Leveraging Modern C++ in High-Level Synthesis

Sakari Lahti, Matti Rintala, and Timo D. Hämäläinen, *Member, IEEE*

***Abstract*—High-level synthesis (HLS) enables the automated conversion of high-level language algorithms into synthesizable register-transfer level code, allowing computation-intensive algorithms to be accelerated on FPGAs. Most HLS tools have C++ as their input language, as it is widely known in both software and hardware industry. However, even though C++ receives a new standard every three years, the HLS tool vendors have mostly provided support and examples using C++98/03. Limiting to early C++ standards imposes a productivity penalty, since the newer standards provide both compilation time reductions and more concise, expressive, and maintainable way of writing code. In this study, we make the case for adopting modern C++ in HLS. We inspect the language features of C++11 and forward, and consider their benefits for HLS. We also test the present support for the modern language features with two state-of-the-art commercial HLS tools. Finally, we provide an extended example, demonstrating the increased clarity of code achieved using the newer standards. We note that the investigated HLS tools already have good support for modern C++ features, and urge their adoption to increase designer productivity.**

***Index Terms*—Algorithms implemented in hardware, C++ language, high-level synthesis (HLS), reconfigurable hardware.**

I. Introduction

FOR MORE than 30 years, register-transfer level (RTL) methods have been the dominant way to describe and verify digital circuits and systems with languages, such as VHDL and Verilog. These languages have evolved rather slowly, especially if compared to the advancements in software programming languages during the same time period. While robust, the RTL languages require special expertise and have limited support for many of the features that have enabled productivity increases in the software domain. For these reasons, accelerating computation-intensive parts of algorithms in CPU+FPGA co-systems has been out of reach for most software engineers.

High-level synthesis (HLS) promises to bridge the gap between the algorithmic design style and RTL synthesis by transforming the high-level behavioral description into the RTL code [1]–[3]. This mapping onto the hardware description is directed by the user who selects the amount of parallelism for loop iterations, implementation of arrays on memory components, and so forth. The HLS design methodology promises to increase productivity by skipping over the time-consuming step of manually converting the behavioral high-level model into RTL code and then verifying it. Indeed, this productivity increase has been demonstrated in several studies as surveyed by Lahti *et al.* [4]. In the ideal case, the behavioral description could be used directly as the input for the HLS tool. Consequently, someone with only software background could implement their algorithms either partly or completely on FPGAs without significant extra knowledge required [5]. This is especially attractive for algorithms that can benefit from the massive parallelization made possible by FPGA circuits.

However, this kind of ideal case is hampered by a few facts. First, the HLS tools are not sophisticated enough to produce efficient hardware structures from a completely behavioral description without any input about the intended hardware architecture [6]–[8]. The user must infer hierarchy, provide efficient communication and memory handling, and adapt bit-accurate data types in ways that are unfamiliar to a software engineer. The work in [9] demonstrates the scale of transformations that are needed. Second, even after these required transformations, the micro-architectural design space exploration (DSE) is a time-consuming task, which greatly affects the quality of the results of the final product [10], [11]. HLS accelerates this step compared to manual RTL coding by enabling different DSE options with pragmas and GUI options, but finding the pareto-optimal solution front is not trivial.

This article concentrates on the third reason that hinders direct automated algorithm-to-RTL conversion: while many HLS tools use widespread software programming languages as their input language, they usually limit the user on what language features they can use [12]. A salient example is C++, which is the most widely used input language in commercial HLS tools, as it is a well-known language in the embedded systems design field [7], [13], [14]. However, most HLS tools forbid the use of dynamic memory allocation, recursion, function pointers, and large portions of the standard template library (STL). Especially, dynamic memory handling and the use of STL are ubiquitous in software C++ programming and most software engineers would feel hindered without access to them. Removing these structures from the source code is an involved task [15].

C++ is also a rapidly developing language with new standards being published every three years. However, the HLS vendor given code examples and function libraries usually demonstrate a quite C-like coding style with perhaps classes and templates added. This begs the question, do the tools have robust support for features introduced in C++11 and beyond, even when promised in the tool's user guide. Omitting modern C++ features not only reduces designer productivity and limits the language's expressiveness but also further alienates software engineers from adopting HLS. This problem with HLS has been only very rarely discussed in prior research articles. da Silva *et al.* [16] proposed a C++11

Manuscript received 21 April 2022; accepted 1 July 2022. Date of publication 25 July 2022; date of current version 21 March 2023. This article was recommended by Associate Editor Z. Zhang. *(Corresponding author: Sakari Lahti.)*

The authors are with the Unit of Computing Sciences, Tampere University, 33720 Tampere, Finland (e-mail: sakari.lahti@tuni.fi).

Digital Object Identifier 10.1109/TCAD.2022.3193646

driven methodology for HLS using Xilinx Vivado HLS, but to the best of our knowledge, no other studies have widely experimented with modern C++ language features in HLS. The motivation of this study is therefore to promote the usage of modern C++ with the following contributions.

1) We go through the most important features of modern C++ and discuss their relevance for increasing HLS productivity.
2) We explore the present support of two widely used commercial HLS tools for the modern C++ features.
3) We then present a code example demonstrating some of the benefits of adopting modern C++.
4) Finally, we give suggestions for the HLS tool users and developers based on the study.

