High-Level Synthesis versus Hardware Construction

Alexander Kamkin
ISP RAS
Plekhanov RUE
MSU, MIPT, HSE
kamkin@ispras.ru

Mikhail Chupilko
ISP RAS
Plekhanov RUE
Moscow, Russia
chupilko@ispras.ru

Mikhail Lebedev
ISP RAS
Plekhanov RUE
Moscow, Russia
lebedev@ispras.ru

Sergey Smolov
ISP RAS
Plekhanov RUE
Moscow, Russia
smolov@ispras.ru

Georgi Gaydadjiev
University of Groningen
Plekhanov RUE
Groningen, Netherlands
g.gaydadjiev@rug.nl

***Abstract*—Application-specific systems with FPGA accelerators are often designed using high-level synthesis or hardware construction tools. Nowadays, there are many frameworks available, both open-source and commercial. In this work, we aim at a fair comparison of several languages (and tools), including Verilog (our baseline), Chisel, Bluespec SystemVerilog (Bluespec Compiler), DSLX (XLS), MaxJ (MaxCompiler), and C (Bambu and Vivado HLS). Our analysis has been carried out using a representative example of 8×8 inverse discrete cosine transform (IDCT), a widely used algorithm engaged in JPEG and MPEG decoders. The metrics under consideration include: (a) the degree of automation (how much less code is required compared to Verilog), (b) the controllability (possibility to achieve given design characteristics, namely a given ratio of the performance and area), and (c) the flexibility (ease of design modifications to achieve certain characteristics). Rather than focusing on computational kernels only, we use AXI-Stream wrappers for the synthesized implementations, which allows adequately evaluating characteristics of the designs when they are used as parts of real systems. Our study shows clear examples of what impact specific optimizations (tool settings and source code modifications) have on the overall system performance and area. It emphasizes how important is to be able to control the balance between the communication interface utilization and the computational kernel performance and delivers clear guidelines for the next generation tools for designing FPGA-accelerator-based systems.**

***Index Terms*—electronic design automation, application-specific computing, hardware construction, high-level synthesis, field-programmable gate array, inverse discrete cosine transform**

I. Introduction

The traditional flow for *electronic design automation* (*EDA*) starts with the development of a *register-transfer-level* (*RTL*) *model* in a *hardware description language* (*HDL*), such as Verilog or VHDL [1]. Appeared in the 1980-90s, the HDLs revolutionized the hardware design, but now, despite the minor updates made in the early 2000s, they look outdated and do not provide high productivity, especially when it comes to certain application domains, such as *digital signal processing* (*DSP*).

Currently, there are two directions for rapid hardware development: (a) *high-level synthesis* (*HLS*) and (b) *hardware construction* (*HC*) [2], [3]. In the first case, an RTL model is synthesized from a high-level description that ignores the implementation details (roughly speaking, from an algorithm). In the second case, hardware microarchitecture is specified explicitly, but in a flexible, highly parameterized way. Both approaches enable users to perform *design-space exploration* (*DSE*) and optimize their designs according to the given constraints on performance, power consumption, and area.

The aim of the work presented here is (a) to study the state of the art in HLS/HC for FPGA, (b) to engage different tools, and (c) to evaluate their effectiveness. We use 8×8 *inverse discrete cosine transform* (*IDCT*) [4] as a benchmark and the following comparison metrics: (a) the *degree of automation* (how much less code is required as compared to Verilog), (b) the *controllability* (possibility to achieve given design characteristics, i.e. a point in the $Performance \times Area$ space), and (c) the *flexibility* (ease of modifying a design and/or finding tool settings to achieve certain characteristics).

To understand our motivation, a broader context should be considered. We are working on an infrastructure for organizing IP libraries in an FPGA-focused EDA system. The main requirements are (a) the ease of describing IP cores (mainly mathematical ones) and (b) the ability to adjust them to specified constraints. Accordingly, IP core generators (based on HLS/HC) are more suitable than predefined HDL descriptions, and it is important to understand which tools are working best. The IDCT example has been chosen for the following reasons: (a) this is a well-known algorithm used in JPEG and MPEG decoders [5], [6]; (b) it represents an "average" computational kernel; (c) there exist many ready-to-use implementations (e.g., in C [6], DSLX [7], and BSV [8]).

The main contributions of this work are as follows:

- novel methodology for assessing HLS/HC tools;
- careful comparison of the existing HLS/HC solutions.

All results presented in this paper and the related source code are available on GitHub [9].

