An Evaluation of Vivado HLS for Efficient System Design

Konstantinos Georgopoulos, Grigorios Chrysos, Pavlos Malakonakis, Antonis Nikitakis, Nikos Tampouratzis, Apostolos Dollas, Dionisios Pnevmatikatos, Yannis Papaefstathiou
Microprocessor and Hardware Laboratory, Technical University of Crete, Chania, Greece
kgeorgopoulos@isc.tuc.gr

***Abstract*—High-Level Synthesis (HLS) tools are hailed as one of the most promising ways to bridge the design productivity gap, especially for reconfigurable systems. These tools increase designer productivity at a possible performance and/or silicon cost, although the designer can play a significant role in the minimisation of both. Currently, one of the key challenges for the designer is to efficiently use the vendor-defined methodology and the design guidelines of the HLS tool. Hence, this work aims at assisting designers in taking full advantage and making optimal use of the official HLS methodology when implementing three fundamental algorithms used in a variety of video and image processing applications. One is a sorting algorithm while the other two are algorithms used in traversing tree/graph data structures. The work presented here concerns a highly popular HLS tool, namely Vivado HLS, and the experiences and analysis of the authors helps in the efficient use of the toolset, which can significantly increase design productivity when compared to the naive datasheet-based approach; the performance of all the resulting synthesisable models is also provided for completeness.**

***Keywords*—High-Level Synthesis; Reconfigurable Computing; Field-Programmable Gate Array; Data Structures**

I. Introduction

High-Level Synthesis (HLS) is currently a rapidly growing Electronic Design Automation (EDA) area and over thirty different tools [1] have been proposed in recent years. This trend is primarily due to the significant need for a fast and reliable design platform that, starting from a high-level model, described in C, C++ or C-based variants such as SystemC, generates its low-level counterpart, which can then be implemented in different technologies, e.g. Field Programmable Gate Arrays (FPGAs) or ASICs.
Such technology can help overcome challenges associated with working directly at the Register Transfer Level (RTL) [2]. Programmers proficient in VHDL or Verilog are very few hence non-experts have to attain a high-level of specialisation for efficient model development. Also, VHDL and Verilog are not as powerful as high-level languages leading to significantly long source codes, which increases the probability for coding mistakes and makes the process of refining the design for modifications a time consuming process.
Thus, HLS tools have seen a rapid rise in popularity since they specialise in generating production-quality RTL models based on high-level models. One of the most popular HLS tools is Vivado HLS and is an integral part of the Vivado Design Suite [3]. Input models are coded in C or C++ and cycle-accurate or untimed VHDL/Verilog RTL models (as well as SystemC) are generated at the output. Moreover, it offers automatic testbench generation, which facilitates behavioural and functional verification through the use of co-simulation.
Nonetheless, using Vivado HLS remains a process characterised by issues that must necessarily be taken into account. These issues sometimes deviate from the official vendor-supplied guidelines and this is the main focus of this paper. It unfolds many unexpected issues that the authors came across for three key image and video processing algorithms, i.e. the Mergesort sorting algorithm [4] and two graph traversing algorithms, the Depth-First Search (DFS) and the Breadth-First Search (BFS) [5].
In addition, the algorithms have been used in the context of three test cases in order to evaluate the tool within a complete application, i.e. Mergesort is used for calculating the Kendall Correlation Coefficient [6], DFS in a Maze Solving algorithm and BFS in a Graph Cuts [7] algorithm.
Section II discusses the high-level description of the selected algorithms. Section III contains information that improve and complement the official Vivado HLS flow and guidelines. Section IV contains the typical HLS flow steps that the authors find to be the most critical and, finally, conclusions are presented in Section V.

II. HLS Implementations

This section discusses important aspects of the implementations while it lists several actual code snippets. An important component of the design process has been the utilisation of key enhancers supported by Vivado HLS in the form of *directives*. These guide the way the tool treats a specific segment of code during the RTL-model generation. The ones considered and/or used within this work have been *dataflow*, *pipeline*, *unroll*, *array_partition*, *inline*, *loop_flatten*, and *interface* [8].

