FPGA-BASED K-MEANS CLUSTERING USING TREE-BASED DATA STRUCTURES

Felix Winterstein[*12]

[1]Ground Station Systems Division
European Space Agency
Robert-Bosch-Str. 5
64293 Darmstadt, Germany
f.winterstein12@imperial.ac.uk

Samuel Bayliss[2]*, George A. Constantinides*[2]

[2]Dept. of Electrical and Electronic Engineering
Imperial College London
South Kensington Campus
London, SW7 2AZ
g.constantinides@imperial.ac.uk

ABSTRACT

K-means clustering is a popular technique for partitioning a data set into subsets of similar features. Due to their simple control flow and inherent fine-grain parallelism, K-means algorithms are well suited for hardware implementations, such as on field programmable gate arrays (FPGAs), to accelerate the computationally intensive calculation. However, the available hardware resources in massively parallel implementations are easily exhausted for large problem sizes. This paper presents an FPGA implementation of an efficient variant of K-means clustering which prunes the search space using a binary kd-tree data structure to reduce the computational burden. Our implementation uses on-chip dynamic memory allocation to ensure efficient use of memory resources. We describe the trade-off between data-level parallelism and search space reduction at the expense of increased control overhead. A data-sensitive analysis shows that our approach requires up to five times fewer computational FPGA resources than a conventional massively parallel implementation for the same throughput constraint.

1. INTRODUCTION

Clustering is a technique for the unsupervised partitioning of a data set into subsets, each of which contains data points with similar features. Automatic partitioning is used in a wide range of applications, such as machine learning and data mining applications [1], radar target tracking [2], image colour quantisation [3], or the spectral clustering of multi- and hyperspectral images [4, 5].

A popular technique for finding clusters in a data set is K-means clustering. K-means algorithms partition the D-dimensional point set $X = \{x_j\}, j = 1, ..., N$ into clusters $\{S_i\}, i = 1, ..., K$, where K is provided as a parameter. The goal is to find the optimal partitioning which minimises the objective function given in (1) where μ_i is the geometric centre (centroid) of S_i.

$$J(\{S_i\}) = \sum_{i=1}^{K} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2 \quad (1)$$

Finding optimal solutions to this problem is NP-hard [6]. A popular heuristic version uses a simple iterative refinement scheme which, for every data point, computes the nearest cluster centre based on the smallest squared distance to it and then updates each cluster centre position according to the data points assigned to it. The simple control flow and inherent parallelism at the granularity of distance computations makes the algorithm suitable for dedicated hardware implementations. Consequently several field programmable gate array (FPGA) and very large scale integration (VLSI) implementations have been developed to accelerate the process. However, the speed-up gained by these implementations is limited by the amount of available hardware resources and real-time clustering remains challenging.

In contrast to massively parallel hardware implementations, sophisticated software implementations have been proposed which gain speed-up from search space reductions. A major contribution in this field has been made by Kanungo *et al.* [7] which is referred to as the *filtering* algorithm. The idea is to organise the data points in a multi-dimensional binary search tree, called a *kd-tree*, and to turn the centre update into a tree traversal. This allows to prune data point-centre pairs at an early stage in the process, so as to reduce the number of computations required. The authors show that the efficiency gained outweighs the drawback of a more complicated control flow. While the efficiency of this approach has been established for sequential software implementations, to the best of our knowledge, the algorithm remains unexplored for K-means clustering hardware implementations. In this work, we investigate the practicality of the *filtering* algorithm in the context of an FPGA implementation. We show that cycle-accurate synchronisation and the customisable distributed memory architecture makes FPGAs a suitable platform for this application. Fur-

[*]This work is sponsored by the European Space Agency under the Networking/Partnering Agreement No. 4000106443/12/D/JR

thermore, the fine-grained control of parallelism combined with reconfigurability allows us to scale the implementation to meet application throughput constraints. The contributions of this paper are:

- An FPGA application which performs a binary kd-tree traversal and requires only a fraction of point-centre examinations compared to a conventional implementation. We show how the implementation can efficiently exploit the distributed memory architecture in FPGAs.
- The use of on-chip dynamic memory allocation which allows us to allocate the average amount of memory required during runtime instead of statically pre-allocating the worst-case amount resulting in more than 70× reduction of on-chip memory resources.
- A data-sensitive analysis of the achieved throughput as well as an analysis of the impact of algorithmic modifications of the original filtering algorithm made to facilitate an FPGA implementation.
- A comparison of the FPGA-based filtering algorithm with a conventional massively parallel FPGA implementation of the K-means clustering algorithm. This comparison is based on the hardware resource consumption under a throughput constraint.

