Resolve: Generation of High-Performance Sorting Architectures from High-Level Synthesis

Janarbek Matai*, Dustin Richmond*, Dajung Lee†, Zac Blair*, Qiongzhi Wu*, Amin Abazari*, and Ryan Kastner*
*Computer Science and Engineering, †Electrical and Computer Engineering
University of California, San Diego, La Jolla, CA 92093, United States
{jmatai, drichmond, dal064, zblair, qiw035, maabazari, kastner}@ucsd.edu

ABSTRACT

Field Programmable Gate Array (FPGA) implementations of sorting algorithms have proven to be efficient, but existing implementations lack portability and maintainability because they are written in low-level hardware description languages that require substantial domain expertise to develop and maintain. To address this problem, we develop a framework that generates sorting architectures for different requirements (speed, area, power, etc.). Our framework provides ten highly optimized basic sorting architectures, easily composes basic architectures to generate hybrid sorting architectures, enables non-hardware experts to quickly design efficient hardware sorters, and facilitates the development of customized heterogeneous FPGA/CPU sorting systems. Experimental results show that our framework generates architectures that perform at least as well as existing RTL implementations for arrays smaller than 16K elements, and are comparable to RTL implementations for sorting larger arrays. We demonstrate a prototype of an end-to-end system using our sorting architectures for large arrays (16K-130K) on a heterogeneous FPGA/CPU system.

1. INTRODUCTION

Sorting is an important, widely studied algorithmic problem [13] that is applicable to nearly every field of computation: data processing and databases [6, 11, 20], data compression [5], distributed computing [9], image processing, and computer graphics [4, 15]. Each application domain has unique requirements. For example, text data compression applications requires sorting arrays with few hundred elements. MapReduce sorts millions of elements. Database applications sort both large and small size arrays.

The importance of sorting has led to the development and study of parallel sorting algorithms [1] on CPUs [8], GPUs [22], and FPGAs [14]. Each platform has its advantages. CPUs are relatively easy to program, but often lack performance compared to GPU and FPGA counterparts. GPUs are more difficult to program than CPUs, but they provide high performance. FPGAs typically provide the best performance for per Watt compared to CPUs and GPUs, but they are the most difficult to develop.

Designing efficient sorting applications using FPGAs is difficult because it requires substantial domain specific knowledge about hardware, the underlying FPGA architecture, and the compiler tools. High-level synthesis (HLS) tools aim to improve the accessibility of FPGAs by minimizing required domain specific knowledge by raising the level of programming abstraction, which results in an increase in productivity. Unfortunately, HLS is not a panacea. As reported in previous works, HLS generates efficient hardware when the input code is written in a specific coding style [10, 17], which we call *restructured* code. Therefore, creating optimized hardware using HLS still requires intimate understanding of the underlying hardware architecture and knowledge about how to effectively utilize the HLS tools.

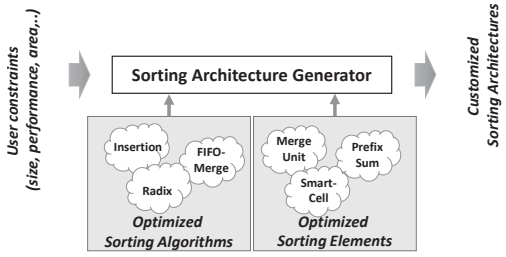

Figure 1: The Resolve sorting framework.

In this paper, we develop a framework that generates high performance sorting architectures by composing basic sorting architectures implemented with optimized HLS primitives. This concept is shown in Figure 1. We note that this is similar to `std::sort` routine found in standard template library (STL), which selects a specific sorting algorithm from a pool of sorting algorithms. For example, STL uses insertion sort for small lists (less than 15 elements), and then switches to merge sort for larger lists. We believe a routine like `std::sort` for HLS is important to facilitate FPGA designs for non-hardware experts. Our framework uses RIFFA [12] to integrate sorting cores into a fully functional heterogeneous CPU/FPGA sorting system. The result is a system that minimizes knowledge required to design high performance sorting architectures for an FPGA.

The specific contributions of this paper are:

1. The design and implementation of highly optimized sorting primitives and basic sorting algorithms.

FPGA '16, February 21–23, 2016, Monterey, CA, USA.

DOI: http://dx.doi.org/10.1145/2847263.2847268

2. A framework to generate hybrid sorting architectures by composing these basic primitives.

3. A comparison of these generated sorting architectures with other sorting architectures implemented on a FPGA.

4. Integration with RIFFA [12] to demonstrate full end-to-end sorting system.

This paper is organized as follows: Section 2 provides a case study of insertion sort to demonstrate HLS optimizations. Section 3 discusses related work. Section 4 describes the optimization of standard sorting primitives, and how to use them to create efficient architectures for ten basic sorting algorithms. Section 5 presents our Resolve framework. Section 6 provides experimental results. We conclude in Section 7.

2. CASE STUDY: INSERTION SORT

Listing 1 shows a common implementation of an insertion sort algorithm. Implementing this directly using a high-level synthesis (HLS) tool would not provide an efficient architecture. We must optimize it specifically for a hardware implementation.

```
1 void InsertionSort(int array[n])
2 {
3   L1:
4   int i, j, index;
5   for (i=1; i < n; i++)
6   {
7       L2:
8       index = array[i];
9       j = i;
10      while ( (j > 0) && (array[j-1] > index) )
11      {
12        L3:
13        array[j] = array[j-1];
14        j--;
15      }
16      array[j] = index;
17  }
18 }
```

Listing 1: Typical source code for insertion sort. This does not create an optimized architecture using HLS tools.

HLS tools typically provide optimization directives that are embedded in input source code as a *pragma*. Throughout this work, we use semantics specific to the Xilinx Vivado HLS tool. However, these ideas are generally applicable to other HLS tools. Some common optimization directives are pipeline, which exploits instruction level parallelism, unroll, which vectorizes loops, and partition, which divides arrays into multiple memories. We denote three potential locations for these directives: `L1`, `L2`, and `L3`. For example, we can direct the HLS tool to exploit instruction level parallelism by applying the `pipeline` pragma to the body of the inner for loop at point `L3`; similarly, we can apply other HLS optimizations at `L1`, `L2`, and `L3`. Unfortunately, as we will shortly see, designers cannot rely on these directives alone, and must often write special code, which we call restructured code, to generate the best results. This restructured code requires substantial hardware design expertise [10, 17].

Table 1 presents the Initiation Interval (II), achieved clock period, and utilization (slices) results for five different optimizations at the different locations `L1`, `L2`, and `L3`. Design 6 is a restructured implementation, i.e., it completely refactors the code with an eye towards an HLS style of coding. We will discuss Design 6 in more detail in Section 4.