A. Depth-First Search & Maze Solver

Depth-First Search (DFS) is a method for traversing graphs [9] based on finding all possible paths from a specific node to every other node of the graph.
Subsequently, the main DFS function is recursive and since recursion cannot be directly mapped into hardware, the algorithm has been modified into a repetitive form function, which uses as input the graph edges in a streaming way and the ID of the start vertex as a stable value. Similarly, it outputs, in a streaming fashion, the vertices in the order that have been visited.
For streaming the data at both the input and output ends, FIFOs were used and in this context, directive *interface* was employed. The same directive was used for defining the input

```
1 const int NODES = 4000;
2 const int NODE_ORDER = 40;
3
4 Function run_DFS(int * from_node,int * to_node,
                                    int * visited_nodes, int first_node)
5 {
6   #pragma HLS INTERFACE ap_fifo port=from_node
7   #pragma HLS INTERFACE ap_fifo port=to_node
8   #pragma HLS INTERFACE ap_fifo port=visited_nodes
9   #pragma HLS INTERFACE ap_stable port=first_node
10
11     int visited_temp[NODES];
12     edge edge_array[NODES* NODE_ORDER];
13
14     int i, j;
15
16     run_dfs_label0:for(i = 0; i ¡ NODES; i++)
17     {
18   #pragma HLS unroll factor=2
19   #pragma HLS PIPELINE
20         visited_temp[i] = -1;
21     }
22
23     run_dfs_label1:for(i = 0; i ¡ no_edges; i++)
24     {
25   #pragma HLS unroll factor=2
26   #pragma HLS PIPELINE
27         edge_array[i].edge_weight = 0;
28         edge_array[i].from_id =from_id[i];
29         edge_array[i].id = 0;
30         edge_array[i].to_id = to_id[i];
31     }
32     dfs(edge_array, visited_temp, first_id, nodes, no_edges);
33
34     run_dfs_label2:for(i = 0; i ¡ nodes; i++)
35     {
36   #pragma HLS unroll factor=2
37   #pragma HLS PIPELINE
38 }
```

of the first vertex as stable. The DFS top-level function consists of three *for* loops that initialise the input data structures as well as transfer results to the output stream. For this, directives *unroll* and *pipeline* have been used for mapping in-parallel each of the initialisation loops.

Furthermore, the DFS algorithm itself is implemented within a function that is comprised of a *while* loop used to explore the graph as well as two *for* loops that move over the graph's nodes and each node's edges. The former cannot be optimised with the use of directives since the depth of the graph is not predefined, however, the two *for* loops can be unrolled and pipelined since they have an upper limit, i.e. the number of node connections and their neighbouring nodes. In addition, they can be flattened, using directive *flatten*, since they belong to the same level of code.

Moreover, graph representation has been carefully addressed with a static incidence list used for graph edges. A static table has been employed for the vertices due to the advantages that these structures can offer in space and time complexity. Finally, in order to set up the actual graph and invoke the DFS function, additional code has been included in the top-level model as shown in the code segment above.

B. Breadth-First Search & Graph Cuts

Breadth First Search (BFS) was first invented for finding the shortest path out of a maze and is here used on a Graph Cuts [7] algorithm, which stems from graph theory. Graph Cuts are used extensively in computer vision in order to address energy minimisation problems approached through the utilisation of the *max-flow min-cut* theorem. A highly popular algorithm for computing the max-flow is the BK algorithm

```
1   /* process neighbours */
2   for (a0=nodes[i].first_arc; a0; a0=arcs[a0].next_arc)
3   {
4     j = arcs[a0].head_node;
5     if ( !nodes[j].is_sink && (a=nodes[j].parent_arc) )
6     {
7       if (arcs[ arcs[a0].sister_arc ].r_cap) set_active(j);
8       if (a!=TERMINAL && a!=ORPHAN && arcs[a].head_node==i)
9       {;
10        set_orphan_rear(j); // add j to the end of the adoption list
11      }
12    }
13  }
```

[10] used in this work.

The second code segment on this page shows the breadth-first traversal of the graph. Here, function *set_active()* tracks the visited nodes while function *set_orphan_rear()* adds a node to a list for further processing.

Graph Cuts are global optimisation methods, which means that some *communication* is implied within the graph for a global solution to be achieved. This restricts the parallelisation possibilities in hardware as it induces significant I/O overhead. Hence, the adopted strategy has been based on splitting the existing graph problem into smaller sub-graphs.