This paper is organised as follows. Section 2 describes the filtering algorithm and related work. Section 3 presents a brief analysis of the algorithm. Our FPGA implementation is described in Section 4. Section 5 presents an analysis of the hardware implementation and a comparison with a conventional implementation. Section 6 concludes the paper.

2. BACKGROUND

K-means clustering is based on the minimisation of an objective function which is given by the sum of squared distances from each data point to its nearest centre as shown in (1). In the version considered here, every centre is moved to the centroid of the points it is assigned to after a nearest centre has been identified for all data points. This process is repeated for several iterations either until the squared distance criterion converges or until a preset number of iterations is reached. The algorithm starts off with K initial centres randomly sampled from the point set. Given N data points and L iterations, $N \times K \times L$ distances in D-dimensional space need to be computed in the version we refer to as the *conventional* algorithm.

The *filtering* algorithm [7] aims to reduce the computational burden. To this end, the point set is recursively divided into two subsets. In each step, the axis-aligned bounding box of the subset is computed and splitting occurs along the longest dimension of the bounding box. This leads to the well-known binary kd-tree structure whose root node represents the bounding box of all data points and whose children

Algorithm 1 The filtering algorithm introduced in [7].

```
1:  function FILTER(kdNode u, CandidateSet Z)
2:      C ← u.C
3:      if u is leaf then
4:          z* ← closest point in Z to u.point
5:          z*.wgtCent ← z*.wgtCent + u.point
6:          z*.count ← z*.count + 1
7:      else
8:          z* ← closest point in Z to C's midpoint
9:          for all z ∈ Z\{z*} do
10:             if z.isFarther(z*, C) then Z ← Z\{z}
11:         end for
12:         if |Z| = 1 then
13:             z*.wgtCent ← z*.wgtCent + u.wgtCent
14:             z*.count ← z*.count + u.count
15:         else
16:             FILTER(u.left, Z)
17:             FILTER(u.right, Z)
18:         end if
19:     end if
20: end function
```

nodes represent recursively refined bounding boxes. Each tree node stores the bounding box (C) information as well as the number ($count$) and the vector sum of the associated points (the weighted centroid, $wgtCent$) which is used to update the cluster centres when each iteration completes. During clustering, the tree is traversed and a set of candidate centres Z is maintained and propagated down the tree. At each visited node, the closest candidate centre z^* to the mid point of the bounding box is found. Remaining candidates are pruned if no part of the bounding box is closer to them than the closest centre. The pruning greatly reduces the number of point-centre pair examinations (and thus distance computations) and subtrees can be pruned if only one candidate remains. As the point set does not change during clustering, the kd-tree needs to be built up only once and the additional overhead is amortised over all iterations.

Alg. 1 shows pseudo code for one iteration of the filtering algorithm. In light of the description of our hardware implementation in Section 4, we identify four kernels here: 1) Line 4 and 8 constitute a search of the nearest centre to either C's mid point or the weighted centroid of the node (which can be resource-shared). The closest centre search involves the computation of the Euclidean distances. 2) Lines 9-11 perform the pruning of ineligible candidate centres. 3) Lines 4-6 and 13-14 form the centre update if a leaf is reached or only one candidate remains. 4) The remaining lines wrap the three kernels above in the tree traversal control flow.

2.1. Related Work

Previous work proposing FPGA implementations for pixel clustering of hyperspectral images was presented by Leeser *et al.* [8]. Their approach trades clustering quality for hardware resource consumption by replacing the Euclidean distance norm with multiplier-less Manhattan and Max metrics. This trade-off is extended to bit width truncations on the input data in [5] where speed-ups of 200× over the software

implementation are reported. More recent work in [9] builds on the same framework and extends it by incorporating a hybrid fixed- and floating-point arithmetic architecture. Contrary to our work, these approaches are based on the conventional K-means algorithm and aim to gain speed-up by an increased amount of parallel hardware resources for distance computations and nearest centre search.