This article does not delve into testing the support of various STL containers and functions in HLS, as there are hundreds of them, warranting a dedicated paper.

The remainder of this article is structured as follows: Section II presents the prominent features of C++11 and newer standards and considers their use in HLS, after which Section III explores their support in two state-of-the-art HLS tools. Next, Section IV provides a motivating example for adopting modern C++ in HLS, and finally, Section V concludes this article with discussion based on the results.

II. Features of Modern C++

In this section, we go through the major language features of C++ revisions 11, 14, 17, and 20 in alphabetical order. We shortly introduce the features and discuss their relevance to hardware description based on our tests (Sections III and IV) and analytical considerations. Often, the benefit in HLS is the same as in software programming: more expressive and readable code or reduced compilation times. In these cases, we have usually not explicitly repeated the fact when considering the feature.

We will see that many of the features involve template classes and functions. These can be used to make HLS code more generic with respect to the type and number of IO parameters and internal storage elements. As this is of tremendous use in hardware design, we emphasize features that make using templates easier or more versatile.

The usage details of the features are only sparingly discussed, so we refer the reader to other sources in regards to them (e.g., [17]). Furthermore, some language features only make sense in the context of software programming, by referencing concepts, such as dynamic memory allocation or call stack. We will therefore omit those from our discussion. Finally, some minor features and changes to the language have been omitted from the discussion as well.

A. C++11

1) Attributes: Traditionally, compiler and tool-specific information about code has been provided with the `#pragma` directive. C++11 introduced the concept of attributes, which are syntactically provided within double square brackets: [[*attribute*]]. Attributes can target separate elements of code, whereas pragmas can only target entire lines of code.

```
int accum(std::initializer_list<int> list)
{
    int res = 0;
    for (auto& p : list) // range-based for-loop
    {
        res += p;
    }
    return res;
}

int sum = accum({x0, x1, x2});
```

Fig. 1. Using initializer list to add together an arbitrary number of variables.

HLS tools commonly use pragmas to direct the synthesis. For example, the top-level component is often indicated by a pragma, and whether a loop should be unrolled or not. The HLS tool vendors should replace pragmas with the more modern and versatile C++ attributes. We did not test this feature for this article, since the studied HLS tools still use pragmas instead of attributes.

2) Constexpr: `Constexpr` is a very helpful specifier that tells the compiler that a function can possibly be executed at compile time. In hardware, this saves resources and computation time, as the result can be stored as a constant value in a register without any computation logic. Moreover, `constexpr` functions can be used to replace compile-time template recursion to generate certain parallel hardware structures, making the code more readable and concise. The example in Section IV demonstrates this.

The usage of C++11 `constexpr` functions was rather limited in that they could not contain many ubiquitous control statements, such as `if`, `switch`, and `for`. Furthermore, local variables were disallowed. These restrictions were lifted in C++14.

3) Extern Templates: A template can be defined with the keyword `extern` to prevent its instantiation in a translation unit. This can be used to reduce compilation time if the same template is instantiated with the same arguments in another translation unit in the same project. There is no effect on the synthesized hardware, regardless.

4) Initializer Lists: Initializer lists allow building constructors and other functions that take {}-lists as arguments. This is convenient in creating classes and functions that are agnostic about the number of input arguments of the same type. The number of arguments is decided only when creating an object or calling a function. The usage requires the `std::initializer_list` template, and is demonstrated in Fig. 1.

In HLS, initializer lists provide the ability to infer functional units with a variable number of ports from a single function. It should be noted that the number of elements in the list must be determinable at compile time upon each function call to such a function.

5) Lambda Expressions: Lambda expressions implement anonymous functions in C++. These are functions that do not have a name and are often used as arguments for higher order functions. A common reason to use them is to avoid populating code with small separate functions that are only called once. Lambdas are especially useful with many STL

functions, but they also have utility in HLS-oriented coding as shown in the example of Section IV. An example of using lambdas to implement synthesizable recursion can be found in [18].

6) Range-Based for Loop: Range-based for loop gives a simpler syntax for iterating over each element in an array, initializer list, or container with `begin` and `end` functions. A usage example can be found in Fig. 1. It is good practice to use a reference with a range-based for loop range declaration. This prevents copying the values in the iterated list for the loop. Instead they are accessed from the list directly, saving memory in simulation. For synthesized hardware, this can mean a difference between copying the values to a separate register or accessing the values directly from the registers or memory where the list is stored.

7) Rvalue References, Move Constructors, and Perfect Forwarding: C++11 introduced *r*value references primarily to allow move semantics without creating costly deep copies when objects are passed by value [19, pp. 193–196]. As temporary objects are not an issue with move semantics on hardware, this usage of *r*value references is not a relevant feature for HLS, except during algorithm development and RTL/C++ co-simulation. It would still be convenient for the HLS tool to allow them to reduce the number of needed modifications between the software algorithm and the HLS source code. The hardware implementation should be the same for normal references and *r*value references.

*R*value references are also used to allow perfect forwarding, which is useful, for example, in creating flexible constructors (factory methods) [20]. Perfect forwarding passes template function arguments to a subfunction retaining their *l*value or *r*value nature. In C++, this also requires support for the `std::forward` function. Perfect forwarding is a convenience feature for generic programming and should be supported by HLS tools, as template functions and classes are often employed to instantiate components.

