Synthesizable Higher-Order Functions for C++

Dustin Richmond, *Member, IEEE,* Alric Althoff, *Member, IEEE,* and Ryan Kastner, *Member, IEEE*

***Abstract*—State-of-the-art C/C++ synthesis tools lack abstractions and conveniences that are pervasive in modern software languages. Higher-order functions are particularly important as they increase productivity by concisely representing common design patterns. Providing these in hardware design environments would improve the accessibility of hardware tools for software engineers by providing familiar interfaces and abstractions. We have created an open-source library of higher-order functions synthesizable in C/C++ hardware development tools. We implement six common algorithms on a PYNQ board and conclude that our library produces results that are generally statistically indistinguishable non-recursive techniques.**

***Index Terms*—FPGAs, High-Level Synthesis, Higher-Order Functions, Programming Languages**

I. Introduction

Hardware development tools have been gradually raising their level of abstraction from specifying transistors, to defining gate level circuits, to describing register transfer operations. C/C++ hardware development tools [1], [2], [3], [4], [5] further this trend by enabling the designer to provide algorithmic descriptions of the desired hardware. Yet, despite much progress, there are calls to make hardware design even more like software design, which will allow more software engineers to write hardware cores [6].

A major impediment to this lies in the fact that C/C++ hardware development tools lack many of the conveniences and abstractions that are commonplace in modern productivity languages. Higher-order functions are a prime example. They are a pervasive representation of computational patterns that take other functions as arguments. For example, the higher-order functions `map` and `reduce` shown in Figure 1 are the eponymous operators of Google's MapReduce framework [7] and the function `filter` is the semantic equivalent to SQL's `WHERE` clause [8]. Higher-order functions are also useful in hardware development where they can represent common parallel patterns [9], [10] like fast fourier transforms (Figure 2), argmin reduction trees [11], sorting networks [12], [8], and string matching [13]. Despite their benefits, higher-order functions are not found in C/C++ hardware development tools.

Higher-order functions are difficult to implement in C/C++ hardware development tools because parallel hardware must be defined *statically*: types, functions, interfaces, and loops must be resolved at compile time. In contrast, higher order functions typically rely on *dynamic* features: dynamic allocation, dispatch, typing, and loop bounds. Prior work has added higher-order functions to Hardware Development Languages (HDLs) [14], [3], [15], added higher-order functions to domain-specific languages [10], or proposed extensions to C/C++ development tools[9]. None have not created synthesizable higher-order functions in a widespread language like C/C++.

```
def mulby2(x):
    return x * 2
def add(x, y):
    return x + y
l = [1, 2, 3, 4]
m = map(mulby2, l)
r = reduce(add, m)
print(r) # Prints '20'
```

(a)

```
array<int, 4> l = {1, 2, 3, 4};
m = map(mulby2, l);
r = reduce(add, m);
```

(b)

Fig. 1. (a) Higher-order functions in Python that multiply all values in a list by 2 (`map`), and take the sum (`reduce`). (b) An equivalent in C++ using our library.

In this paper we develop a library of synthesizable higher-order functions for C/C++ hardware development tools with a syntax similar to modern productivity languages. Our work leverages recent additions to the C++ language standard to enable seamless integration into a C/C++ hardware design flow.

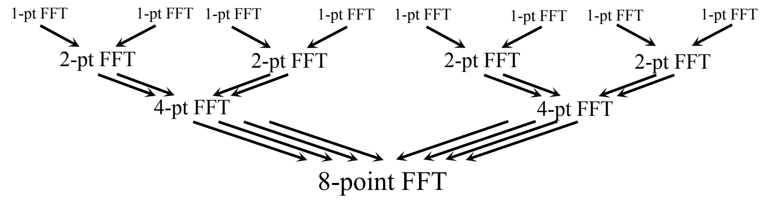
This work has four main contributions:

- A demonstration of C++ techniques that enable synthesizable higher-order functions
- An open-source library of higher-order functions for C/C++ hardware development tools
- A statistical comparison between our work and loop-based C/C++ implementing six common algorithms on a PYNQ board
- A qualitative comparison between the syntax of our library and a modern high-level language

Our paper is organized as follows: Section II describes the C++ techniques we use to develop our higher-order functions. Section III demonstrates how our library can be used to implement the well-known the Fast Fourier Transform (FFT) algorithm, one of many examples in our repository. Section IV presents a comparison of compilation results between our library and loop-based constructs on six common algorithms. Section V describes related work. We conclude in Section VI.

All authors are affiliated with Department of Computer Science and Engineering, University of California, 9500 Gilman Drive, San Diego, San Diego, CA, 92103 USA e-mail: drichmond@eng.ucsd.edu

This article was presented in the International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS) 2018 and appears as part of the ESWEEK-TCAD special issue



(a)

```python
def fft(sig):
    return divconq(nptfft, bitreverse(sig))
```

(b)

```cpp
template <typename T, size_t LEN>
auto fft(array<T, LEN> sig){
  return divconq(nPtFFTStage, bitreverse(sig));
}
```

(c)

Fig. 2. (a) A graphical representation of the divide-and-conquer structure of a Fast Fourier Transform (FFT). (b) A python implementation of the FFT algorithm using the higher-order function `divconq` and functions `NPtFFTStage` and `bitreverse`. (c) A C++11 implementation of the Fast Fourier Transform algorithm using `divconq` from our library of higher-order functions.

II. Building Higher-Order Functions

Higher-order functions are a pervasive abstraction that encapsulate common programming patterns by calling other functions provided as input arguments. Figure 1 shows two higher-order functions: `map` applies a function to every element in an array, and `reduce` iterates through an array from left to right applying a function and forwarding the result to the next iteration. Higher-order functions can also implement recursive patterns. Figure 2 demonstrates how the recursive divide and conquer function `divconq` is used to implement the fast fourier transform algorithm. By encapsulating common patterns, higher-order functions encourage re-use.

