

Chapter 2

High-Level Synthesis of Dynamic Data Structures

HLS promises significant shortening of the design cycle compared to a design entry at RTL. However, many HLS implementations require extensive code alterations to ensure synthesizability and to achieve latency, throughput and resource utilisation comparable to handwritten RTL designs. These are especially important for programs with ‘irregular control flow’ and ‘complicated data dependencies’. In this chapter, we describe these terms in detail and elaborate on their implications for efficient HLS. To this end, we present a case study comparing the implementations of two algorithms for a compute-intensive machine learning application (*K*-means clustering). Algorithmically, both implementations solve the same problem, but they differ significantly in their computational properties: the first is a data flow-centric, ‘regular’ implementation with simple control flow, whereas the second is based on a recursive traversal of a pointer-linked tree data structure and uses dynamic memory allocation. The latter application thus exhibits highly ‘irregular control flow’ and ‘complicated data dependencies’. Despite this irregularity, software implementations of this algorithm have been shown to be significantly faster than their data flow-centric counterparts because it effectively reduces the algorithmic complexity of the problem [1].

Our evaluation fits in the line of works that present designer’s experiences with HLS tools. For example, a broad selection of 12 state-of-the-art HLS tools, academic and commercial, is evaluated by Meeus et al. [2]. Their overview, attesting Vivado HLS excellent test results, targets FPGA as well as ASIC flows and is based on a large set of criteria grouped into language support, ease of use, QoR and the capability of a rapid design space exploration. The goal is to perform a broad comparison across different tools mainly using a *Sobel* edge detector [3] as a test case. Sarkar et al. [4] present a more refined designer’s experience with three HLS tools for ASICs using stream-based video processing applications. Their conclusion highlights the importance of fine-grained re-architecturing their test cases to optimise area and power consumption, and an evaluation by experienced users to obtain solid comparisons. BDTI present an explicit evaluation of AutoPilot (later renamed into Vivado HLS after the acquisition by Xilinx) [5]. Their evaluation uses video processing and stream-based wireless communications benchmarks, reporting QoR comparable with

manual RTL implementations. The evaluations above share the commonality that the chosen benchmark cases are data flow-centric stream-based applications with simple control flow. A recent survey in [6] compares three academic tools and one commercial HLS tool using and four data-flow centric benchmarks in addition to the CHStone [7] benchmark suite, which covers a broader spectrum of applications. Heap-manipulating code, however, is not included. In contrast to the above evaluations, with our pointer-based benchmark, we aim to operate the HLS flow on test cases outside its ‘comfort zone’.

The outcome of our case study is three-fold: Firstly, we can show that the performance result obtained for software implementations can be repeated with hand-optimised RTL implementations of both algorithms. This result is interesting in that irregular algorithms are often believed to be inefficient once mapped into hardware. Furthermore, it shows that the use of dynamic, pointer-linked data structures, which are central to the second algorithm, can result in very efficient FPGA applications if implemented well. Secondly, we repeat the case study with an HLS implementation using a state-of-the-art HLS tool and show that our previous result is reversed if the source code is not substantially altered prior to HLS. Thirdly, we analyse the efficiency with which the HLS tool maps specific program features into RTL and propose source-to-source transformations that improve the QoR of the irregular algorithm by a factor of eight in terms of latency, significantly narrowing the gap between HLS and hand-written RTL implementations. This chapter describes:

- An efficient RTL implementation of the irregular tree-based K -means clustering algorithm which preserves the algorithmic advantage over the conventional regular implementation. We show how the implementation can efficiently exploit the distributed memory architecture in FPGAs.
- A comparative case study using a data-flow centric clustering implementation and an implementation based on recursive traversal of a pointer-linked tree structure which incorporates data-dependent control flow. The case study comprises hand-written RTL and HLS implementations. Code transformations necessary to enable HLS of unsupported program features are highlighted.
- 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 a $57\times$ reduction of on-chip memory resources.
- An end-to-end QoR comparison between the automatically generated RTL code for both variants and both functionally equivalent, hand-written RTL implementations.
- An analysis of how efficiently specific program features are synthesised into RTL. We propose source-to-source transformations that improve QoR by a factor of eight in terms of latency.

The two algorithms for K -means clustering form the basis of our case study. Figure 2.1 shows our design flow. The initial C++ model is modified in order to include custom precision for operands of the basic arithmetic operations. From this model, we implement a hand-written RTL design written in VHDL (bottom branch, Sect. 2.3) and a C++-based HLS design (top branch). The HLS

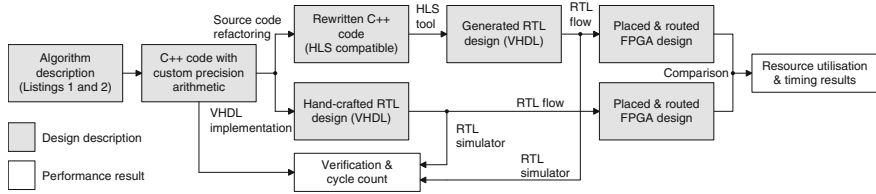


Fig. 2.1 Design flow of the case study

implementation requires further code refactoring which we discuss in Sect. 2.4. The generated and hand-crafted RTL design entries are verified using standard RTL simulation tools. Finally, QoR is compared in terms of latency and resource usage taken from the placed and routed FPGA designs (Sect. 2.5). The evaluation flow in Fig. 2.1 is repeated for both clustering algorithms. The following section discusses both algorithms.

2.1 Background

The test cases we chose for this case study are two implementations of a clustering application, a technique for unsupervised partitioning of a data set commonly used in a wide range of applications, such as machine learning and data mining [8, 9], radar tracking [10], image colour or spectrum quantisation [11–14]. A popular technique for finding clusters in a data set is K -means clustering, which partitions the D -dimensional point set $X = \{x_j\}$, $j = 1, \dots, N$ into clusters $\{S_i\}$, $i = 1, \dots, K$, where K is provided as a parameter. The goal is to find the optimal partitioning which minimises the total sum of squared Euclidean distances (squared-error distortion) given in (2.1) where μ_i is the geometric centre (centroid) of S_i .

$$J(\{S_i\}) = \sum_{i=1}^K \sum_{x_j \in S_i} \|x_j - \mu_i\|^2 \quad (2.1)$$

Finding optimum solutions to this problem is NP-hard [15]. A popular heuristic version uses an iterative refinement scheme. The standard algorithm begins by choosing K initial centres $Z = \{\mu_1, \dots, \mu_K\}$ sampled randomly from the point set. The set Z is iteratively refined until it no longer changes. On each iteration, it splits X into K partitions, according to which is the nearest mean of each partition. These means (geometrical centres) form the next generation of Z (Z'). Using one algorithm for this problem, which we refer to as *Lloyd's algorithm*, $N \cdot K \cdot L$ distances in D -dimensional space are computed where N is the number of data points and L , the number of required iterations. Listing 1 shows pseudo code of the main processing loop for one iteration of Lloyd's algorithm. Line 12 searches among K candidate