8) Static Assertions: Static assertions, using the declaration `static_assert` allow checking for assertions at compile time. This is especially useful for testing template argument properties. Also, compiler-specific assumptions, such as the size of various standard data types can be tested with static assertions. A usage example can be found in the extended example in Section IV. Assertions are one of the most important tools in both software and hardware verification, so compile-time support for them is of great utility.

9) Strongly Typed Enumerations: Strongly typed enumerations allow for safer, more portable, and more flexible enumeration types. From C++11 forward, enumeration types should be declared with the `enum class` keywords to use the strongly typed enumerations. This prevents declaring the enumerators in the enclosing scope, which is usually undesirable. Another benefit is that the underlying type of enumerations can now be any integral type instead of just *int*. Bit widths to represent the enumerators can thus be reduced manually in case the HLS tool does not optimize them automatically.

10) Type Aliases and Alias Templates: C++11 introduced type aliases with the `using` keyword that can be used much in the same way as the old `typedef` specifier. They have no difference in semantics. However, the `using` keyword allows declaring alias templates, whereas `typedef` does not. Alias templates allow creating, for example, partially bound templates from previous template definitions, which is a beneficial feature in generic HLS code that can be template-heavy. An example can be found in Section IV.

11) Type Inference (`auto`, `decltype`): The `auto` keyword allows the compiler to deduce the type of a variable. This is beneficial when the type is determined by, for example, the return value of a template function, saving programmer effort. The verbosity of the code is also reduced by using `auto` instead of some long type name. The `decltype` specifier, on the other hand, can be used to determine the type of an expression at compile time, which is especially useful in determining the actual type of `auto` variables. Again, the template-rich HLS code will greatly benefit from these features. A usage example is in Section IV.

12) Variadic Templates: Traditional C++ templates allowed for a fixed number of template arguments. Variadic templates, on the other hand, allow an arbitrary number of template arguments, i.e., zero or more. This makes templates even more flexible in generic programming. For example, a generic matrix class could be implemented with a variable number of dimensions in the following manner:

```
template<typename T, size_t dim0, size_t... dims>
class Matrix.
```

Here, *T* is the data type of the elements and `size_t...` `dims` represents the arbitrary number of dimensions and their sizes. (The second parameter makes sure that there is at least one dimension.) Due to the usefulness of generic programming in hardware description, the support for variadic templates is recommended for any modern HLS tool.

B. C++14

1) Binary Literals: Binary-valued literals can be declared using the prefixes 0b and 0B. This is useful in conjunction with bit-accurate data types, allowing arbitrary-length bit vectors akin to std_logic_vectors in VHDL, which are ubiquitous in that language. Decimal-valued literals should usually be preferred for readability reasons, but binary values can be a better choice with, for example, bit vectors related to control signals, where decimal interpretation would be meaningless.

2) Function Return-Type Deduction: C++14 provided the convenient ability for functions to automatically deduce their return type based on the return statement. Such a function is denoted by using the keyword `auto` as its return type. The usage of this is demonstrated in the example of Section IV. This is one of the features that make using templates easier, as they can produce complex return types.

3) Generic Lambdas: C++11 demanded lambda function parameters to be declared with explicit types, but C++14 relaxed this by allowing type deduction with the *auto* keyword. These types of lambdas are demonstrated in Section IV.

4) Variable Templates: Variable templates allow variables whose type is a template parameter that can be determined

```
int my_abs(int x)
{
    return (x < 0 ? -x : x);
}

template<typename... Args>
int abs_sum(Args&... args)
{
    return (my_abs(args)+ ...);
}
```

Fig. 2. Summing absolute values with fold expressions.

upon instantiation. An example would be declaring a variable template for the constant pi

```
template<class T> constexpr T pi = T(3.14159265358);
ac_fixed<16,8> area = pi<ac_fixed<11,3> > * r * r;.
```

HLS uses bit-accurate data types, so a variable of desired accuracy could be instantiated from this definition. This example uses the Siemens Algorithmic *C* library, with a fixed point value of `<X,Y>`, where *X* denotes the total bit width and *Y* the number of integer bits.

C. C++17

1) Class Template Argument Deduction: In C++17, a class constructor can deduce its type parameters from its constructor arguments. For example, `std::pair(5, true)` can be used instead of `std::pair<int, bool>(5, true)`. In HLS, this feature should be used with care to avoid accidentally inferring non bit-accurate data types that waste more bit width than is necessary to represent the intended value range.

2) Constexpr If: The `constexpr` specifier (Section II-A2) was expanded in C++17 to allow conditional compiling and compile-time calculations with the `if constexpr(condition)` structure. This can be combined with an `else if/else` structure as with normal conditional statements. Example utilization of this useful feature can be found in Section IV. This kind of conditional compiling is more readable than using preprocessor directives, so it should be used when possible.

3) Fold Expressions: C++11 introduced variadic templates (Section II-A12). C++17 expanded their usefulness with fold expressions, which allow to repeat an operator or function over a variadic template pack. Fold expressions allow many convenient code structures, such as the `abs_sum` function shown in Fig. 2 function that calculates the absolute value for an arbitrary number of parameters and sums the results.

