Composable, Parameterizable Templates for High-Level Synthesis

Janarbek Matai, Dajung Lee, Alric Althoff and Ryan Kastner
Dept. of Computer Science and Engineering, University of California, San Diego
La Jolla, CA 92093, United States
{jmatai, dal064, aalthoff, kastner}@cs.ucsd.edu

***Abstract*—High-level synthesis tools aim to make FPGA programming easier by raising the level of programming abstraction. Yet in order to get an efficient hardware design from HLS tools, the designer must know how to write HLS code that results in an efficient low level hardware architecture. Unfortunately, this requires substantial hardware knowledge, which limits wide adoption of HLS tools outside of hardware designers. In this work, we develop an approach based upon parameterizable templates that can be composed using common data access patterns. This creates a methodology for efficient hardware implementations. Our results demonstrate that a small number of optimized templates can be hierarchically composed to develop highly optimized hardware implementations for large applications.**

I. Introduction

Field programmable gate arrays (FPGA) are seeing widespread adoption in applications, including wireless communication, image processing, and data center. Despite the benefits of FPGA, programming an FPGA largely requires an expert hardware designer. Recently, high-level synthesis (HLS) tools aim raise the level of programming abstraction of FPGAs in order to make FPGAs accessible to application programmers. While current HLS tools are meant to be used by a larger number of designers and increase productivity, creating an optimized implementation requires substantial code transformations [22]. This transformations requires intimate knowledge of microarchitectural tradeoffs and domain expertise of application at hand. In order to successfully use today's HLS tools one needs to have: 1) domain knowledge about the application, 2) hardware design expertise, and 3) the ability to translate the domain knowledge into an efficient hardware design.

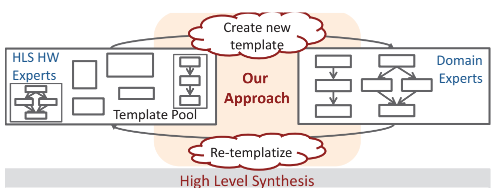

Fig. 1: An abstraction layer that separates domain knowledge from hardware skills.

In this work, we develop an approach to help separate domain knowledge from hardware expertise, in order to create more efficient implementation of an application on an FPGA. The general process is shown in Figure 1. There are number of basic kernels that share the same or similar computational primitives in range of applications [2, 6]. This indicates that these kernels can and should be built using a highly optimized template that is efficiently synthesized by the HLS tool. These templates are developed by hardware designers that have intimate knowledge of both the domain, hardware design, and the HLS tools.

The basic building block of our approach is a *composable, parameterizable template*. These templates are easily composed to create new templates that are automatically optimized for efficient synthesis by HLS tools. This is enabled by utilizing existing templates that follow pre described rules and common data access patterns. These composed templates are added to the template pool and can be later used to compose more complex templates. In this way, domain experts simply need to use an existing template, or form a new template for their specific applications. Similar to platform - based design [4], our approach is a structured methodology. Platform-based design "'theoretically limits the space of exploration, yet still achieves superior results in the fixed time constraints of the design"'.

Hardware design expertise is required at the initial stage of the process to contribute primitive templates for composition. However, we show that the number of primitive templates is small for many applications across several domains. We also show that it is possible to automatically generate efficient, high performance hardware implementations through the careful use of composable, parameterizable templates. Our method targets towards application programmers who have little hardware design expertise and HLS expertise.

The specific contributions of this paper are:

1) A novel approach based on composable, parameterizable templates that enables the design of applications on an FPGA by separating the domain knowledge from hardware design skill.
2) A theoretical framework based on trace theory [17] for the composibility and parameterization of templates to combine basic templates into more complex ones.
3) The development of basic templates across application domains, and case studies of how to compose these templates into more complex applications

This paper is organized as follows. Section 2 provides a motivational example. Section 3 and Section 4 formalizes the notation of a template and composition of templates. Section 5 presents results. Section 6 and Section 7 presents related work and conclusion.

II. Motivating Example: Sorting

The goal of this section is to motivate the research by stepping through an example that demonstrates how small number of basic templates can be composed to create an efficient hardware sorter implementation. We show how two basic optimized templates, prefix sum and histogram, can be combined in a hierarchical manner to create highly efficient implementations of different sorting algorithms. First, we combine the prefix sum and histogram templates to create a counting sort template. This in turn will be used to develop several parameterized implementations of a radix sort template. We use two data access patterns (bulk-synchronous and fork/join) to compose these basic templates into more complex ones.

We will quickly and briefly discuss the basics of the counting and radix algorithms. Counting sort has three primary kernels which are ripe for basic templates. These code blocks are: histogram, prefixsum, and another kernel which uses histogram operation. Figure 2 a) and b) shows how these histogram and prefix sum templates are used to build a counting sort template. Creating an efficient counting sort template requires functional pipelining between the three templates using a data access pattern that we call *bulk-synchronous*. We will later argue, and show in a number of examples, that this sort of functional pipelining is generalizable to a large range of applications.