An example of a tree-based algorithm for data-mining applications other than clustering, implemented on a parallel architecture, is presented in [10] for regression tree processing on a multi-core processor. Chen *et al.* [11] present a VLSI implementation for K-means clustering which is notable in that it, in the spirit of our approach, recursively splits the data point set into two subspaces using conventional 2-means clustering. Logically, this creates a binary tree which is traversed in a breadth-first fashion and results in computational complexity proportional to $\log_2 K$. This approach, however, does not allow any pruning of candidate centres.

Saegusa *et al.* [3] present a simplified kd-tree-based FPGA implementation for K-means image clustering. The data structure stores the best candidate centre(s) at its leaf nodes and is looked up for each data point. The tree is built independently of the data points, *i.e.* the pixel space is subdivided into regular partitions which leads to *empty* pixels being recursively processed. The tree also needs to be rebuilt at the beginning of each iteration and centre sets are not pruned during tree traversal in the build phase, which are essential features of the original filtering algorithm.

3. ANALYSIS OF THE FILTERING ALGORITHM

The filtering algorithm can be divided into two phases: building the tree from the point set (pre-processing), and the iterated tree traversal and centre update. In order to obtain information about the computational complexity of both parts, we profiled the software implementation of the algorithm using synthetic input data. Here, we chose the number of Euclidean distance computations performed as our metric for computational complexity. Since the tree creation phase does not compute any distances but performs mainly dot product computations and comparisons, we introduce distance computation equivalents (DCEs) to obtain a unified metric for both parts which combines several operations that are computationally equivalent.

For the test case, we generated point sets of $N = 10000$ three-dimensional real-valued samples. The data points are distributed among 100 centres following a normal distribution with varying standard deviation σ, whereas the centre coordinates are uniformly distributed over the interval $[-1, 1]$. Finally, the data points are converted to 16bit fixed-point numbers. We start off with $K = 100$ initial centres sampled randomly from the data set and run the algorithm either until convergence of the objective function or until 30 iterations are reached. We note that the clustering output is

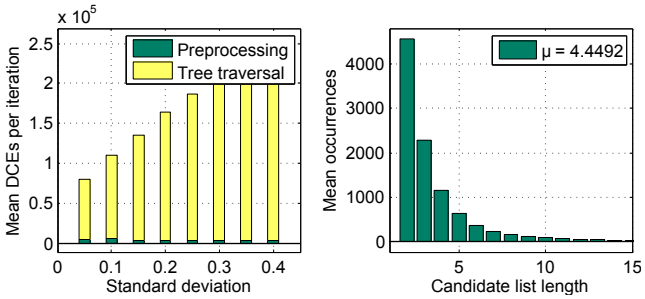

Fig. 1: Left: Contribution of the pre-processing to the overall computation performed. Right: Frequency of set sizes.

exactly the same as the one produced by the conventional algorithm which we implemented for comparison. Fig. 1 (left) shows the profiling results obtained from this test case for several standard deviations σ (ranging from well-distinguished clusters to a nearly unclustered point set). For all cases, the number of DCEs performed during tree creation is only a fraction of the total number of DCEs. We perform the pre-processing in software and the FPGA implementation described in the following section focuses on the tree traversal phase only. Fig. 1 shows the frequency of candidate centre set sizes averaged over all cases above. During tree processing, most sets contain only 2 or 3 centres which shows the effectiveness of the search space pruning.

4. FPGA IMPLEMENTATION

Three out of the four algorithm kernels described in Section 2 are data-flow centric: 1) The closest centre search computes Euclidean distances to either C's mid point or the node's weighted centroid, followed by a min-search. 2) The pruning kernel performs two slightly modified distance computations to decide whether any part of the bounding box C crosses the hyperplane bisecting the line between two centres z^* and z (See [7] for a more detailed description). Those centres z for which this is not the case are flagged and no longer considered by subsequent processing units. 3) When the centre update is performed on a leaf node or a one-element candidate set, it is a simple accumulation operation which updates a centre buffer. After a tree traversal is completed, the information contained in this buffer is used to update the centre positions memory before the next iteration starts. All three kernels are implemented in a pipelined, stream-based processing core. This core has a hardware latency of 31 clock cycles and can accept a node-candidate-centre pair on every other clock cycle.

