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.

978-3-9815370-6-2/DATE16/ ©2016 EDAA
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].

![Diagram](image)

---

**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 the template and their composability properties based upon forward/backward compatibility.

**Def 2.** (Forward/Backward Compatibility) \( \forall p_1, p_2 \in P, FC(p_1, p_2) = 1 \iff d(p_1) = \text{out} \land d(p_2) = \text{in} \land i(p_1) = i(p_2) \) and \( BC(p_1, p_2) = 1 \iff d(p_1) = \text{in} \land d(p_2) = \text{out} \land i(p_1) = i(p_2) \).

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

\( IN(t) \), \( OUT(t) \), 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 \( A_i^j = \{ t_{ij} \mid t_{ij} \text{ is the } j^{th} \text{ instance template of abstract template } t_i \}, \) 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 \( t_{ij} = (I, a) \) where \( I \) is the throughput of \( t_{ij} \) and \( a \) is the area of \( t_{ij} \). We use \( I(t_{ij}) \) 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. **BSCComposability** 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 \( t_1, t_2, \ldots, t_n \in T \), we check the composability of every consecutive pair of templates using a binary composability property, **BinaryComposability**.

**Rule 1.** \( \forall t_1,t_2 \in T, \text{BinaryComposability}(t_1, t_2) = 1 \) if \( |\text{OUT}(t_1)| = |\text{IN}(t_2)| \) and \( \forall p \in \text{OUT}(t_1) \exists! p_2 \text{ s.t. } p_2 \in \text{IN}(t_2) \).

Based on **BinaryComposability**, we can define **BSCComposability** as follows:

**Rule 2.** **BSCComposability**\( (t_1, t_2, \ldots, t_n) = 1 \) if \( \forall t_i, t_{i+1} \in T \text{ BinaryComposability}(t_i, t_{i+1}) = 1 \) 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.

---

2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)
Rule 3. *ForkComposability*\((t_1, t_2, \ldots, t_n) = 1\) where \(t_1, \ldots, t_n \in T\) if 1) \([IN(t_1)] = 1 \land [OUT(t_1)] = n - 1; 2) \([IN(t_k)] = [OUT(t_k)], \forall k = 2, n\), 3) \(\forall p_k \in OUT(t_1)\) if \(\exists p_k \in IN(t_k)\) s.t. *ForwardCompatibility*\((p_k, p_1) = 1 \land *BackwardCompatibility*\((p_k, p_1) = 1\)

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

Rule 5. *FJComposability*\((t_1, t_2, \ldots, t_n) = 1\) where \(t_1, \ldots, t_n \in T\) if 1) *ForkComposability*\((t_1, t_2, \ldots, t_{n-1}) = 1\), 2) *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 \(f(IN_1) = OUT_1, f(IN_2) = OUT_2, \ldots, f(IN_n) = OUT_n\), 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*\((t_1, t_2, \ldots, t_n) = 1\) for \(t_1, \ldots, t_n \in T\), then *BS*\((t_1, t_2, \ldots, t_n)\) 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., \(A = \{k_1, k_2, \ldots, k_n\}\). 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 \(UC = (f, H, a_u)\) where \(f\) is frequency, \(H\) 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.

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 implemented 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 $t_{ij}$ where the $H(t_{ij})$ is minimum. The result of this algorithm returns a graph $G$ which starts from a selected $t_{ij}$ 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 close 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 $t_{ij}$ of $G_2$, then GetAllComposableTo returns $t_{ij}$ and $t_{ij+1}$. 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 $(II_{ai}, \alpha_{ai})$ and a $H(t_{ij})$, $\alpha(t_{ij})$ 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 fail to meet $II_{ai}$, 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.

### 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, $RS_1$ and $RS_2$ 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, $RS_1$ and $RS_2$ 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, throughputs, $H(t_{ij})$, and area, $\alpha(t_{ij})$, are linear functions of input data size. They can be determined differently for regular and irregular programs; for regular programs, $H(t_{ij})$ and $\alpha(t_{ij})$ 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 $H(t_{ij})$ and $\alpha(t_{ij})$.

### 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 HLSCodeGen 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.