The fold expression is indicated with the ellipsis notation. In this example, the addition operator in the body of the `abs_sum` function, followed by the fold expression, unpacks the argument list and applies the `my_abs` function on all the arguments. Using fold expressions avoids the need for passing arguments in arrays and iterating over the elements. In HLS, the number of parameters used should be determinable at compile time upon each invocation, as hardware resources cannot be reserved dynamically.

4) Initializers For If and Switch: C++17 makes it possible to give an initial statement within if and switch statements in the manner of

```
if (init-statement; condition) {...}.
```

The init-statement can be, for example, a function call that initializes a variable with a value that is used in the if condition. The benefit of this feature is to simplify some common code patterns and prevent variables from leaking outside their scope. Without the init-statement, a return value from a function that is only used for the if condition is also seen outside the scope of the if statement. With C++17, this variable can be kept strictly within the scope of the if expression.

5) Inline Variables: In C++17, variables can be declared as inline, similar to how the `inline` specifier is used for functions. Inline variables ease defining global constants that are used in multiple compilation units. The `inline` specifier informs the linker that only one instance of the variable should exist, even in the case that the variable is present in multiple compilation units. Inlining helps avoid a set of workarounds that used to adversely affect either code readability or performance.

6) Structured Bindings: Structured bindings allow the ability to declare multiple variables initialized from an array or nonunion class type, which makes code cleaner and easier to understand. For example, with array `int arr[3] = {0, 1, 2};` separate variables could be initialized to the corresponding elements of the array using structured binding: `auto[x0, x1, x2] = arr;`.

7) Template Parameters Declared With Auto: Since C++17, compilers can deduce the type of nontype class template parameters. This means that `template <auto N> class MyClass` is valid, and the type of N will be deduced upon instantiation. This will help make templatized code more concise and understandable.

D. C++20

1) Coroutines: Coroutines are defined as functions that can be suspended and resumed later. C++20 enables the creation of such functions with the `co_yield`, `co_return`, and `co_await` keywords that each infer a coroutine when encountered in a function body. A coroutine stores its state when suspended and continues from that state when resumed. In the software domain, the state is often stored in the heap from which space is dynamically allocated. In hardware, the implication is that the number of coroutine instances should be statically determinable and each instance should have its own register storage for state.

Coroutines provide interesting alternative ways to implement state machines that are so commonly encountered in digital logic. They can also be employed to perform nonblocking reads and create generators that produce elements of a sequence only when needed. The C++ support for coroutines is still developing, but they seem to be potentially a very useful addition when employed in HLS. The benefits in, for example, state machine description are beyond the scope of this article, but should be investigated.

2) Concepts: Concepts provide a way to name sets of behavioral constraints on a type. This allows type-checked generic programming up front, where previously programming errors would be caught only later at template instantiation. This results in more understandable compiler messages, the ability

to overload templates based on parameter type properties, and also faster compilation times. Due to the ubiquitousness of templates in HLS code, concepts will provide a productivity boost when supported by HLS tools. They do not have any implications on the generated hardware, being a compile-time checking mechanism.

3) Modules: Header files have been the standard way to share language objects between source code files since the original C language. However, there are many problems associated with them, such as increased compilation times with multiple inclusions, side effects of inclusion order, and multiple inclusion errors. Some of these problems have traditionally been solved with cumbersome workarounds, such as inclusion guards. The modules introduced in C++20 solve these problems by being compiled independently of the units that import them. Just as in conventional software programming, HLS benefits from compartmentalization of code and shareable libraries that can contain utility functions or whole IP blocks. Modules will therefore be of benefit to prevent the problems associated with header files.

4) Three-Way Comparison <=>: C++20 introduces three-way comparison with the so-called spaceship operator. The operation $a <=> b$ returns less than 0 if $a < b$, greater than 0 if $a > b$, and 0 if $a = b$. This operator reduces the need for boilerplate code when overloading comparison operators for self-defined classes. In hardware, the natural representation of a three-way comparison is a generic comparator component. The operator should be used with care as a generic comparator is relatively area hungry.

5) Consteval and Constinit: The `consteval` specifier is similar to the `constexpr` specifier in that it enables compile-time functions. The difference is that `consteval` ensures compile-time execution, whereas `constexpr` converts to run-time function if compile time is not possible. Using the new specifier prevents accidentally generating run time code that would be synthesized in hardware components.

The new `constinit` specifier can also be applied to variables to ensure compile-time initialization. The main use is to prevent the static initialization order fiasco during compilation [21].

III. HLS Support of Modern C++ Features

We tested the C++ features listed in Section II with two commercial, state-of-the-art HLS tools: Siemens Catapult HLS and Xilinx Vitis HLS, which ships with the Vivado synthesis tool suite [22], [23]. Officially, the language standard support extends to C++14 for the newest versions of both the tools. However, C++11/14 features are rarely seen in code examples provided by the HLS tool vendors, so there was some room for doubt about the reality of the support in practice.

In addition, we had the assumption that by choosing a front-end compiler that supports C++17, we could enable some features that are not officially supported. If the front-end compiler analyzes the source code and creates a language-agnostic intermediary representation of it, the HLS tool should not notice the usage of the modern features, unless it is explicitly checking the source code. We did not test the C++20 features, however, as even compiler support is still partially experimental. HLS tool support for C++20 should not be expected until few years from now.