Listing 1 Main kernel of Lloyd’s algorithm (one clustering iteration).

```

1: Parameters:
2:  $N, K$ 
3: Input:
4: point set  $X = \{x_1, x_2, \dots, x_N\}$ 
5: initial centre set  $Z = \{\mu_1, \mu_2, \dots, \mu_K\}$ 
6: Output:
7: new centre set  $Z' = \{\mu'_1, \mu'_2, \dots, \mu'_K\}$ 
8: Variables:
9: centroid information  $C = \{c_1, c_2, \dots, c_K\}$ 

10: function LLOYDS
11:   for all  $x_j \in \{x_1, x_2, \dots, x_N\}$  do                                     ▷ iterate over all data points
12:      $i \leftarrow \operatorname{argmin}_{i' \in Z} (||x_j - \mu_{i'}||^2)$            ▷ find closest centre to  $x_j$  among  $K$  candidates
13:      $c_i \leftarrow$  select  $i$ th element in  $C$ 
14:      $c_i.wgtCent \leftarrow c_i.wgtCent + x_j$ 
15:      $c_i.count \leftarrow c_i.count + 1$ 
16:     update  $c_i$  in  $C$ 
17:   end for
18:   for all  $c_i \in C$  do                                                         ▷ update centre positions
19:      $\mu'_i \leftarrow c_i.wgtCent / c_i.count;$ 
20:   end for
21: end function

```

centres for the closest centre to a data point x_i . The index i of this centre is used to update the correct entry in the centroid information table C (Lines 13–16). C contains K vector sums of data points which we refer to as ‘weighted centroids’ ($wgtCent$). After all data points have been processed, the final output centre set $\{\mu'_1, \mu'_2, \dots, \mu'_K\}$ is computed from the weighted centroids in C (Lines 18–20).

In contrast to massively parallel hardware implementations, sophisticated software implementations have been proposed which gain speed-up from search space reductions. Kanungo et al. [1] present one possible implementation. Their *filtering algorithm* organises the data points in a multi-dimensional binary search tree, called a ‘kd-tree’, and finds nearest centres at each iteration using a tree traversal. 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 subdivided. This leads to a (generally not perfectly balanced) binary kd-tree structure whose root node represents the bounding box of all data points and whose children nodes represent recursively refined, non-empty disjoint bounding boxes. Each tree node stores the bounding box (*bndBox*) information as well as the number (*count*) and the vector sum of its associated points (the weighted centroid, *wgtCent*) which is used to update the cluster centres when each iteration completes. The weighted centroid of leaf nodes is the data point itself.

Listing 2 shows a simplified version of the recursive kernel function of the filtering algorithm for one iteration. During clustering, the tree is traversed starting from the root node. The set of input centres in Lloyd’s algorithm is replaced by sets of

candidates for the closest centre to a subset of data points. The algorithm propagates multiple candidate sets down the tree. These are of variable size and are created and disposed at run-time. At each non-terminal visited tree node, the closest candidate centre to the mid point (*midPoint*) of the bounding box is found. Some of the remaining candidates are pruned if no part of the bounding box is closer to them than the closest centre (Line 22). The pruning greatly reduces the number of computed distances since the average number of ‘close’ cluster-centre candidates is significantly smaller than K . Additionally, entire sub-trees 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. In fact, our profiling results show that, on average, the tree construction demands less than 2% of the total computation required. Therefore, we perform the pre-processing in software and the FPGA accelerator discussed in the following focuses only on the tree traversal phase.

In light of this case study, we identify the most important features of both applications. Because the min-search in Listing 1 (Line 12) is implemented as a `for`-loop over K centres, the main kernel of Lloyd’s algorithm consists of two nested `for`-loops with constant bounds. The simple control flow and inherent parallelism at the granularity of distance computations makes the computationally expensive algorithm suitable for hardware implementations so as to accelerate K -means clustering for real-time implementations if N and K are large. Computational parts of the filtering algorithm in Listing 2 are the closest centre searches (Lines 14, 20) and the candidate pruning (Line 22, containing two distance calculations), and the centroid buffer update. The loops in the min-searches and candidate pruning have variable bounds $2 \leq k \leq K$. The implementation uses dynamic memory allocation (Line 21) and de-allocation (Lines 32, 36) enclosed in data-dependent conditionals. Memory space is freed upon backward traversal, i.e. after an allocated centre set has been read twice. The implementation uses recursive function calls (beyond tail recursion) which requires the presence of a *stack*. The stack is implicitly handled in the software program, but it needs to be explicitly implemented in an FPGA application. The data passed between recursive instances are the tree node u and the set of candidate centre set Z .

Previous hardware implementations of Lloyd’s algorithm are proposed in [14, 16–19]. Pioneering work by Leiser et al. [16] implemented FPGA-clustering for the analysis of hyperspectral images. 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 by Estlick et al. [14] who report a speed-up of up to $200\times$ over the software implementation. More recent work in [17] builds on the same framework and extends it by incorporating a hybrid fixed- and floating-point arithmetic architecture. These approaches aim to gain acceleration from an increased amount of parallel hardware resources for distance computations and nearest centre search.

Listing 2 Main kernel of the filtering algorithm (one clustering iteration) [1].

```

1: Parameters:
2:  $N, K$ 
3: Input:
4: kd-tree
5: initial centre set  $\{\mu_1, \mu_2, \dots, \mu_K\}$ 
6: Output:
7: new centre set  $Z' = \{\mu'_1, \mu'_2, \dots, \mu'_K\}$ 
8: Variables:
9: node in the kd-tree  $u$ 
10: multiple sets of candidates for the closest centre to a point cloud ( $Z$ )
11: centroid information  $C = \{c_1, c_2, \dots, c_K\}$ 

12: function FILTER( $u, Z$ )
13:   if  $u$  is leaf then
14:      $i^* \leftarrow \operatorname{argmin}_{i', \mu_{i'} \in Z} (||u.wgtCent - \mu_{i'}||^2)$   $\triangleright$  find closest centre to  $u.wgtCent$ 
15:      $c_{i^*} \leftarrow$  select  $i^*$ -th element in  $C$ 
16:      $c_{i^*}.wgtCent \leftarrow c_{i^*}.wgtCent + u.wgtCent$ 
17:      $c_{i^*}.count \leftarrow c_{i^*}.count + 1$ 
18:     update  $c_{i^*}$  in  $C$ 
19:   else
20:      $i^* \leftarrow \operatorname{argmin}_{i', \mu_{i'} \in Z} (||u.midPoint - \mu_{i'}||^2)$   $\triangleright$  find closest centre to  $u.midPoint$ 
21:      $Z_{new} \leftarrow$  new centre set  $\triangleright$  allocate new centre set (empty)
22:     for all  $\mu_j \in Z$  do  $\triangleright$  prune candidate centres
23:       if pruningTest( $i^*, \mu_j, u.bndBox$ ) is false then
24:          $Z_{new} \leftarrow Z_{new} \cup \{\mu_j\};$   $\triangleright$  insert surviving candidates into  $Z_{new}$ 
25:       end if
26:     end for
27:     if  $|Z_{new}| = 1$  then
28:        $c_{i^*} \leftarrow$  select  $i^*$ -th element in  $C$ 
29:        $c_{i^*}.wgtCent \leftarrow c_{i^*}.wgtCent + u.wgtCent$ 
30:        $c_{i^*}.count \leftarrow c_{i^*}.count + u.count$ 
31:       update  $c_{i^*}$  in  $C$ 
32:       delete  $Z_{new}$   $\triangleright$  immediately delete allocated  $Z_{new}$ 
33:     else  $\triangleright$  recurse on children
34:       FILTER( $u.left, Z_{new}$ );
35:       FILTER( $u.right, Z_{new}$ );
36:       delete  $Z_{new}$   $\triangleright$  delete allocated  $Z_{new}$  on the way back
37:     end if
38:   end if
39: end function
40: for all  $c_i \in C$  do  $\triangleright$  update centre positions
41:    $\mu'_i \leftarrow c_i.wgtCent / c_i.count;$ 
42: end for

```

Contrary to these works, the first contribution in this thesis chapter is an efficient implementation of the filtering algorithm, which gains acceleration largely from search space pruning. Chen et al. [20] present a VLSI implementations for K -means clustering which is notable in that it, in line with 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. [12] present a simplified kd-tree-based implementation for K -means image clustering. The data structure stores the best candidate centre (or generally a few ‘best’ candidates) 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. Other disadvantages are that the tree needs to be rebuilt at the beginning of each iteration and that the centre lists are not pruned during tree traversal in the build phase, which are essential features of the filtering algorithm.