This enables the increase of the level of on-chip parallelisation while it allows for the scaling up to larger problems that would otherwise pose demand for on-chip memory.

Graph splitting is comprised of two main stages, first, the data along the split are duplicated and weighed down by half in both sub-graphs while a certain margin defines the number of overlapping sub-graph nodes. Second, through the use of specific dual variables, the two sub-graphs are forced to have the same weight at all the crossover edges.

Subsequently, a top-level engine has been developed to include all individual sub-graphs and perform data loading. For data communication, a data bus is defined through the intermediation of an arbiter, which loads the data on a set of FIFOs ranging from *1* to *N*, where *N* is the total number of BK sub-graphs.

The key high-level blocks for the Graph Cuts implementation are *i)* a graph structure definition, *ii)* a multiple core instantiation, *iii)* the node information extraction, and *iv)* the initiation of the algorithmic execution.

In this implementation no directives are utilised due to the interconnection between the different cores, which does not allow automatic parallelisation whilst the maximum number of cores used has been set to 16.

C. Mergesort & Kendall Correlation

Mergesort is a divide and conquer comparison-based algorithm for array sorting. First, it divides the array into two halves, second, it sorts the two halves recursively and, finally, it merges the results. One application that makes use of a sorting algorithm such as the Mergesort is the Kendall Correlation Coefficient. Specifically, the Kendall tau (τ) rank correlation coefficient is a measure of the strength of dependence between two variables. Subsequently, various algorithms have been proposed for calculating the Kendall Coefficient, and one of the most popular based on the Mergesort is that proposed by Knight in [11] and used in this work.

In this implementation, the arrays have been stored in a BRAM, which provides two read/write ports. Unrolling further than 2 results in low clock frequencies and a slower hardware

```
1   //decompose input
2
3   for(i = 0; i ¡ len/2; i++)
4   {
5     #pragma HLS PIPELINE
6     buf1[i] = arr1[i];
7     buf2[i] = arr1[i+len/2];
8     bufc1[i] = arr2[i];
9     bufc2[i] = arr2[i+len/2];
10    }
11
12  //merge sort segments
13
14  swapCount+=mergesort( buf1,temp1,bufc1,tempc1,len/2);
15  swapCount+=mergesort( buf2,temp2,bufc2,tempc2,len/2);
16
17  //merge sorted arrays
18
19  for(i = 0; i ¡ len/2; i++)
20  {
21    #pragma HLS PIPELINE
22    arr1[i] = buf1[i];
23    arr1[i+len/2] = buf2[i];
24    arr2[i] = bufc1[i];
25    arr2[i+len/2] = bufc2[i];
26  }
27
28  swapCount+=merge(arr1,bufs,arr2,bufs2,0,len/2,len);
```

design in addition to greater resource utilisation.
The top level function of the high level model synthesised is presented at the top of this page. This function contains three major tasks, i.e. *i)* it decomposes the input into two segments, *ii)* it invokes two Mergesort function calls and *iii)* it merges their results.
The Kendall Correlation Mergesort function sorts one array while, at the same time, re-arranging the second array. In this work, a non recursive implementation has been used since Vivado HLS does not support such an approach. For this purpose, two iterative functions have been used along with directives *pipeline* and *unroll* (factor of two) for performance enhancement.

III. Tool Evaluation

In this section we present observations and practical notes on Vivado HLS' ability to generate a low-level RTL model from its equivalent high-level representation.

A. Algorithms-Centred Evaluation

The RTL-model generation as well as the simulation process have been based on the following characteristics in terms of data sets used, targeted hardware and CPUs employed. The DFS Maze Solver algorithm uses a 30x30 maze puzzle with the same starting and goal positions; the hardware implementation is based on a Virtex-7 XC7VX415T FPGA while software execution is performed with an Intel i5 at 2.5GHz and 4GB of RAM.
The BFS algorithm utilises 16K nodes and 64K edges on-chip while hardware implementation is based on a Virtex-7 XC7VX330T FPGA and software execution is performed with an Intel i5 at 3GHz. Finally, the Mergesort algorithm is based on a 2x10K input time series while hardware/software executions target a Virtex-7 XC7V690T FPGA and an Intel i5 at 3.2GHz respectively. Finally, the version of Vivado HLS used in this work has been 2014.2.