Higher-order functions for are difficult to implement in C/C++ hardware development tools because parallel hardware must be defined *statically*: types and functions must be resolved, lists that define parallel interfaces must be statically sized, and parallel loops must be statically bounded. In contrast, higher order functions in productivity languages such as Python typically rely on *dynamic* features: polymorphic functions are overloaded with a table of function pointers, functions are passed as global memory addresses for dynamic dispatch, lists are created and resized by memory allocators, and the stack is dynamically resized for recursion. While it is possible to define hardware with dynamic memory allocation, function pointers, and dynamic dispatch the main drawback is efficiency and similarities to general-purpose processors.

In the following subsections we describe how to replace these dynamic features with static techniques to implement synthesizable higher-order functions for C/C++ hardware development tools. By using standard compiler features our work is not limited to a single toolchain. The result of our work is a library of higher-order functions that mimics the behavior of modern productivity languages.

A complete listing of the functions used in this paper are shown in Table I. The remaining functions can be found in our repository.

A. Templates (Parametric Polymorphism)

In this section we describe how to use C++ templates to provide the polymorphism required by higher-order functions. Polymorphism is the ability of a data type or function to be written *generically*. For example, the higher-order function `map` must be written generically so that its array argument can be a array of integers, array of arrays, array of classes, or an array of any other type. `map` must also have a generic output type so that it can produce any type of output array. Container classes such as arrays must be able to store integers or booleans. Polymorphism provides the ability to represent repeatable patterns across various input and output types.

1) Class Templates: Class templates are used to parameterize container classes and algorithms with types, length, and functions. They are pervasive in the C++ the Standard Template Library (STL). We use templated classes like those shown in Figure 3 extensively in our work.

Three examples of the the STL `array` class are shown in Figure 3a. `arr1` is an `array` of four integers, `arr2` is an `array` of four floats, and `arr3` is a `array` of two array classes, each with four integers (a 2-by-4 matrix). This example demonstrates how template parameters provide generic classes.

Figure 3b shows how templated classes are defined. Template parameters can be type names, class names, values, or functions. Template variables can be used to define the type of other template variables. For example `T` is used to define the type of the template parameter `VALUE`.

```cpp
int main(){
  array<int, 4>   arr1 = {1, 2, 3, 4};
  array<float, 5> arr2 = {1.0, 2.0, 3.0, 4.0};
  array<array<int, 4>, 2> arr3 = {arr1};
  return 0
}
```

(a)

```cpp
template <typename T, T VALUE>
class foo{
  T fooval = VALUE;
};
```

(b)

Fig. 3. (a) Three parameterized instances of the STL `array` class. (b) Defining a templated class `foo`.

2) Function Templates: Templates are also used to implement polymorphic functions that can handle multiple types with one definition. For example, the higher-order function `map` must be written generically so that its array argument can be a array of integers, array of arrays, array of classes, or an array of any other type. Templates can also be used to pass compile-time constants to a function. This functionality is required for functions that use the STL `array` class and will be used heavily in our higher-order functions.

Lines 1-10 in Figure 4 show two templated functions: `add` and `arrayfn`. The template parameter `T` provides static type polymorphism to both functions. This means they can be applied to integers, floats, or classes. `arrayfn` has an additional parameter `LEN` that specifies the length of its `array` argument. These functions are called without template inference on Lines 16-19.

```
1  template<typename T>
2  T add(T l, T r){
3    return l + r;
4  }
5
6  template<typename T, unsigned long LEN>
7  int arrayfn(array<T, LEN>& arr){
8    /* Do some array processing*/
9    return 0;
10 }
11
12 int main(){
13   array<int, 3> arr = {0, 1, 2};
14
15   // Three examples without template inference
16   int res1 = add<int>(0, 1);
17   float res2 = add<float>(0.0f, 1.0f);
18   int res3 = arrayfn<int, 3>(arr);
19
20   // The same examples with template inference
21   int res4 = add(0, 1);
22   float res5 = add(0.0f, 1.0f);
23   int res6 = arrayfn(arr);
24   return 0;
25 }
```

Fig. 4. Two templated functions: `add` and `arrayfn`.

3) Template Inference: Template parameter inference allows template parameters to be inferred from the call site, and is critical for creating succinct higher-order functions that mimic dynamically typed software languages. Template inference starts with the last template parameter, and stops when a parameter cannot be inferred, or all parameters have been inferred.

Figure 4 also demonstrates an example of template inference on Lines 21-24. The template parameters of calls `add` and `arrayfn` infer the `T` and `LEN` based on the types of the input arguments at those callsites. The effect of template inference is to allow designers to write less verbose code.

4) Functions as Template Parameters (First-Class Functions): C++ functions can also be passed as template parameters. Unlike software languages, where functions are passed as pointers and dynamically resolved during runtime, functions passed as template parameters are static and synthesizable.

Figure 5 demonstrates how the function `truncate` can be passed to the higher-order function `apply` as the template parameter `FN`.

Template inference cannot be applied to the example in Figure 5. `truncate` depends on the type parameters `TI` and `TO`, so it must follow those parameters in the parameter list.

```
template <typename TI>
unsigned char truncate(TI IN){
  return (unsigned char)IN;
}

template <typename TO, typename TI, TO (FN)(TI)>
TO apply(TI IN){
  return FN(IN);
}

int main(){
  int i = 0x11223344;
  unsigned char res;

  res = apply<unsigned char, int, truncate<int> >(i);
  // res = 0x44
  return 0;
}
```

Fig. 5. A C++ function passed as a template parameter.