It must be mentioned that it is quite important that the initial templates are optimized in a manner that enables them to be efficiently composed. While we do not have space to describe such optimizations for histogram and prefix sum, it is not simply creating a functionally correct implementation. Care must be taken to insure that they can be composed efficiently. This largely boils down to insuring that each template can be suitably pipelined. Details on how to make these subtle, but extremely important transformations can be found in [15].

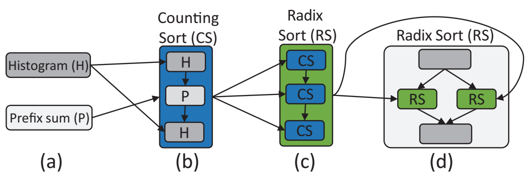

Fig. 2: Hierarchically composing templates from primitive templates. a) Initial templates for histogram and prefix sum. b) A counting sort template built using histogram and prefix sum templates. c) Radix sort built with counting sort templates using a bulk-synchronous data access pattern. d) A different implementation of radix sort composed using a fork/join data access pattern on two radix sort templates.

Radix sort performs counting sort n times for n radix (digits). Therefore, we can compose a radix sort template using counting sort templates. Again, to develop an efficient implementation, we require functional pipelining of the counting sort templates. In this case, we can build multiple implementations of radix sort, as depicted in Figure 2 c) and d). Figure 2 c) depicts a bulk-synchronous implementation using three counting sort templates. This hierarchical composition provides an optimized radix sort implementation that can be added into the template library. Figure 2 d) provides another implementation of radix sort. Here we combine two of the previously developed radix sort templates hierarchically using a fork-join data access pattern. This provides another option in the design space with different performance and area. Both of these can be added into the template library and later be used to compose more complex ones.

Based upon this example and our experience developing applications on FPGAs, we argue that domain programmers can efficiently build hardware implementations if they are provided with templates that are easily composable. We aid the composability by using a number of common data access patterns, e.g., bulk-synchronous, fork/join. By providing some optimized basic templates, and methods for automatically composing new templates in an efficient manner using these data access patterns, we show that it is possible to create highly efficient hardware implementations across a wide swath of application domains.

III. Templates and Compositions

In this section, we present a theory behind templates and necessary conditions for their composition. The template composition algebra and resulting process calculus is explained below, and is based on the theory of traces [18]. Readers should note that the rules given below fit into the framework of process calculi and trace theory. We will avoid an excessive introduction to those subjects, but those familiar with the concepts of trace theory will see that our rules form a process calculus, we will highlight the analogous operators towards the end of the section. For those interested in the abstract algebra of traces we refer the reader to [17] and for a well known treatment with respect to VLSI verification [7]. In the following, an *abstract template* is a functionally specific process. A *template* is a member of an *abstract template* super-class, but having a specific architecture. Let us start by defining necessary properties for elements of the set of *abstract templates*. We define three sets: T, P, and I. $T = \{t_1, t_2, \ldots, t_n\}$ is the set of the *abstract templates*. $P = \{p_1, p_2, \ldots, p_m\}$ is the set of ports. P_i and P_o are subsets of input ports and output ports, respectively. $P = P_i \cup P_o$ and $P_i \cap P_o = \emptyset$. $I = \{i_1, i_2, \ldots, i_n\}$ is a set of template interfaces applicable to ports.

Def 1. *A port $p = (d, i) \in P$ is a tuple where $d \in \{in, out\}$ is a direction and $i \in I$. We use $d(p)$ and $i(p)$ to represent the direction and interface of the port p.*

Templates are composed based upon their *port* properties. *Def 2* defines port properties which are defined by forward/backward (FC/BC) compatibility which are defined below. The *Def 3* defines templates and their composability properties based upon forward/backward compatibility.

Def 2. *(Forward/Backward Compatibility) $\forall p_1, p_2 \in P$, $FC(p_1, p_2) = 1 \Leftarrow d(p_1) = out \wedge d(p_2) = in \wedge i(p_1) = i(p_2)$ and $BC(p_1, p_2) = 1 \Leftarrow d(p_1) = in \wedge d(p_2) = out \wedge i(p_1) = i(p_2)$.*

Def 3. *An abstract template is a tuple $t = (IN, OUT, f) \in T$ with following properties: **1)** $IN \subset P_i \wedge OUT \subset P_o$, **2)** $|IN| \geq 1 \wedge |OUT| \geq 1$, **3)** $f(IN) = OUT$.*