Table 1: Case study for insertion sort optimization in HLS

	Optimizations	II	Period	Slices	Category
1	L3: pipeline II=1	661	3.75	29	slow/small
2	L3: unroll factor=2 cyclic partition array by factor=2	730	3.84	112	slow/small
3	L2: pipeline II=1	1194	3.06	47	slow/small
4	L2: unroll factor=2 and cyclic partition array by factor=2	1193	3.50	144	slow/small
5	L1: pipeline II=1 and complete partition array	1	440.85	27291	faster/huge
6	Code restructuring	64	2.90	374	fastest/small

We categorize the performance and area results from Table 1 into three groups: 1) slow/small, 2) faster/huge, and 3) fastest/small. Ideal design from HLS would be fast with small area. The first four designs are very slow and have small area. Design 5 achieves higher performance (II=1 and very large clock period) with unrealistically large area due to aggressive HLS optimizations. Design 6 is hand written by an expert HLS designer to create an optimal architecture; it achieves the highest performance with small area.

This case study demonstrates several concepts: First, writing efficient HLS code requires that the designer must understand hardware concepts like unrolling and partitioning. Second, the HLS designer must be able to diagnose any throughput problems, which requires substantial HLS tool knowledge. Third, and most importantly, in order to achieve the best results – high performance and low-area, it is typically required to re-write the software-centric code to create an efficient hardware architecture.

The aim of this work is to make it easy to design optimized sorting algorithms (like that of Design 6) from higher-level languages by providing a framework of optimized sorting algorithms in HLS. This requires several steps: 1) understand the sorting algorithms, 2) study existing hardware implementations (often written in register transfer level Verilog or VHDL), and 3) modify the sorting algorithms to optimally synthesize to the FPGA. In the remainder of this paper, we address each of these issues.

3. RELATED WORK

There are two main bodies of past work related to this paper; these are hardware sorting architectures and high-level synthesis code generation.

Hardware sorting architectures: The first body of work focuses on implementing hardware sorters (usually a single algorithm) on an FPGA. There are a variety of published works exploring sorting architectures on FPGA platforms. Several works have implemented a single sorting algorithm on a FPGA [3, 7, 16, 18, 21, 23], and some have explored high performance sorting of large size inputs [6, 7, 14].

All the above work focuses on designing a specific hardware architecture for a particular algorithm. Our work enables the user to generate a vast number of different sorting architectures from high-level languages without writing low-level code. Additionally, our work does this automatically from high-level languages, where previous works have used low level hardware description languages. Our framework allows full parameterization, the composition of hybrid ar-

chitectures from multiple algorithms, and the ability to perform quick design space exploration. Finally, the sorting architectures generated from our work can be integrated with RIFFA to provide an an end-to-end system.

There are also a few works that study sorting in the context of high-level languages. Arcas-Abella et al. [2] looked at the feasibility of implementing bitonic sort and spatial insertion sorting units using existing HLS tools (BlueSpec, Chisel, LegUP, and OpenCL). This work is similar to ours since it studies the implementation of sorting algorithms using HLS tools. Zuluaga et al. [23] presented a method for generating sorting network architectures from a domain-specific language. At a high level, the use of a domain-specific language seems similar to our architecture-generation approach. There are several main differences between aforementioned work and our work. First, we study multiple algorithms instead of focusing on a single algorithm. Second, we generate optimized sorting architectures by composing one or more algorithms. Finally, we can address much larger input sizes and the architectures generated from our work are orders of magnitude better than [23]. Section 6 provides a more detailed comparison of these works and the results generated from our Resolve framework.

HLS code generation: The work by George et al. [10] proposed a domain-specific language based FPGA design using existing high-level synthesis tools. This is similar to our approach by allowing non-hardware designers to write code (in their case using Scala) to generate optimized HLS code. Their work targets specific computational patterns. Our work targets a specific domain (sorting) and creates a framework for the user to explore a vast number of different sorting architectures using sorting primitives and basic sorting algorithms.

4. HARDWARE SORTING

Figure 1 shows the structure of our framework. It has three components: 1) Block 1 is a library of optimized parameterizable sorting primitives. These sorting primitives are the building blocks of our framework. Block 2 represents our basic sorting algorithms. The algorithms use the sorting primitives to implement all the basic sorting algorithms on an FPGA using high-level synthesis. Block 3 is the sorting architecture generator. Here we use the sorting primitives and basic algorithms to generate optimized hybrid sorting architectures to meet different system constraints. The following describes each of these components in more detail.

4.1 Sorting Primitives

This section presents optimized HLS implementations of sorting primitives. Previous works presented a list of several common sorting primitives, e.g., compare-swap, select-value, and a merge unit [14]. After analyzing more common sorting algorithms, we added three more primitives to this list: prefix-sum, histogram, and insertion-cell. Our basic sorting algorithms (presented in Section 4.2) are implemented efficiently in hardware using these six sorting primitives. Figure 2 shows the initial hardware architectures generated from HLS code for our sorting primitives. Section 2 described how *restructured* HLS code is necessary to generate an efficient hardware from HLS. We now present the optimization of prefix sum, merge, and insertion-cell.

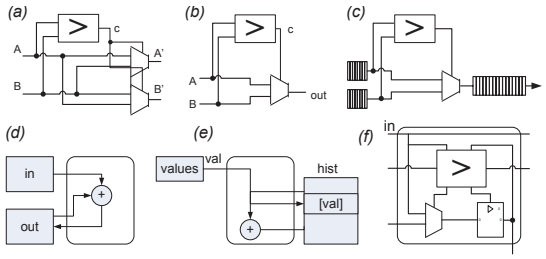

Figure 2: Initial hardware architecture of sorting primitives generated from HLS. a) compare-swap, b) select-value element, c) merge, d) prefix-sum, e) histogram, f) insertion cell

```
1 #pragma PARTITION out
      cyclic factor=4
2 #pragma PARTITION in
      cyclic factor=4
3 for(i=0;i<SIZE;i++){
4   #pragma UNROLL
        factor=4
5   #pragma PIPELINE
6   out[i]=out[i-1]+in[i]
7 }
```

Listing 2: Prefix sum (SW)

```
1 A=in[0];
2 #pragma PARTITION out
      cyclic factor=4
3 #pragma PARTITION in
      cyclic factor=4
4 for(i=0;i<SIZE;i++){
5   #pragma UNROLL
        factor=4
6   #pragma PIPELINE
7   A = A+in[i];
8   out[i] =  A;
9       }
```

Listing 3: Prefix-sum (HW)