`truncate` is not a function argument to `apply` it cannot be inferred. Figure 6 demonstrates how we can can aid template inference by wrapping the `truncate` function inside of a struct.

Figure 6 demonstrates how the body of the function `truncate` and its template parameters are relocated to the `()` operator inside of the `Truncate` struct. This is often called a class/struct-wrapped function, or functor. By passing the struct `Truncate` we "hide" the template parameters of its function from the template parameter list in `array`. Instead, the compiler infers them when `Truncate` is instantiated and the `()` operator is called on Line 10 of Figure 6.

```
1  struct Truncate{
2    template <typename TI>
3    unsigned char operator()(TI IN){
4      return (unsigned char)IN;
5    }
6  };
7
8  template <typename TO, class FN, typename TI>
9  TO apply(TI IN){
10   return FN()(IN);
11 }
12
13 int main(){
14   int i = 0x11223344;
15   unsigned char res;
16   // Previously:
17   //   apply<unsigned char, int, truncate<int> >(i);
18   res = apply<unsigned char, Truncate>(i);
19   // res = 0x44
20   return 0;
21 }
```

Fig. 6. Wrapping a function inside of a struct.

We can simplify this example further and deduce `FN` by passing it as an argument to `apply`, as shown in Figure 7. In Figure 7, `Truncate` is defined and instantiated as the variable `truncate`. The variable `truncate` is passed as a function argument to `apply`. Passing `truncate` as an argument allows the compiler to infer the template parameter `FN`. Because the variable `__ignored` is never used the example in Figure 7 is synthesizable. However, we still cannot infer `TO` unless it is passed as a function argument. To deduce `TO` we must use a new feature from the C++ specification covered in Section II-A5.

```
struct Truncate{
  template <typename TI>
  unsigned char operator()(TI IN){
    return (unsigned char)IN;
  }
} truncate;

template <typename TO, typename TI, class FN>
TO apply(FN __ignored, TI IN){
  return FN()(IN);
}

int main(){
  int i = 0x11223344;
  unsigned char res;
  // Prev: apply<unsigned char truncate<int> >(i);
  res = apply<unsigned char>(truncate, i);
  // res = 0x44
  return 0;
}
```

Fig. 7. Class-wrapped-functions can be inferred by passing them as instances at the callsites.

5) Type Inference: Section II-A4 we showed that we can automatically infer input types using template inference, but could not infer output types since output types are not arguments. Higher-order functions like `map` can return a multitude of types depending on what function is provided. To correctly mimic the behavior of dynamically-typed software languages we must be able to infer the output types automatically.

Figure 8 demonstrates how we can remove the `TO` template parameter from `apply` using the new `auto` keyword. This causes the compiler to deduce the return value from the type of `FN`. Figure 8 is functionally identical to Figure 7, but the call to `apply` is now less verbose.

```
struct Truncate{
  template <typename TI>
  unsigned char operator()(TI IN){
    return (unsigned char)IN;
  }
} truncate;

// Previously: <typename TO, typename TI, class fn>
template <class fn, typename TI>
// Previously: TO apply(fn _, TI IN)
auto apply(fn _, TI IN){
  return fn()(IN);
}

int main(){
  int i = 0x11223344;
  unsigned char res;
  // Previously: apply<unsigned char>(truncate, i);
  res = apply(truncate, i);
  // res = 0x44
  return 0;
}
```

Fig. 8. Applying the new `auto` keyword to Figure 7 allows us to remove the template parameter `TO`

B. Arrays

Lists are the most common target of higher order functions. In software productivity languages lists are implemented as dynamically allocated chunks of memory (contiguous or linked) that can be created, duplicated, resized, and modified during runtime. Lists in hardware circuits describe static structures that cannot be dynamically modified. To describe these structures in C/C++ we must use static arrays.

```
int main(){
  // Constructing three arrays
  array<int, 10> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  array<int, 10> arr1 = range<10, 0, 1>(); // {0...9}
  array<int, 2>  arr2 = construct(1, 2); // {1, 2}

  // Manipulating arrays
  int h = head(arr); // 0
  array<int, 9> t = tail(arr); // {1...9}
  array<int, 10> arr3 = prepend(h, t); // {0...9}
  array<int, 10> arr4 = append(t, h) // {1..9, 0}
  array<int, 20> arr5 =
    concatenate(arr, arr); // {0...9, 0...9}
  return;
}
```

Fig. 9. A variety of array constructors and manipulations.

For our higher-order functions we use the `array` class from the C++ Standard Template Library to provide list-like functionality for our higher-order functions. The `array` class has major benefits over pointers. Unlike pointers an `array` is parameterized by its length, and propagates this parameter through function arguments and for template inference. Second, `array` is a class with a copy operator. This means it can be returned from a function directly, unlike pointers, which must be passed as an argument to be modified in order to be "returned". This maintains a more Python-like feel for our functions.

Figure 9 shows severals examples of how C++ `array` objects are constructed and can be manipulated. While these arrays cannot be dynamically resized, our library also provides a collection of functions for static list manipulation. A few simple examples of list manipulations are shown in Figure 9. Thus, the `array` class allows us to provide software-like syntax for C++ hardware tools.