1) Depth-First Search & Maze Solver: For the generated RTL-code to be easily ported or simulated by other tools, the designer must focus on the design's I/O. In particular, scalar inputs must be set to a valid value at least one clock cycle before the start signal activation. This refers to one of the I/O ports of the RTL module generated such as *ap_start*, which is essentially activated in order to move from an idle state to a processing state. For instance, the scalar parameters of the top-level function have been passed with directive *interface* through the *ap_stable* mode. The *ap_stable* mode indicates that the design will consider the corresponding input port as stable throughout the processing cycles.
In addition, using directive *interface* along with a FIFO interface means that the FIFO module control signal, i.e. *signal_empty_n()*, controls the total simulation process and should, therefore, be set to a logic "one" after the data enter the design since a logic "zero" forces the simulation to stop.
The throughput demand for this particular implementation is minimal, hence, the I/O issues do not influence the performance of the system which is very close to the initial software solution. In other words no notable speedup has been measured although changing the I/O interface to a FIFO-like structure slightly improves design performance, which may otherwise offer a latency that is longer than that of software. Specifically, directive *pipeline* caused small performance gains mainly due to the control-based nature of the DFS algorithm and its data dependencies.
Nevertheless, the mapping of the memory I/O technique seems to be less effective than the I/O FIFO streaming scenario, especially in cases where the data are streamed and stored inside the FPGA. Directive *pipeline* can worsen the performance of the design if it is applied on code with data dependencies or in loops where the limits are not predefined. Also, directive *dataflow* could offer really good results in case of easily parallelisable code, whereas in our control-based algorithm it did not work. Finally, directive *unroll* offers little performance improvements.
The major advantage here is that, using HLS, the effort was concluded in a few days, which constitutes a considerable shortening of design implementation time.

2) Breadth-First Search & Graph Cuts: The key observation related to this implementation is on how parallel cores are initiated in Vivado HLS.

```
1   // this is not allowed
2
3     Graph graph_cores[32];      //Graph is an object (a hardware core)
4
5     for (int i = 0; i ¡ 32; i++) graph_cores[i].set_trcap(i, trcap);
```

The coding style shown here cannot be handled by the tool because there can be no explicit reference to each core, such as in the case of a loop, especially in a parametric way.
Consequently, a reliable way to resolve this problem is through the implementation of a wrapper function, which implements a parametric controller to the multiple cores and, thus, it is efficiently and properly handled by Vivado HLS.
Subsequently, Table I presents latency and speedup results for the BK algorithm implementation. Note that these results are not a function of different HLS directives since there was no effective way to speed up this particular algorithm; rather, the results vary depending on the level of parallelisation. Hence, the greater the number of parallel cores, the smaller

the HW latency and, consequently, the greater the speedup over the software implementation, which cannot implement any parallelisation whatsoever and, therefore, achieves only one latency figure.

TABLE I. BFS & BK ALGORITHM PERFORMANCE RESULTS

Config/ Parallelisation Level	Total Nodes	HW Latency (ms)	SW Latency (ms)	HW Speedup
1 Core	16K	12.17	6.1	0.5x
8 Cores	16K	1.79	6.1	3.3x
16 Cores	16K	0.95	6.1	6.2x

3) Mergesort & Kendall Correlation: The algorithm that calculates the Kendall Correlation is based on two calls of the Mergesort algorithm. Parallelisation can be achieved by splitting the array into smaller arrays, sorting them and merging the sorted arrays. However, the gain in speed is not significant, especially with small arrays as the merge operations have to be performed sequentially.
Hence, the algorithm is based on counting the swaps a Mergesort needs to sort the second array while the first array has to be sorted before the count of the swaps. As a result, each call of the algorithm needs two Mergesort calls.
The parallel Mergesort was implemented by copying parts of the input into new arrays in order for the tool to implement them using separate BRAMs so that the creation of dependencies was avoided. After a partial array is sorted it is merged with its adjacent array. This allowed the resulting design to sort in parallel, which would otherwise require large multiplexers on the input BRAMs, resulting in a slow clock frequency or a sequential operation.

