High-Level Synthesis of Dynamic Data Structures: A Case Study Using Vivado HLS

Felix Winterstein[12]
[1]Ground Station Systems Division
European Space Agency
Robert-Bosch-Strasse 5, 64293 Darmstadt, Germany
Email: f.winterstein12@imperial.ac.uk

Samuel Bayliss[2], George A. Constantinides[2]
[2]Department of Electrical and Electronic Engineering
Imperial College London
South Kensington Campus, London, SW7 2AZ
Email: g.constantinides@imperial.ac.uk

Abstract—**High-level synthesis promises a significant shortening of the FPGA design cycle when compared with design entry using register transfer level (RTL) languages. Recent evaluations report that C-to-RTL flows can produce results with a quality close to hand-crafted designs [1]. Algorithms which use dynamic, pointer-based data structures, which are common in software, remain difficult to implement well. In this paper, we describe a comparative case study using Xilinx Vivado HLS as an exemplary state-of-the-art high-level synthesis tool. Our test cases are two alternative algorithms for the same compute-intensive machine learning technique (clustering) with significantly different computational properties. We compare a data-flow centric implementation to a recursive tree traversal implementation which incorporates complex data-dependent control flow and makes use of pointer-linked data structures and dynamic memory allocation. The outcome of this case study is twofold: We confirm similar performance between the hand-written and automatically generated RTL designs for the first test case. The second case reveals a degradation in latency by a factor greater than $30\times$ if the source code is not altered prior to high-level synthesis. We identify the reasons for this shortcoming and present code transformations that narrow the performance gap to a factor of four. We generalise our source-to-source transformations whose automation motivates research directions to improve high-level synthesis of dynamic data structures in the future.**

I. INTRODUCTION

High-level synthesis (HLS) raises the abstraction level for hardware description promising significant shortening of the design cycle compared with RTL-based design entry. To achieve latency and resource utilisation comparable to handwritten RTL, high-level synthesis often requires extensive code alterations to ensure synthesisability. These are especially important for programs with irregular control flow and complicated data dependencies. In this paper, we use Vivado HLS as an exemplary state-of-the-art tool to investigate HLS support for this type of programs. Our test cases are two algorithms for a compute-intensive machine learning application (K-means clustering). Algorithmically, both implementations produce exactly the same result, but they differ significantly in their computational properties: The first is a data-flow centric implementation with simple control flow, whereas the second is based on a recursive tree traversal. The latter application uses dynamic memory allocation to significantly reduce on-

chip memory requirements. We use hand-written RTL implementations of both algorithms as comparison points. The contributions of this paper are:

○ 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 and makes use of dynamic memory allocation. We highlight code transformations necessary to enable program synthesis.

○ 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 provide source-to-source transformations that improve QoR by a factor of eight and propose research directions aimed to automate these transformations in the future.

Section II discusses the technical background and describes the test cases. The C-based HLS implementations are described in Section III. Section IV presents QoR comparisons and Section V concludes the paper.

II. BACKGROUND

We use 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 [2] or ROCCC [3]. Similar to LegUp and ROCCC, it is based on an LLVM intermediate representation. 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. Similar to LegUp and ROCCC, the C-based input is restricted to a synthesisable subset excluding system calls, arbitrary pointer casting, arbitrary recursive functions and dynamic memory allocation (*new*, *delete*).

Listing 1 Main kernel of Lloyd's algorithm.

```
1: function LLOYDS(parameters N, K)
2:    for all x_j ∈ {x_1, x_2, ..., x_N} do
3:        // find closest centre z^ to data point x_j
4:        for all z_i ∈ {z_1, z_2, ..., z_K} do
5:            tmpDist = ‖x_j − z_i‖^2;
6:            if tmpDist < minDist or i = 0 then
7:                minDist = tmpDist;
8:                i^ = i; // index of z^
9:            end if
10:       end for
11:       // update centroid buffer
12:       centroidBuffer[i^].wgtCent += x_j;
13:       centroidBuffer[i^].count += 1;
14:   end for
15: end function
16: // update centre positions with information in centroid buffer
```