A. Catapult HLS

We used Catapult HLS version 2021.1 running on Red Hat Enterprise Linux 7 to perform the tests. This version of Catapult officially supports C++14 standard with the following limitations: Dynamic memory allocation, recursion (except with template functions), system calls, and function pointers are prohibited. Most standard libraries as well as most features of the STL are unsupported as well. However, Catapult allows choosing a path to the used compiler, which may support newer C++ versions. We used the GCC 10.2.0 compiler, which has almost complete support for C++17. In the Catapult options we indicated using C++14, the newest version that could be selected. Therefore, we told Catapult we were using the C++14 standard but we were actually using a compiler that has support for C++17.

As a first step in the synthesis, Catapult uses Edison Design Group front-end compiler (version 6.1) to analyze the source code [24]. The front-end translates the program into a high-level, tree-structured intermediate language from which Catapult produces the source code information for synthesis. Since the front-end supports a newer C++ version than Catapult itself, we hypothesize that the front-end can hide modern C++ features from Catapult when it performs its transformations. This is likely the cause why Catapult does accept many C++17 features, as was demonstrated by our tests.

The synthesis target was the Catapult default (a 45-nm ASIC library) with 100-MHz clock frequency. The target should not affect the reported results, however, so it could be an FPGA as well. The tests were based on either the determinant example of Section IV or on simple functions demonstrating the feature. The synthesis results were checked with the Catapult design analyzer tool after running pure C++ and C++/RTL co-simulations in Catapult. We considered a simulation successful if Catapult was able to compile and simulate the C++ and the results were correct. The correctness of the generated RTL was then checked with a simulation against the C++ model.

Table I summarizes the test results. The first column lists the features discussed in Section II and the second column shows whether Catapult was able to perform C++ simulation ("Sim") and final RTL synthesis ("Syn") on the feature. In most cases, both simulation and synthesis were successful, but for some features, there was only partial or no support.

The *r*value references was the only C++11 feature that was not fully supported by Catapult. The C++ simulation produced correct results, but synthesis failed to include an adder for an operation where a variable value was added to an *r*value reference. All other C++11 and all C++14 features were fully supported. C++17 proved more problematic. Structured bindings, class template argument deduction, and template parameters declared with auto were not supported, but compilation errors were issued. Since these same features were accepted in a later test with Vitis HLS, we assume that the problem lies in the Edison Design Group front end.

TABLE I
C++11/14/17 FEATURE SUPPORT WITH CATAPULT AND VITIS HLS

Feature name (and C++ version)	Catapult	Vitis
Alias templates (11)	Sim+Syn	Sim+Syn
Constexpr (11)	Sim+Syn	Sim+Syn
Extern templates (11)	Sim+Syn	Sim+Syn
Initializer lists (11)	Sim+Syn	Sim+Syn
Lambda expressions (11)	Sim+Syn	Sim+Syn
Range-based for loop (11)	Sim+Syn	Sim+Syn
Rvalue references (11)	Sim	Sim+Syn
Static assertions (11)	Sim+Syn	Sim+Syn
Strongly typed enumerations (11)	Sim+Syn	Sim+Syn
Type aliases (11)	Sim+Syn	Sim+Syn
Type inference (11)	Sim+Syn	Sim+Syn
Variadic templates (11)	Sim+Syn	Sim+Syn
Binary Literals (14)	Sim+Syn	Sim+Syn
Function return type deduction (14)	Sim+Syn	Sim+Syn
Generic Lambdas (14)	Sim+Syn	Sim+Syn
Variable templates (14)	Sim+Syn	Sim+Syn
Class template argument deduction (17)	No support	Sim+Syn
Constexpr if (17)	Sim+Syn	Sim+Syn
Fold expressions (17)	Sim+Syn	Sim+Syn
Initializers for if and switch (17)	Sim+Syn	Sim+Syn
Inline variables (17)	Sim+Syn	Sim+Syn
Structured bindings (17)	No support	Sim+Syn
Template params. declared with auto (17)	No support	Sim+Syn

Despite this, we can still note that linking a modern compiler to Catapult HLS enables some language features that are not officially supported.

B. Vitis HLS

The Vitis HLS version used in the tests was 2020.2 running on Red Hat Enterprise Linux 7. Vitis promises support up to the C++14 standard, but with the same limitations as Catapult, prohibiting dynamic memory allocation, recursion, system calls, function pointers, and most of the STL. We directed Vitis to use the same GCC 10.2.0 compiler as with Catapult tests and ran all the same feature tests. First, we ran C++ simulation for the code in Vitis, then synthesized to VHDL, ran C++/RTL co-simulation, and finally checked the synthesis result in the Xilinx Vivado synthesis tool. The synthesis target was the Vitis default (xcvu11p-flga2577 FPGA chip) with a 100-MHz clock.

As Table I shows, Vitis HLS was able to simulate and correctly synthesize all the tested features, including those of C++17, which are not officially supported by Vitis. Considering that the used compiler was the same for both Catapult and Vitis, but Catapult did not accept all of the features during compilation, the difference is likely due to the Edison Design Group front end compiler. This tool used by the Catapult appears to restrict the usability of some modern C++ features even when the C++ compiler accepts them.