TABLE II. MERGESORT & KENDALL CORRELATION PERFORMANCE RESULTS

	Time (ms)	Speedup *(Without Directives)*	Speedup *(With Directives)*
S/W (i5@3.1GHz)	4.6	1x	-
No Directives	17.8	0.26x	1x
Pipeline	12	0.38x	1.48x
Pipeline + Unroll	10.1	0.46x	1.76x
Parallel Mergesort 4	6.1	0.75x	2.9x
Parallel Mergesort 16	3.2	1.4x	5.6x

Table II presents the runtime results for different (directive) configurations of the hardware implementation. The software runtime is presented followed by the implementation without directives. Consequently, directives are introduced and the results of the parallel Mergesort implementation are presented. Parallelising even further yields marginally better results and almost reaches the limits of the FPGA's BRAM resources which, in turn, leads to lower clock frequencies. This happens because each Mergesort core needs its own BRAM allocation. Finally, speedup is also a function of input size since smaller datasets allow for more parallel Mergesort cores thereby leading to reduced latencies.

B. Generic Evaluation

An initial observation is that the high-level C models are simulated on a C/RTL co-simulator, which, based on our measurements, generates accurate estimates on the number of cycles required. Hence, this ensures that the generated RTL models are indeed cycle-accurate.
Based on the time spent for the development process, it can certainly be supported that a major underlining factor for using Vivado HLS is the ability to attain considerably shorter implementation times. This applies to all three case studies while the more complex the system, the higher the development speedup.
On the other hand, Vivado HLS offers weak support for SystemC-described models while it suffers from limited support for the *math.h* function. For instance, the *pow* function is not supported nor are most of the logarithmic operations, hence, such operations must be manually developed.
Furthermore, Vivado HLS does not guarantee that function calls with no dependencies will be implemented in-parallel. This problem can be remedied using the *array_partition* directive, however, it is somewhat "temperamental" in the sense that it can cause the tool's flow to crash for reasons that are not yet entirely clear.
In terms of the latency estimation offered at the end of the C-synthesis stage, it has been noted that it sometimes deviates significantly and is, therefore, unreliable. On the other hand, directives *pipeline* and *unroll* perform very well together especially when *unroll* is tied with a factor of a small value.
Subsequently, for an application that uses the same arithmetic operation on multiple and independent sets of data, the designer is strongly advised to try out the *unroll* and *pipeline* directives. Both those directives have been proven to increase the parallelism of the design and help in generating efficient RTL models.
On the other hand, applications that have low levels of inherent parallelism require more manual intervention. Such applications are similar to DFS and BFS, i.e. problems that have to do more with memory accesses rather than computational issues.
Such a scenario can be approached as a wider problem that is broken down to a selection of smaller ones, which can then operate in parallel. Hence, a manually written reproducible/reusable small-size design core can be defined as a C++ object and then manually instantiated multiple times so that each one can process its corresponding set of data. Finally, Table III presents a short commentary on the Vivado HLS directives used.

TABLE III. VIVADO HLS DIRECTIVES - OVERVIEW

Name	Comments
PIPELINE	Very efficient overall
UNROLL	Very efficient - Achieves good speedups
ARRAY_PARTITION	Can slow down the design
DATAFLOW	Works well with no dependencies
INLINE	Efficient but generates HW overhead
INTERFACE	Always used for I/O interfacing
LOOP_FLATTEN	Works well at an additional area cost

IV. PROPOSED HLS FLOW

The observations presented so far are merged with the typical HLS flow in order to structure an efficient design methodology. Fig. 1 shows both the standard steps of the HLS flow as well as those identified as very important by the authors, indicated with a dashed red line. The high-level description of the algorithm, in the form of a C/C++ model, is initially modified in order to eliminate non-HLS compatible