C. Recursion and Looping

Higher-order functions use loops or recursion to iterate over list elements. Dynamically typed languages like Python can use loops to implement recursion since intermediate type state is propagated during runtime. Statically typed languages like C++ must use recursion since the output of the previous iteration must type-check with the current at compile time. Since C++ hardware development tools are statically typed and no dynamic stack, we must use static recursion.

```
int main(){
  array<int, 10> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  int s = sum(arr); // s = 45
  return 0;
}
```

Fig. 10. Using the recursive array-sum from Figure 11

C++ static recursion uses a technique known as *template metaprogramming* [16]. Template metaprogramming is synthesizable because its is unrolled at compile time, bounded by compile-time template parameters, it eliminates the need for a dynamic stack. Template metaprogramming makes the resulting functions concise. This is shown in Figure 10, which calls `sum` to obtain the sum of an array of elements, and `sum`'s implementation in Figure 11.

```
1  template <size_t LEN>
2  struct _sum{
3    auto operator()(array<int, LEN> IN){
4      return head(IN) + _sum<LEN-1>()(tail(IN));
5    }
6  };
7
8  template <>
9  struct _sum<0>{
10   int operator()(array<int, 0> IN){
11     return 0;
12   }
13 };
14
15 template <size_t LEN>
16 int sum(array<int, LEN> IN){
17   return _sum<LEN>()(IN);
18 }
```

Fig. 11. An array-summation implementation using recursive class templates.

Figure 11 shows an implementation of `sum` that uses template recursion to iterate through the array. Lines 15-18 define the `sum` method with the template parameter `LEN`. This is preceded by two definitions of the `_sum` helper class. The first definition on Lines 1-6 is the recursive definition that is used when the template parameter `LEN` is non-zero. The second definition on Lines 8-13 is the base case for when `LEN` is zero. Together these implement the `sum` method.

When the `sum` method is called in Figure 11 the function creates an instance of `_sum<LEN>` and then calls the `()` operator. The `()` operator instantiates an instance of `_sum<LEN-1>` and calls `_sum<LEN-1>`'s `()` operator. This process continues until `LEN` is equal to 0 and `_sum<0>` return 0. When the program is run in software the program unwinds the call tree and adds the elements together. Since `LEN` is a static template parameter this recursion is bounded at compile time and can be synthesized.

D. Higher-Order Functions

We now have all of the pieces to develop our synthesizable higher-order functions: templates, arrays, functions, and recursion. We emphasize that the implementations of our higher-order functions are complex but that using our functions is quite simple as demonstrated in these examples.

We demonstrate our techniques by implementing the higher-order function `reduce` in Figure 12 and follow with an example in Figure 13. `reduce` is defined on Lines 18-22. When the function `reduce` is called templates are inferred as described in Section II-A3. The template parameter `LEN` specifies the length of the array, parameters `TI` and `TA` provide input polymorphism on the initial value and the array value respectively, and `FN` is the function class from Section II-A4. The output type is deduced by the `auto` keyword. `LEN` parameterizes the recursive class `_rhelp` defined on Lines 1-8. The base case when `LEN` is zero is defined on Line 10-16. The recursive behavior follows the description in Section II-C.

```
1  template <size_t LEN>
2  struct _rhelp{
3    template<typename TI, typename TA, class FN>
4    auto operator()(FN F, TI INIT, array<TA, LEN> IN){
5      return _rhelp<LEN-1>()(F, FN()(INIT, head(IN)),
6                             tail(IN));
7    }
8  };
9
10 template <>
11 struct _rhelp<0>{
12   template<typename TI, typename TA, class FN>
13   TI operator()(FN F, TI INIT, array<TA, 0> IN){
14     return INIT;
15   }
16 };
17
18 template <size_t LEN, typename TI, typename TA,
19           class FN>
20 auto reduce(FN F, TI INIT, array<TA, LEN> IN){
21   return _rhelp<LEN>()(F, INIT, IN);
22 }
```

Fig. 12. Implementation of the function `reduce` using all of the features described in this section.

Figure 13 shows how the array summation function from Figures 10 and 11 can be re-written using `reduce`. Again, this demonstrates that using our functions is quite simple despite the implementation complexity.

```
int main(){
  array<int, 10> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  int s = reduce(add, arr, 0); // s = 45
  return 0;
}
```

Fig. 13. Array summation from Figure 10 re-written using `reduce` defined in Figure 12

III. Examples

We demonstrate our work by implementing the Fast Fourier Transform (FFT) algorithm with our higher-order function library. We use this example to demonstrate our library and compare its syntax to Python, a modern, dynamically-typed productivity language. We have chosen FFT because it uses many of our higher order functions, is a well-known algorithm in the hardware development community, and has been used as a motivating related hardware development language work, [14], [17]. Further examples are available in our repository, and results are shown in Section IV.

The FFT algorithm is developed in several parts. Section III-A demonstrates `interleave`, which is used in the `bitreverse` function in Section III-B. Section III-C demonstrates how to implement an N-point FFT Stage function. Section III-D combines the previous sections into an implementation of the FFT algorithm.

A. Interleave

The `interleave` function interleaves two lists as shown in Figure 14. Figure 14a shows a graphical example of interleaving two lists. Figure 14b demonstrates a C++ implementation using our synthesizable library, and Figure 14c demonstrates a Python implementation.

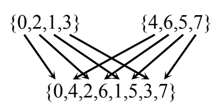
Figure 14b uses `zipWith` to apply the `construct` function to combine both arrays into a pair-wise array of arrays.

