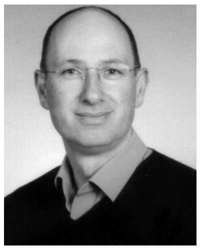
[43] Intel. (2019). *Benefitting From Implicit Vectorization*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/iocl-opg/top/coding-for-the-intel-cpu-opencl-device/benefitting-from-implicit-vectorization.html

[44] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanaa, and C. Le, "RAPL: Memory power estimation and capping," in *Proc. 16th ACM/IEEE Int. Symp. Low Power Electron. Design (ISLPEDNew)*, York, NY, USA: Association for Computing Machinery, 2010, pp. 189–194, doi: 10.1145/1840845.1840883.

[45] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *Amer. Statistician*, vol. 46, no. 3, pp. 175–185, Aug. 1992.

[46] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 17, no. 8, pp. 790–799, Aug. 1995.

**NUNO PAULINO** received the M.Sc. degree in electrical and computer engineering from the Faculty of Engineering, University of Porto, in 2011, and the Ph.D. degree in electrical and computer engineering from the University of Porto, in 2015. He is currently an Assistant Professor with the University of Porto. He is also a Researcher with INESC Technology and Science, where his research interests include run-time reconfigurable systems, embedded systems in FPGAs, co-processor hardware acceleration, and tools for hardware/software codesign automation.



**JOÃO CANAS FERREIRA** (Senior Member, IEEE) received the Licenciatura and Ph.D. degrees in electrical and computer engineering from the University of Porto, Portugal, in 1989 and 2001, respectively. Since then, he has been an Assistant Professor with the Faculty of Engineering, University of Porto, and a Senior Researcher with INESC TEC. His current research interests include dynamically reconfigurable systems, application-specific architectures for cognitive radio and sensor networks, and adaptive embedded systems. He is a member of ACM and Euromicro.

**JOÃO M. P. CARDOSO** (Senior Member, IEEE) received the D.Eng. degree from the University of Aveiro, Portugal, in 1993, and the M.Sc. and Ph.D. degrees in electrical and computer engineering from the Technical University in Lisbon (IST/UTL), Portugal, in 1997 and 2001, respectively. He is currently a Full Professor with the Department of Informatics Engineering, Faculty of Engineering, University of Porto, and a Senior Researcher with INESC TEC. Before, he was with IST/UTL, from 2006 to 2008, a Senior Researcher with INESC-ID, from 2001 to 2009, and with the University of Algarve, from 1993 to 2006. In 2001 and 2002, he worked for PACT XPP Technologies, Inc., Munich, Germany. His research interests include compilation techniques, domain-specific languages, reconfigurable computing, and application-specific architectures. He is a Senior Member of ACM.