With Vitis, it seems that any modern C++ feature can be used when a new enough compiler is linked to the tool. We expect that C++20 features will similarly be well supported by Vitis when they become robustly backed by compilers.

IV. EXTENDED EXAMPLE

This section demonstrates some of the benefits to be gained by using modern C++ in the HLS source code. We show an algorithm adapted for HLS for calculating the determinant of an $n \times n$ matrix A using the Laplace expansion

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} A_{i,j} M_{i,j}$$

where $A_{i,j}$ is the element of the ith row and jth column of A, and $M_{i,j}$ is the determinant of the submatrix (called minor) that is extracted by removing the ith row and the jth column from A. This is a recursive formula, as it requires the calculation of the determinant of the minors.

Generally, HLS tools do not support recursion as it involves runtime memory allocation, but if the matrix size is known at compile time, then the recursion depth is also known. We should therefore be able to express the algorithm so that a fixed hardware architecture for $n \times n$ determinant calculation is generated for a given value of n. For example, for the 3×3 matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

the Laplace expansion along the first row gives

$$\det(A) = 1 \cdot (5 \cdot 9 - 6 \cdot 8) - 2 \cdot (4 \cdot 9 - 6 \cdot 7) + 3 \cdot (4 \cdot 8 - 5 \cdot 7).$$

We can see that all of the multiplications, additions, and subtractions of this formula are possible to do in parallel, and this holds true for any value of n assuming that there are enough arithmetic components available on the target platform. We demonstrate an implementation, which creates such a fully parallel determinant calculation and is generic with respect to n.

A. C++03 Implementation

In C++03, this requires using recursive template functions. The code is shown in Fig. 3. We utilize a matrix class whose implementation is not shown. The class simply stores a 2-D array and provides some matrix operations. In this example, we use only the constructor and a getter function for matrix elements. The matrix class is templatized with respect to the data type of the matrix elements. In this case, we use the bit accurate algorithmic C data types with bit width W and signedness S [25].

Line 49 starts the top-level function for the determinant block. The actual templatized determinant calculation function (line 42) is called from the top-level function. This function is templatized with respect to the data type of the matrix elements and the size of the matrix. However, we cannot use this function directly to implement the compile-time recursion. This is because the end of the recursion needs to be specialized for the value $n = 1$. This is complicated by the fact that the matrix element type is another template parameter, whose type should then also be mentioned for the specialization. This is infeasible, as the number of possible element types is practically limitless. To circumvent this problem, line 2 starts a template struct, which contains the actual recursive function that is called on line 45. This struct is templatized only with respect

```
1  // Helper struct for determinant calculation
2  template<int N_det>
3  struct determinant_helper
4  {
5    // Return the minor matrix of element (i,j) for determinant calculation
6    template<typename T>
7    static Matrix<T, N_det-1> get_minor(const Matrix<T, N_det>& matrix, int i, int j)
8    {
9      ... // Function implementation omitted
10   }
11
12   // Recursively calculate determinant
13   template <typename T>
14   static T do_determinant(const Matrix<T, N_det>& param)
15   {
16     T det_val = 0;
17     // Calculate determinant columnwise
18     for (int col = 0; col < N_det; col++)
19     {
20       T temp = param.getElement(0,col) *
21         determinant_helper<N_det-1>::do_determinant(get_minor(param,0,col));
22       (col & 1) ? temp = -temp : temp = temp; // Flip sign of every other cofactor
23       det_val += temp;
24     }
25     return det_val;
26   }
27 };
28
29 // Determinant specialization for 1x1 matrix
30 template<> struct determinant_helper<1>
31 {
32   // getMinor intentionally missing, no minor in 1x1 matrix
33   template<typename T>
34   static T do_determinant(const Matrix<T, 1>& param)
35   {
36     return param.getElement(0,0);
37   }
38 };
39
40 // Function for determinant calculation
41 // Uses helper struct to work around partial specialization
42 template<typename T, int N_det>
43 T determinant_calc(const Matrix<T, N_det>& param)
44 {
45   return determinant_helper<N_det>::do_determinant(param);
46 }
47
48 // Top-level function
49 void determinant(ac_int<W,S> input[N][N], ac_int<W,S> &result)
50 {
51   Matrix<ac_int<W,S>, N> input_mat(input); // Create Matrix object from the input array
52   result = determinant_calc(input_mat);
53 }
```

Fig. 3. Determinant calculation code using C++03 style.

to n. Template substitution is used to pass the datatype to the functions `get_minor` and `do_determinant` within the struct. The `get_minor` is a helper function used to extract a minor of a matrix given as a parameter (implementation not shown). The `do_determinant` function recursively calculates the determinant by calling itself on line 21. Finally, the `determinant_helper` specialization for 1×1 matrix is shown in line 30.