List Function	Description
`array<TA, 2> construct(TA, TA)`	Turn two elements into a 2-element array
`array<TA, LEN + 1> prepend(TA, array<TA, LEN>)`	Add an element to the head of an array
`array<TA, LEN + 1> append(array<TA, LEN>, TA)`	Add an element to end of an array
`array<TA, LENA + LENB> concatenate(array<TA, LENA>, array<TA, LENB>)`	Concatenate two lists into a single list
`TA head(array<TA, LEN>)`	Get the first element (head) of an array
`array<TA, LEN-1> tail(array<TA, LEN>)`	Get a list with all elements except the head (tail)
`array<pair<TA, TB>, LEN> zip(array<TA, LEN>, array<TB, LEN>)`	Combine two lists into a list of pairs
`pair<array<TA, LEN>, array<TB, LEN> unzip(array<pair<TA, TB>, L)`	Split list of pairs into a pair of two lists
Higher-Order Function	**Description**
`auto flip(FN)`	Return a new function with FN's input arguments swapped
`auto compose(FNA, FNB)`	Return a function where FNA is called on the output of FNB
`auto map(FN, array<TA, LEN>)`	Apply a function to each element in a list
`auto reduce(FN, array<TA, LEN>, TI)`	Iterate from left to right applying FN and carrying the result
`auto rreduce(FN, array<TA, LEN>, TI`	Iterate from right to left applying FN and carrying the result
`auto divconq(FN, array<TA, clog2(LOGLEN)>)`	Divide a list into elements and apply a function to pairs
`auto zipWith(FN, array<TA, LEN>, array<TB, LEN>)`	Combine two lists with a function

TABLE I

SUMMARY OF THE LIST-MANIPULATION AND HIGHER-ORDER FUNCTIONS USED IN THIS PAPER. FN IS SHORTHAND FOR A WRAPPED FUNCTION FROM SECTION II-D

Then, `rreduce` applies the `merge` function to attach the front of each 2-element array to the end of the previous array and produce an interleaving. The corresponding Python implementation is shown in Figure 14c, with `zip` instead of `zipWith` because Python tuples are easily converted to arrays.



(a)

```
struct Interleave{
  template <typename T, size_t LEN>
  auto operator()(array<T, LEN> L, array<T, LEN> R){
    auto pairs = zipWith(construct, L, R);
    return rreduce(concatenate, pairs,
                   array<T, 0>());
  }
} interleave;
```

(b)

```
def interleave(L,R):
    pairs = zip(L, R)
    concatenate = lambda p, lst: list(p) + lst
    return rreduce(concatenate, pairs, [])
```

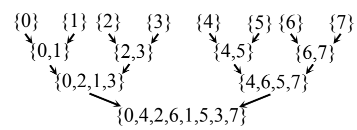
(c)

Fig. 14. (a) Interleaving two lists graphically (b) Interleaving two lists in C++ (c) Interleaving two lists in Python

B. Bit-Reverse

Figure 15 shows a bit-reverse permutation for arrays. In the permutation, the element at index N is swapped with the value at P, where P is equal to the a reversal of the bits of N. For example in an 8-element list, if $N = 1 = 3'b001$, then $P = 4 = 3'b100$. Figure 15a shows a bit-reversal permutation applied to the list $\{0, 1, 2, 3, 4, 5, 6, 7\}$ as a recursive interleaving. This is followed by the synthesizable C++ implementation in Figure 15b and the Python implementation in Figure 15c.

Figure 15b implements the bit-reverse permutation using the higher-order functions we have developed. `divconq` from Table I is used to divide the input array into single-element arrays. The function `interleave` from Section III-A is used to interleave the resulting arrays to produce the result.



(a)

```
struct Bitreverse{
  template <typename T, size_t LEN>
  auto operator()(array<T, LEN> IN){
    return divconq(interleave, IN);
  }
} bitreverse;
```

(b)

```
def bitreverse(in):
    return divconq(interleave, in)
```

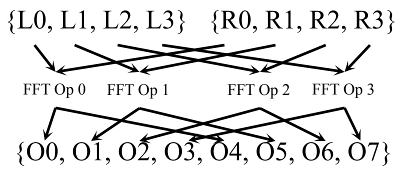
(c)

Fig. 15. (a) Bit-reverse permutation of a list graphically (b) Bit-reverse in C++ (c) Bit-reverse in Python

C. N-Point FFT Stage

The FFT algorithm is implemented by recursively applying an N-Point FFT function to two outputs of two N/2-Point functions in the previous stage. The N-point FFT stage is shown graphically in Figure 16a. In an N-point FFT inputs from the "left" and "right" inputs are passed with the context (tree level and index) to the `fftOp` function. `fftOp` performs computation and then produces two outputs that are de-interleaved in a similar fashion.

We define an `nPtFFTStage` function in Figure 16b. This function first computes its level (`LEV`) using the `LEN` template parameter. `LEV` is replicated and paired with an index to produce a context for each `fftOp` function. The function calls the `fftOp` function using `zipWith` to pass the context and input data. The output is de-interleaved using `unzip` and `merge`.



(a)

```
struct NPtFFTStage{
  template <typename T, size_t LEN>
  auto operator()(array<T, LEN> l, array<T, LEN> r){
    static const std::size_t LEV = log2(LEN) + 1;
    auto contexts = zip(replicate<LEN>(LEV),
                        range<LEN>());
    auto inputs = zip(l, r);
    auto outpairs = zipWith(fftOp, contexts, inputs)
    auto outputs = unzip(outpairs);
    return concatenate(outputs.first,
                       outputs.second);
  }
} nPtFFTStage;
```

(b)

```
def nptfftstage(L, R):
    lev = log2(len(L) + 1)
    contexts = zip([lev]*L, range(len(L)))
    inputs = zip(L, R)
    outpairs = map(fftOp, zip(contexts, inputs))
    outputs = zip(*outpairs)  # Unzip
    return outputs[0] + outputs[1]
```

(c)

Fig. 16. (b) N-Point FFT Stage using synthesizable higher-order functions (c) N-Point FFT Stage using Python

D. *Fast Fourier Transform*

We now have all the tools we need to implement widely-used Fast Fourier Transform (FFT) algorithm from Figure 2. The divide and conquer, recursive structure of the FFT algorithm is shown in Figure 2a. Each N-point stage of an FFT combines the results of two N/2-point transforms.