2.2 Analysis of the Filtering Algorithm

We analyse several properties of the filtering algorithm that provide insight into the advantage over Lloyd’s algorithm. To this end, we profile a software implementation of the algorithm. The input data sets that we use throughout this chapter are point sets of $N = 16,384$ three-dimensional real-valued samples. The data points are distributed among 128 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 choose $K = 128$ initial centres sampled randomly from the data set and run the algorithm either until convergence of the objective function or until a maximum of 30 iterations are reached. In addition to synthetic input data, we include a working set with $N = 16,384$ randomly sampled pixels from the well-known Lena benchmark image and quantise the colour space into $K = 128$ clusters. Note that the clustering output is exactly the same for both the implementation of Lloyd’s and the filtering algorithm.

The filtering algorithm can be divided into two phases: building the tree from the point set (pre-processing), and the repeated tree traversal and centre update (clustering phase). In order to obtain information about the computational complexity of both parts, we profile 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 which are computationally equivalent. Table 2.1 shows the profiling results of the computational complexity of the filtering algorithm broken down into clustering and pre-processing phases for different working sets. The parameter σ is varied such that the synthetic input data ranges from well-distinguished clusters ($\sigma = 0.05$) to a nearly unclustered point set ($\sigma = 0.35$). For all cases, the number of DCEs performed during tree creation is only a fraction of the total number of DCEs (2% geometric mean). Because of the small contribution of

Table 2.1 Computational complexity of the filtering algorithm broken down into clustering and pre-processing phases

| Input data $N = 16,384$, $K = 128$ | DCEs in clustering | DCEs in pre-processing | Contribution of pre-processing (%) |
|---|--------------------|---------------------------|---------------------------------------|
| Synthetic $\sigma = 0.05$ | 1,09,207 | 4963 | 4.3 |
| Synthetic $\sigma = 0.10$ | 1,56,464 | 4712 | 2.9 |
| Synthetic $\sigma = 0.15$ | 2,12,670 | 4574 | 2.1 |
| Synthetic $\sigma = 0.20$ | 2,59,146 | 4494 | 1.7 |
| Synthetic $\sigma = 0.25$ | 2,94,173 | 4423 | 1.5 |
| Synthetic $\sigma = 0.30$ | 3,21,841 | 4432 | 1.4 |
| Synthetic $\sigma = 0.35$ | 3,39,831 | 4424 | 1.3 |
| Lena benchmark (subset) | 2,24,418 | 4923 | 2.1 |

the pre-processing, we perform this part in software and the FPGA implementation described in the following section focuses on the tree traversal phase only.

We also evaluate the search space pruning. The major complexity reduction is due to the fact that the repeated searches for the closest centre need to consider significantly fewer centres than Lloyd’s algorithm for which this number is always K . Figure 2.2 (left) shows the frequency of candidate centre set sizes averaged over all synthetic cases above. During tree processing, most sets contain only 2 or 3 centres and the average centre set size is 4.36 (3.78 for the Lena image benchmark), which shows the effectiveness of the search space pruning. We quantify the overall search complexity of the filtering algorithm in terms of the aggregate number of *node-centre pairs*, i.e. the cumulative number of candidate centres processed at the visited tree nodes per clustering iteration. This number is sensitive to the input data.

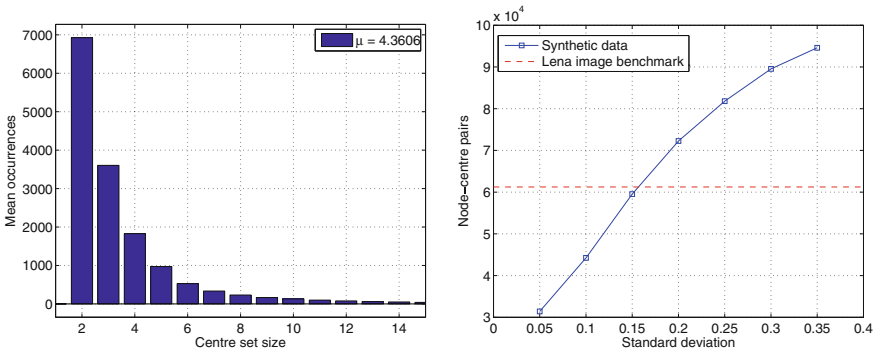


Fig. 2.2 Left Frequency of candidate centre set sizes for synthetic input data. Right Computational complexity of the filtering algorithm in terms of node-centre pairs (Lloyd’s algorithm has a constant complexity of 209.7×10^4 point-centre pairs for this data set)

Figure 2.2 (right) shows the number of node-centre pairs over different values of σ in the synthetic data sets. The complexity ranges from 31,399 to 94,590. We also include the Lena benchmark with 61,230 node-centre pairs for a comparison with real-world data. For Lloyd’s algorithm, an equivalent metric of data point-centre pairs can be defined which is $N \cdot K = 20,97,152$ for all input sets in Fig. 2.2. Even for unfavourable input data ($\sigma = 0.35$), the filtering algorithm thus achieves a $22\times$ reduction of search complexity. In a sequential software implementation [1], this reduction translates directly into a run-time advantage of the filtering algorithm. The next sections investigate if, how, and to what extent this result can be reproduced in hardware implementations.

2.3 RTL Implementations

This section describes efficient hand-crafted FPGA implementations of Lloyd’s and Kanungo’s filtering algorithm implementations, which will be compared in Sect. 2.5.1. Both RTL implementations are fully pipelined designs and their computational parts mainly consist of the same basic elements, Euclidean distance and dot product computations, but their control structures and memory architectures are substantially different. We made the source code of the RTL implementations discussed below available in an open source repository.¹ The following description motivates later discussion of how we direct the HLS flow to produce competitive designs from a C description. Specific features discussed here and implemented later in the HLS flow (Sect. 2.4) will disclose particular limitations.

2.3.1 Lloyd’s Algorithm

The implementation consecutively fetches data points from memory, computes the Euclidean distance to each centre μ_i , $1 \leq i \leq K$, and selects the closest centre before fetching the next data point. The distance computation is fully parallelised for a parametric data point dimensionality D . Parallelism is further increased by performing P distance computations concurrently which reduces the number of sequential steps per iteration from $N \cdot K$ to $(N \cdot K)/P$. A centroid buffer stores the centroid information C and maintains the intermediate results during one iteration which are continuously updated. The accumulated weighted centroids (*wgtCent*) are then divided by the *count* value at each index to obtain the centre positions for the next iteration. The data set memory and centroid buffer are implemented as on-chip block random access memory (BRAM) and distributed look-up table (LUT) RAM, respectively. The position update uses a pipelined divider core.