The rest of the paper is organized as follows. Section II classifies HLS/HC approaches and reviews existing HLS/HC surveys. Section III describes our evaluation methodology: the metrics, the benchmark, and the procedure. Section IV presents our experimental analysis of the tools: the obtained indicators and the overall comparison. Section V summarizes the results and outlines directions for further research.

II. Background and Related Work

The classical HDLs, such as Verilog and VHDL, are still the main means of designing hardware. However, the situation is changing, and nowadays we have a wide range of languages and tools. Their appearance and development is dictated by the obvious desire to reduce labor costs

978-3-9819263-7-8/DATE23/© 2023 EDAA

for system design and verification. Depending on how this goal is implemented, we identify the following groups of tools: (a) *highly specialized generators* aimed at building efficient hardware implementations of particular algorithms (e.g., FloPoCo [10]); (b) *HC tools built on top of programming languages* (e.g., Chisel/Scala [11], MyHDL/Python [12], and JHDL/Java [13]); (c) *HC tools based on high-level HDLs* (e.g., Bluespec Compiler [14]); (d) *HLS tools for imperative programming languages*, such as C/C++ and Fortran (e.g., Bambu [15], LegUp [16], and Vivado HLS [17]); (e) *HLS tools for parallel programming languages*, such as CUDA C, OpenCL, and DPC++ (e.g., Vitis HLS [19], Intel FPGA SDK for OpenCL [20], and oneAPI [21]); (f) *HLS tools for dataflow programming languages* based on both the imperative and the functional paradigms (e.g., MaxCompiler [22] and XLS [7]); and (g) *HLS tools for domain-specific languages*, such as MATLAB, TensorFlow, and OpenVX (e.g., HDL Coder [23], Vitis AI [24], and HiFlipVX [25]).

In the last decade, a number of HLS/HC surveys were published. In [26], W. Meeus et al. have evaluated a broad selection of tools. The comparison criteria are as follows: (a) source language and abstraction level (untimed, cycle-accurate or mixed); (b) ease of description and learning curve; (c) support for fixed- and floating-point numbers; (d) ease of DSE (optimization options, source code modification, etc.); (e) verification facilities (namely testbench generation); (f) resource usage of the generated designs (number of FPGA slices). As a benchmark, the 3×3 Sobel filter was applied. At present, that survey looks somewhat outdated. Moreover, being targeted mainly at design productivity, it lacks some information relevant to our study: how much the tools can optimize the designs compared to hand-written RTL models.

In [27], L. Daoud et al. concentrate on using HLS/HC for developing heterogeneous systems with FPGA-based accelerators. Existing tools have been divided into four categories according to the input language: (a) C/C++ and its derivatives; (b) non-C/C++ languages: MATLAB, Python, etc.; (c) visual modeling languages: LabVIEW, Simulink, etc.; (d) GPU-oriented languages: CUDA and OpenCL. The review provides a good idea of the HLS/HC landscape, but does not compare the frameworks with each other. Good many of the mentioned solutions are no longer supported or available for download.

In [28], R. Nane et al. have attempted to overview recent HLS/HC frameworks and evaluate some of them (four popular C-to-HDL compilers). The comparison is based on such indicators as license type, input/output languages, application domain, testbench automation, and support for floating- and fixed-point numbers. A distinctive feature of the work is that it unveils how HLS/HC works "under the hood" and describes basic optimizations. In-depth evaluation has been made for three academic tools (Bambu, DWARV, and LegUp) and a commercial one (undisclosed). The authors performed two sets of experiments: first, they executed each tool with the default settings; then they modified the source code and the options to maximize the performance. In both cases, there were assessed the frequency and wall-clock time as well as the LUT, DSP and BRAM usage. Most of the benchmarks were taken from the CHStone suite [29]. The idea of conducting two sets of experiments looks interesting and has been used in our work. However, we have compared tools of different types (not only C-to-HDL compilers) and applied other metrics.

To summarize this section, the available surveys of HLS/HC solutions are mostly obsolete and either contain qualitative, non-measurable information, or focus on a rather small number of tools. Our aim is to mitigate these disadvantages.

III. Evaluation Methodology

In this section, we describe the methodology used to assess HLS/HC tools: the metrics, the benchmark, and the procedure.

A. Metrics

Our study exploits the following primary indicators:

- *source code size* (L) – the number of lines of code (LOC), including tool settings (directives, parameters, etc.);
- *performance*, or *throughput* (P) – the number of operations per second (OPS);
- *area*, or *resource usage* (A) – the number of utilized lookup tables (LUT) and flip-flops (FF).