$IN(t)$, $OUT(t)$, and $f(t)$ represent a set of input ports, a set of output ports, and the functionality of t, respectively. An abstract template is useful much like an *abstract class* in software engineering, and is useful for validating composition. Actual architectures are represented by *optimized architectural instance templates* which are instantiations of an abstract template. The *abstract template* $t \in T$ can have many variants of *optimized architectural instance template*, each of which has the same functionality and ports as the abstract template t. We use a set $A_I^{t_i} = \{t_{ij} | t_{ij}$ is the j^{th} instance template of abstract template $t_i\}$, so t_{ij} for $j = 1, \ldots, k$ denote k instances of an abstract template t_i. In the rest of the paper, we use *instance template* to refer optimized architectural instance template and *template* to refer both abstract and instance template. An instance template t_{ij} is a tuple $t_{ij} = \{II, a\}$ where II is the throughput of t_{ij} and a is the area of t_{ij}. We use $II(t_{ij})$ and $a(t_{ij})$ to represent throughput and area of instance template t_{ij}.

Now we define rules and functions that must hold in order to compose two or more templates to form a new template. $BSComposability$ and $FJComposability$ functions that check if two or more templates can be composed to form a new template based on bulk-synchronous or fork/join data access patterns, respectively. In order to define bulk-synchronous composability of $t_1, t_2, ..t_n \in T$, we check the composability of every consecutive pair of templates using a binary composability property, $BinaryComposability$.

Rule 1. *$\forall t_1, t_2 \in T$, $BinaryComposability(t_1, t_2) = 1$ if $|OUT(t_1)| = |IN(t2)|$, and $\forall p_i \in OUT(t_1)\ \exists! p_j$ s.t. $p_j \in IN(t_2)$.*

Based on $BinaryComposability$, we can define $BSComposability$ as follows:

Rule 2. *$BSComposability(t_1, t_2, \ldots, t_n) = 1$ if $\forall t_i, t_{i+1} \in T$ $BinaryComposability(t_i, t_{i+1}) = 1$ where $i = 1, n-1$*

In order to define fork/join composability of $t_1, t_2, \ldots, t_n \in T$ we assume t_1 is the fork, and t_n is the join. Then the fork join composability function, $FJComposability$, is defined by checking $ForkComposability$ and $JoinComposability$ rules.

Rule 3. *$ForkComposability(t_1, t_2, ..t_n) = 1$ where $t_1, \ldots, tn \in T$ if **1)** $|IN(t_1)| = 1 \wedge |OUT(t_1)| = n-1$, **2)** $|IN(t_k)| = |OUT(t_k)|$, $\forall k = 2, n$, **3)** $\forall p_i \in OUT(t_1)$ if $\exists p_k \in IN(t_k)$ s.t. $ForwardCompatibility(p_i, p_k) = 1 \wedge BackwardCompatibility(p_k, p_i) = 1$*

Rule 4. *$JoinComposability(t_1, t_2, ..t_n) = 1$ where $t_1, \ldots, tn \in T$ if **1)** $|IN(t_i)| = 1 \wedge |OUT(t_i)| = 1$, $\forall i = 1, n-1$ $|IN(t_n)| = n-1$, **2)** $\forall p_k \in OUT(t_i)$ for $i = 1, n-1$ if $\exists p_j \in IN(t_n)$ s.t. $ForwardCompatibility(p_k, p_j) = 1 \wedge BackwardCompatibility(p_j, p_k) = 1$*

Rule 5. *$FJComposability(t_1, t_2, ..t_n) = 1$ where $\{t_1, \ldots, t_n\} \in T$ if $ForkComposability(t_1, t_2, \ldots, t_{n-1}) = 1$ $\wedge JoinComposability(t_2, t_3, \ldots, t_n) = 1$. As a corollary, for a template t, if $IN = IN_1 \cup IN_2, \ldots, IN_n$, $OUT = OUT_1 \cup OUT_2, \ldots, OUT_n$ and $\exists f(t)$ such that $f(IN1) = OUT1, f(IN2) = OUT2, \ldots, f(IN_n) = OUT_n$, then t is n-way FJ composable.*

Based on the definitions and rules above, we define template composition functions. Template composition is a way of structurally composing existing templates from T based on data access patterns such as bulk-synchronous and fork/join. We omit the proofs here for the sake of brevity.