¹<https://github.com/FelixWinterstein/Vivado-KMeans> [21].

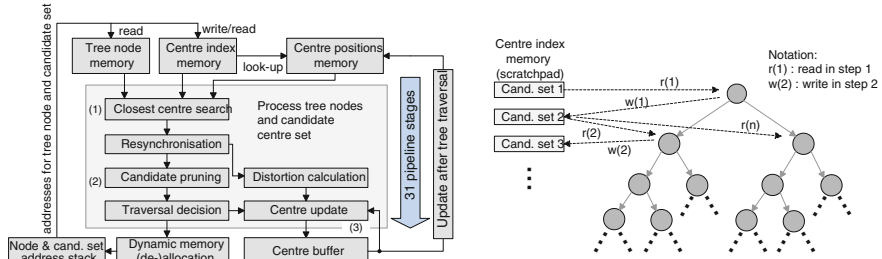


Fig. 2.3 *Left* FPGA implementation of the filtering algorithm. *Right* Read-write accesses to the scratchpad memory for centre sets during tree traversal

2.3.2 Filtering Algorithm

Figure 2.3 (left) shows a high-level block diagram of our RTL design of the filtering algorithm. Our RTL implementation contains three computational kernels: (1) The closest centre search computes Euclidean distances to either the mid point of a bounding box or the tree 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 crosses the hyperplane bisecting the line between two centres. A more detailed description of the pruning algorithm is given in [1]. Those centres μ_i for which the pruning test returns false are flagged and no longer considered by subsequent processing units. (3) The centroid buffer is updated and used in the same way as for Lloyd’s algorithm. All three sub-kernels are integrated in a pipelined, stream-based processing core. This core has a hardware latency of 31 clock cycles and can accept a node-centre pair on every other clock cycle. Thus, if fully utilised, the pipeline is usually filled with several tree nodes and their associated candidate centre sets.

The heart of the filtering algorithm is the traversal of the kd-tree which is implemented using the recursive calls shown in Listing 2. Our implementation controls this tree traversal using a stack which contains pointers to a tree node and to its associated set of candidate centres as well as the current set size. After fetching the pointers from stack, the data referenced by them is processed. At the output of the pipeline, we obtain a new traversal decision which is based on whether we have not yet reached a leaf node and whether there is more than one centre in the pruned candidate set left. If so, new pointers (left and right child and a new centre set) and the new set size are pushed onto the stack. Otherwise, nothing is pushed onto the stack. In the latter case, a pointer to a non-visited node further up in the tree will be fetched for processing in the next cycle. This process is repeated until the stack and pipeline are empty which terminates the tree traversal. Because all memories (tree nodes, centre indices, centre positions, centroid buffer, and stacks) are mapped to physically disjoint memories, all accesses can be made simultaneously in each clock cycle.

Pipelining and Parallelisation

The profiling results in Sect. 2.2 show that a candidate set (associated with a tree node and processed item by item) has an average size of 4.36 centres in the scenarios considered here, which is smaller than the pipeline depth of 31 stages. In order to hide pipeline latency, we need to overlap the processing of multiple node-centre set pairs in the pipeline, which is possible in the absence of feedback dependencies. Figure 2.3 (right) illustrates the read and write accesses. Memory accesses are indicated by dashed lines, pointer links are drawn as solid lines. The diagram shows that a read-write data dependency exists only between centre sets whose associated tree nodes have a direct parent-child relation. In fact, all pointers residing on the stack point to data structures that has already been written to and hence can be processed independently. The scheduler in the stack management fetches new pointers as described above as soon as the pipeline is ready to accept new data. Independent centre sets are read and written simultaneously using dual-port memory. For parallelism beyond pipelining the processing units are duplicated. To process independent subsets of such pairs, we split the tree into P disjoint sub-trees and distribute them across several computational units for parallel processing. We note that for both pipelining and parallelisation, we exploit knowledge about dependencies carried by data structures accessed through pointers.

Dynamic Memory Allocation

The centre index memory (Fig. 2.3, left) serves as a *scratchpad* memory for storing centre sets 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. The memory space then can be freed and reused. The duration for which a centre set must be retained in memory depends on the shape of the (generally unbalanced) tree. The results in Sect. 2.2 are obtained under the assumption that the application can allocate as much scratchpad memory as needed. However, the requested amount may exceed the available on-chip memory resources. The worst-case number of candidate sets is $N - 1$ which is required in the case of a degenerate kd-tree where every internal node's right child is a leaf and its left child is another internal node. If we consider an FPGA application supporting $N_{max} = 16,384$ data points and a maximum of $K_{max} = 256$ centres, we require $(N_{max} - 1) \cdot K_{max} \cdot \log_2 K_{max} \approx 33.6$ Mbits worst-case memory space which consumes 912 on-chip 36k-BRAM resources ($\sim 89\%$ in a medium-size Virtex 7 FPGA) and does not leave enough resources for the other memories in the implementation. However, in the average case, the tree is unlikely to be degenerate as described above and therefore the lifetime of a centre set is much shorter and the instantaneous memory requirement is significantly lower.

As a result of this resource advantage, we implement a memory management unit which dynamically allocates space and frees it once the candidate set has been read for the second time, rather than a static allocation. The implementation of the fixed-size allocator uses a *free-list* that keeps track of occupied memory space. In our implementation, the scratchpad memory and free-list are sized to accommodate an

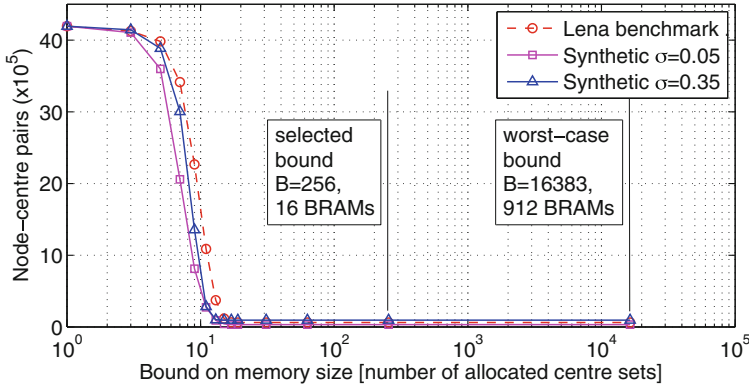


Fig. 2.4 Trade-off between heap size and run-time of the filtering algorithm (profiling)

‘average-case’ number of centre-candidate sets. Our approach is to limit the memory to a size of $B \ll N - 1$ sets. When inadequate memory is available to service an allocation request, the algorithm allows us to abandon the pruning approach and instead consider all candidate centres. This modification does not compromise the functionality of the algorithm, but it increases its run-time (the number of node-centre interactions). Figure 2.4 shows the result of profiling the software implementation clustering $N = 16,384$ pixels (RGB vectors) sampled from the Lena image benchmark and the two extreme cases for synthetic data in Table 2.1. If we allow the algorithm to allocate memory for only a single centre, the search complexity degrades to the worst case of $(2 \cdot N - 1) \cdot K$ node-centre pairs to be examined. The search complexity, however, greatly decreases for $B > 10$ in all test scenarios. We select a bound of $B = 256$ centre sets (16 36k-BRAMs) which practically causes no run-time degradation in the scenarios considered in this case study.