We do not treat L as a perfect metric, but it is an objective one. It can even be adopted for visual modeling languages (since the tools should produce code at some stage); however, such languages and tools are out of the scope of this paper.

The ratio $Q = \frac{P}{A}$ captures the design *quality*, or its *efficiency*. Let Φ be an optimization criterion being maximized. Unless otherwise stated, we assume that $\Phi \equiv Q$. Based on L and Φ we measure the *degree of automation* (α), the *controllability* (C_Φ), and the *flexibility* (F_Φ).

Let $\mathcal{A}$ be an algorithm (a benchmark), $\mathcal{L}$ be an HDL, and $\mathcal{T}$ be an HLS/HC tool for $\mathcal{L}$.

1) Degree of Automation: α shows how easy it is to describe $\mathcal{A}$ in $\mathcal{L}$ and setup $\mathcal{T}$ compared to Verilog (L_V):

$$\alpha = \frac{(L_V - L)}{L_V} \times 100\%. \tag{1}$$

2) Controllability: By *control abilities* we mean the spectrum of facilities that allow a hardware designer to influence the synthesis result (in terms of Φ) without changing the functionality. They include tool settings, code annotations, and source code itself. The metric is defined as follows:

$$C_\Phi = \frac{\Phi^*}{\Phi_V^*} \times 100\%, \tag{2}$$

where Φ^* is the maximum value of Φ achieved by the tool $\mathcal{T}$ and Φ_V^* is the "absolute" maximum expected with Verilog.

3) Flexibility: F_Φ indicates how easy it is to improve the Φ value by setting up $\mathcal{T}$ and modifying the source code:

$$F_\Phi = \frac{\Phi^* - \Phi_0}{\Delta L}, \tag{3}$$

where values Φ^* and Φ_0 relate respectively to the "optimal" and "initial" designs, while $\Delta L = \Delta L^+ + \Delta L^-$ is the number of changed lines of code (added and removed), including code annotations and parameters.

B. Benchmark

As a benchmark, we use 8×8 IDCT (known as DCT-III). All the implementations are IEEE 1180-1990-compliant [4] and mostly based on IDCT from the ISO/IEC 13818-4:2004 conformance test suite [6], [32]. The input is a matrix of 12-bit numbers, while the output is a matrix of 9-bit numbers. For more information, see the source code on GitHub [9].

C. Procedure

For each language considered in this study (Verilog, Chisel, Bluespec SystemVerilog, DSLX, MaxJ, and C), the "initial" IDCT description is developed in accordance with the Chen-Wang's butterfly algorithm [30], [31]. When RTL languages (Verilog and Chisel) are used, such a description is likely to be a combinational circuit. For the resulting functional unit, LOC is counted (excluding comments and blank lines) — L^{FU}.

For each tool, the "optimal" design is created with the aim to maximize Φ. It is done by using the command-line options, pragmas, and/or code modifications. It worth noting that the most frequent optimization is pipelining.

To make designs AXI-Stream-compliant [33] (which is quite common for IP libraries [34]), an interface adapter is created. Input/output data are transmitted row-by-row. The adapter is either generated automatically (by the HLS/HC tool) or written manually (in the source language or Verilog). In the second case, LOC is counted — L^{AXI}. Being a system-level tool, MaxCompiler generates the PCIe adapter. In this case, we evaluate a kernel without the interface wrapper ($L^{AXI} = 0$).

If the tool requires tuning, the size of configuration files and the number of parameters are counted — L^{Conf}. For the initial experiment, the default settings are applied ($L^{Conf} = 0$).

The labor costs for creating the functional unit are measured as $L = L^{FU} + L^{AXI} + L^{Conf}$. The degree of automation (α) is estimated according to equation (1).

To synthesize the implementation, we use the Vivado design suite (v2017.4) [18] with the default settings. We determine the minimum clock period, for which the synthesis is successfull (T_{clk}), and estimate the maximum frequency and throughput:

$$\nu_{max} = \frac{1}{(T_{clk} - T_{wns})}, \qquad P = \frac{\nu_{max}}{T_P},$$

where T_{wns} is the *worst negative slack* provided by Vivado and T_P is the *periodicity*, which is the minimum number of cycles between starts of two subsequent operations. *Latency* (T_L), i.e. the number of cycles required to execute an operation (including I/O transmission) is also of interest.