The heart of the filtering algorithm is the traversal of the kd-tree which is built from the data points. During these traversals, the algorithm maintains and propagates a set of potential candidates for the nearest centre to a data point (or generally a subset of points). Ineligible candidates are discarded so that this set monotonically shrinks during traversal

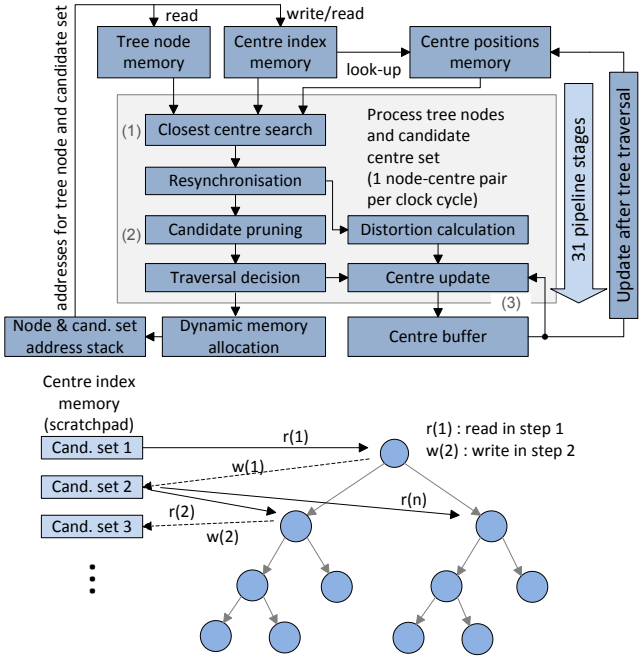

Fig. 2: Top: FPGA implementation of the filtering algorithm. Bottom: Read-write accesses to the centre index set memory during tree traversal.

towards the leaf nodes. This allows for efficient search space pruning, which reduces the overall number of necessary Euclidean distance calculations. We control the tree traversal using a stack. Pointers to a tree node and the associated set of candidate centres are fetched from the head of the stack and processed by kernels 1) and 2). At the output of the processing core, we obtain a new traversal decision. If we continue to traverse towards the leaf nodes, we push pointers to both left and right child and to the associated candidate centre set onto the stack. If a leaf is reached or only one candidate remains after pruning, kernel 3) performs the centre update. In this case, we do not push child nodes onto the stack and instead, a pointer to a non visited node further up in the tree will be fetched and processed. This process is repeated until the stack and pipeline are empty which terminates the tree traversal. Fig. 2 (top) shows a high-level block diagram of the FPGA application. Because all memories (tree nodes, centre indices, centre positions, centre buffer, and stacks) are mapped to physically disjoint memories, all accesses can be made simultaneously in each clock cycle.

4.1. Pipelining and Parallelisation

The decision whether to continue traversal toward the leaf nodes of the tree is available after a latency of at least 31 cycles. Delaying the next traversal step until this decision is made results in a classical control hazard which would leave the pipeline under-utilised. In order to overlap the processing of node-centre set pairs in the pipeline and to keep the computational units busy, as many as possible independent node-centre set pairs must be scheduled for processing. Two node-centre pairs are independent if they do not exhibit any read-write dependencies. Fig. 2 (bottom) illustrates that the read-write dependencies exist only between the candidate centre sets of a node and its immediate children: Processing the parent node writes the new (thinned out) candidate set into memory and it is read when processing the two child nodes. Thus, all nodes (and their associated centre sets) which do not have any direct parent-child relation can be processed independently of each other in the pipeline.

We implement this scheduling task using the stack: Every pointer to a node-centre set pair pushed onto the stack naturally points to children whose parent has previously been processed, and thus all items on stack point to disjoint subtrees and are independent. The scheduler in the stack management fetches a new node and centre set pointer from the stack as soon as the pipeline is ready to accept new data. Independent centre sets are read and written simultaneously using the dual-port memory available in the FPGA.