The next section describes the re-implementations of both algorithms using a C-based HLS tool, which finally allows us to compare the FPGA resource usage and speed of all four designs.

2.4 HLS Implementations

We choose Vivado HLS for this case study as an exemplary state-of-the-art tool which shares many similarities with other modern C-to-FPGA flows such as LegUp [22], ROCCC [23], Dwarv [24] and GAUT [25]. RTL generation is guided by synthesis directives which are manually invoked and configured. Exploring design options and optimisations using directives ideally does not require the source code to be altered. The most important directives we use to control the RTL generation are loop pipelining and loop unrolling directives. Loop pipelining overlaps loop iterations in the pipeline. The interval between the start of two iterations is given by the initiation

interval (II). Loop unrolling is used to force parallel instantiations of the loop body. In order to remove the bottleneck of an insufficient number of memory ports in a parallelised application, on-chip memories can be split into multiple banks using an *array partitioning* directive. As for LegUp, ROCCC, Dwarv and GAUT, the C-based input is restricted to a synthesisable subset. Vivado HLS allows pointers to be used as references to statically allocated arrays. However, it does not synthesise dynamic memory allocation (`new`, `delete`) and heap memory. In this thesis, we refer to pointer variables which obtain their value from a call to the `new` function as *heap-directed* pointers. Other disallowed features are system calls, arbitrary pointer casting and arbitrary recursive functions.

Our goal is to bring the generated RTL designs produced by the HLS flow as close as possible to the highly optimised manual RTL designs in the previous section. We distinguish between optimisations using synthesis directives and manual source code modifications.

2.4.1 *Lloyd's Algorithm*

The C code for Lloyd's algorithm corresponding to Listing 1 is directly synthesisable and does not contain any unsupported language features. We unroll all `for`-loops over the three dimensions of the input data points which results in a parallel implementation of the distance computation $||x_j - \mu_{i'}||^2$. Most of the computation is contained within the inner `for`-loop which implements the min-search in Line 12 (bound K). Pipelining this loop ($II=1$) leads to performance comparable to hand-coded RTL. For acceleration beyond pipelining, we control the degree of parallelism just as in the case of the manual RTL design by partially unrolling the outer loop to degree P (replicating pipelines). In order to match the parallelism of computational units and memory ports, we partition the centre positions and centroid buffer arrays into P banks using the array partitioning directive. Overall, using synthesis directives and a minor source code modification to ensure correct indexing of the parallel instances of the centroid buffer, we are able to produce an RTL design which is architecturally similar to its hand-written counterpart.

2.4.2 *Filtering Algorithm*

The synthesisability of the main kernel as in Listing 2 requires the removal of the recursive function calls and the calls to `new` (Line 21) and `delete` (Lines 32, 36), and code transformations to improve QoR of the synthesis of the pointer-linked data structures and the circuits operating on these.

Listing 3 Iterative replacement for the recursive kernel in Listing 2.

```

1: push to stack (root, { $\mu_1, \mu_2, \dots, \mu_K$ }, true);
2: while stack not empty do
3:   u, Z, d  $\leftarrow$  fetch from head of stack
4:   if (d is true) then
5:     delete Z
6:   end if
7:   Znew  $\leftarrow$  new centre set
8:   ... ▷ original body in Listing 2 (contains two variable-bound sub-loops)
9:   if (u is not a leaf) and ( $|Z_{new}| > 1$ ) then
10:    push to stack (u.right, Znew, true)
11:    push to stack (u.left, Znew, false)
12:   else
13:     delete Znew
14:     ... ▷ update centroid buffer
15:   end if
16: end while

```

Recursive Tree Traversal

Recursion is replaced by a `while`-loop and a stack data structure. As in the RTL implementation, our C-based HLS design now contains three heap-allocated data structures: the pointer-linked kd-tree, the pool of centre sets and the stack. The program accesses these data structures through pointers. The stack contains the pointers to a heap-allocated tree node *u* and a set of candidate centres *Z* (and its size), as well as a flag *d* indicating that the centre set can be de-allocated. Listing 3 shows the rewritten code that avoids recursion.

Dynamic Memory Allocation

We replace the basic C++ routines for dynamic memory allocation to ensure synthesizability by off-the-shelf HLS tools. Occurrences of `new` and `delete` statements are replaced by calls to custom allocator functions that we provide in an additional header file. The implementation of the fixed-size allocator is in Line with Sect. 2.3.2. Heap memory is replaced by arrays that are mapped to on-chip memory. We translate pointer dereferencing into array indexing and instantiate an array for each data structure type. We choose the same heap sizes as in the RTL implementation. The memory for centre sets is limited to the same bound *B* as selected in Fig. 2.4. We implement the same fall-back solution when inadequate memory is available to service an allocation request as described in Sect. 2.3.2.

Parallelisation

As in the manual RTL design, we split the tree structure into *P* independent sub-trees to parallelise the application by instantiating *P* parallel processing kernels. Heap memories for tree nodes and centre set memory are by default monolithic memory spaces which need to be divided into *P* disjoint regions (sub-trees, and segments for private centre sets). The access through (dynamically allocated) pointers, however, hides this disjointness information, which renders the array partitioning directive

Listing 4 Loop distribution to enable pipelining.

```

1: while stack not empty do
2:   while (stack not empty) and (queue not full) do
3:     u, Z, d  $\leftarrow$  fetch from head of stack
4:     enqueue(u, Z, d) in queue ▷ newly introduced queue
5:   end while
6:   while queue not empty do
7:     u, Z, d  $\leftarrow$  dequeue from queue
8:     ... ▷ remaining loop body in Listing 3 (Lines 4–15)
9:   end while
10: end while

```

ineffective and does not lead to parallel execution. In fact, applying automatic partitioning through HLS directives even leads to a degradation in latency as we show in the performance comparison in Sect. 2.5. Instead, we manually partition the tree node memory and privatise heap space for centre sets for each instance. This ensures that the scheduler of the HLS tool recognises the parallelisation opportunity. Automating this step requires a program analysis capable of identifying disjoint regions (in terms of access patterns) in the monolithic heap memory space.

Inter-Iteration Dependencies and Pipelining

Apart from replication, acceleration of the manual RTL design is obtained from pipelining the tree traversal. This corresponds to pipelining the loop nest in Listing 3 which must take two (potential) inter-iteration dependencies into account. The first occurs between fetching pointers to data from the stack and pushing new pointers onto the stack, which hinders pipelining. However, because there are two push statements and one fetch statement, the items stored on the stack (pointers *u* and *Z*, *d*) accumulate if the condition in Line 9 holds in several iterations. Once there are multiple pointers on the stack, these do not cause any read-write dependencies between iterations and hence can be overlapped in pipelined execution. Listing 4 shows a transformation of the loop in Listing 3 to implement this schedule. The transformation distributes the execution of the original loop body over two (pipelineable) inner loops which exchange data via a newly inserted queue. The second inner loop ensures that multiple items on stack will be immediately scheduled for processing. However, this loop still contains sub-loops with variable bounds which prevents the tool from pipelining it. An additional manual loop nest flattening transformation is required to enable pipelining the loop with $II=1$. Because of the variable bounds of the inner loops, this loop nest is not a perfectly or semi-perfectly nested loop, which prevents the application of Vivados loop flattening directive. Without loop flattening, only the inner loops can be pipelined, which would result in less speed-up compared to the manually flattened loop.