When synthesizing from this code, all the loops should be fully unrolled and the main loop pipelined with initiation interval of 1. This gives a fully parallel architecture with a throughput of 1 sample/clock cycle after pipeline ramp-up latency. Even though this coding style gives the desired result, it is complex and the employed template meta-programming is prone to errors. This coding style has been explored for HLS in [26]–[28]

B. C++17 Implementation

Fig. 4 shows the code listing for the same algorithm, where C++17 has been adopted. We can immediately see that the code is more concise than with C++03. The major change is that in addition to the top-level function (line 33), there is only one extra function, whereas the previous example contained an extra function along with two structs. The actual determinant calculation function starts on line 6, and it is only templatized with respect to the matrix size. The `constexpr` keyword on line 7 tells the compiler that this function can be evaluated

```
1  // Template alias for Matrix class
2  template<int WIDTH, int SGN, int SZ>
3  using matrix_t = Matrix<ac_int<WIDTH,SGN>, SZ>;
4
5  // Compile-time function for determinant calculation
6  template<auto N_det> // Template argument deduction
7  constexpr auto determinant_helper()
8  {
9    static_assert(N_det < 10, "Matrix size too large!");
10   if constexpr (N_det == 1) // Recursion end
11   {
12     return [](auto const& param){ return param.get_element(0,0); };
13   }
14   else
15   {
16     return [](auto const& param) // Recursively calculate determinant
17     {
18       auto det_val = 0;
19       // Calculate determinant columnwise
20       for (int col = 0; col < N_det; col++)
21       {
22         // Extract the minor matrix of element (0,col) ... omitted
23         auto temp = param.get_element(0,col) * determinant_helper<N_det-1>()(minor);
24         (col & 1) ? temp = -temp : temp = temp; // Flip sign of every other cofactor
25         det_val += temp;
26       }
27       return det_val;
28     };
29   }
30 };
31
32 // Top-level function
33 void determinant(ac_int<W,S> input[N][N], ac_int<W,S> &result)
34 {
35   matrix_t<W,S,N> input_mat(input);
36   result = determinant_helper<N>()(input_mat);
37 }
```

Fig. 4. Determinant calculation code using C++17 style.

at compile time, which ensures that no dynamic recursion is implied by it. Instead, the compiler will generate a tree-like structure of function calls that is implemented directly as concurrent logic on the target platform. The matrix element type has been discarded as a template parameter thanks to type inference and function return-type deduction.

The `constexpr if` structure within the function implements the recursion. It checks for the size of *N*_det in the template parameter and either returns the lambda in line 12 or 16. The first lambda, which terminates the recursion, returns just the single element in the 1×1 minor. The second lambda is returned for larger matrices. It extracts the minor and gives that as the parameter to the recursive call to `determinant_helper` in line 23.

Other modern C++ features used in this example are:

1) alias template in lines 2–3, which is used in line 35. This reduces code if objects from the Matrix class are often instantiated;
2) template parameter-type deduction in line 6 to reduce programmer effort;
3) static assertion in line 9 to ensure at compile time that the recursive algorithm is not used in too large matrices, which could cause excessive resource usage on the target platform because of the recursion.

We were able to simulate and synthesize this code with Vitis HLS. Catapult HLS accepted the code when the *auto* keyword in line 6 was changed to *int*, as Catapult does not support class template argument deduction, a C++17 feature.

It should be stressed that the benefit gained from using modern C++ is the clarity of code, ease of expression, faster compilation and simulation, and fewer bugs. Furthermore, DSE becomes easier with features, such as type aliases and deduction and *constexpr if*, which allow switching data types and design parameters with minimal code changes. All of these translate to better design and verification productivity, a vital metric for competitiveness. Performance, energy consumption, or area benefits on hardware are less likely, unless the compiler can generate a more efficient intermediary representation from the modern source code. For example, the determinant implementation produced the exact same microarchitecture and area score from both input codes.

V. Conclusion

In this article, we have briefly summarized the most salient features of modern C++ language and what their benefits for HLS are. We tested the features with two state-of-the-art commercial HLS tools and noted that even though code examples from the tool vendors usually omit using modern C++, the tools support it well. However, this may require bypassing the default compilers used by the tools and linking newer compiler versions instead, when going beyond officially supported language versions. This article has also made the case

for adopting the new language features by demonstrating the code simplification to be gained with an example application coded in both traditional and modern C++.

HLS promises to increase the productivity of digital system design and verification, and facilitate relatively straightforward porting of parts of application to be accelerated on FPGAs. This promise has been proved true in studies. However, in our experience, many hardware-oriented HLS tool users still write source code in C++98/03, whereas software-oriented users may have felt that the tools do not support the more modern language versions that they are used to. For the former group, we urge to familiarize with modern C++ and leverage it to increase productivity. The tool support is already largely present. For the latter group, we have shown in this study that they do not have to resort to early C++, but can employ most of the language features they are used to in modern C++. The HLS tool vendors, for their part, should ensure that their tools do not lag too much behind the C++ standard development. Furthermore, they should produce examples and tutorials that demonstrate the usage of the modern features, as opposed to being satisfied with traditional C-like coding style sugar-coated with classes and templates.

What is still missing from HLS is the ability to fully use the STL. The STL is used extensively by software programmers for its convenient data structures and functions. HLS tool users, on the other hand, must be content with C arrays and self-written functions, which is a major drawback for productivity. Those STL features that do not require dynamic memory handling may have support in HLS tools, but the list of these is usually missing from the tool manuals, and it is impractical for the user to try and test the support on a case-by-case basis. The HLS tool vendors should therefore make it one of their top priorities to provide their own implementations of STL constructs that take into account the limitation of not being able to dynamically reserve memory. The ultimate goal of HLS should be to make the transition from software code to HLS-targeted code as invisible as possible, which requires this kind of STL emulation.