Prefix Sum: Listing 2 shows "software-style" C code for prefix sum. Even for this simple prefix sum primitive, we have to restructure the code in non-intuitive ways to produce optimized hardware. First, we apply unroll and pipeline optimizations to expose data and instruction level parallelism. We also perform cyclic partitioning on the arrays *in* and *out* to match the memory access patterns required by unrolling. By pipelining the loop, we expect to get $II = 1$, and by unrolling, we expect to get a speed up by a factor of 4. However, the data dependencies between $out[i-1]$ and $out[i]$ prevent us from achieving the expected results. Figure 2 (d) shows the hardware architectures for the code in Listing 2. Optimized architecture with an $II = 1$ is shown in Figure 3 (a), and Listing 3 shows the HLS code for this optimal hardware architecture.

```
1 #pragma HLS DATAFLOW
2 //omitted partition
3 //pragmas
4 stage1(IN, TEMP);
5 ...
6 stage(TEMP, OUT);
7 }
```

Listing 4: Prefix sum dataflow

```
1 stage1(in, t) {
2   for(i=0; i<SIZE;
        ++i) {
3     #pragma HLS UNROLL
          factor=4
4     #pragma HLS
          PIPELINE
5     t[i] =
          in[i-1]+in[i];
6   }}
```

Listing 5: Prefix sum stages

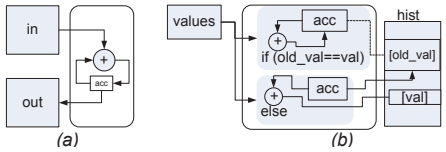

Figure 3: Optimal hardware architectures for prefix sum and histogram that give $II = 1$

As an additional example, we present another optimized HLS block for prefix sum which implements the reduction

pattern. The reduction pattern uses $\log(n)$ parallel stages to compute a prefix sum of size n in parallel. The individual stages do not have the data dependency seen in the previous example. Listing 4 shows a high-level prefix sum implementation using a reduction pattern. The `stage` functions are implementations of the parallel stages without the data dependency. Listing 5 shows the code for the first `stage` function. Since there is no data dependency, it is straightforward to get a speed up of 4× or more by unrolling and cyclically partitioning as in Listing 5. Multiple versions of optimized sorting primitives such as in Listing 3 and Listing 4 will facilitate to do easy design space exploration with these primitives. For example, the prefix sum in Listing 3 achieves the desired unrolling factor with reduced frequency, while the prefix sum in Listing 4 with the same unrolling factor achieves higher frequency.

```
1 void MergeUnit(hls::stream<int> &IN1,
      hls::stream<int> &IN2, hls::stream<int> &OUT ,
      int n){
2   int a,b;
3   int subIndex1 = 1, subIndex2 = 1;
4   IN1.read(a); IN2.read(b);
5   for(int i=0; i < n; i++){
6   #pragma HLS PIPELINE
7       if(subIndex1 == n/2+1) {
8           OUT[i] = b;
9           IN2.read(b);
10          subIndex2++;
11      } else if (subIndex2 == n/2+1) {
12          OUT[i] = a;
13          IN1.read(a);
14          subIndex1++;
15      } else if (a < b) {
16          OUT[i] = a;
17          IN1.read(a);
18          subIndex1++;
19      } else {
20          OUT[i] = b;
21          IN2.read(b);
22          subIndex2++;
23      }
24  }
25 }
```

Listing 6: FIFO based streaming merge primitive

Merge: The merge primitive combines two sorted $n/2$ size arrays into a sorted array of size n. Figure 2 (c) shows the hardware architecture. Listing 6 shows the HLS implementation of streaming FIFO-based merge unit. Implementation of merge unit with C arrays is straightforward. Here the `IN1` and `IN2` are two sorted arrays and `OUT` is the merged output. The `for` loop in Line 5 runs n times where $n/2$ is the size of `IN1` and `IN2`. It reads one element from either `IN1` or `IN2` on each iteration and writes it to the output until the end of the FIFO is reached. We pipelined this loop to get an $II = 1$ so it does one read operation every cycle.

Insertion Cell: Insertion cell is a hardware sorting primitive for insertion sort algorithms. The hardware architecture has an input, an output, a comparator, and a register – see Figure 2 (f). The insertion-cell compares the current input with the current value in current register. The smaller (or larger depending sort direction) of current register and the current input is given as an output.

```
1 T InsertionCell(hls::stream<int> &IN,
      hls::stream<int> &OUT){
2   static int CURR_REG=0;
3   int IN_A=IN.read();
4   if(IN_A>CURR_REG) {
5     OUT.write(CURR_REG);
6     CURR_REG = IN_A;
7   } else
8     OUT.write(IN_A);
9   return CURR_REG;
10 }
```

Listing 7: The code for the sorting primitive insertion-cell.

The code for insertion-cell is shown in Listing 7. The function takes one input argument `IN` and one output argument `OUT`. It uses a `hls::stream<>` type to indicate that these input and outputs can use a FIFO interface. The cell holds the previous value in the `CURR_REG` static variable. It must save this value across function calls, and thus declares it as a `static` variable. The architecture compares the input value to the previous value, and outputs the larger of these two values. The next section shows how to use this primitive to create a linear insertion sort algorithm.

4.2 Sorting Algorithms

In this section, we elaborate on the HLS implementations of four kinds of sorting algorithms: nested loop, recursive, non-comparison, and sorting network. Table 2 summarizes the results of our HLS implementations.

4.2.1 Nested Loop Sorting Algorithms

The *selection sort* algorithm iteratively finds the minimum element in an array and swaps it with the first element until the list is sorted . This algorithm runs in $O(n^2)$, where n is the number of array elements. In HLS, we can pipeline the inner loop to get $II = 1$, which still gives us $O(n^2)$ time. We can create a better design by sorting from both "sides", i.e., finding the minimum and maximum elements in parallel, which reduces the number of iterations in the outer loop by 2×. This gives us $O(n^2/2)$ time. In general, selection sort does not translate into high performance hardware using HLS. However, selection sort can be used to produce an area-efficient sorting algorithm implementation.

The *rank sort* algorithm sorts by computing the rank of each element in an array, and then inserting them at their rank index. The rank is the total number of elements greater than or less than the element to be sorted. Sequential rank sort has a complexity of $O(n^2)$. The rank sort algorithm can be fully parallelized in HLS: sorting an array of size n has n units operating in parallel computing the rank of each element. However, this process uses $2 \times n^2$ storage to sort the array of size n. Rank sort can be useful when designing sorting hardware in HLS because it is a good algorithm for exploring area and performance trade-offs.