The other (potential) inter-iteration dependency is due to the pointer references to *Z* and *Z_{new}* in Listing 3. This is a false dependency because, after the loop transformation, the pointers to *Z* and *Z_{new}* never alias across iterations. Inserting a ‘dependence false’ directive makes Vivado HLS aware of the non-existence of this dependency.

Enabling automatic pipelining for pointer-based programs thus crucially depends on an automated analysis capturing the semantics of `new` and `delete` and reasoning about such ‘pointer-carried’ dependencies which we will explore in Chap. 4.

2.5 Performance Comparison

We evaluate the four implementations (RTL and HLS designs for both algorithms) based on their execution time (latency) and resource consumption. For a latency comparison, we ran simulations on the synthetic data described in Sect. 2.2 for different values of σ . All hardware implementations produce the same clustering result as a software implementation that we implemented for validation. The algorithms ran until convergence or until 30 iterations were reached. All latency results below are per clustering iteration (average). This section begins with a comparison of the two RTL implementations. The latter part of the section then shows how close our manually optimised HLS designs can get to these results.

2.5.1 RTL Designs

Figure 2.5 shows the average number of clock cycles per iteration of the FPGA-based filtering algorithm (left) as well as the average speed-up over the FPGA implementation of Lloyd’s algorithm (right). We synthesise both RTL implementation of the filtering algorithm and Lloyd’s algorithm for a Xilinx Virtex 7 FPGA (7vx485tffg-2) for varying degrees of parallelism. We use Xilinx Vivado 2014.4 for netlist

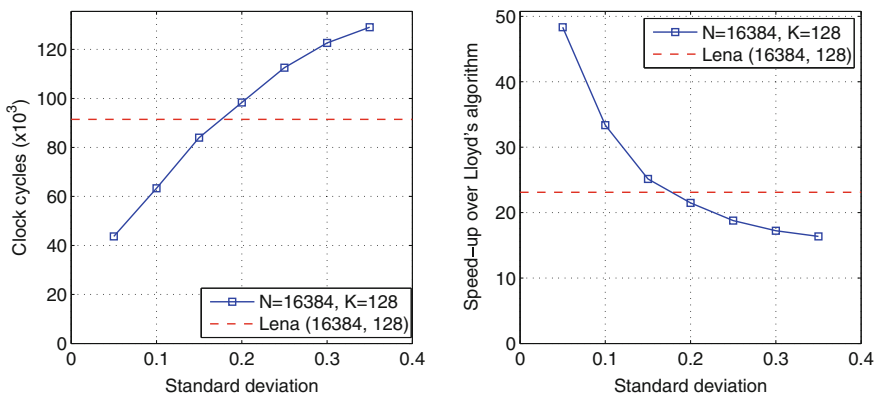


Fig. 2.5 *Left* Average cycle count per iteration for the manual RTL implementation of the filtering algorithm ($P = 1$). *Right* Speed-up over an RTL implementation of Lloyd’s algorithm ($P = 1$ in both cases)

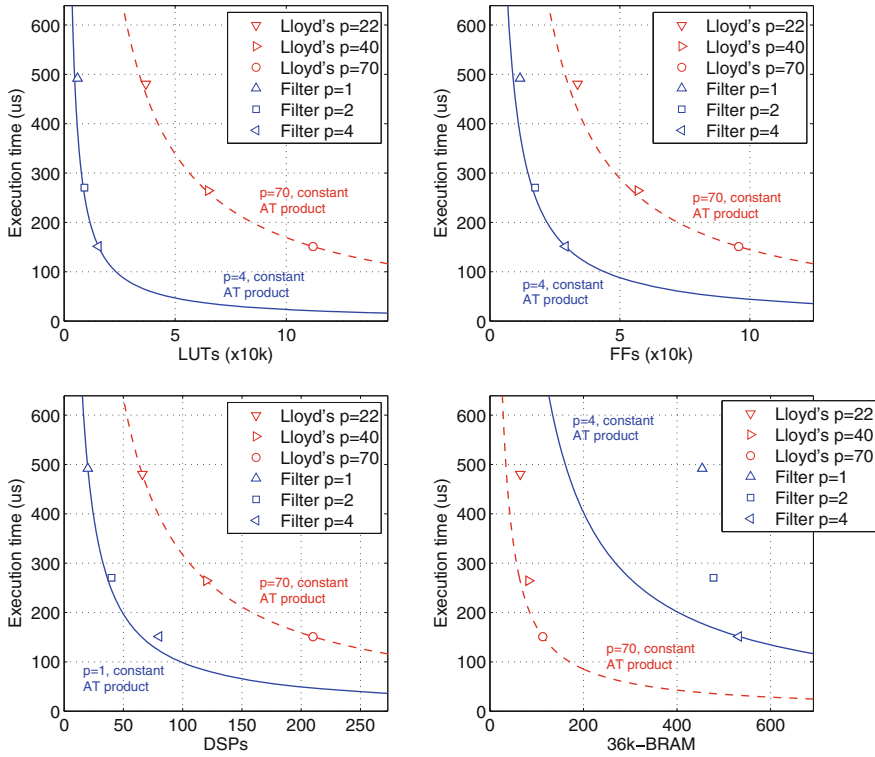


Fig. 2.6 Mean execution time per iteration over FPGA resources for $N = 16384$, $K = 128$, $\sigma = 0.2$ (Xilinx Virtex7 7vx485tffg-2)

synthesis, placement and routing. We report the FPGA resource consumption for the different design points in terms of look-up tables (LUTs), flip-flops (FFs), FPGA slices (containing four LUTs and eight FFs), digital signal processing slices (DSPs) and 36k-BRAM resources. All designs are synthesised for 200MHz target clock frequency and all results are taken from fully placed and routed designs meeting the timing constraint. For the resource comparison of both implementations, we select the performance point in Fig. 2.5 with $\sigma = 0.2$, which lies amid the range of execution times and is close to the performance measured for the Lena benchmark. The degree of parallelism we choose in both implementations is given by the target latency which is expressed as average execution time per iteration. Figure 2.6 shows the area-time (AT) diagram, i.e. the amount of FPGA resources required to meet a target throughput. For ease of comparison of the two algorithms, we draw an area-time frontier with a constant AT product through the design points with the smallest AT product for each algorithm (solid blue and dashed red line; note that only the intersections of these lines with the data points are feasible designs). The inherent run-time advantage of the filtering algorithm needs to be countered by significantly increased parallelism of computational units in the implementation of Lloyd's

Table 2.2 Resource comparison for a 270 μ s-latency constraint (input parameters: $N = 16,384$, $K = 128$, $\sigma = 0.2$)

| P : parallelisation degree, R : resource overhead for LLoyd’s algorithm | | | |
|---|-------------------------------|--------------------------------|-------------|
| | Lloyd’s algorithm $P = 40$ | Filtering algorithm $P = 2$ | R |
| LUT | 64,922 (21.4 %) | 9148 (3.0 %) | $7.3\times$ |
| FF | 56,975 (9.4 %) | 17397 (2.9 %) | $3.3\times$ |
| Slices | 19,843 (26.1 %) | 4915 (6.5 %) | $4.0\times$ |
| DSP | 120 (4.3 %) | 40 (1.4 %) | $3.0\times$ |
| 36k-BRAM | 83 (8.1 %) | 478 (46.4 %) | $0.2\times$ |

algorithm ($22\times$ - $70\times$). Table 2.2 shows a resource comparison as well as the absolute and relative utilisation for a fixed latency constraint of 270 μ s (which corresponds to the latency achieved by the filtering algorithm with $P = 2$).