With regard to FPGA area, we use the following indicators: (a) N_{LUT} – the number of LUTS; (b) N_{FF} – the number of FFs; (c) N_{DSP} – the number of DSP blocks; (d) N_{IO} – the number of inputs and outputs. As synthesis differently maps different designs to resources (first of all to DSP blocks), we use the normalized indicator:

$$A = N^*_{LUT} + N^*_{FF},$$

where N^*_{LUT} and N^*_{FF} are respectively the numbers of utilized LUTs and FFs when disabling the DSP usage (this can be done by setting `maxdsp=0` in Vivado). As designs do not use on-board memory, BRAM consumption is not considered.

Language	Paradigm	Tool	Type	Openness
Verilog	Classical RTL	Vivado	LS/PR	Commercial
Chisel	Functional/RTL	Chisel	HC	Open-source
BSV	Rule-based/RTL	BSC	HC	Open-source
DSLX	Functional	XLS	HLS	Open-source
MaxJ	Dataflow	MaxCompiler	HLS	Commercial
C	Imperative	Bambu	HLS	Open-source
		Vivado HLS	HLS	Commercial

TABLE I
LANGUAGES AND TOOLS UNDER EVALUATION

After P and A are known, the objective function Φ is calculated. To evaluate HLS/HC tool capabilities, we optimize the design according to Φ (as much as possible) and measure C_Φ and F_Φ as stated in equations (2) and (3).

IV. EVALUATION

The evaluated languages and tools are presented in Table I (LS/PR stands for *logic synthesis/place & route*). We used Vivado to synthesize the designs, obtain their characteristics, and generate the FPGA bitstreams. The device is Xilinx's Virtex Ultrascale+ (XCVU9P-FLGB2104-2-E): $N_{LUT} = 1,182,240$, $N_{FF} = 2,364,480$, $N_{DSP} = 6,840$, and $N_{IO} = 702$.

Below we report our results grouped by input languages. The quantitative data are summarized in Fig. 1 and Table II.

Verilog is a classical HDL created in the mid 1980s [35]. A description is a hierarchy of modules; each module contains input/output ports and encapsulates net declarations, processes (`always` blocks and continuous assignments), and instances of other modules. Nets require the bit widths to be explicitly specified; blocks are written procedurally in a C-like syntax. To design reusable IP cores, the language supports module parameterization and `generate` statements.

Initial Design: The starting point is a naive combinational circuit with the row-by-row AXI-Stream adapter. It contains eight IDCT^{row} and eight IDCT^{col} instances performing row- and column-wise IDCTs respectively. Obviously, such organization requires lots of resources and leads to low frequency. The major bottleneck here is the sequential adapter (in theory, the implementation could run 8 times faster).

Optimization(s): Two optimized designs have been tried. The first consists of one IDCT^{row} and eight IDCT^{col} instances. There is no point in having eight IDCT^{row} if only one row arrives at each clock cycle. The throughput grows by 1.8 times (as goes the frequency), and the area reduces by 1.7 times. Accordingly, the quality is more than tripled.

The second design contains one IDCT^{row} and one IDCT^{col}. Such architecture increases the latency from 17 to 24 cycles. However, the throughput is doubled (compared with the initial version), the area reduces by 4.6 times, and the quality increases by the factor of 9.4.

Chisel is an open-source Scala-based language for RTL design [11]. In addition to conventional HDL constructs, it provides object-oriented and functional programming facilities (polymorphism, abstract data types, and recursion), which makes it possible to build flexible IP core generators [3]. To

reduce efforts, the language supports type inference: bit widths of ports and registers are specified manually, but the remaining widths can be derived automatically.

Initial Design: As before, we started with a simple combinational circuit. Interestingly, compared to Verilog, this design has slightly better performance (105.7%) while requiring less area (94.6%). It might be explained as follows. The Verilog description uses 32-bit arithmetic (as in [6]), while Chisel infers the bit widths automatically and more accurately.

Optimization(s): The best solution is also a pipelined design with one IDCT^{row} and one IDCT^{col}. It is slightly inferior to Verilog: performance is 98.7% and area is 109.5%. However, the Chisel description was developed about 2–3 times faster.