Lemma 1. *if $BSComposability(t_1, t_2, ..t_n) = 1$ for $\{t_1, \ldots, t_n\} \in T$, then $BS(t_1, t_2, ..t_n)$ maps to a new template t_{new} where t_{new} has all properties in Definition 3.*

Lemma 2. *if $FJComposability(t_1, t_2, \ldots, t_n) = 1$ for $\{t_1, \ldots, t_n\} \in T$, then $FJ(t_1, t_2, \ldots, t_n)$ maps to a new template t_{new} where t_{new} has all properties in Definition 3.*

These lemmas demonstrate that our rules give templates algebraic closure and transitivity. To see that these rules have indeed defined a process calculus, notice that we have a set of processes that 1) can be executed in parallel, 2) have communication channels, 3) may be nested recursively, 4) have abstracted interaction semantics, and 5) are sequentially composable. In practical terms BS and FJ functions are ways of constructing a new template t_{new} with new functionality, which later can be used as an existing template. We also define a *general strategy* to parameterize any code block. If there is functionality which does not have a corresponding template in T, we rely on users making a new template and contributing it to our system. The users add the new template to the system by defining the abstract properties of the template. Loop optimization works discussed in [5, 11, 23] can be used here. This area of research needs further investigation. We believe that our approach, as in Figure 1, will eventually fill functionality gaps by producing more and more templates in a disciplined manner.

IV. Template Parameterization

In this section, we describe how to compose templates in a highly optimized manner and provide trade-offs on performance and area. When composing new templates based on the rules defined in previous section, we have two constraints: 1) composition algorithm, 2) parameterizable architecture generation.

Composition Algorithm: A domain expert is designing an application A with n kernels, i.e., $A = \{k_1, k_2, ..k_n\}$. Assume that there exists at least one template that can be used to implement each k_i. The input to the algorithm is a set of templates T, user input data, and an optional user constraints UC. UC is a tuple $UC = (f_u, II_u, a_u)$ where f_u is frequency, II_u is throughput, and a_u is area. The area, a_u, is considered as a weighted combination of FPGA elements such as BRAMs, LUT/FFs. Next, we present an algorithm for constructing a new template using bulk-synchronous function (BS) in Algorithm 1. Due to limited space, we only present an algorithm for BS. The same principle and algorithm applies to FJ function using Rule 5.

Algorithm 1: Procedures of BS construction algorithm

Procedure `BulkSynchronous()`
    Data: $UC = \{f_u, II_u, a_u\}$, $D = data$ $T = \{t_1, t_2, ..t_n\}$
    //Call the subroutines here
Procedure `FindInstances()`
    Data: $T = \{t_1, t_2, ..t_n\}$
    Result: M_A=set of instance templates for each t_i
    forall the $t_i \in T$ **do**
        in=GetAllInstancesOf(t_i, D), $i = 1, ..n$
    end
    return M_A
Procedure `ComposabilityCheck()`
    Data: M_A
    Result: G=A set of graphs of composable templates
    return G
Procedure `ConstructBulkSynchronous()`
    Data: $G(V, G)$
    Result: BS= $BSComposability$ set of templates
    $currentVertex = v_0$
    if UC *is* $\emptyset$ **then**
        while $i < n$ **do**
            Select next $v \in V$ s.t. v(II) is minimum
            BS.AddToBS(v)
            $currentVertex$ = selected(v)
        end
    end
    else
        foreach *(v_i)* **do**
            $templates$ = GetAllComposableTo(v_i)
            BS.AddToBS(v)
            foreach *($templates$)* **do**
                $cost$ = CalculateClosestPair(v_i, $templates$)
            end
            $currentVertex$ = VertexWithMinCost()
            BS.AddToBS($currentVertex$)
        end
    end
    return BS
Procedure `CodeGeneratorBS()`
    //Omitted for brevity

The algorithm has four sub routines. The $FindInstances$ calls $GetAllInstance$ sub routine for each abstract template t_i. The $GetAllInstance$ returns a set M_A containing optimized instance templates. As discussed in previous section, abstract template is a black box, and each abstract template has a number of instance templates. This is because in our framework, we want to separate functionality from the underlying microarchitectural hardware, and letting our framework choose the one based on user constraints. For example, as shown in Figure 3, matrix multiplication abstract template has a number of instances. Each instance is implemented in different microarchitecture (streaming, 1 processing element (PE), 4 PE with streaming) having different performance and area based on user constraint. Based on user constraint (e.g., input data size), the algorithm selects different instance templates. This is important because some applications have a intersection points between different instance templates where certain instance template is better than another around that point for different user constraints (e.g., input size). We call it *performance breaking* point. This will be discussed in more detail in experimental section.