The FFT algorithm is implemented in Figure 17 using the higher-order function `divconq` with the `nPtFFTStage` function. The input list is bit-reversed to obtain the correct output ordering.

```
template <typename T, size_t LEN>
auto fft(array<T, LEN> sig){
  return divconq(nPtFFTStage, bitreverse(sig));
}
```

Fig. 17. FFT Implementation using the functions described in this section

As demonstrated, we have created a library of higher-order functions with a syntax that is similar to a modern productivity language, Python. This is evident from comparing examples in Figures 1, 2, 14, 15, 16, and 17. In all examples, the syntax and structure of the C++ code is very similar to Python. This is in spite of extensive use of C++ templates. More examples can be found in our repository, and in Section IV.

IV. Results

We report the quantitative results of our work by synthesizing six application kernels with our higher-order functions and compare them to loop-based implementations. The kernels we target are described in Section IV-A, followed by the experimental setup in Section IV-B and results in Section IV-C.

A. *Application Kernels*

1) Fast Fourier Transform: The *Fast Fourier Transform (FFT)* kernel was presented in Section III-D. The FFT kernel is widely used in signal analysis, often for compression. As demonstrated in Section III, the FFT kernel can be implemented with the higher-order `divconq` and `zipWith` functions. The loop-based equivalent is written with a pair of nested loops.

2) Argmin: Argmin is a function to compute value and index of the minimum element in an array. This function can be used in many streaming data processing applications such as databases [8], [11]. The *Argmin* kernel can be written using the higher-order function `divconq` and function `argminop` as an input. The `argminop` function returns the value and index of the minimum value from its subtrees. The loop-based equivalent is implemented with a pair of nested loops.

3) Finite Impulse Response Filter: A Finite Impulse Response (FIR) filter is a common signal processing kernel that takes samples from an input signal and performs a dot-product operation with an array of coefficients. These filters can be used for simple high-pass, and low-pass audio filtering. Our FIR filter is composed of two `zipWith` to perform the element-wise multiplication of the input signal array and coefficient array, and `divconq` to produce an addition tree to take the sum of all of the elements. The loop-based equivalent is written as a single for-loop that computes the element-wise multiplication, and a pair of nested for loops to implement the addition tree.

4) Insertion Sort: Insertion Sort is a streaming function that computes sorted lists. In our kernel, an array of values is streamed into the kernel. The kernel maintains a sorted order for the N minimum or maximum values seen and ejects others. This is identical to the implementation in [12]. The *Insertion Sort* kernel can be written using the higher-order function `reduce`, and a `compareswap` function. The `compareswap` function is applied to each element in the list and swaps the element depending on the sort criteria. The ejected element is carried forward for further comparison.

5) Bitonic Sort: Bitonic sort is a highly parallel sorting function used widely on GPU and FPGA architectures [12]. Its structure can be described recursively as a tree of butterfly networks.

6) Smith-Waterman: A systolic array is a common hardware structure for solving dynamic programming problems. In this example we use the Smith-Waterman algorithm from [13]. The systolic array is written as a composition of `zipWith` and `reduce`.

B. *Experimental Setup*

For each of the six algorithms described in Section IV-A we developed a loop-based and higher-order-function-based implementation, resulting in twelve designs. Each design targeted 16-element arrays, with types described in Table II. For each design we gathered performance, resource utilization,

Function Name (`Data Type`)	Higher-Order Functions FF	SRL	LUT	BRAM	DSP	Loop Based FF	SRL	LUT	BRAM	DSP
Fast Fourier Transform (FFT) (`ap_fixed<32,16>`)	21263	2487	8096	0	77	21240	2494	8096	0	77
Argmin (`int`)	2670	8	1573	0	0	2666	10	1575	0	0
FIR Filter (`float`)	14388	277	7306	0	48	14388	272	7305	0	48
Insertion Sort (`int`)	2300	0	935	0	0	2300	0	935	0	0
Bitonic Sort (`int`)	11929	1	4869	0	0	11929	1	4869	0	0
Smith-Waterman (`ap_int<2>`)	895	11	1187	0	0	895	11	1186	0	0

TABLE II
POST-IMPLEMENTATION RESOURCE UTILIZATION FOR SIX ALGORITHMS ON A PYNQ (XC7Z020) IN VIVADO 2017.4.

and maximum frequency results for loop-based and higher-order-function-based variants of the six algorithms. Our results were gathered in Vivado 2017.4 and implemented on a PYNQ development board with a Zynq XC7Z020 SoC.

Performance results were gathered from the Vivado HLS synthesis tool. The tool targeted a clock period of 2 nanoseconds (500 MHz) to guarantee that the architecture was fully pipelined and completely unrolled. These results are reported in Table III.

Resource utilization and $\mathrm{F}_{\mathtt{max}}$ results are reported from a sweep of thirteen Vivado Implementation goals. Resource utilization did not vary across these thirteen runs and are reported in Table II. For each goal we performed a binary search for the maximum frequency, varying the output frequency of the Xilinx Clock Wizard attached to the hardware core. The resulting statistics are reported in Table IV. Finally, Table V presents a statistical analysis of the maximum frequency data we collected.

C. Analysis

Performance results are shown in Table III. Column 1 displays the name for each of the six application kernels. Columns 2 and 3 show the initiation interval and latency for each higher-order-function-based application kernel. Likewise, columns 4 and 5 show the initiation interval and latency for each loop-based application kernel.

Function Name	HOFs Interval (Cycles)	HOFs Latency (Cycles)	Loop-Based Interval (Cycles)	Loop-Based Latency (Cycles)
FFT	1	59	1	59
Argmin	1	7	1	7
FIR Filter	1	65	1	65
Insertion Sort	1	31	1	31
Bitonic Sort	1	21	1	21
Smith-Waterman	1	16	1	16

TABLE III
PERFORMANCE RESULTS FROM VIVADO HLS 2017.4 FOR SIX APPLICATION KERNELS ON 16-ELEMENT LISTS

From Table III we conclude that our higher-order functions produce equal performance to fully-unrolled loop-based designs. This is evident from comparing the initiation intervals of columns 2 and 4 and the latencies of columns 3 and 5 in Table III. For all designs, higher-order implementations are equal to fully-unrolled loop-based implementations. We conclude there are no performance penalties associated with our higher-order functions.