Bluespec SystemVerilog (**BSV**) is a rule-based HDL [14]. As in other HDLs, descriptions are hierarchical, but modules are written differently. Each of them includes state elements and rules that modify the state. A rule is an atomic guarded action: the guard specifies the condition under which the rule is applied; the action describes the state modification. Compared to Verilog and Chisel, BSV offers higher timing abstraction: the programming paradigm implies the one-rule-at-a-time semantics, but a compiler may optimize the design and schedule multiple rules within a clock cycle.

We have created two designs and synthesized 26 circuits with **Bluespec Compiler** (**BSC**) by varying the tool options and code attributes. It has been shown that the settings have a negligible impact on the performance and area (at least for our example); the data below are for the default configuration.

Initial Design: At first, we manually translated the original C program [6] to BSV. It turned out that the resulting implementation is slightly better than the initial design in Verilog: the performance is 110.3%, while the area is 97.2%.

Optimization(s): Although the optimized version is similar to ones in Verilog and Chisel, its quality is slightly worse: the performance is 80.2%, and the area is 107.1% (compared to Verilog). The main reason for the lower performance is that the periodicity is one cycle higher (9 instead of 8). In theory, this "bubble" could be eliminated, which would make the characteristics comparable to Chisel's.

DSLX is a dataflow-oriented functional language for designing computational kernels [36]. It mimics Rust, while extending the latter with hardware-specific features (fixed-size objects and fully analyzable call graphs). Supported data types include bit vectors, tuples, structures, and arrays. It is worth highlighting the following facilities: parametric structures and functions, type inference, and `for` expressions ("loops" with known number of iterations). DSLX ignores timing issues, allowing a compiler to schedule computations.

We have adapted an existing IDCT example [7] (changed the bit widths of input/output matrix elements) and synthesized 19 implementations with **XLS** by varying the following options: (a) the circuit type (combinational or pipelined) and (b) the number of pipeline stages.

Initial Design: The starting point is a combinational circuit with a hand-crafted AXI-Stream adapter. This implementation is noticeably better than the initial designs mentioned above:

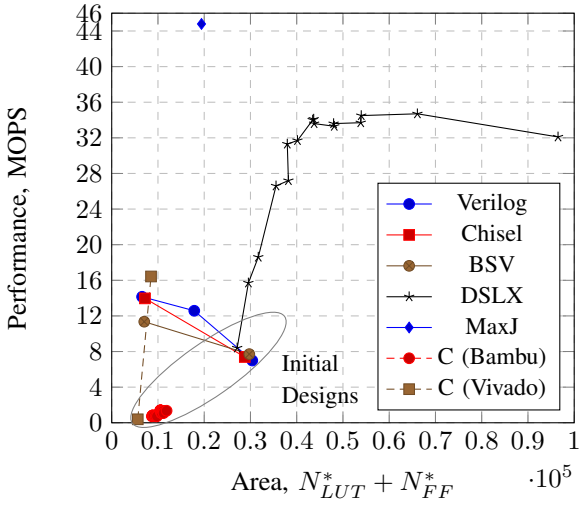

Fig. 1. Design space exploration for IDCT

the performance is 120.3%, and the area is 89.2% (compared to the initial Verilog description).

Optimization(s): Our experiments have shown that the maximum quality is achieved when the number of pipeline stages is set to 8 (for unknown reasons, operation in this case takes 3 cycles): the performance is 221.2%, and the area is 578.1% (compared to the optimized Verilog design). The quality score of 38.3% does not look good; the problem, again, is in the sequential adapter (the throughput could be 8 times higher).

MaxJ is an imperative-style dataflow language for developing high-performance systems [37]. A system description is split into three parts: computational kernels, a manager (connecting kernels to the CPU, RAM, and other kernels), and software. MaxJ is about kernels and a manager. It is based on Java and can be viewed as a means of constructing dataflow graphs with the following types of nodes: values (constants and runtime parameters), computation nodes (arithmetic and logic operations), offsets (access to past and future elements of data streams), multiplexers (decisions), counters (looping), and inputs/outputs (data stream connections).

Unlike the other tools evaluated here, **MaxCompiler** allows developing a whole system, not only an FPGA part. It provides an API and a runtime library for transferring data to kernels and controlling computations. The software and hardware parts are linked through the PCIe interface. So, it is not quite right to compare MaxJ designs to the other implementations.

Initial Design: First we developed a kernel that can input and output an 8×8 matrix every clock cycle. The synthesized implementation has a pipeline of 47 stages and operates at 403.13 MHz, the highest frequency among all of the designs. The bottleneck here is the PCIe bus. The throughput has been estimated as the bandwidth of the PCIe 3.0 x16 interface (about 16 GB/s) divided by the input data size (1024 bits). Although the implementation consumes a lot of resources, its quality is superior to the handwritten Verilog's (963%).