After $FindInstances$ routine, M_A set contains all instance templates necessary to implement an application A. M_A has a matrix-like structure where column i represents the same class of templates that can be used to implement kernel k_i. To illustrate this process better, we give an example in Figure 3 (a). The k_1, k_2, k_3, and k_4 are kernels which can be implement by *abstract templates* t_1, t_2, t_3, and t_4. In the next step, we call subroutine $ComposabilityCheck$ which returns a set of graphs (each graph contains a set of templates composable based on BS model). The routine checks Rule 1 for each t_{ij}, t_{ij+1} pair, and Rule 2 for the selected set of t_{ij}. In the case of fork/join, we use Rule 5 to check composability. After this step, we obtain one or more sub-graphs of M_A as shown in the Figure 3. The optimal algorithm (maximizes throughput) to find instance templates runs checks all possible paths in each graph which runs $O(n \times k \times k)$. We use a greedy algorithm which selects a graph that has t_{1i} where the $II(t_{1i})$ is minimum. The result of this algorithm returns a graph G which starts from a selected t_{1i} as a source. The next step, $ConstructBulkSynchronous$, accepts input G and outputs a path from a source of G to a sink of G. This procedure returns path that contains a set of instance templates for the given application A based on BS. In this process, we consider two cases; When $UC = \emptyset$, the algorithm selects each next instance template greedily which maximizes throughput. If $UC \neq \emptyset$, then we model the selection as a cost function using *closest point problem* [19] between UC and a set of candidate instance templates. The function $GetAllComposableTo$ returns all composable templates from the current vertex v_i. For example, in Figure 3, if we are on t_{14} of G_2, then $GetAllComposableTo$ retuns t_{22} and t_{24} The $CalculateClosestPair$ function calculates cost from the current vertex v_i to all other vertices returned by $GetAllComposableTo$. The next instance template t_{ik} is selected based on a value of closest point between pair of (II_a, a_u) and a $II(t_{ij}), a(t_{ij})$ where $t_{ij}$is a set of all candidate instance templates. This process is performed in $VertexWithMinCost$ function. Based on UC, if a certain template fall to meet II_u, we apply *parameterized template generation and selection*, which will be discussed in Section IV. The final step, $CodeGeneratorBS$, generates optimized HLS code based on compositions and templates.

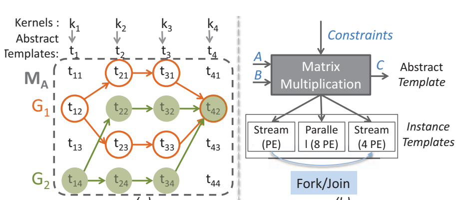

Fig. 3: a) Composition example for bulk-synchronous model, b) An example relationship between abstract and instance templates.

Parameterization: Instance templates allow users to have parameterizable architectures. This enables instance templates to provide flexibility that leverages area and performance trade-off by providing different instances of an abstract template. The flexibility of instance templates provides two benefits: 1) adjusting throughput to global throughput or to user specified constraints when composing template, 2) template selection based on user constraints or user input data size. Both of these benefits are crucial when composing templates. The former benefit provides easy way to achieve throughput increase/decrease based on user constraints. We name it *parameterizable speed-up* in this paper. The latter benefit of instance template is essential to provide optimized architecture for the user, which selects certain architecture based on user constraint or user input data. For example, $RS1$ and $RS2$ in Figure 3 (b) are the same instance templates for radix sort that represent the architecture as in Figure 2 (c). Based on different parameters and user input data size, $RS1$ and $RS2$ has different throughputs for a given data size as shown in Figure 3 (b). We will discuss this example in detail in Section V. Currently, this process of selecting optimized architecture for specific user constrain is being done manually by HLS experts. With our approach, we will automate this process by leveraging user constraints and analyzing user input data. Since templates have pre-defined high-level structures, throughput, $II(t_{ij})$, and area, $a(t_ij)$, are linear functions of input data size. They can be determined differently for regular and irregular programs; for regular programs, $II(t_{ij})$ and $a(t_ij)$ are defined by exploiting the user input data and instance template structures. For irregular programs such as sparse matrix vector multiplication, we rely on design space exploration to determine $II(t_{ij})$ and $a(t_ij)$.

V. Experimental Results

We used Vivado HLS 2014.1 as a back-end HLS tool. In our prototype framework, each abstract template is modeled as a Python class. Each instance template is a Python class inherited from an abstract template class. An abstract template class consists of fields to model ports, interfaces, and functionality for its child classes. In this work, we define interfaces based on Vivado HLS interface specification [1]. The abstract template implements the $HLSCodeGenerator$ function which writes HLS code based on domain-specific functionality. Each instance template inherits this method and calls it with template specific parameters, e.g., optimization parameters, bit width, size, number of functional units, etc.

```
Radix(array, N)
Loop0: for(i=0;i<N;) {
L0:
  LP1: for (...) {
  L1:
   ... }
  LP2: for (...) {
  L2:
   ... }
  LP3: for (...) {
  L3: ...
  }}
```

Listing 1: Radix sort (sw)

```
MergeSort(array, N)

 LP0: for(i=0;i<N;){
 L0:
  LP1: for (...) {
  L1:
   ...
    LP2: for (...) {
    L2
      ...
    }
  }}
```

Listing 2: Merge sort

TABLE I: Case study: SWO versus Template. (Period = clock period)