elements such as pointers and is supplemented by the HLS tool's directives. This results in a model that is ready to be processed, i.e. pre-synthesis model.
Subsequently, the tool generates the RTL equivalent model, which is then combined with a testbench file for co-simulation. In co-simulation, the RTL design's functionality is confirmed and performance characteristics, such as latency, resource utilisation, clock speed and others, can be studied.
All this comes in the form of report files that the designer can take into account for the potentially necessary modifications that need to be introduced to the high-level model of the target algorithm. Hence, this process is repeated until all target criteria have been met and the RTL model can be used within a translate, map, and place & route tool for bitstream generation.
With respect to the key flow steps, first, the designer must set up the I/O to the high-level model such that the produced RTL model is portable. Furthermore, the high-level model must be studied carefully and, in case of no inherent parallelism, should be modified by manually breaking down the algorithm into smaller independent units that can then operate in-parallel.
For the same reason, the *array_partition* directive can be employed in the high-level model. Furthermore, in case an explicit number of cores is to be used in the loops, this must be approached alternatively through the use of a wrapper function similar to the one described earlier.
Directives *unroll* and *pipeline* are highly beneficial to the synthesis process since they help in bringing forward parallelism and pipelining, especially in cases where arithmetic operations are used many times and on multiple and independent sets of data.
Finally, Vivado HLS lacks any serious math functions which eventually have to be coded manually. Also, any latency estimates prior to the co-simulation have been found unreliable.

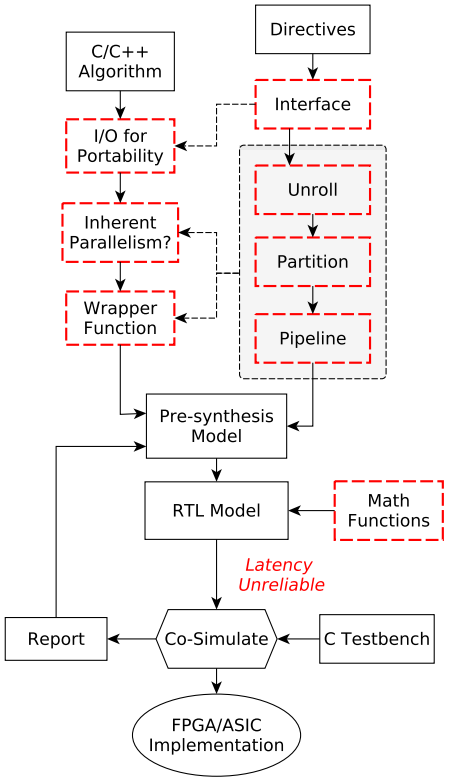

Figure 1. High Level Synthesis Flow - Enhanced

V. Conclusions

This paper outlines the distilled knowledge accumulated by the authors when using the Vivado HLS tool in three separate high-to-low level algorithm transition cases. This has helped in evaluating the official guidelines as well as extracting useful conclusions as to the type of actions one needs to take.
The three different computing algorithms of interest have been the DFS, BFS and Mergesort, selected due to their noted popularity and utilisation in numerous image and video processing systems.
Hence, based on our hands-on observations, if the algorithm is inherently parallel it can easily and efficiently be synthesised by an HLS tool, like Vivado HLS, without significant manual intervention. Moving to the non-easily parallelisable data processing schemes, we present a divide-and-conquer approach that can be implemented manually and will allow for the efficient synthesis of a high-level functional description.
In two of the three case studies, the implemented modules were faster than the software execution on multi-GHz state-of-the-art CPUs. In addition, in all cases, most of the available Vivado HLS directives were utilised and the majority of them were found to be efficient.

References

[1] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *24th International Conference on Field Programmable Logic and Applications (FPL)*, September 2014.

[2] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architects Workbench*. The Kluwer International Series in Engineering and Computer Science 85, Springer US, 1 ed., 1990.

[3] Xilinx Inc., *Vivado Design Suite User Guide v2015.1*, 2015.

[4] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[5] D. Kozen, *The Design and Analysis of Algorithms*. New York: Springer-Verlag, 1991.

[6] R. B. Nelsen, "Kendall tau metric," 2001.

[7] D. M. Greig, B. T. Porteous, and A. H. Seheult, "Exact maximum a posteriori estimation for binary images," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 51, no. 2, pp. 271–279, 1989.

[8] R.-T. S. L. Wiki, "Vivado hls knowledge base," 2015.

[9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[10] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 9, pp. 1124–1137, 2004.

[11] W. R. Knight, "A computer method for calculating kendall's tau with ungrouped data," *Journal of the American Statistical Association*, vol. 61, no. 314, 1966.