For parallelism beyond pipelining the processing units are duplicated. As we see in Fig. 1 (right), parallelising the centre processing is not efficient for more than 2 or 3 parallel units. Instead, we parallelise at the granularity of node-centre pairs. To process independent subsets of such pairs, we split the tree into subtrees and distribute them across several banks which can be accessed independently. The speed-up gained for P parallel pipelines is generally less than P since the subtrees do not contain an equal number of points.

4.2. Memory Usage and Dynamic Allocation

A tree node contains bounding box information, the weighted centroid and the number of associated points, as well as pointers to left and right child nodes. The centre index memory (Fig. 2, top) serves as a scratchpad memory for storing centre candidate sets (containing centre indices) and retaining them for later usage during the tree traversal. A new set is written when child nodes are pushed onto the stack and must be retained until both left and right child nodes have been processed. For example, candidate set 2 in Fig. 2, bottom, must be retained until step $n+1$. After step $n+1$, the space allocated for set 2 can be freed and reused. Therefore, allocating memory space dynamically can reduce the size of the centre index memory compared to allocating the cumulative worst-case size statically. To give an indication how crucial dynamic memory allocation is in this case, we consider an FPGA application supporting $N = 10000$ data points and a maximum of $K = 256$ centres. The worst-case number of candidate sets is $N - 1$ and is required in the case of a degenerated kd-tree where every internal node's right child is a leaf and its left child is another internal node. In this case, we would require $(N-1) \cdot K \cdot \log_2 K = 20$ Mbits of memory space which consumes 313 on-chip 36K-Block RAM resources in a Xilinx Virtex FPGA ($\sim$50% in a medium-size Virtex 6 FPGA). Obviously, statically allo-

Algorithm 2 Dynamic memory management for centre sets.

```
1:  if scratchpad write requested then
2:      if heap utilisation < bound B then
3:          newAddress ← allocateNextFreeAddress()
4:      else
5:          newAddress ← 0 (default full-size set)
6:      end if
7:  end if
8:  if scratchpad read requested then
9:      if readSecondTime(readAddress) then
10:         free(readAddress)
11: end if
```

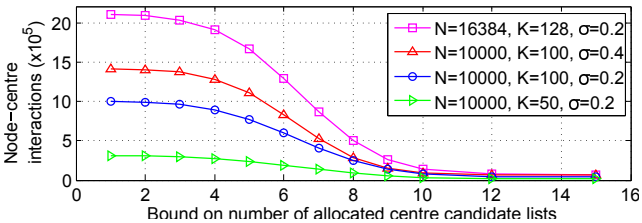

Fig. 3: Impact of the scratchpad memory bound B on the algorithm complexity (runtime).

cating this cumulative worst-case size is not a viable approach since it easily exceeds the available on-chip memory resources. In fact, our experiments show that, for this scenario and in average, memory space for 15 candidate sets is fully sufficient if dynamic allocation and de-allocation is used, which requires 1 Block RAM (BRAM) only.

Given this resource advantage, we implement a memory management unit for the centre index memory which dynamically allocates space and frees it once the candidate set has been read for the second time. The implementation of the fixed-size allocator uses a *free list* which keeps track of occupied memory space. However, even if dynamic memory allocation is used and the average-case memory space is much smaller than the worst-case, we need to be able to support the (unlikely) worst-case. The only worst-case bound we can guarantee from an algorithmic point of view is $N-1$ sets as described above and, of course, we want to limit the memory to a much smaller size. Our approach is to limit the memory to a size of $B \ll N-1$ sets and keep track of the current number of sets allocated during execution. If a new set needs to be allocated and the memory is full, this information cannot be stored and the allocator returns default address 0. The candidate set at this address is the full-size (default) set containing all K entries $0 \leq i \leq K-1$, which is never de-allocated. Alg. 2 describes the protocol.