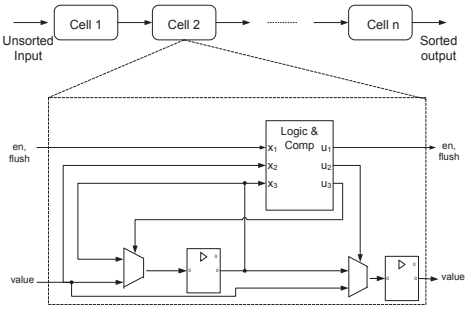

Figure 4: Hardware architecture of linear insertion sort

Insertion sort iterates through an input array maintaining sorted order for every element that it has seen. Insertion

*Table 2: Sorting Algorithms evaluations when implementing them using HLS. n=number of elements to sort. *n=number of insertion sort cells, t*= number of compare-swap elements*

		Parallel HLS Implementation			
Algorithm name	SW Complexity	Parallel tasks	Complexity (II)	Storage	Main Sorting Primitives
Selection sort	$O(n^2)$	2	$O(n^2/2)$	$O(2 \times n)$	Compare-swap
Rank sort	$O(n^2)$	n	O(n)	$O(n^2)$	Histogram, Compare-swap
Bubble sort	$O(n^2)$	2	$O(2 \times n^2)$	$O(2 \times n)$	Compare-swap
Insertion sort	$O(n^2)$	-	$O(n)$	n*	Compare-swap, insertion-cell
Merge sort	$O(n \log n)$	-	$O(n)$	$O(2 \times \sum logn)$	Merge Unit
Quick (Sample) sort	$O(n \log n)$ or $O(n^2)$	t	$O(n/t \log n/t)$	$O(n \times t)$	Prefix sum
Counting sort	$O(n \times k)$ (k=3)	3	$O(n)$	$O((k-1)n$	Prefix sum, Histogram
Radix sort	$O(n \times k)$ (k=4)	4	$O(n)$	$O((k-1)n$	Prefix sum, Histogram, Counting Sort
Bitonic sort	-	t	$O(log^2 n)$	$O(n \times t)$	Compare-swap
Odd-even transposition sort	$O(n^2)$	t*	$O(n^2/t^*)$	$O(t*)$	Compare-swap

sort has a complexity of $O(n^2)$. Listing 1 shows a software-centric HLS implementation of insertion sort. We discussed some naive HLS optimizations for insertion sort in Section 2. These used different optimization directives (`pragma`s) in an attempt to create a better hardware implementations. These designs (Designs 1 - 5 in Table 1 did not result in the optimal implementation. Design 6 give the best result. Here we describe code restructuring optimizations of Design 6.

An efficient hardware implementation of insertion sort uses an linear array of insertion-cells [2, 3, 16, 21] or a sorting network [19]. Here we focus on a linear insertion sort implementation; we discuss sorting network implementation later. Figure 4 shows architecture from Arcas-Abella et al. [2]. In this architecture a series of cells (insertion-cell primitives) operate in parallel to sort a given array. It compares the current input (IN) with the current value in current register (CURR_REG). The smaller of current register and the current input is given as an output to OUT.

Listing 8 shows the source code that represents the hardware architecture in Figure 4. A cascade of insertion-cells is implemented in a pipelined manner using the `dataflow` pragma, and series of calls to the `InsertionCell` function from Listing 7. Note that we have four different versions of the function – `InsertionCell1`, `InsertionCell2`, etc.. It is necessary to replicate the functions due to the use of the `static` variable. Each of these functions has the same code as in Listing 7. This implementation achieves $O(n)$ time complexity to sort an array of size n.

```
1 void InsertionSort(hls::stream<T> &IN,
      hls::stream<T> &OUT){
2   #pragma HLS DATAFLOW
3   hls::stream<T> out1, out2, out3;
4   // Function calls;
5   InsertionCell1(IN, out1);
6   InsertionCell2(out1, out2);
7   InsertionCell3(out2, out3);
8   InsertionCell4(out3, OUT);
9 }
```

Listing 8: Insertion Sort code for HLS design based on the hardware architecture in Figure 4. The `InsertionCell` functions use the code from Listing 7.

4.2.2 Recursive Algorithms

A pure software implementation of *merge sort* and *quick sort* are not possible in HLS due to the use of recursive functions. HLS tools (including Vivado HLS) typically do not allow recursive function calls. Changing from a recursive implementation to one that is synthesizable requires a modification of software implementation to remove the recursive function calls in the code.

Merge sort has two primary tasks. The first task partitions the array into individual elements, and the second merges them. The majority of the work is performed in the merging unit, which is implemented with a merge primitive. This was described in Section 4.1.

Merge sort is implemented in hardware using merge sorter tree [14] or using odd-even merge sort. Listing 9 provides an outline of the code for streaming merge sorter tree. In this code, `IN1`, `IN2`, `IN3` and `IN4` are $n/4$ size inputs, and `OUT` is a size n output. `MergePrimitive1` and `MergePrimitive2` merges two sorted lists of array size $n/4$ and $n/2$, respectively. Using the `dataflow` pragma, we can perform a functional pipeline across these three functions. Merge sort based on odd-even merge also uses merge sorting primitive to sort a given n size array with II of n. Merge sort can be optimized in hardware by running $n \log n$ tasks in parallel.

```
1 void CascadeMergeSort(hls::stream<int> &IN1,
2       hls::stream<int> &IN2, hls::stream<int> &IN3,
3       hls::stream<int> &IN4, hls::stream<int>
            &OUT){
4   #pragma HLS DATAFLOW
5   #pragma HLS stream depth=4 variable=IN1
6   for(int i=0;i<SIZE/4;i++) {
7     //read input data
8   }
9   MergePrimitive1(IN1, IN2, TEMP1);
10  MergePrimitive1(IN3, IN4, TEMP2);
11  MergePrimitive2(TEMP1, TEMP2, OUT);
12 }
```

Listing 9: FIFO based streaming merge sorter tree

Quick sort uses a randomly selected pivot to recursively split an array into elements that are larger and smaller than the pivot. After selecting a pivot, all elements smaller than pivot are moved left of the pivot, i.e., they are in a lower index in the array. This process is repeated for the left and right sides separately. The software complexity of this algorithm is $O(n^2)$ in the worst case and $O(n \log n)$ in the best case. Non-recursive (iterative) version of quick sort can be implemented in HLS with slow performance. Instead, we chose to implement a parallel version of quick sort known as sample sort. In sample sort, we can run t tasks to divide the work of `pivot_function` to sort n size array into n/t. The integration of t results from tasks can be done using the prefix sum primitive. Essentially, this implementation sorts an n size array in $O(n)$ time with higher BRAM usage.

4.2.3 Non-comparison based

Counting sort has three stages. First the counting sort computes the histogram of elements from the unsorted input array. The second stage performs a prefix sum on the histogram from the previous stage. The final stage sorts the array. Final stage first reads the value from the unsorted input array. Then it finds the first index of that element from the prefix sum stage and writes it to the output array. Then it increments the index in the prefix sum by one. Figure 5 (a) shows an example of the counting sort algorithm on an 8 element input array. The first stage performs a histogram on the input data. There are only three values (2, 3, 4), and they occur 3, 2, and 3 times in the unsorted input array, respectively. The second stage does a prefix sum across the histogram frequencies. This tells us the starting index for each of the three values. The value 2 starts at index 0; the value 3 starts at index 3; and the value 4 starts at index 5. The final stage uses these prefix sum indices to fill in the sorted array. Parallel counting sort can be designed using function pipelining of three stages. It runs in $O(n)$ time using $O(n \times k)$ (k is constant) memory storage.