Optimization(s): To reduce the area, we have developed another kernel that receives a matrix row at each clock cycle and stores intermediate results in the on-board memory. In contrast to the initial design, the performance here is limited by the frequency. The occupied area is roughly 2.8 times less than of the previous design, the throughput is 2.7 times lower, and the quality is 4% higher.

C is a well-known imperative programming language developed in the 70s [38]. It offers only a single-thread execution model, making parallel hardware design synthesis challenging.

We have made experiments with two C-to-HDL compilers: the open-source **Bambu** and the commercial **Vivado HLS**. In both cases, the original C code [6] has been slightly modified: rounding in IDCT^{col} is implemented as a function (`iclip`), not a pre-filled array [9]. Bamboo, unlike Vivado HLS, cannot generate an AXI-Stream adapter; that component has been developed manually in Verilog.

Bambu has a rich set of options covering all HLS stages: scheduling, operation binding, memory allocation, etc. Our efforts have been concentrated on the `experimental-setup` presets aimed at different optimization criteria: performance, area, and something in between. As we have seen, most of the options do not have a tangible impact on the design quality. During evaluation, we have tried 42 configurations.

Initial Design (Bambu): We started with the following configuration: `channels-type=MEM_ACC_11` (one read port and one write port) and `memory-allocation-policy=LSS` (local/static variables and strings are allocated in BRAMs). Despite the relatively high frequency (471.4% compared to the initial Verilog design), the implementation, being sequential, has the low performance (11.7%). The small area (29.2%) does not save the situation.

*Optimization(s) (Bambu):*The best quality has been achieved on the preset `BAMBU-PERFORMANCE-MP` (two read ports and two write ports in the memory) in conjunction with the following options: `speculative-sdc-scheduling` and `memory-allocation-policy=LSS`. The results obtained are close to the initial ones and are significantly inferior to the ones of the other optimized designs: the performance is 9.8% and the area is 160.1% (compared to Verilog).

In Vivado HLS, the synthesis is controlled by preprocessor directives (pragmas). There are ones for pipelining, function inlining, loop unrolling, array optimization, structure packing, interface synthesis, etc. It is worth noting that the tool automates interface generation. To make the AXI-Stream adapter, we have added an extra function and applied `#pragma HLS INTERFACE axis port=`$\langle name \rangle$.

Initial Design (Vivado HLS): The implementation constructed in the push-button mode is not good: the throughput is almost 18 times lower compared to the initial Verilog design. The main problem is that the tool does not inline the IDCT^{row} and IDCT^{col} units and, moreover, generates superfluous AXI-Stream interfaces to communicate with them.

Optimization(s) (Vivado HLS): To overcome the problem, the source code has been modified: `short buf[8]` in the IDCT^{row} and IDCT^{col} functions has been replaced with `short buf0, ..., short buf7`; in addition, `#pragma HLS PIPELINE` has been injected. As a result, the quality has become close to the optimized Verilog design (89.7%): the performance is 116.1% and the area is 129.5%.

The data obtained in our study are represented in Fig. 1 and Table II. The figure shows the design space exploration in the $Performance \times Area$ space. The table contains details on the initial and optimized designs developed with different HLS/HC tools. The best characteristics are highlighted in bold. The highest *automation* is provided by MaxCompiler and Vivado HLS. The former is a more specialized tool, while the latter suits as a better solution for rapid prototyping. Speaking about the *controllability* (and the *quality* of the resulting designs), we should highlight Chisel and BSC. Vivado HLS also demonstrates good results. MaxCompiler controllability of more that 100% is due to higher bandwidth of PCIe comparing to AXI-Stream. The most *flexible* tools are XLS and (again) Vivado HLS, though they are configured differently: the former utilizes only one parameter (the number of pipeline stages), while the latter uses source code pragmas.

The resource utilization for the IDCT benchmark is low due to the simplicity of its design. We are confident that our results cannot be easily extrapolated to more complex benchmarks. During FPGA bitstream generation PR tools are used that can perform differently if near all available resources are used. However, our goal is not to test PR; it is about HLS/HC transformations being carried out before the LS/PR.