	Optimizations	II	FF/LUT	Period
Radix OSC1	Pipeline L1, L2, L3	14425	320/544	6.38
Radix OSC2	Pipeline L1, L2, L3, Unroll L0 factor=2	14422	497/936	6.38
Radix OSC3	Pipeline L1, L2, L3, Unroll L0 factor=2	14420	849/1688	6.38
Radix Template	Inner loop pipeline	2476	2065/4100	7.58
Merge OSC1	Pipeline L3	8742	1503/831	4.21
Merge OSC2	Pipeline L2	14551	1531/832	5.25
Merge OSC3	Unroll L2 factor=4	2476	2065/4100	4.21
Merge Template	Inner loop pipeline	262	3475/6897	2.69

We present area and performance results of different primitive templates. We use $Template$ and *Optimized Software Code (OSC)* to indicate code generated from our method and *optimized synthesizable code* for HLS, respectively. *OSC* is a HLS code *highly optimized using HLS #pragmas*. This codes is not rewritten to target low level architectural features. It is code written optimized for software and using *only* HLS pragmas. It is how a software programmer would write the code and use the HLS tools based on our experience in teaching HLS over three years of a graduate student class. To illustrate how software code is optimized in HLS using only pragmas, we present two examples in the following; one is easy to parallelizable (radix sort) and other is inherently recursive (merge sort) algorithms. By presenting optimization of these examples, our goal is to give an intuitive example which shows that re-writing software code is an essential step (not easy for software programmers) in order to generate an efficient hardware from HLS. Important parts of pseudo code to optimize radix and merge sort is given in Listing 1 and Listing 2. In the first *OSC*, we can pipeline all inner loops $L1, L2, L3$ of radix sort. This will gives better performance but performance will not increase largely because inner loop pipeline only improves instruction level parallelism. Second, in order to improve performance, we unrolled $L0$ which tries to unroll the loop by a given factor (factor=2). However, due to dependency between different iterations of loop, the unroll attempt does not success; in fact, this gives worse results than *OSC1*. We summarized different OSC results and template results for radix and merge sort in Table I. OSC results for radix and merge sort does not improve performance; in fact area gets larger than *OSC1*. On the other hand, template approach improve performance while area is linearly increasing due to achieved II. Template uses higher area (2X to 4X) than OSCs because it achieves higher parallelism. The particular merge sort and radix sort templates uses 8 and 14 BRAMs, respectively which is 4-8X more than *OSC1*.

In the following, we start by comparing $Template$ against OSC. Then, we present two examples of achieving parameterizable speed-ups using templates. Finally, we use several of those highly parameterizable templates to design five large applications. Due to space constraints, we present only few results of applications designed using templates. The applications are Canny edge detection and matrix inversion. All results are obtained from HLS place and route.

Template vs. OSC: Figure 4 shows throughput of *Template* and *OSC* designs for various templates designed. Level 0 is a primitive template and includes the *0prefsum, 0histo, 0gaussian, 0conv, 0hufftre, 0thrh, 0Imgadjt, 0bicubic, 0dilation, 0erosion, 0bit_rev, 0butter, 0SpMV_0* kernels. For templates *0dilation, 0erosion*, and *0thrh* kernel, *Template* is better than *OSC* by around $1.1 - 1.5X$. This is because those templates can be highly optimized using only HLS only directives. For kernels *0gaussian, 0conv, 0bicubic, 0bit_rev, 0butter* and *0hufftre*, we see several orders of magnitude of improvement. This is because these templates require low-level microarchitectural knowledge in order to generate efficient hardware. The second level kernels are designed using the templates from the *Level 0*. For example, *1CntSort* is built on using *0histo* and *0prefsum*. *FFT* is built using *0bit_rev* and *0butter*. *1SpMV* is built by composing several of *0SpMV_0* templates using the fork join data access pattern. Several templates in Level 1 use linear algebra primitives such as vector-vector multiplication. Level 2 are five applications composed using existing templates from Level 0 and Level 1. Next, we will elaborate parameterizability of templates.

Parameterization: Parameterization plays a vital role in Algorithm 1 when composing templates to meet a throughput requirements. Here we describe the parameterizable templates for prefix sum

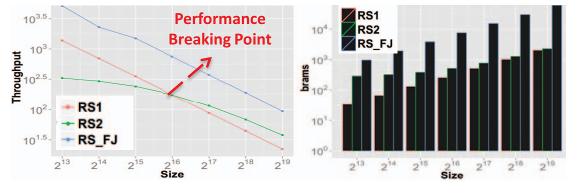

Fig. 5: Radix sort

and histogram. The result is shown in Figure 6. First, we optimized both of them targeting low level hardware architecture by removing data and read after write dependencies. This is same as the template in Level 0 in Figure 4. We call this $Lev0$. Then using these $Lev0$ designs, we applied different combinations of parameterizable speed-up factors using FJ and BS data access patterns. The prefix sum is composed based on FP pattern while histogram is composed based on FJ pattern. The speed-up factor (shown as *Factor* in the Figure 6) is the unrolling factor for OSC and $Lev0$ designs. $OSC - X$ means speed-up factors of X. The same convention follows for $Lev0$ designs. For BS and FJ designs, the speed-up factor is the task level parallelism factor. In both cases, OSC designs does not give the desired throughput regardless of unrolling factor. In the $Lev0$ designs, the throughput does increase, but it does not scale as expected. This is because the clock frequency is also increasing with higher speed-up factor. BS and FJ, both designs perform and scale as expected according (4, 8, 16) to speed-up factor.