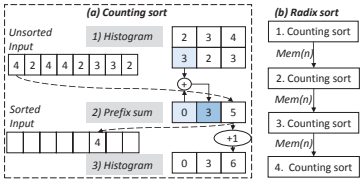

Figure 5: An example hardware architectures for counting sort and radix sort

Radix sort works by applying counting sort for each digit of the input data. For example, to sort 32-bit integers, we can apply counting sort four times to each of the four hexadecimal (radix 8) digits. We can implement a fully parallel radix sort in HLS using functional pipelining of each counting sort. An individual counting sort operation has a throughput of n, thus fully parallel radix sort will also have a throughput of n. To store the outputs of intermediate stages, we need $n \times k$ storage. Here k is usually 4 for 32-bit number or 8 for 64-bit number. Thus to sort 32-bit number in parallel, we use $3 \times n$ storage (3 intermediate memory storage) as shown in Figure 5 (b).

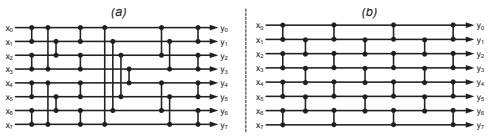

Figure 6: a) Bitonic sort, b) Odd-even transposition sort

4.2.4 Sorting networks

Sorting networks [19] is a set of compare-swap primitives connected by wires. *Bubble sort* is an instance of a sorting network. Two examples of sorting networks (bitonic and odd-even transposition) are shown in Figure 6. For each vertical connection, the minimum of two inputs is assigned to the upper wire and the maximum goes to the lower wire.

Due to parallel nature of sorting networks, they are easier to implement in HLS than other sorting algorithms. However, sorting networks does not scale well in hardware [14] due to required IO throughput. This requires balancing the parallelism and area in HLS and will be discussed later. For example using parallel n compare-swap elements, odd-even transposition sort can sort an n size array in $O(n)$.

5. SORTING ARCHITECTURE GENERATOR

In this section, we describe our framework for generating sorting architectures. A user can perform design space exploration for a range different application parameters. And once she has decided on a particular architecture, the framework generates a customized sorting architecture that can run on out of the box on a heterogeneous CPU/FPGA system. It creates the RTL code if the user wishes to integrate it into the system in another manner.

The flow for our sorting framework is shown in Figure 8. We define user constraint as a tuple $UC(T, S, B, F, N)$ where T, S, B, F and N are throughput, number of slices, number of block rams, frequency, and the number of elements to sort. We define V as a set of sorting designs that can perform sorting on an input array of size N. The sorting architecture generation is a problem to find a design D of the form $D(T, S, B, F, N)$ that satisfies the UC.

```
a) RD::= | RD v1 | RD v2| BD v3| RD v4| RD v5
   BtS::= | BtS v1 | BtS v2| BtS v3| BtS v4| BtS v5
   …
b) Sort ::=                c) match Sort (n, v)::=
     | SS n                      | SS n → emit SS (v)
     | RS n                      | RS n → emit RS (v)
     | BS n                      | BS n → emit BS (v)
     | IS n                      | IS n → emit IS (v)
     | MS n                      | MS n → emit MS (v)
     | QS n                      | QS n → emit QS (v)
     | RD n                      | RD n → emit RD (v)
     | BtS n                     | BtS n → emit BtS (v)
     | OET n                     | OET n → emit OET (v)
     | OEM n                     | OEM n → emit OEM (v)
     | Merge(Sort,Sort)
```

Figure 7: Grammar of domain-specific language. SS=Selection sort, RS=Rank sort, BS=Bubble sort, IS=Insertion sort, MS=Merge sort, QS=Quick sort, RD=Radix sort, BtS=Bitonic sort, OET=Odd-even transposition sort, OEM=Odd even merge sort. a) Sorting architectural variants for particular algorithm, b) Sort function grammar, c) Code generator

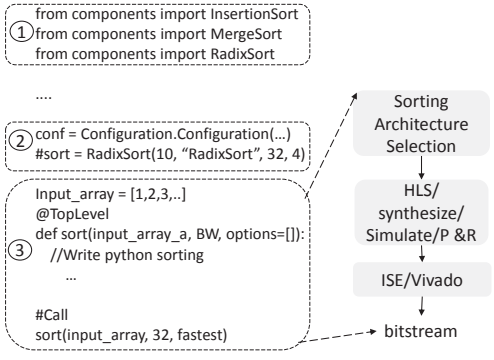

Figure 8: Design flow of Resolve.

Our framework is implemented as a small domain-specific language. Figure 7 shows simplified grammar of the language. The sorting architectures defined in previous sections are defined by types for instance, RD and IS. Each

sorting algorithm has a number of different implementations, called *variants*. For example, radix sort (RD) has five variants: $RD\_v1, RD\_v2, RD\_v3, RD\_v4, RD\_v5$,. The *sort* function can use any sorting algorithm or a composition of one or more algorithms. If we wanted to create an implementation that sorts n elements, we could define it as any of the basic sorting algorithms from Figure 7. For example, `SS n` creates a selection sort implementation, and `BS n` uses the bubble sort algorithm for the implementation. If we wish to create a hybrid sorting architecture we could perform `Merge(QS n/2, QS n/2)`, which uses quick sort on the two halves of the input data and merges the results together. The expression: `Merge(Merge(RD n/4, RD n/4), Merge(RD n/4, RD n/4))` splits the input data into quarters, and then merges them twice. The elements for the quarter arrays can be sorted using different sorting algorithms in our framework. In this example, radix sort is used to sort the quarter arrays. Based on the `Sort` function, the `emit` function generates specific variant of sorting architecture. Thus our framework completely abstracts the underlying architectural details from the user, and allows the user to generate an optimized architecture in a matter of minutes.

To use the framework, the user writes Python code as described in Figure 8. It has three components: Part ① is a library of the template generator classes for existing sorting algorithms (e.g., `InsertionSort`, `MergeSort`). There are currently eleven classes, some with multiple architecture variants. All these classes inherit from base class called `Sorting`. The `Sorting` class provides common class methods and members (e.g., size, bit width) for all the sorting algorithms. Each class provides parameterizable functions tailored to specific sorting algorithm. For example, $RadixSort.optimized\_II1(size, bit-width)$ generates optimized Radix sort with $II = 1$, while $functional\_pipelining$ $(size, bit\_width)$ generates a dataflow pipelined radix sort for a given parameters. Part ② is HLS project generator and configuration class. The configuration class accepts several parameters. These are the FPGA device, frequency, clock period, simulate_true, implement_true, and name of the module. If `simulate_true=1` then the generated design is simulated and verified with a selected simulator inside HLS. If the `implement_true=1`, then the design is physically evaluated by RTL synthesis.