Listing 2 Recursive tree traversal of the filtering algorithm.

```
1: function FILTER(variables treeNode, centreSet, k)
2:    // find closest centre z^ to treeNode.bndBox.midPoint
3:    for all z_i ∈ centreSet do
4:        // ... distance calculation and min-search as above
5:    end for
6:    // z^ = closest centre, i^ = index closest centre
7:    if treeNode is leaf then
8:        // ... update centroid buffer as above
9:    else
10:       centreSet_new = malloc(k centre indices);
11:       k_new = 0;
12:       // prune candidate centres
13:       for all z_i ∈ centreSet do
14:           if !(pruningTest(z^, z_i, treeNode.bndBox)) then
15:               k_new++;
16:               insert z_i into centreSet_new;
17:           end if
18:       end for
19:       if k_new = 1 then
20:           // ...update centroid buffer as above
21:           free(centreSet_new);
22:       else
23:           // recurse on children
24:           FILTER(treeNode.left, centreSet_new, k_new);
25:           FILTER(treeNode.right, centreSet_new, k_new);
26:           // free allocated heap on the way back
27:           free(centreSet_new);
28:       end if
29:   end if
30: end function
31: // update centre positions with information in centroid buffer
```

Meeus *et al.* [4], Sarkar *et al.* [5] and BDTI [1] present evaluations of state-of-the-art HLS tools. Their work shares the commonality that the chosen benchmark cases are data-flow centric stream-based applications with simple control flow. On the contrary, this work aims to operate an HLS flow on a test case outside its 'comfort zone'.

The test cases we chose for this case study are two implementations of a K-means clustering application, a technique for unsupervised partitioning of a data set commonly used in a wide range of applications, such as machine learning, data mining, radar tracking, image colour or spectrum quantisation. K-means clustering partitions 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. Each cluster is represented by its geometrical centre. We consider two K-means algorithms: *Lloyd's algorithm* exhaustively computes Euclidean distances between data points and candidate centres to assign a closest centre to each point, whereas the *filtering algorithm* [6] uses a tree data structure (kd-tree, [6]) to prune unfavourable candidates early in the search process. A detailed description as well as a case study showing the advantage of the filtering algorithm in hardware are given in [7]. Simplified versions of the main kernels are shown in Listings 1 and 2.

To aid in the explanation of this case study, we identify the most important features of both applications. Lloyd's algorithm (Listing 1) captures most of the computation in a two-dimensional regular loop nest, which includes the Euclidean distance calculation (line 5) and a min-search (lines 6-9). All loop bounds (N, K) are constant. The key computational parts of the filtering algorithm (Listing 2) are the closest centre search (lines 2-6) and the candidate pruning (lines 12-18, $pruningTest$, remove centres form the current set). The loops enclosing closest centre search and candidate pruning have runtime-variable bounds $2 \leq k \leq K$. The implementation uses dynamic memory allocation (line 10, spawning a new object $centreSet_{new}$) and de-allocation (lines 21, 27) enclosed in data-dependent conditionals. Memory space is freed after traversal. In addition, the implementation uses recursive function calls (beyond tail recursion) which usually 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 objects $treeNode$, $centreSet$ (set of candidate centres), and the variable k (current set size).

III. HLS Implementations

Our goal is to bring the RTL designs produced by the HLS flow as close as possible to the highly optimised manual RTL designs in [7]. We distinguish between optimisations using synthesis directives and manual source code modifications.

A. Lloyd's Algorithm

The C code for Lloyd's algorithm corresponding to Listing 1 is directly synthesisable and does not contain any unsupported language constructs. We unroll all *for*-loops over the data point dimensionality D which results in a parallel implementation of the distance computation $\|x - z\|^2$. Most of the computation is contained in the inner *for*-loop (Listing 1, lines 4-10) with a 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 partitioning 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.

B. Filtering Algorithm

The synthesisability of the main kernel as in Listing 2 requires the removal of the recursive function calls and the calls to *malloc* and *free* (discussed in 1 and 2) and code transformations to improve QoR (discussed in 3 and 4).

Listing 3 Iterative replacement for the recursive kernel.

```
1: push(root, centreSet_initial, K, true);
2: while stack not empty do
3:    treeNode *u; uint *centreSet; uint k; bool d;
4:    pop(&u, &centreSet, &k, &d);
5:    ... = *centreSet;
6:    if (d == true) then
7:       free(centreSet);
8:    end if
9:    centreSet_new = malloc(K centre indices);
10:   //... original body in Listing 2 (contains two sub-loops)
11:   *centreSet_new = ... ;
12:   if (*u is not a leaf) and (k_new > 1) then
13:      push(u → right, centreSet_new, k_new, true);
14:      push(u → left , centreSet_new, k_new, false);
15:   else
16:      free(centreSet_new);
17:   end if
18: end while
```

1) Recursive Tree Traversal: Recursion is replaced by a *while*-loop and a stack which contains pointers to the heap-allocated tree node and set of candidate centres, as well as the set size k and a flag d indicating that the centre set can be de-allocated. Listing 3 shows the rewritten code.