It remains to determine a reasonable bound B. The functionality of our implementation is not compromised by this algorithmic modification, *i.e.* the clustering output is the same as before. The value of B, however, affects the runtime of the algorithm since the candidate set size is reset to K if B is exceeded. In order to get a notion of the runtime degradation, we ran tests with the same input data as

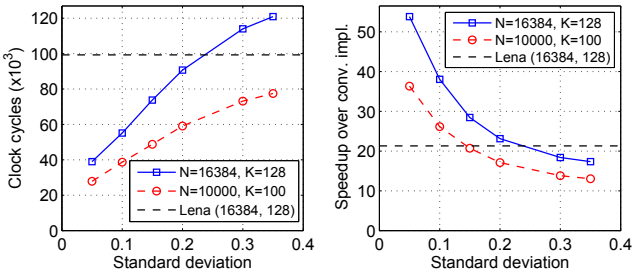

Fig. 4: Left: Average cycle count per iteration for the FPGA implementation of the filtering algorithm ($P = 1$). Right: Speed-up over an FPGA implementation of the conventional algorithm ($P = 1$ in both cases).

described in Section 3 and recorded the accumulated number of node-centre interactions per iteration as a measure for the architecture-independent algorithm complexity. Fig. 3 shows the profiling results for different values for N, K and standard deviation σ. The runtime of our implementation is practically not degraded for bounds $B \geq 15$. Our current implementation uses $B = 64$ which consumes 4 BRAMs.

5. PERFORMANCE AND RESOURCES

We generated synthetic test data as described in Section 3 for different values of N, K and σ. The hardware implementation produces the same clustering results as the software implementation. For comparisons, we developed an FPGA implementation of the conventional K-means algorithm performing $N \times K$ distance computations in each iteration as described in Section 2. The implementation is data-flow centric and does not use the tree structure, stack, or dynamic memory allocation. Both algorithms ran until convergence of the total squared-error distortion or until 30 iterations were reached. All throughput results given below are per iteration (average). Fig. 4 shows average clock cycles per iteration of the FPGA-based filtering algorithm (left) as well as the average speed-up over the FPGA implementation of the conventional algorithm (right). We also include a colour space clustering application of $N = 16384$ pixels sampled randomly from the *Lena* benchmark image.

We synthesised both our implementation of the filtering algorithm and the conventional implementation for a Xilinx Virtex 7 FPGA (7vx485t-1) for varying degrees of parallelism. The FPGA resource consumption for the different design points is given by the utilisation of look-up table (LUT), slice register (REG), digital signal processing slice (DSP48) and 36K-BRAM resources. All designs were synthesised for 200 MHz clock frequency and all results are taken from fully placed and routed designs meeting the timing constraint. The degree of parallelism in both implementations is given by the target throughput which is expressed as average execution time per iteration. Fig. 5 shows the *area-time* (AT) diagram, *i.e.* the amount of FPGA resources

Table 1: Resource comparison (Virtex7 7vx485t-1).

	Conv. $P = 82$	Filt. $P = 4$	Ratio
LUT	108937 (36%)	14167 (5%)	7.7×
REG	54652 (9%)	24486 (4%)	2.2×
DSP48	1062 (38%)	186 (7%)	5.7×
BRAM	101 (10%)	240 (23%)	0.4×

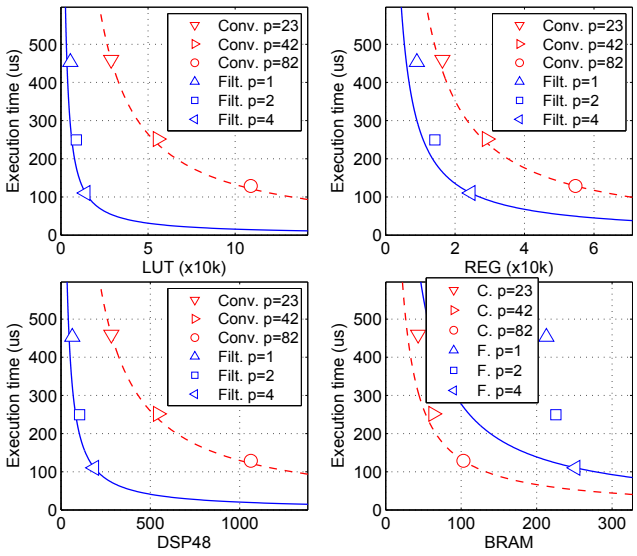

Fig. 5: Mean execution time per iteration over FPGA resources for $N = 16384$, $K = 128$, $D = 3$, $\sigma = 0.2$.