For DSP, LUT and FF resources, the efficiency advantage of the filtering algorithm in hardware is obvious. We also note that the data set used here is relatively unclustered ($\sigma = 0.2$) and the performance advantage will be greater for values $\sigma < 0.2$ as shown in Fig. 2.5. However, our implementation of the filtering algorithm requires more memory compared to Lloyd’s algorithm. This is mainly due to the increased memory space required to store the data points in the kd-tree structure. We can conclude that the availability of on-chip BRAM resources is the limiting factor in scaling this algorithm through increased parallelism, but the advantage of its RTL implementation in terms of computational resources is compelling.

2.5.2 HLS Designs

We compare the performance of both HLS to both RTL designs based on different metrics: clock cycles count per iteration (through RTL simulations), execution time per iteration (includes the clock period), resource usage and AT product (in logic slices \times ms). We implement the HLS designs with Xilinx Vivado HLS 2014.4. As in the previous section, all designs are synthesised for a 200 MHz target clock rate and all results are taken from fully placed and routed designs (not all designs meet the timing constraint in which case we report the best achievable clock period). The input data set to all implementations is the same data set as used above ($\sigma = 0.2$). In order to account for the inherent runtime advantage of the filtering algorithm due to search space pruning and to compare all four designs on a common basis, we increase the parallelisation degree for the final implementations of Lloyd’s algorithm to $P = 40$, which equalises the cycle count of the hand-written RTL designs.

Table 2.3 shows the performance comparison based on the metrics above. The resource consumption of both HLS designs compared to their RTL counterparts is remarkably similar. The utilisation of flip flops is notable in that it is substantially

Table 2.3 Performance comparison using the hand-written RTL designs as reference

| Architecture: $N_{max} = 32, 768, K_{max} = 256, B = 256$; input data (synthetic): $N = 16,384, K = 128, \sigma = 0.2$ | | | | | | | | | |
|---|-------------------|------------------------------|-----------------|---------------------|--------------------------------|--------------------------------|----------------------------------|--|--|
| | Lloyd's algorithm | | | Filtering algorithm | | | | | |
| | RTL (reference) | HLS | RTL (reference) | HLS (reference) | HLS (directives only) | HLS (manual partitioning) | HLS (manual loop transformation) | | |
| P | 40 | 40 | 2 | 2 | 2 | 2 | 2 | | |
| Slices | 19,843 | 22,711 ($\times 1.1$) | 6950 | | 5263 ($\times 0.8$) | 5161 ($\times 0.7$) | 6540 ($\times 0.9$) | | |
| LUT | 64,922 | 68,484 ($\times 1.1$) | 10,418 | | 12,865 ($\times 1.2$) | 12,717 ($\times 1.2$) | 15046 ($\times 1.4$) | | |
| FF | 56,975 | 47,895 ($\times 0.8$) | 19,008 | | 11,517 ($\times 0.6$) | 11,293 ($\times 0.6$) | 13612 ($\times 0.7$) | | |
| DSP | 120 | 120 ($\times 1.0$) | 40 | | 36 ($\times 0.9$) | 36 ($\times 0.9$) | 36 ($\times 0.9$) | | |
| 36 k-BRAM | 83 | 75 ($\times 0.9$) | 448 | | 506 ($\times 1.1$) | 506 ($\times 1.1$) | 507 ($\times 1.1$) | | |
| Clock period | 5.0 ns | 8.4 ns ($\times 1.7$) | 5.0 ns | | 5.0 ns ($\times 1.0$) | 5.0 ns ($\times 1.0$) | 5.5 ns ($\times 1.1$) | | |
| Cycles/iteration | 53 k | 66 k ($\times 1.2$) | 54 k | | 1440 k ($\times 26.6$) | 583 k ($\times 10.8$) | 165 k ($\times 3.0$) | | |
| Time/iteration | 264 μ s | 555 μ s ($\times 2.2$) | 270 μ s | | 7200 μ s ($\times 26.6$) | 2915 μ s ($\times 10.8$) | 902 μ s ($\times 3.3$) | | |
| AT product | 5243 | 12,594 ($\times 2.4$) | 1880 | | 37,892 ($\times 20.2$) | 15,043 ($\times 8.0$) | 5899 ($\times 3.1$) | | |

lower in both HLS designs. There is only a 20% overhead in terms of cycle count for both implementations of Lloyd’s algorithm which indicates similar scheduling of operations. However, the HLS implementation has a significantly longer critical path (8.4ns compared to 5.0ns) which results in a performance gap of a factor of $2.1\times$ in terms of latency and $2.4\times$ in terms of AT product. The BRAM utilisation of the HLS design is lower because the synthesis tool decides to map some of the memories into LUT RAM. The last three columns show different variants of the HLS designs for the filtering algorithm. The design in Column 5 includes only code alterations that enable synthesizability and only uses Vivado’s synthesis directives to improve QoR which results in a $20.2\times$ degradation in terms of the AT product compared to the manual RTL design. Columns 6 and 7 show the importance of additional source-to-source transformations as discussed in Sect. 2.4.2. The manual partitioning of the heap memory narrows the performance gap from $20.2\times$ to $8.0\times$ (Column 6). The loop distribution in Listing 4 that enables pipelining in the tree traversal loop in addition to manual memory partitioning further improves the AT product to a factor of $3.1\times$ larger than that of the manual RTL design (Column 7). The final AT product is more than two times smaller than that for Lloyd’s algorithm.

2.6 Summary

This chapter presents a comparative case study for a C-to-FPGA flow using Xilinx Vivado HLS as an exemplary state-of-the-art tool. Our test cases are two alternative algorithms for K -means clustering, referred to as Lloyd’s algorithm and the filtering algorithm. The former is a data flow-centric brute-force approach and has regular control flow and regular memory accesses, whereas the implementation of the filtering algorithm uses dynamic memory management and is based on recursive traversal of a pointer-linked tree structure. The search space pruning applied by the latter algorithm translates into a substantial run-time advantage in sequential software implementations. We first investigate the practicality of the algorithm in the context of an FPGA implementation and show that a carefully optimised parallel RTL implementation achieves the same execution time with four times fewer logic slices and three times fewer DSP slices. We also show how a custom implementation of dynamic memory allocation greatly reduces the on-chip memory consumption for the filtering algorithm. The implementations and evaluations of this part of the study were first published in [26].

The second part of this case study repeats the comparison for HLS designs of both algorithms. The performance gap between the HLS and hand-written RTL implementations of Lloyd’s algorithm is approximately a factor of two in terms of area-time product, which is a remarkable result given the enormous difference in design time. The HLS design of the filtering algorithm also consumes a ‘close-to-hand-written’ amount of FPGA resources, but latency is initially degraded by a factor of $26.6\times$. The limited acceleration gained from semi-automatic design optimisations with synthesis directives results in a reversal of the previous finding: the AT product of the

initial HLS implementation of the filtering algorithm is larger than that for Lloyd's algorithm. We subsequently apply manual code transformations to partition and privatise data structures accessed through pointers in order to promote parallelisation and to enable pipelining of the loop traversing the pointer-linked data structure which results in an overall $8\times$ improvement of latency. The code transformations ultimately narrow the performance gap in terms of the AT product from $20.2\times$ to $3.1\times$ larger than that of the hand-crafted RTL design. The results of the HLS-based case study and guidelines for source code refactoring were first published in [27].