#### Listing 1: Radix sort (sw)
```python
1. RadixSort(array, N)
2. For i=0; i < N; i++)
3. L0: For i=0; i < N; i++)
4. L1: For i=0; i < N; i++)
5. L2: For i=0; i < N; i++)
6. L3: For i=0; i < N; i++)
7. L4: For i=0; i < N; i++)
8. L5: For i=0; i < N; i++)
9. L6: For i=0; i < N; i++)
10. L7: For i=0; i < N; i++)
11. L8: For i=0; i < N; i++)
12. L9: For i=0; i < N; i++)
```

#### Listing 2: Merge sort
```python
1. MergeSort(array, N)
2. L0: For i=0; i < N/2; i++)
3. L1: For i=0; i < N/2; i++)
4. L2: For i=0; i < N/2; i++)
5. L3: For i=0; i < N/2; i++)
6. L4: For i=0; i < N/2; i++)
7. L5: For i=0; i < N/2; i++)
8. L6: For i=0; i < N/2; i++)
9. L7: For i=0; i < N/2; i++)
10. L8: For i=0; i < N/2; i++)
11. L9: For i=0; i < N/2; i++)
12. L10: For i=0; i < N/2; i++)
```

#### Table 1: Case study: SWO versus Template. (Period = clock period)

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>II</th>
<th>FF/LUT</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix OSC1</td>
<td>320/544</td>
<td>14425</td>
<td>6.38</td>
</tr>
<tr>
<td>Radix OSC2</td>
<td>375/689</td>
<td>14422</td>
<td>6.38</td>
</tr>
<tr>
<td>Radix OSC3</td>
<td>544/936</td>
<td>14420</td>
<td>6.38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L0 factor=2</th>
<th>Unroll L1, L2, L3</th>
<th>Unroll L1, L2, L3, L4</th>
<th>Unroll L1, L2, L3, L4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix Template</td>
<td>Inner loop pipeline</td>
<td>2476</td>
<td>7.58</td>
</tr>
<tr>
<td>Merge OSC1</td>
<td>Pipeline L3</td>
<td>8742</td>
<td>4.21</td>
</tr>
<tr>
<td>Merge OSC2</td>
<td>Pipeline L2</td>
<td>14551</td>
<td>5.25</td>
</tr>
<tr>
<td>Merge OSC3</td>
<td>Unroll L2 factor=4</td>
<td>2476</td>
<td>4.21</td>
</tr>
</tbody>
</table>

2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)
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 code 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 $L_1, L_2, L_3$ of radix sort. This will give 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 $L_0$ 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 succeed; in fact, this gives worse results than OSC1. We summarized different OSC results and template results for radix and merge sort in Table 1. 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 Oprefixum, Ohisto, Ogaussuan, Oconv, Oluafftre, Othrrh, Omgadgtj, Obicubic, Odilation, Oerosion, Obit_rev, Obutter, OSpMV_0 kernels. For templates Odilation, Oerosion, and Othrrh 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 Ogaussuan, Oconv, Obicubic, Obit_rev, Obutter and Oluafftre, 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 Ohisto and Oprefixum. FFT is built using Obit_rev and Obutter. 1SpMV is built by composing several of OSpMV_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 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 Level0. Then using these Level0 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 Level0 designs. OSC – $X$ means speed-up factors of $X$. The same convention follows for Level0 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 Level0 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 template in Figure 5. $RS_1, RS_2$ 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 usually BRAM is important area metric. Thus, in the Figure 5 we presented throughput and BRAM utilization. $RS_1$ and $RS_2$ have similar area usage, and $RS_FJ$ has 8 times larger area usage than $RS_1$ and $RS_2$ due to higher parallelism. In this case if $UC$ is maximizing throughput with minimum area, the Algorithm 1 selects $RS_1$ for input data size $2^{13} = 2^{15}$ and $RS_2$ for input data size $2^{16} = 2^{19}$ as shown in the Figure 3. We call the intersection of $RS_1$ and $RS_2$ 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. 5: Radix sort](image)

**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
smoothing, edge strength identification, 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 F/J 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], FUCUDA [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