The users write their top level function in Part ③; this calls the sorting routine. `TopLevel` is a Python decorator which allows us to add additional information to the existing Python function. Once `TopLevel` decorator starts executing, it does several things. First, it generates a customized sorting architecture tailored to user provided parameters using Algorithm 1. Here V is a set of all different variants of existing sorting architectures, and D and R are returned sorting design and respective simulation/implementations results. The user provides UC. UC must contain at least one element which is size of array to sort (N). If UC is one, then sorter generates a design from existing designs which has the highest throughput using `SorterGenerator` function. The `emitCode` function generates optimized sorting architectures using existing HLS architectures (templates) wrapped in python code. The `SorterGenerator` includes `CalculateThroughput` function that calculates throughput TS of current design using initial II of each variant. We assume the II of each variant is known. For example, we know linear insertion sort (LIS) has $II = 1$, so the $TS(LIS) = 1 \times N$. Then it generates design D and returns report R. In the case of $|UC| > 1$, we must satisfy user constraints; In Algorithm 1, we present a case where there is not a design in the current pool that satisfies UC (other case where there is a D that satisfies UC is straightforward). We use a heuristic approach that continuously divides N into halves until it finds a design that satisfies UC. For a returned design D from `SorterGenerator`, we call `CheckUserConstraints` to check these conditions: $UC(T) > D(T)$, $UC(S) < D(S)$, $UC(B) < D(B)$, $UC(F) > D(F)$. If these conditions meet, then emitMerge generates HLS code from pre-wrapped templates in python.

Algorithm 1: Customized Sorting Architecture Generation

```
   Data: UC={T, S, B, F, N}, V={V_1, V_2, ..V_m},
         P={N/2, N/4..}
   Result: D=architecture for UC, R=performance area
           results
 1 if UC is 1 then
 2 |  [D, R]=SorterGenerator(V, N)
 3 end
 4 else
 5 |  foreach (P) do
 6 |  |  [D, R]=SorterGenerator(V, P)
 7 |  |  if CheckUserConstraints(UC) then
 8 |  |  |  emitMerge(D, P)
 9 |  |  |  if sim/impl is 1 then
10 |  |  |  |  R = Simulate D
11 |  |  |  |  R =Implement D
12 |  |  |  end
13 |  |  end
14 |  end
15 end
16 Procedure SorterGenerator(V, N)
   |  Data: V, N
   |  Result: D : Design, R : Report
17 |  TS(1, 2, .., m)=CalculateThroughput(V, N)
18 |  S = min(V_1(t), V_2(t), ..V_m(t))
19 |  [D, R]=emitCode S
20 |  if sim/impl is 1 then
21 |  |  Simulate D Implement D
22 |  end
```

6. EXPERIMENTAL RESULTS

In this section, we present the performance and utilization results for a representative set of architectures generated by our framework, and the end-to-end (CPU/FPGA) implementation of selected sorting architectures. Finally, we compare our designs with existing implementations of sorting hardware architectures.

Basic Sorting Algorithms: We implemented basic sorting algorithms – selection sort, rank sort, linear insertion sort, merge sort (two variants), sample sort, radix sort (two variants), bitonic sort, and transposition sort (two variants) – for three different problem sizes (32, 2014, 16384). The results are shown in Table 3. Results presented in Table 3 are obtained after RTL synthesis targeting the Xilinx xc7vx1140tflg1930-1 chip using Vivado HLS 2014.3. The performance results are presented in terms of megabytes per

Table 3: Implementation results for different sorting architectures. Tasks=number of parallel sorting processes. Entries with '-' are omitted since the sorting architecture is not good for that particular size (e.g., the utilization is too high to fit on the target device).

		32				1024				16384			
Algorithm name	Tasks	Slices	BRAM	Freq	MB/s	Slices	BRAM	Freq	MB/s	Slices	BRAM	Freq	MB/s
Selection sort	2	26	0	266	50	410	12	232	3.5	599	192	171	97
Rank sort	2	119	4	389	508	162	16	419	4	504	256	348	< 10
Linear insertion sort	n	374	0	345	1380	12046	0	310	1243	-	-	-	-
Merge sort (P)	$\log n$	1526	18	164	954	2035	40	239	482	484	608	155	1244
Merge sort (UP)	$\log n$	666	18	180	550	1268	40	281	899	2474	832	177	567
MergeStream (P)	$\log n$	529	8	211	794	1425	20	189	756	2487	140	166	666
Sample sort	-	-	-	-	-	2777	218	228	911	5174	2838	127	510
8-bit Radix sort	4	1420	19	227	42	1500	36	230	202	1743	456	222	220
4-bit Radix sort	8	2146	30	353	223	2470	60	362	356	3352	960	289	289
Bitonic sort	-	4391	0	268	1073	3239	56	268	1048	7274	1280	230	922
Odd-even trans	8*2	929	33	342	96	1254	36	301	15	1361	128	225	0.8
Odd-even trans	16*2	1326	0	323	70	2209	68	270	29	2370	128	212	1.64
Merge (Stream)	-	221	0	395	1407	231	0	374	1490	255	0	368	1474
Merge4 + Radix	-	-	-	-	-	-	-	-	-	1010	168	244	411
Merge8 + Radix	-	-	-	-	-	-	-	-	-	2584	240	245	782
Merge16 + Radix	-	-	-	-	-	-	-	-	-	4786	320	148	858

second (MB/s). We show a broad set of implementations to highlight the ability of our framework to create a broad number of Pareto optimal designs rather than simply show the best results.

Selection sort and rank sort both have small utilization with limited throughput especially as the input size increases. Linear insertion sort has very high throughput, but it does not scale well as the number of slices has a linear relationship (to sort n size array, n insertion-cell is required) with the input size since we are directly increasing the number of insertion sort cells. Thus linear insertion sort architecture should only be used to sort arrays with small sizes (e.g, 512).

The designs Merge sort (P) and Merge sort (UP) are pipelined and unpipelined versions of cascade of odd-even merge [13]. Merge Stream (P) is the streaming version of the cascade of odd-even merge sort. Pipelined version of merge sort achieve better II except for size 1024. This is caused because HLS tool is doing loop level transformations when we do not have pipeline for size 1024. Sample sort tends to achieve higher throughput but uses more BRAMs than other sorting architectures.

The 8-bit radix sort has four parallel tasks; the 4-bit radix sort has eight parallel tasks. Radix sort provides a good area-throughput tradeoff. In the 4-bit implementation, doubling the area produces a greater than 4× speedup for 32 inputs. This trend does not continue for larger input sizes though the throughput does increase in all cases. This indicates that radix sort is suitable for medium size arrays. Bitonic sort achieves high throughput for, but it tends to use more BRAMs than merge sort. Thus, bitonic sort is suitable for sorting medium size arrays.