2) Dynamic Memory Allocation: A new centre set is created in line 9 in Listing 3 and disposed in line 7 after it has been read for the second time. The duration for which a candidate set must be retained in memory depends on the shape of the (generally unbalanced) tree. In the worst-case the tree is degenerate (fully unbalanced) and more than one item remains in all centre sets after pruning. In this case, $N - 1$ centre sets must be retained in memory before they can be disposed, and hence, the heap memory for centre sets must be able to accommodate $N-1$ sets to ensure functional correctness in this worst-case scenario. For $N_{max} = 16384$ and $K_{max} = 256$, this memory then consumes 512 on-chip 36k-Block RAM (BRAM) resources ($\sim$81% in a medium-size Virtex 6 FPGA). In the average case, however, the tree is unlikely to be fully degenerate and the instantaneous memory requirement is significantly lower because centre sets can be disposed earlier. Dynamic memory allocation allows to exploit this to use the available memory space more efficiently by freeing unused space. Our custom implementation of the fixed-size allocator uses a *free-list* which keeps track of occupied memory space and the on-chip heap memory can accommodate an 'average-case' number of centre 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 [7]. This modification does not compromise the functionality of the algorithm, but it increases its runtime (number of node-centre interactions). Fig. 1 shows the result of profiling the C application clustering 16384 pixels (RGB vectors) randomly sampled from a benchmark image (Lena). We select a bound of $B = 256 \ll N_{max} - 1$ centre sets (8 BRAMs) which practically causes no runtime degradation in the scenarios considered here. A generic framework to infer such an average-case bound (semi-)automatically while still supporting the worst case requirement would be a valuable tool to support dynamic memory allocation in an HLS context.

3) 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
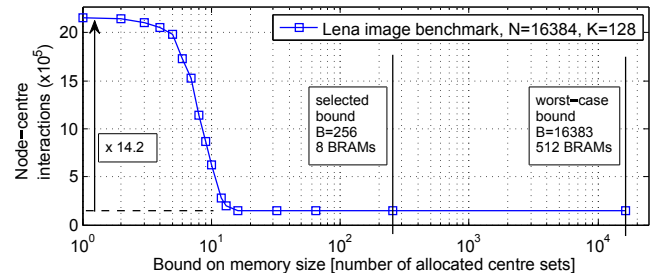

Fig. 1: Trade-off between heap size and runtime (profiling).

Listing 4 Loop distribution to enable pipelining.

```
1: while stack not empty do
2:    while (stack not empty) and (queue not full) do
3:       pop(&u, &centreSet, &k, &d);
4:       enqueue(u, centreSet, k, d); // newly introduced queue
5:    end while
6:    while queue not empty do
7:       dequeue(&u, &centreSet, &k, &d);
8:       //... remaining loop body in Listing 3 (lines 5-17)
9:       //... (contains two sub-loops with variable bounds)
10:   end while
11: end while
```

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 in Listing 3, however, hides this disjointness which renders the array partitioning directive ineffective and does not lead to parallel execution. In fact, applying automatic partitioning even leads to a degradation in latency as shown in the following section. Instead, we manually partition the tree node memory and privatise heap space for centre sets for each instance. This ensures that the scheduler 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.

4) Inter-Iteration Dependencies and Pipelining: Apart from replication, acceleration of the manual RTL design in [7] is obtained from pipelining the tree traversal. This corresponds to pipelining the loop nest in Listing 3 which requires to reason about two (potential) inter-iteration dependencies. The first is between *pop* and *push* statements on the stack and stack pointer which hinders pipelining. However, because there are two *push* statements and one *pop* statement, the items stored on the stack (pointers u and $centreSet$, k and d) accumulate if the condition in line 12 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. This loop, however, 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.

The other (potential) inter-iteration dependency is due to the pointer accesses in lines 5 and 11 in Listing 3. This is a false dependency because, after the loop transformation, the

TABLE I: Performance comparison using the hand-written RTL designs as reference.