V. Conclusion

In this work, we considered a number of HLS/HC tools: Chisel, BSC, XLS, MaxCompiler, Bambu, and Vivado HLS. For each of the tools, we have evaluated the ease of creating an FPGA-based implementation of 8×8 IDCT compared to manually writing Verilog code, the quality of that implementation (performance-to-area ratio), the flexibility (how easy it is to optimize the quality), and the controllability (to what extent it is possible). Our experiments provide helpful tips (which tool to use for best performance, resource utilization, etc.) and are a "snapshot" of tool choices today. Explaining the differences in metrics output would require deeper insights of the considered tools (a topic for future research).

Our study shows that the most balanced solution (among those considered) is the commercial Vivado HLS. However, we believe that the future lies in open-source and extensible frameworks. The concept of such a tool can be sketched as follows. At the top level, there are *domain-specific languages* and related translation and optimization engines. Next comes a *general-purpose language* oriented towards parallel computing (e.g., MaxJ) and the corresponding intermediate representation (computation graph). Individual units (nodes) can be designed using various *lower-level tools*, both universal (XLS, Chisel, BSV, Verilog, etc.) and specialized (e.g., FloPoCo). An important feature is the ability to generate external and internal interfaces. The development of such open source HLS/HC framework is the main direction of our future work.

Language	**Verilog**		**Chisel**		**BSV**		**DSLX**		**MaxJ**		**C**			
EDA Tool	**Vivado**		**Chisel**		**BSC**		**XLS**		**MaxCompiler**		**Bambu**		**Vivado HLS**	
Configuration	*Initial*	*Opt*	*Initial*	*Opt*	*Initial*	*Opt*	*Initial*	*Opt*	*Initial*	*Opt*	*Initial*	*Opt*	*Initial*	*Opt*
LOC, including options	247	316	195	222	238	199	242	243	**121**	163	183	191	125	130
Modification, ΔL	258		131		434		**3**		231		29		71	
Automation, α	0%	0%	21.1%	29.8%	3.6%	37.0%	2.0%	23.1%	51.4%	48.4%	25.9%	39.6%	49.0%	**58.9%**
Quality, $Q = P/A$	230	**2,155**	257	1,942	259	1,614	310	825	2,215	**2,308**	91	132	69	1,933
Controllability, C_Q	**100%**		90.1%		74.8%		38.3%		**100% [107.1%]**		6.1%		89.7%	
Flexibility, F_Q	7.5		12.9		3.1		**171.7**		0.4		1.4		26.3	
Frequency, MHz	55.88	113.21	59.15	111.77	100.25	102.18	67.30	250.50	**403.13**	**403.13**	263.44	257.33	132.61	131.46
Throughput, MOPS	6.99	14.15	7.39	13.97	7.71	11.35	8.41	31.31	**123.08**	44.79	0.82	1.39	0.39	16.43
Latency, cycles	**17**	24	**17**	24	21	26	**17**	19	47	60	323	185	340	26
Periodicity, cycles	**8**	**8**	**8**	**8**	13	9	**8**	**8**	1	9	323	185	340	**8**
Area, $N^*_{LUT} + N^*_{FF}$	30,396	6,567	28,778	7,194	29,549	7,036	27,127	37,965	55,580	19,413	8,879	10,514	**5,633**	8,501
N^*_{LUT} (maxdsp=0)	29,059	**3,909**	27,441	4,530	27,565	4,781	25,805	26,960	19,704	5,941	5,443	7,134	4,103	4,974
N^*_{FF} (maxdsp=0)	1,337	2,658	1,337	2,664	2,184	2,255	**1,322**	11,005	35,876	13,472	3,436	3,380	1,530	3,527
N_{LUT}	13,850	**2,106**	9,283	2,205	26,560	4,643	25,805	26,010	19,704	5,941	5,090	6,614	2,334	3,123
N_{FF}	1,337	2,658	1,337	2,661	2,184	2,255	**1,322**	10,717	35,876	13,472	3,436	3,156	1,481	3,346
N_{DSP}	160	20	184	23	40	4	0	16	384	48	5	9	22	19
N_{IO}	172	170	172	172	174	172	170	170	59	59	174	174	270	267

TABLE II
HLS/HC TOOLS EVALUATION RESULTS

ACKNOWLEDGMENT

As a member of the Maxeler University Program MAX-UP, our team would like to thank Maxeler Technologies.

REFERENCES

[1] N. M. Botros, HDL Programming Fundamentals: VHDL and Verilog. Charles River Media, 2005. 506 p.