In the second part of Table 3, we present four hybrid sorting architectures. Merge (Stream) is a streaming version of merge sort that operates on pre-sorted inputs. It is designed for heterogeneous CPU/FPGA sorting where the smaller arrays are pre-sorted in CPU. Merge4+Radix is generated with the user constraints $UC(T = H, n = 16384, S < 1500, B < 170)$. This architecture uses merge primitve to combine four 4096-element radix sorts, which gives the highest throughput design with less than 170 Block RAMs ($B < 170$). Merge8+Radix and Merge16+Radix architectures divides the input array (similar to Merge4+Radix except they use more parallelism 8-way and 16-way) into 8 and 16, respectively. Then uses radix sort to sort the sub arrays.

Table 3 presents some of the basic sorting architectures. Once we have these kinds of sorting architectures, it is straightforward to generate even more sorting architectures for different user constraints. For example, we presented slices, achieved clock period and throughput results for streaming merge sort (pipelined (P) and unpipelined (UP)) in Figure 9. These results are obtained for different sizes and different user specified clock period. We only presented one case study here; we can generate broad number of Pareto optimal designs for aforementioned different sorting algorithms to meet different user constraints.

End-to-end sorting system: To the best of our knowledge, there is no published end-to-end system implementation of large sorting problems using architectures created from HLS. We implemented and tested a number of different sorting algorithms on a hybrid CPU/FPGA system using RIFFA 2.2.1 [12]. The HLS sorting architectures use AXI stream. The corresponding AXI signals are connected to signals of RIFFA. We present the area and performance of the several prototypes (sizes) in Table 4. In the first row of Table 4, we present the area results for RIFFA using only loop-back HLS module (i.e., an empty HLS module). This shows the overhead of RIFFA. The remaining results include RIFFA and the sorting algorithm. Results for 16384 and 65536 are obtained using the xc7vx690tffg1761-2 FPGA running at 125MHz, and PC with Intel Core i7 CPU at 3.6 GHz and 16 GB RAM. The CPU is used only to transmit and receive data. The sorting implemented on the FPGA can sort data at a rate of 0.44 - 0.5 GB/s. Our end-to-end system does not overlap communication and sorting times. Thus, it has an average throughput of 0.23 GB/s. The last line of Table 4 shows hybrid sorter results for 131072 size formed by two 65536 size sorters. CPU merges outputs of sub sorters. These results can be improved linearly by using more channels on RIFFA or increasing the clock frequency.

Comparison to previous work: We compare the results from our framework with the sorting networks from the Spiral project [23], interleaved linear insertion sort (ILS) [21], and merge sort [14]. We selected these because insertion sort is usually best suited for small size arrays, sorting networks are used for both small and medium size arrays, and a

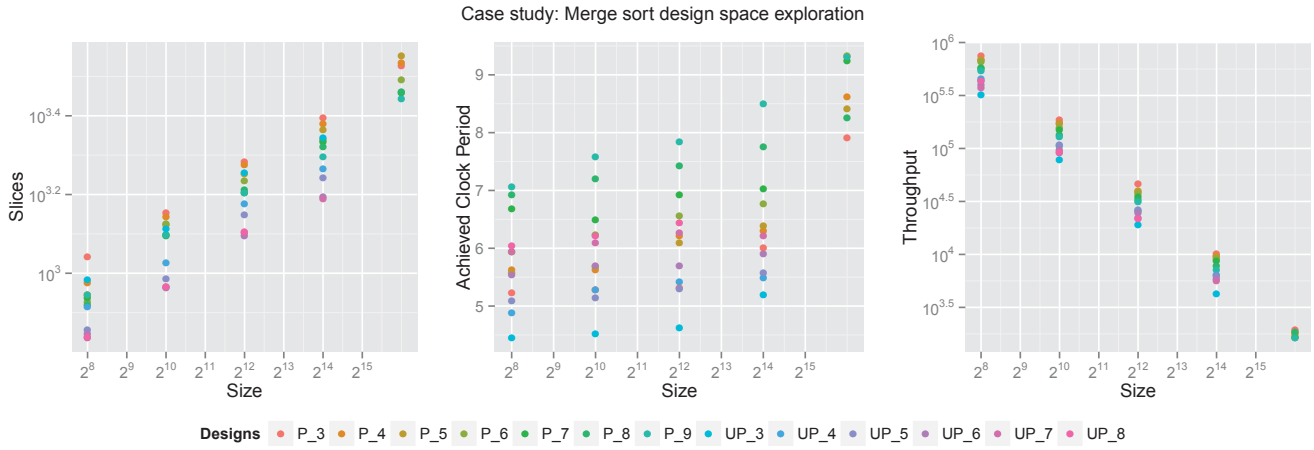

Figure 9: Design space expiration of generated architectures: P_X (X is user specified clock period and X = 3 to 10): piplelined and UP_X (X=3 to 10): unpipelined versions of merge sort.

*Table 4: Area and performance of end-to-end system. *HLS result of 131072 size hybrid sorter. +indicates CPU merging time).*

Design	Size	FF/LUT	BRAM	II
RIFFA	N/A	19472/16395	71	N/A
RIFFA+Sorting IP	16384	25118/20368	141	18434
RIFFA+Sorting IP	65536	26353/21707	333	73730
RIFFA+Sorting IP	131072	38436/31816	609	*73730+

merge sort is best for larger size arrays. Finally, we compare against the sorting architectures implemented in various different high-level languages [2].

First we compare our results (streaming merge sort) to sorting architectures from the Spiral project [23]. We used the same parameters in both cases: 32-bit fixed point type for all architectures, Xilinx xc7vx690tffg1761-2, streaming width of one (one streaming input and one streaming output), and 125 MHz frequency. Spiral generates five different sorting architectures (SN1, SN2, SN3, SN4, and SN5). SN1 and SN3 are high performance fully streaming architectures with large area. SN2 and SN4 balance area and throughput. And SN5 is an architecture optimized for area [23]. We compare against SN1, SN2, and SN5 because they provide a good balance between performance and area. For SN2, we generate fully streaming (SN2_S) and iterative (SN2_I) versions. We only compared our result against to the SN5 fully steaming version because the iterative version of SN5 has a very low performance (e.g., throughput of SN5 iterative version for size 1024 is 102621). We implemented these designs (SN1, SN2_I, SN_S, and SN5) using Vivado 2015.2. All of the results are presented after place-and-route.