Architecture: $N_{max} = 32768$, $K_{max} = 256$, $D = 3$, $B = 256$; Input data (synthetic): $N = 16384$, $K = 128$, $D = 3$, $\sigma = 0.2$						
	Lloyds RTL (ref.)	Lloyds HLS	Filt. RTL (ref.)	Filt. HLS (orig., directives only)	Filt. HLS (man. partitioning)	Filt. HLS (man. loop transf.)
P	40	40	2	2	2	2
Slices	25185	24103 ($\times 1.0$)	6950	6112 ($\times 0.9$)	5670 ($\times 0.8$)	7054 ($\times 1.0$)
LUT	66472	68360 ($\times 1.0$)	10418	14912 ($\times 1.4$)	13649 ($\times 1.3$)	16106 ($\times 1.5$)
REG	62168	63878 ($\times 1.0$)	19008	13324 ($\times 0.7$)	12601 ($\times 0.7$)	17013 ($\times 0.9$)
DSP	120	120 ($\times 1.0$)	40	46 ($\times 1.2$)	38 ($\times 1.0$)	38 ($\times 1.0$)
BRAM	83	89 ($\times 1.1$)	448	500 ($\times 1.1$)	516 ($\times 1.2$)	507 ($\times 1.1$)
Clock per.	5.0 ns	9.7 ns ($\times 1.9$)	5.0 ns	5.7 ns ($\times 1.1$)	5.7 ns ($\times 1.1$)	6.3 ns ($\times 1.3$)
Cycles/Iter.	53 k	66 k ($\times \mathbf{1.2}$)	54 k	1440 k ($\times \mathbf{26.6}$)	583 k ($\times \mathbf{10.8}$)	165 k ($\times \mathbf{3.0}$)
Time/Iter.	264 us	637 us ($\times \mathbf{2.4}$)	270 us	8208 us ($\times \mathbf{30.3}$)	3323 us ($\times \mathbf{12.3}$)	1036 us ($\times \mathbf{3.8}$)
AT prod.	6655	15361 ($\times \mathbf{2.3}$)	1880	50165 ($\times \mathbf{26.7}$)	18841 ($\times \mathbf{10.0}$)	7305 ($\times \mathbf{3.9}$)

pointers $centreSet$ and $centreSet_{new}$ 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 *malloc* and *free* and reasoning about such 'pointer-carried' dependencies which we propose as a promising area to explore in improving future HLS flows.

IV. PERFORMANCE COMPARISON

We compare the performance of the four designs based on different metrics: clock cycles count per clustering iteration (through RTL simulations), execution time per iteration (includes the clock period), FPGA resource usage and area-time product (AT product, given in logic slices $\times$ ms). We generate point sets of $N = 16384$ three-dimensional fixed-point samples (16 bit) which are distributed among $K = 128$ centres and follow a normal distribution with standard deviation σ. We implement the HLS designs (Vivado HLS 2012.2) and the hand-written RTL designs on a Xilinx Virtex 7 FPGA (7vx485t-1) using standard synthesis tools (Vivado 2012.2 RTL flow). The FPGA resource consumption is given by the utilisation of LUT, slice register (REG), digital signal processing (DSP), and 36k-BRAM resources. All designs are synthesised for a 200 MHz target clock rate and all results are taken from fully placed and routed designs. In order to account for the inherent runtime advantage of the filtering algorithm due to search space pruning and to compare all 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 I 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 cycle count for both implementations of Lloyd's algorithm is similar which indicates similar scheduling of operations. The last three columns show different variants of the HLS designs for the filtering algorithm. The design in column 5 includes only code alterations to enable synthesisability (discussed in 1) and 2) above) and uses only synthesis directives to improve QoR. Columns 6 and 7 show the effect of subsequent source-to-source transformations discussed in 3) and 4), respectively, narrowing the performance gap from a factor of 30.3 to a factor of 3.8 compared to the manual RTL design.

V. CONCLUSION

We present a comparative case study for a C-to-FPGA flow using Xilinx Vivado HLS as an exemplary 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 data-flow centric 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 performance gap between HLS-derived 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 $30\times$. We 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. We conclude that design automation optimisations for code using dynamic data structures are currently limited. We propose an analysis for finding tight bounds on the dynamically allocated heap memory, 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 research directions to improve the HLS support for (widely used) programs operating on dynamic, pointer-based data structures.

REFERENCES

[1] BDTI, "An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool," 2010. [Online]. Available: http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems," in *Proc. 19th ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2011, pp. 33–36.

[3] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *Proc. Symp. on Field-Programmable Custom Comput. Mach.*, 2010, pp. 127–134.

[4] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An Overview of Todays High-Level Synthesis Tools," *Design Automation for Embedded Systems*, pp. 1 – 21, Aug. 2012.

[5] S. Sarkar, S. Dabral, P. Tiwari, and R. Mitra, "Lessons and Experiences with High-Level Synthesis," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 34–45, Jul. 2009.

[6] 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. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002.

[7] F. Winterstein, S. Bayliss, and G. Constantinides, "FPGA-Based K-Means Clustering using Tree-Based Data Structures," in *Proc. Int. Conf. on Field Programmable Logic and Appl.*, 2013, pp. 1–6.