Next we present three different radix sort templates in Figure 5. $RS1$, $RS2$ are templates composed as in Figure 2 (c) with different parameters, and RS_FJ is a template as in Figure 2 (d). Sorting algorithms use less slices, and usaully BRAM is important area metric. Thus, in the Figure 5 we presented throughput and BRAM utilization. $RS1$ and $RS2$ have similar area usage, and RS_FJ has 8 times larger area usage than $RS1$ and $RS2$ due to higher parallelism. In this case if UC is maximizing throughput with minimum area, the Algorithm 1 selects $RS1$ for input data size $2^{13} - 2^{15}$ and $RS2$ for input data size $2^{16} - 2^{19}$ as shown in the Figure 3. We call the intersection of $RS1$ and $RS2$ *performance breaking point*. Our algorithm transparently selects an architecture based on user constraint balancing performance breaking point. If UC is empty or maximum throughput, the algorithm selects RS_FJ.

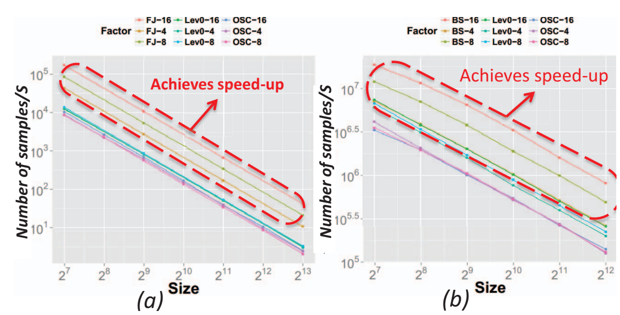

Fig. 6: Developing parameterizable templates for a) Histogram, b) Prefixsum

Canny Edge Detection / Matrix Inversion: Next, we argue that the hardware generated from our approach has competitive area and performance results. We compare area and performance of applications composed with templates with other published work. We use two cases: Canny edge detection and matrix inversion. The Canny edge detection algorithm is divided into four stages, Gaussian

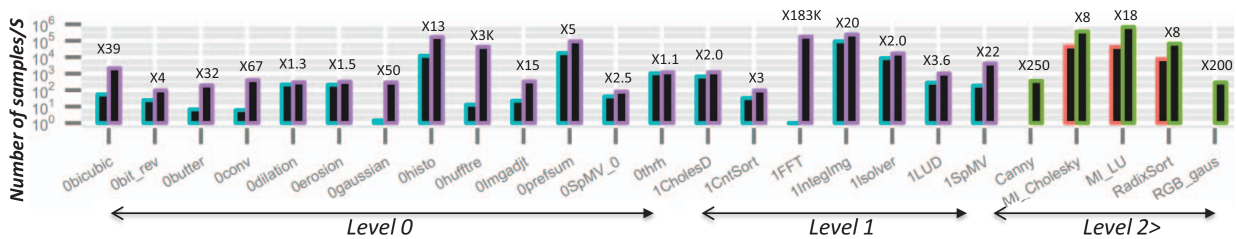

Fig. 4: The performance of OSC versus Templates. The OSC results are always on the left and the Template results on the right. In each application we give the actual speedup above the bars. The Template approach is always better than OSC. The "level" indicates the maximum amount of hierarchy used for the templates. Level 0 is a primitive template. Level 1 is composed of at least one primitive template. Level 2 is composed of at least one Level 1 (or higher) template.

TABLE II: Hardware area and performance results. ***T****=Throughput,* ***S****=Size,* ***F****=Freq,* ***B****=BRAM,* ***W****=bits, *=QVGA or VGA size*

	Canny Edge			Matrix Inversion		
	Q/VGA	R [12]	R [8]	MI	R [13]	R [14]
T	464/134	400	240	2.44	0.33	0.13
S	*	360x280	512x512	4x4	4x4	4x4
F	121/140	27	120	180	-	115
B	9/9	-	-	0	1	9
W	8/8	-	-	32	20	20

smoothing, edge strength identification, non maxima suppression, and double thresholding. All four stages can be designed using highly parameterizable convolution and histogram templates from our template pool. We composed a new template for Canny edge using BS function. Our designs of Canny edge runs for Q/VGA sizes. Table II shows the throughput as frames per second and hardware utilization for our designs and previous work. Our results are comparable to these published works. Matrix inversion application uses FJ pattern. Table II shows a comparison between our results (synthesized on xc4vfx140) and previously published works [13, 14] for 4×4 matrix inversion. In general, our performance is 7-18X better than [13, 14] but our area is 2-8X larger (we use 85 DSP, [13] use 12 DSP) than those works.