Table 5 compares the four architectures from Spiral to our work. The throughput (II) is the number of clock cycles need to sort an array of n elements. We obtained Spiral throughput results from the report generated by online tool (http://www.spiral.net/hardware/sort/sort.html). The throughput of our work is obtained from Vivado HLS cosimulation. In each case, this is the II for sorting one n size array. The best design (fastest, small area) from Spiral project is SN2_S for 1024. SN_S uses 17.9× more BRAMS, 4.6× more FFs, 2.1× more LUTs than our merge sort implementation for the 1024 element array. The smallest design from Spiral is SN2_I. For example, to sort a 16384 element array, SN2_I uses 13.7× more BRAMs, and its throughput is 14× worse than our merge sort implementation. SN1 and SN5 for 16384 size could not fit on target device (e.g., SN5 requires 8196 BRAMs while target device has only 1470).

We also compared our results to work by Chen et al. [7] which designs an energy efficient bionic sort on the same target device. Their designs uses 19927 LUTs and 2 BRAMS for sorting 1024 elements, and it uses 36656 LUTs and 88 BRAMs for sorting 16384 elements. The LUTs and BRAMs are calculated using the utilization percentage from [7].

Table 6: Streaming insertion sort generated in this paper (Resolve) vs. Interleaved linear insertion sorter (ILS) [21].

	64	128	256
ILS Throughput (MSPS) [21]	4.6	2.33	1.16
Resolve Throughput (MSPS)	5.3	2.54	1.29
Ratio	**1.13X**	**1.08X**	**1.1X**
ILS Slices [21]	1113	2227	4445
Resolve Slices	792	1569	3080
Ratio	**0.7X**	**0.7X**	**0.69X**

Table 6 presents the throughput and utilization results of interleaved linear insertion sorter (ILS) and our streaming insertion sort for different sizes (64, 128, 256). We calculated the slices of ILS by using slices per node × number of elements (size). The slices per node for $w = 1$ is obtained from [21]. The throughput is the number of MSPS for a given size (64, 128, 256). Our insertion sorter has average 1.1X better throughput while using 0.6X fewer slices.

Arcas-Abella et al. [2] develop a spatial insertion sort and bitonic sort using Bluespec, LegUp, Chisel, and Verilog. Table 7 shows comparison of our spatial insertion / bitonic sort designs to implementations of this work. We achieve higher throughput and use less area. Our bitonic sort achieves the same throughput with comparable area results.

Koch et al. [14] use partial reconfiguration to sort large arrays. They achieve a sorting throughput of 667 MB/s to 2 GB/s. We can improve our throughput by increasing the frequency (our HLS cores run at 125 MHz) and using additional RIFFA channels. Our system consumes more BRAMs because they implement a FIFO-based merge sort using a

Table 5: Comparison to Spiral [23]. II is the number of clock cycles to produce one sorted array.

	64			1024			16384		
	FF/LUT	BRAM	II	FF/LUT	BRAM	II	FF/LUT	BRAM	II
Spiral SN1	5866 / 1775	10	64	34191 / 28759	162	1024	-	-	-
Spiral SN2_I	2209 / 880	5	397	4053 / 2002	45	10261	6790/2547	964	229405
Spiral SN2_S	5912 / 1803	10	64	16165 / 5991	125	1024	62875 /2744884	1395 / 16384	16384
Spiral SN5	9386 / 3023	18	64	27130 / 11104	225	1024	-	-	-
Resolve	1560 / 1401	2	68	3486 / 2848	7	1028	6515 / 4901	70	16388

*Table 7: Comparison of our work to [2]. * calculated with II=1*

	Spatial Insertion		Bitonic	
	FF/ LUT	MB/s	LUT/FF	MB/s
Verilog	2081/ 641	1301	10250/ 2640	38016
BSV	2012/ 1701	1310	10250/ 2640	38326
Chisel	2012/ 720	1317	10272/ 2649	38447
LegUp	1115/ 823	3.13	4210/ 5180	1034
Resolve	605/ 661	1415	6404/ 9827	38016*

shared memory blocks for both input streams. Writing to a FIFO using two different processes during functional pipelining is not supported by HLS tools that we used.

7. CONCLUSION

The Resolve framework generates optimized sorting architectures from pre-optimized HLS blocks. Resolve comes with a number of highly optimized sorting primitives and sorting architectures. Both the primitives and basic sorting algorithms can be combined in countless manners using our domain specific language, which allows for efficient design space exploration to enable a user to meet all of the necessary system design constraints. The user can customize these hardware implementations in terms of sorting element size and data type, throughput, and FPGA device utilization constraints. Resolve integrates these sorting architectures with RIFFA, which enables designers to call these hardware accelerated sorting functions directly from a CPU with a PCIe enabled FPGA card.

References

[1] S. G. Akl. *Parallel sorting algorithms*. AP, Inc, 1985.

[2] O. Arcas-Abella et al. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *International Conference on Field Programmable Logic and Applications*. IEEE, 2014.

[3] M. Bednara et al. Tradeoff analysis and architecture design of a hybrid hardware/software sorter. In *International Conference on Application-Specific Systems, Architectures, and Processors*, pages 299–308. IEEE, 2000.

[4] V. Brajovic et al. A sorting image sensor: An example of massively parallel intensity-to-time processing for low-latency computational sensors. In *International Conference on Robotics and Automation*, volume 2, pages 1638–1643. IEEE, 1996.

[5] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[6] J. Casper et al. Hardware acceleration of database operations. In *International symposium on Field-programmable gate arrays*, pages 151–160. ACM, 2014.

[7] R. Chen et al. Energy and memory efficient mapping of bitonic sorting on fpga. In *International Symposium on Field-Programmable Gate Arrays*, pages 240–249. ACM, 2015.

[8] J. Chhugani et al. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[9] J. Dean et al. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] N. George et al. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications*, pages 1–8. IEEE, 2014.

[11] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3):10, 2006.

[12] M. Jacobsen et al. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2015.

[13] D. E. Knuth. *The art of computer programming, volume 3: sorting and searching*. Addison-Wesley Professional, 1998.

[14] D. Koch et al. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *International symposium on Field programmable gate arrays*, pages 45–54. ACM, 2011.

[15] C. Lauterbach et al. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[16] R. Marcelino et al. Sorting units for fpga-based embedded systems. In *Distributed Embedded Systems: Design, Middleware and Resources*, pages 11–22. Springer, 2008.

[17] J. Matai et al. Enabling fpgas for the masses. In *First International Workshop on FPGAs for Software Programmers*, 2014.

[18] R. Mueller et al. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[19] R. Mueller et al. Sorting networks on fpgas. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 21(1):1–23, 2012.

[20] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[21] J. Ortiz et al. A streaming high-throughput linear sorter system with contention buffering. *International Journal of Reconfigurable Computing*, 2011.

[22] N. Satish et al. Designing efficient sorting algorithms for manycore gpus. In *IPDPS*, pages 1–10. IEEE, 2009.

[23] M. Zuluaga et al. Computer generation of streaming sorting networks. In *Design Automation Conference*, pages 1245–1253. ACM, 2012.