[2] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," IEEE Des. Test., vol. 26, no. 4, pp. 8–17, Jul.-Aug. 2009.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a Scala embedded language," in Proc. Des. Autom. Conf. (DAC), Jun. 2012, pp. 1216–1225.

[4] IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform, IEEE Std 1180-1990, March 18, 1991.

[5] Information technology – Digital compression and coding of continuous-tone still images: JPEG File Interchange Format (JFIF) – Part 5: ISO/IEC 10918-5:2013, May 5, 2013.

[6] MPEG-2 decoder from ISO/IEC 13818-4:2004. [Online]. Available: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_13818-4_2004_Conformance_Testing/Video/verifier/mpeg2decode_960109.tar.gz

[7] XLS. [Online]. Available: https://github.com/google/xls

[8] Bluespec Compiler. [Online]. Available: https://github.com/B-Lang-org/bsc

[9] Repository with the results of this work. [Online]. Available: https://github.com/ispras/hls-idct

[10] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," IEEE Des. Test., vol. 28, no. 4, pp. 18–27, Jul. 2011.

[11] Chisel. [Online]. Available: https://github.com/chipsalliance/chisel3

[12] MyHDL. [Online]. Available: https://www.myhdl.org

[13] P. Bellows and B. Hutchings, "JHDL-an HDL for reconfigurable systems," in Proc. IEEE Int. Symp. Field-Program. Cust. Comput. Mach. (FCCM), Apr. 1998, pp. 175–184.

[14] *Bluespec SystemVerilog Reference Guide*, Bluespec, Inc., Framingham, MA, USA, July 21, 2017.

[15] Bambu. [Online]. Available: https://github.com/ferrandi/PandA-bambu

[16] LegUp. [Online]. Available: https://www.legupcomputing.com

[17] *Vivado Design Suite User Guide. Implementation*, Xilinx, San Jose, CA, USA, UG904 (v2020.2), February 26, 2021.

[18] *Vivado Design Suite User Guide. Implementation*, Xilinx, San Jose, CA, USA, UG904 (v2017.4), December 20, 2017.

[19] *Vitis High-Level Synthesis User Guide*, Xilinx, San Jose, CA, USA, UG1399 (v2020.2), March 22, 2021.

[20] Intel FPGA SDK for OpenCL. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html

[21] Intel oneAPI. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

[22] MaxCompiler. [Online]. Available: https://www.maxeler.com/products/software/maxcompiler

[23] HDL Coder. [Online]. Available: https://www.mathworks.com/products/hdl-coder.html

[24] Vitis AI. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html

[25] L. Kalms, A. Podlubne, and D. Göhringer, "HiFlipVX: an Open Source High-Level Synthesis FPGA Library for Image Processing," in Proc. Int. Symp. App. Reconf. Comp. (ARC), Apr. 2019, pp. 149–164.

[26] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," Des. Autom. Embed. Syst., vol. 16, pp. 31–51, Sep. 2012.

[27] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," Adv. Syst. Sci., vol. 240, pp. 483–492, Jan. 2014.

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

[29] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in Proc. IEEE Int. Symp. C&S. (ISCAS), May 2008, pp. 1192–1195.

[30] W. H. Chen, C. H. Smith, and S. C. Fralick, "A fast computational algorithm for the discrete cosine transform," IEEE Trans. Commun., vol. 25, no. 9, pp. 1004–1009, Sep. 1977.

[31] Z. Wang, "Fast algorithm for the discrete W transform and for the discrete Fourier transform," IEEE Trans. Acoust. Speech Signal Process., vol. 32, no. 4, pp. 803–816, Aug. 1984.

[32] Information technology – Generic coding of moving pictures and associated audio information – Part 4: Conformance testing, ISO/IEC 13818-4:2004, December 2004.

[33] *AMBA 4 AXI4-Stream Protocol Specification*, ARM, Cambridge, UK, ARM IHI 0051A (ID030610), March 03, 2010.

[34] *Vivado Design Suite Reference Guide. Model-Based DSP Design Using System Generator*, Xilinx, San Jose, CA, USA, UG958 (v2018.1), April 4, 2018.

[35] IEEE Standard for Verilog Hardware Description Language, IEEE Std 1364-2005, April 7, 2006.

[36] DSLX Reference. [Online]. Available: https://google.github.io/xls/dslx_reference

[37] *Multiscale Dataflow Programming*, Maxeler Technologies, London, UK, Version 2021.1, May 14, 2021.

[38] Information technology – Programming languages – C, ISO/IEC 9899:2018, June 2018.