required to meet a target throughput, as well as the Pareto optimal frontiers (AT=const.) of the smallest AT-product. The runtime advantage of the filtering algorithm needs to be countered by significantly increased parallelism of computational units in the conventional implementation (23×-82×). Table 1 shows a resource comparison as well as the absolute and relative utilisation for a throughput constraint of 128µs per iteration. For DSP48, LUT and REG resources, the efficiency advantage of the filtering algorithm in hardware is obvious. However, the implementation of the filtering algorithm requires more memory compared to the conventional implementation. This is mainly due to the increased on-chip memory space required to store the data points in the kd-tree structure. The availability of BRAM resources is thus the limiting factor in scaling the current implementation to support larger data sets. For the configuration above and a maximum value of $K = 2048$, the 7vx485t-1 FPGA can support a maximum point set size of $N = 65536$.

6. CONCLUSION

This paper presents an FPGA implementation of the computationally efficient filtering algorithm for K-means clustering in order to investigate the practicality of the algorithm in the context of an FPGA implementation. We evaluate the effect of algorithmic modifications necessary for an efficient hardware implementation. We show how the kd-tree processing can be pipelined and parallelised using multiple banks of distributed on-chip memory and how dynamic memory allocation can greatly reduce the on-chip scratch-pad memory size. The comparison with an FPGA implementation of the conventional K-means algorithm performing $N \times K$ distance computations in each iteration shows that the FPGA-based filtering algorithm achieves the same throughput with five times fewer DSP48 resources and seven times fewer LUTs, the types of resources which are exhausted first in the conventional implementation. In future work, we will store the tree structure containing larger data sets in off-chip memory and design an application-specific prefetching and cache interface to alleviate the performance degradation due to the drop of memory bandwidth.

7. REFERENCES

[1] A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognition Lett.*, vol. 31, no. 8, pp. 651–666, June 2010.

[2] D. Clark and J. Bell, "Multi-target state estimation and track continuity for the particle PHD filter," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 43, no. 4, pp. 1441–1453, Oct. 2007.

[3] T. Saegusa and T. Maruyama, "An FPGA implementation of real-time K-means clustering for color images," *Real-Time Image Processing*, vol. 2, no. 4, pp. 309–318, Nov. 2007.

[4] J. P. Theiler and G. Gisler, "Contiguity-enhanced k-means clustering algorithm for unsupervised multispectral image segmentation," in *Proc. SPIE 3159*, 1997, pp. 108–118.

[5] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transformations in the implementation of k- means clustering on reconfigurable hardware," in *Proc. Int. Symp. on Field Programmable Gate Arrays*, 2001, pp. 103–110.

[6] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay, "Clustering large graphs via the singular value decomposition," *Mach. Learn.*, vol. 56, no. 1-3, pp. 9–33, June 2004.

[7] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 24, no. 7, pp. 881–892, July 2002.

[8] M. Leeser, J. Theiler, M. Estlick, and J. Szymanski, "Design tradeoffs in a hardware implementation of the k-means clustering algorithm," in *Proc. IEEE Sensor Array and Multichannel Signal Process. Workshop*, 2000, pp. 520–524.

[9] X. Wang and M. Leeser, "K-means clustering for multispectral images using floating-point divide," in *Symp. on Field-Programmable Custom Comput. Mach.*, 2007, pp. 151–162.

[10] S. Tyree, K. Q. Weinberger, K. Agrawal, and J. Paykin, "Parallel boosted regression trees for web search ranking," in *Proc. Int. Conf. on World Wide Web*, 2011, pp. 387–396.

[11] T.-W. Chen and S.-Y. Chien, "Flexible Hardware Architecture of Hierarchical K-Means Clustering for Large Cluster Number," *IEEE Trans. VLSI Syst.*, vol. 19, no. 8, pp. 1336–1345, Aug. 2011.