VI. Related Work

Several HLS vendors provide libraries, e.g., OpenCV and linear algebra from Xilinx. They provide a first step towards making FPGA designs more accessible. While experienced HLS users find these libraries useful, it is difficult for a domain programmers to use them since they require low level hardware expertise. Our technique goes further than just static libraries; these libraries are typically not composable or parameterizable. In fact, we can use the functions in these libraries as basic templates. The work [6, 16] present similar approaches to this work terms of facilitating composability of reusable components. The work [6] defines accelerator building block as a service for hardware blocks and the work [16] presented a study of IP core composition. Both of these works compose low level IP cores. Therefore, our approach can provide functionality to them by generating composable IP cores or accelerator blocks. Several other works such as Chisel [3], FCUDA [18] and others [9, 20] present domain-specific language based approach to design an FPGA system.

System level design automation [21] and compositional high-level synthesis [10] present an approach to select hardware components while doing inter/intro optimizations among components. The main building blocks (or assumptions) of these works are existing components. These components, in fact, can be modeled as composable parameterizable templates. Thus our work can be used as a components for [10, 21]. Different than these works, we assume the users of our work will be pure software programmers without any hardware knowledge. Thus, our work provides higher-level of abstraction by composable parameterizable templates.

VII. Conclusion

In this work, we described a theoretical framework for parameterizable and composable HLS templates using common data access patterns. We built a highly optimized library of basic parameterizable templates and showed how to compose them to create a number of large applications from various domains. These designs were highly optimized and easily developed using our framework.

References

[1] Vivado design suite user guide: High-level synthesis.
[2] K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
[3] J. Bachrach et al. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference*, pages 1216–1225. ACM, 2012.
[4] L. Carloni et al. Platform-based design for embedded systems. *The Embedded Systems Handbook*, pages 1–26, 2005.
[5] C. Chen. Polyhedra scanning revisited. In *ACM SIGPLAN Notices*, volume 47, pages 499–508. ACM, 2012.
[6] J. Cong et al. Charm: a composable heterogeneous accelerator-rich microprocessor. In *ISLPED*, pages 379–384. ACM, 2012.
[7] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. 1988.
[8] C. Gentsos et al. Real-time canny edge detection parallel implementation for fpgas. In *ICECS*, pages 499–502. IEEE, 2010.
[9] N. George et al. Hardware system synthsis from domain-specific languages. In *International Conference on Field Programmable Logic and Applications*, 2014.
[10] D. Guglielmo et al. A design methodology for compositional high-level synthesis of communication-centric socs. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.
[11] R. J. Halstead et al. Compiling irregular applciations for reconfigurable systems. *International Jounal on High Performance Computing and Networking*, 2014.
[12] W. He et al. An improved canny edge detector and its realization on fpga. In *World Congress on Intelligent Control and Automation*, pages 6561–6564. IEEE, 2008.
[13] A. Irturk et al. A top-to-bottom design methodology for automatic generation of application specific architectures. *TCAD*, 30(8):1173–1183, 2011.
[14] M. Karkooti et al. Fpga implementation of matrix inversion using qrd-rls algorithm. In *Conference on Signals, Systems and Computers*, pages 1625–1629, 2005.
[15] J. Matai et al. Energy efficient canonical huffman encoding. In *ASAP*. IEEE, 2014.
[16] D. A. Mathaikutty. *Metamodeling driven IP reuse for system-on-chip integration and microprocessor design*. PhD thesis, 2007.
[17] A. Mazurkiewicz. Introduction to trace theory. *The Book of Traces*, 1995.
[18] A. Papakonstantinou et al. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors*, pages 35–42. IEEE, 2009.
[19] M. I. Shamos et al. Closest-point problems. In *Symposium on Foundations of Computer Science*. IEEE, 1975.
[20] H. Tabkhi et al. Function-level processor (flp): Raising efficiency by operating at function granularity for market-oriented mpsoc. In *ASAP*. IEEE, 2014.
[21] P. Zhang et al. Cmost: a system-level fpga compilation framework. In *DAC*. ACM, 2015.
[22] Z. Zhang et al. Challenges and opportunities of esl design automation.
[23] W. Zuo et al. Improving high level synthesis optimization opportunity through polyhedral transformations. In *FPGA*, pages 9–18. ACM, 2013.