The AT product results in Table 2.3 show that both a carefully designed RTL and HLS implementation of the filtering algorithm outperform the respective implementation of the data flow-centric brute-force algorithm. This case study quantifies the benefits of hardware implementations of a sophisticated algorithm that uses structured data. We argue that this algorithm is representative of many other benchmarks that operate on tree structures, linked lists or graphs in general and common implementations of these algorithms are based on dynamically allocated data structures and pointer chasing. Due to the significant amount of source code refactoring in the implementation of the filtering algorithm, we conclude from this case study that the current generation of HLS tools lack support for effective design automation optimisations for this type of code. In particular, our code transformations enable memory partitioning, parallelisation and pipelining - optimisations that are essential for efficient FPGA designs. These optimisations require knowledge about data dependencies carried by data structures accessed through pointers.

Our goal in the following chapters of this thesis is to automate the memory partitioning and parallelisation in HLS flows targeting heap-manipulating programs. The difficult part of the automation of these optimisations is the program analysis: regardless of scope, every two heap-directed pointers could potentially reference the same memory cell and hence could create a data dependency. We propose an automated analysis of dependencies carried by data structures accessed through pointers, and an automated analysis to identify and privatise disjoint regions in the monolithic heap memory as the key features to improve the HLS support for (widely used) programs operating on dynamic, pointer-based data structures. Chapter 4 presents our approach to automatic heap partitioning and parallelisation. The HLS design aid in Chap. 4 automates the related code transformations that were applied manually in this chapter.

The synthesis of heap memory from on-chip BRAM in this case study and in Chap. 4 imposes a tight constraint on the working set size. For example, the RTL and HLS implementations in Sects. 2.3 and 2.4 use nearly 50% of the on-chip memory resources on the device. Chapter 5 removes this limitation by extending the technique in Chap. 4 to the automatic generation of application-specific parallel multi-cache systems in a framework where the heap resides in off-chip memory by default and only a fraction of it is held on-chip. This extension enables the HLS implementation of heap-manipulating programs with large memory footprints and alleviates the performance penalty due to the drop of memory bandwidth. Before describing the two core contributions of this thesis in Chaps. 4 and 5, we discuss related research in the following chapter.

References

1. T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, A. Wu, An efficient K-means clustering algorithm: analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.* **24**(7), 881–892 (2002)
2. W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, D. Stroobandt, An overview of today's high-level synthesis tools. *Des. Autom. Emb. Syst.*, pp. 1–21 (2012)
3. Sobel Edge Detector. <http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm>. Accessed 13 Mar 2016
4. S. Sarkar, S. Dabral, P. Tiwari, R. Mitra, Lessons and experiences with high-level synthesis. *IEEE Des. Test Comput.* **26**(4), 34–45 (2009)
5. BDTI, An Independent Evaluation of the AutoESL AutoPilot High-Level Synthesis Tool (2010). <http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot>. Accessed 10 Oct 2012
6. R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y.T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, K. Bertels, A survey and evaluation of FPGA high-level synthesis tools, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. http://janders.eecg.toronto.edu/pdfs/tcad_hls.pdf. Accessed 28 Feb 2016
7. Y. Hara, H. Tomiyama, S. Honda, H. Takada, Proposal and quantitative analysis of the CHStone Benchmark program suite for practical C-based high-level synthesis. *J. Inf. Process.* **17**, 242–254 (2009)
8. A.K. Jain, Data clustering: 50 years beyond K-means. *Pattern Recogn. Lett.* **31**(8), 651–666 (2010)
9. C.M. Bishop, *Pattern Recognition and Machine Learning* (Springer, New York, 2006)
10. D. Clark, J. Bell, Multi-target state estimation and track continuity for the particle PHD filter. *IEEE Trans. Aerosp. Electron. Syst.* **43**(4), 1441–1453 (2007)
11. Y.-C. Hu, M.-G. Lee, K-means-based color palette design scheme with the use of stable flags. *Electron. Imaging* **16**(3), 033 003–033 003–11 (2007)
12. T. Saegusa, T. Maruyama, An FPGA implementation of real-time K-means clustering for color images. *Real-Time Image Process.* **2**(4), 309–318 (2007)
13. J.P. Theiler, G. Gisler, Contiguity-enhanced K-means clustering algorithm for unsupervised multispectral image segmentation. *Proc. SPIE* **3159**, 108–118 (1997)
14. M. Estlick, M. Leeser, J. Theiler, J.J. Szymanski, Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware, in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (2001), pp. 103–110
15. P. Drineas, A. Frieze, R. Kannan, S. Vempala, V. Vinay, Clustering large graphs via the singular value decomposition. *Mach. Learn.* **56**(1–3), 9–33 (2004)
16. M. Leeser, J. Theiler, M. Estlick, J. Szymanski, Design tradeoffs in a hardware implementation of the K-means clustering algorithm, in *Proceedings of the IEEE Sensor Array and Multichannel Signal Processing Workshop* (2000), pp. 520–524
17. X. Wang, M. Leeser, K-means clustering for multispectral images using floating-point divide, in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2007), pp. 151–162
18. H.M. Hussain, K. Benkrid, A.T. Erdogan, H. Seker, Highly parameterized K-means clustering on FPGAs: comparative results with GPPs and GPUs, in *International Conference on Reconfigurable Computing and FPGAs* (2011), pp. 475–480
19. J. Kutty, F. Boussaid, A. Amira, A high speed configurable FPGA architecture for K-means clustering, in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)* (2013), pp. 1801–1804
20. T.-W. Chen, S.-Y. Chien, Flexible hardware architecture of hierarchical K-means clustering for large cluster number. *IEEE Trans. VLSI Syst.* **19**(8), 1336–1345 (2011)
21. Vivado-KMeans: Hand-Written HDL Code and C-Based HLS Designs for K-means Clustering Implementations on FPGAs. <https://github.com/FelixWinterstein/Vivado-KMeans>. Accessed 19 Dec 2015

22. High-Level Synthesis with LegUp, Accessed 20 Oct 2015. <http://legup.eecg.utoronto.ca/>
23. ROCCC 2.0Jacquard Computing, Accessed 12 May 2015. <http://www.jacquardcomputing.com/roccc/>
24. R. Nane, V. M. Sima, B. Olivier, R. Meeuws, Y. Yankova, K. Bertels, DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler, in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)* (2012), pp. 619–622
25. GAUT - High-Level Synthesis Tool From C to RTL, Accessed 21 Mar 2015. <http://hls-labsticc.univ-ubs.fr/>
26. F. Winterstein, S. Bayliss, G. Constantinides, FPGA-based K-means clustering using tree-based data structures, in *Proceedings International Conference on Field Programmable Logic and Applications (FPL)* (2013), pp. 1–6
27. F. Winterstein, S. Bayliss, G. Constantinides, High-level synthesis of dynamic data structures: a case study using Vivado HLS, in *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)* (2013), pp. 362–365

Separation Logic for High-level Synthesis

Winterstein, F.

2017, XIX, 132 p. 19 illus., 7 illus. in color., Hardcover

ISBN: 978-3-319-53221-9