Post-Implementation resource utilization is shown in Table II. Columns 2-6 in Table II show resource utilization for applications written with our higher-order functions and columns 7-11 show resource utilization for applications written with fully-unrolled loops.

From Table II we conclude that our higher-order functions implementations consume similar resources to loop-based implementations. Higher-order function and loop-based implementations consume equal numbers of DSPs and no BRAMs. The two methodologies also consume similar numbers of Flip-Flops (FFs), Look-Up-Tables (LUTs), and Shift-Register-Look-up-tables (SRLs): The maximum resource difference between the two implementation methodologies is less than 10 resources, a less than 1% difference in most cases. Given these small differences, we conclude that our functions do not produce significant differences in resources consumed.

We have a theory for this behavior. Vivado HLS and SDSoC use an LLVM backend to generate verilog circuits. This verilog is emitted from LLVM IR. We theorize that this behavior is because LLVM IR produced by our higher-order functions is identical to the LLVM IR produced by loop-based designs after code transformations have been performed. However, the designs above generate large LLVM IR files, and it is difficult to differentiate the structures. However, results shown here are consistent with this theory.

Post-Implementation frequency statistics are shown in Table IV. The statistics in Table IV are gathered from a sweep of 13 Vivado Implementation goals. Columns 2-4 show the mean, median, and standard deviation of frequency results for application kernels implemented with higher-order functions, and columns 5-7 show statistics for loop-based designs.

Function Name	HOFs F_{max} (MHz) Mean	Med.	Std.	Loop F_{max} (MHz) Mean	Med.	Std.
FFT	123.56	124.22	3.35	123.56	123.44	3.18
Argmin	110.64	110.94	2.51	110.91	109.77	2.89
FIR Filter	166.80	165.63	3.12	165.56	166.41	3.57
Insertion Sort	162.83	162.11	3.22	166.47	164.84	5.02
Bitonic Sort	112.77	112.11	3.50	113.85	115.63	2.76
Smith-W.	103.73	104.69	4.19	103.58	105.47	5.34

TABLE IV
MAXIMUM FREQUENCY STATISTICS FROM 13 IMPLEMENTATION RUNS OF VIVADO 2017.4

To determine whether the average maximum frequency of higher-order-function-based kernels differs statistically from loop-based kernels we perform an exact permutation test with the null hypothesis of equal means [18], [19]. In considering whether to reject the null hypothesis, we adjust for multiple comparisons using Holm-Bonferroni (H-B) correction, this is necessary because we test each of the six designs independently. The resulting p-values and $\alpha = 0.05$ H-B rejection thresholds are reported in Table V.

Function Name	p-value	$\alpha = 0.05$ thresh.
FFT Algorithm	0.97656	0.05
Argmin	0.74976	0.01667
FIR Filter	0.22267	0.01
Insertion Sort	0.00830	0.00833
Bitonic Sort	0.23193	0.0125
Smith-Waterman	0.91406	0.025

TABLE V
p-VALUES AND H-B REJECTION THRESHOLDS FROM A PERMUTATION TEST WITH THE NULL HYPOTHESIS OF EQUAL MEANS

Out of the six designs, only one (Insertion Sort) would be rejected at the $\alpha = 0.05$ level, and only just barely. Given the results of the Table V analysis, we conclude that our functions produce maximum frequency results that are generally statistically indistinguishable from those of loop-based designs.

V. Related Work

A. Hardware Development Languages

There have been several hardware-development projects that bring functional languages to bring higher-order functions to hardware development. Lava [14] and Cλash [17] are functional hardware development languages embedded in Haskell that provide higher-order functions and polymorphism to users. Lava is interesting because the operators are composed from functional definitions of Xilinx primitives, which provides a function abstraction for the user and context for the compiler to improve synthesis.

Higher-order functions originate from, but are not limited to purely-functional languages. The Chisel project [15] uses Scala and provides higher-order functions. Several projects have used Python for hardware development, for example, PyMTL [20] is a project that embeds a hardware development language in Python to raise the level of abstraction. These projects provide higher-order functions, imperative syntax, and polymorphism to generate circuits.

However, HDL projects fail to raise the designer's level of abstraction. The notion of wiring together operations, scheduling, registers, and clocks is pervasive. These concepts are unfamiliar to software developers. In addition, HDL languages do not generate complete systems. C/C++ synthesis tools completely abstract the detailed wiring, scheduling, and clocking concepts and automate core integration with communication and memory interfaces [1], [4], [5].

B. High-Level Synthesis Languages

High-level synthesis tools were developed to eliminate scheduling, wires, and registers from designer control - but few support higher-order functions. The Bluespec [21] language is one tool that provides higher-order functions. Bluespec is written written as a set of rules that are executed when prerequisites are met. These rules are scheduled by the Bluespec compiler to create a Verilog circuit.

Despite its obvious advantages the syntax and structure of the Bluespec language is substantially different than modern software languages. Our work provides a syntax that is similar to modern software languages and still provides higher-order functions, and automatic scheduling.

C. Domain-Specific Languages

In [10] the authors develop a domain-specific language with higher-order functions to generate parallel hardware. This domain-specific language is scheduled, translated into Maxeler's vendor-specific dataflow hardware development language, and finally deployed onto the vendor system. By using higher-order functions the authors can deploy highly-parallel systems, with low verbosity and high productivity for software engineers.