REFERENCES

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

[2] H. Ren, "A brief introduction on contemporary high-level synthesis," in *Proc. IEEE Int. Conf. IC Des. Technol.*, 2014, pp. 1–4.

[3] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[4] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019.

[5] R. Venkatakrishnan, A. Misra, and V. Kindratenko, "High-level synthesis-based approach for accelerating scientific codes on FPGAs," *Comput. Sci. Eng.*, vol. 22, no. 4, pp. 104–109, Jul./Aug. 2020.

[6] J. Matai, D. Richmond, D. Lee, and R. Kastner, "Enabling FPGAs for the masses," 2014, *arXiv:1408.5870*.

[7] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, "Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains," *IEEE Access*, vol. 8, pp. 174692–174722, 2020.

[8] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *Proc. 9th IEEE Int. Conf. ASIC*, 2011, pp. 1102–1105.

[9] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1014–1029, May 2021.

[10] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, Oct. 2020.

[11] L. Ferretti, J. Kwon, G. Ansaloni, G. D. Guglielmo, L. P. Carloni, and L. Pozzi, "Leveraging prior knowledge for effective design-space exploration in high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3736–3747, Nov. 2020.

[12] Z. Zhang and D. Chen, "Challenges and opportunities of ESL design automation," in *Proc. IEEE 11th Int. Conf. Solid-State Integr. Circuit Technol.*, 2012, pp. 1–4.

[13] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in *Advances in Systems Science*, J. Swiatek, A. Grzech, P. Swiatek, and J. M. Tomczak, Eds. Cham, Switzerland: Springer Int., 2014, pp. 483–492.

[14] A. Takach, "High-level synthesis: Status, trends, and future directions," *IEEE Design Test*, vol. 33, no. 3, pp. 116–124, Jun. 2016.

[15] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, 2013, pp. 362–365.

[16] J. S. da Silva, F.-R. Boyer, and J. M. P. Langlois, "Module-per-object: A human-driven methodology for C++-based high-level synthesis design," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2019, pp. 218–226.

[17] "C++ Reference." 2021. [Online]. Available: https://en.cppreference.com/w/

[18] D. B. Thomas, "Synthesisable recursion for C++ HLS tools," in *Proc. IEEE 27th Int. Conf. Appl. Spec. Syst. Archit. Process. (ASAP)*, 2016, pp. 91–98.

[19] B. Stroustrup, *The C++ Programming Language*, 4th ed. Upper Saddle River, NJ, USA: Addison Wesley, 2013, pp. 193–196.

[20] "Std::Forward." 2021. [Online]. Available: https://en.cppreference.com/w/cpp/utility/forward

[21] "Static Initialization Order Fiasco." 2020. [Online]. Available: https://en.cppreference.com/w/cpp/language/siof

[22] "Catapult C++/Systemc Synthesis." Siemens. 2021. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/f

[23] "Vivado ML Editions." Xilinx, Inc. 2021. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[24] "Edison Design Group." Edison Design Group, Inc. 2021. [Online]. Available: https://www.edg.com/

[25] "Algorithmic C (AC) Datatypes." Mentor Graphics Corporation. Apr. 2021. [Online]. Available: https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf

[26] T. R. Mück and A. A. Fröhlich, "Toward unified design of hardware and software components using **C++**," *IEEE Trans. Comput.*, vol. 63, no. 11, pp. 2880–2893, Nov. 2014.

[27] D. Richmond, A. Althoff, and R. Kastner, "Synthesizable higher-order functions for C++," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2835–2844, Nov. 2018.

[28] M. Fingeroff, *High-Level Synthesis Blue Book*. Bloomington, IN, USA: Xlibris, 2010, pp. 137–140.

Sakari Lahti received the first M.Sc.(tech.) degree in engineering physics and the second M.Sc.(tech.) degree in computer engineering from the Tampere University of Technology, Tampere, Finland, in 2002 and 2014, respectively.

He was a Doctoral Researcher with the Tampere University of Technology from 2014 to 2017. He is currently a University Instructor with the Unit of Computing Sciences, Tampere University, Tampere. His current research interests include high-level synthesis of digital systems, and hardware and system-on-chip designing in general.

Matti Rintala received the M.Sc.(tech.) and Ph.D. degrees in computer science from Tampere University of Technology, Tampere, Finland, in 1995 and 2012, respectively.

He works as a University Lecturer with the Unit of Computing Sciences, Tampere University, Tampere. He also participates in the standardization of the C++ programming language. His current research interests include concurrency, generic programming, and exception handling.

Timo D. Hämäläinen (Member, IEEE) received the M.Sc.(tech.) and Ph.D. degrees in electrical engineering from the Tampere University of Technology, Tampere, Finland, in 1993 and 1997, respectively.

He is currently a Full Professor and the Head of the Unit of Computing Sciences, Tampere University, Tampere. He has authored more than 60 journal and 200 conference publications. He holds several patents. His current research and teaching activities include system-on-chip design methodologies and tools, as well as SoC architectures and systems.