Our work does not use a domain specific language. Instead, we provide a familiar software API within C++ synthesis tools. By targeting the C++ compiler toolchain we can rely on a large body of existing work on optimization passes to improve our quality of result.

In addition, [10], [9], and [**?**] are highly complementary to our own. In [10] the authors state: "generating imperative code from a functional language only to have the HLS tool attempt to re-infer a functional representation of the program is a suboptimal solution because higher-level semantic knowledge in the original program is easily lost." Similarly, [9] motivates the need for parallel patterns in C++. We believe our work is a basis for both of these works. We have generated higher-order function interfaces for C++ synthesis, eliminating the need to "re-infer a functional representation".

D. Our Work

In our work we develop a library of higher-order functions for C/C++ synthesis tools. Using C/C++ synthesis tools avoids the pitfalls of HDLs: low-level wiring, registers, scheduling, and interfaces. Unlike prior work in high-level synthesis, our work is synthesizable to hardware and available in standard tools without modifications. Finally, it provides a syntax similar to a modern dynamically typed productivity language within C++.

VI. Conclusion

In this work, we have demonstrated a library of synthesizable library of higher-order functions for C++. These functions mimic the syntax of a modern dynamically-typed productivity language despite being written in a statically-typed language, for a tool with limited memory primitives.

We demonstrated how we build our higher order functions using C++ templates, and new features in the C++ standard. The library we created uses extensive C++ templates but the API we produced is simple and has similar syntax to Python.

Our results demonstrate that our code generates highly-similar hardware to traditional loop-based high-level synthesis: performance results were equal, differences in resources consumed were small, and the distributions of the maximum frequencies were generally statistically indistinguishable.

There are challenges ahead for this work: First, defining functions is more verbose than in other languages. Second, our work currently instantiates completely parallel computation kernels. Further work is needed to break these kernels into iterative sub-problems and provide a trade-off between performance and area.

In summary, we have made measure steps toward increasing the accessibility of C++ synthesis tools by providing common abstractions that are present in software environments.

Acknowledgments

The authors acknowledge the National Science Foundation Graduate Research Fellowship Program, the Powell Foundation, and ARCS foundation for their support.

References

[1] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "Sdsoc: A higher-level programming environment for zynq soc and ultrascale+ mpsoc," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 4–4.

[2] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 531–534.

[3] D. P. Singh, T. S. Czajkowski, and A. Ling, "Harnessing the power of fpgas using altera's opencl compiler," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2013, pp. 5–6.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[5] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field Programmable Logic and Applications*, Sept 2013, pp. 1–4.

[6] L. Ceze, M. D. Hill, and T. F. Wenisch, "Arch2030: A vision of computer architecture research over the next 15 years," *CoRR*, vol. abs/1612.03182, 2016.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] R. Mueller and J. Teubner, "Fpga: What's in it for a database?" in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 999–1004.

[9] L. Josipovic, N. George, and P. Ienne, "Enriching c-based high-level synthesis with parallel pattern templates," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 177–180.

[10] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," *SIGPLAN Not.*, vol. 51, no. 4, pp. 651–665, Mar. 2016.

[11] D. Lee, A. Althoff, D. Richmond, and R. Kastner, "A streaming clustering approach using a heterogeneous system for big data analysis," in *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*. IEEE, 2017, pp. 699–706.

[12] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner, "Resolve: Generation of high-performance sorting architectures from high-level synthesis," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 195–204.

[13] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 161–168.

[14] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," in *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '98. New York, NY, USA: ACM, 1998, pp. 174–184.

[15] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovi, "Chisel: Constructing hardware in a scala embedded language," in *DAC 2012*, June 2012, pp. 1212–1221.

[16] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, ser. C++ in-depth series. Addison-Wesley, 2005.

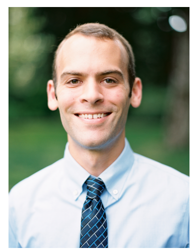
[17] C. Baaij, "Cλash : from haskell to hardware," December 2009.

[18] R. A. Fisher, *The design of experiments*. Oliver And Boyd; Edinburgh; London, 1937.

[19] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[20] D. Lockhart, G. Zibrat, and C. Batten, "Pymtl: A unified framework for vertically integrated computer architecture research," in *MICRO 2014*, Dec 2014, pp. 280–292.

[21] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, June 2004, pp. 69–70.



Dustin Richmond is a PhD Candiate at the University of California, San Diego. Dustin received a MS in Computer Engineering from the University of California, San Diego in 2014, and two BS degrees from the University of Washington in 2012. Dustin's research interests include programming languages, reconfigurable systems, and hardware design. Dustin was awarded a National Science Foundation Graduate Research Fellowship in 2012 and a Powell Fellowship in 2013.

Alric Althoff Alric Althoff is a Research Scientist at Leidos in San Diego. He receieved his PhD in Computer Science in 2018 under professor Ryan Kastner at the University of California, San Diego. He received his B.S. in 2013 in Cognitive Science and Mathematics. His primary areas of interest are statistics, signal processing, and machine learning, focusing on hardware security, experimental design in environments where individual experiments are costly, and high-throughput data analysis. Alric was awarded a National Science Foundation Graduate Research Fellowship in 2014.

Ryan Kastner Ryan Kastner is currently a professor in the Department of Computer Science and Engineering at the University of California, San Diego. He received a PhD in Computer Science at UCLA, a masters degree (MS) in engineering and bachelor degrees (BS) in both Electrical Engineering and Computer Engineering, all from Northwestern University. He leads the Kastner Research Group whose current research interests fall into three areas: hardware acceleration, hardware security, and remote sensing. He is the co-director of the Wireless Embedded Systems Master of Advanced Studies and Engineers for Exploration programs