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

Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, Qiongzhi Wu, Amin Abazari, Ryan Kastner

Department of Computer Science and Engineering
University of California, San Diego
02/23/2016
Motivation: Sorting is everywhere

- Database \[^1\]  
  ~31,500 records (UCSD)

- Compression/ image processing \[^2, 3\]  
  <1000 records

- Map Reduce and Big Data \[^4, 5\]  
  > 1 Billion records (Facebook)

---

[4] Dean et al. "MapReduce: simplified data processing on large clusters", Communications of ACM’08
Motivation: Sorting is everywhere

- Database \[1\]
  \~31,500 records (UCSD)

- Compression and image processing \[2, 3\]
  \<1000 records

- Map Reduce and Big Data\[4, 5\]
  \> 1 Billion records (Facebook)

- Resolve provides **FPGA version of std::sort**
  - `std::sort` uses insertion sort for \<16
  - `std::sort` uses quicksort + (heapsort) for larger arrays

- Resolve aims to generate sorting architectures based on user constraints (e.g., size, frequency, area, ...)

Resolve built upon HLS

- Database[^1] ~31,500 records (UCSD)
- Compression and image processing[^2, 3] <1000 records
- Map Reduce and Big Data[^4, 5] > 1 Billion records (Facebook)

Resolve provides **FPGA version of std::sort**
- std::sort uses insertion sort for <16
- std::sort uses quicksort + (heapsort) for larger arrays

Resolve aims to generate sorting architectures based on user constraints (e.g., size, frequency, area,...)
Resolve: A Domain-Specific Framework

**Sorting Architecture Generator**

Constraints (e.g., size) → Sorting Architecture Generator → Custom Sorter
Resolve: A Domain-Specific Framework

Sorting Architecture Generator

- Insertion Sort
- Merge Sort
- Bitonic Sort
- ...

Based on function composition:

- $g(\cdot) \rightarrow \text{[sorted]}$
- $f(g(\cdot), g(\cdot)) \rightarrow \text{[sorted]}$
Resolve: A *Domain-Specific Framework*

**Sorting Architecture Generator**

Constraints (e.g., size) → Merge Unit → Merge Unit → Merge Unit → Custom Sorter
Resolve: A Domain-Specific Framework

**Sorting Architecture Generator**

<table>
<thead>
<tr>
<th>Sorting Algorithms</th>
<th>Sorting Primitive Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>Merge unit, compare-swap</td>
</tr>
</tbody>
</table>

Constraints (e.g., size) → Custom Sorter
Resolve: A Domain-Specific Framework

Sorting Architecture Generator

<table>
<thead>
<tr>
<th>Sorting Algorithms</th>
<th>Sorting Primitive Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>Merge unit, compare-swap</td>
</tr>
<tr>
<td>Radix sort</td>
<td>Prefix sum, Histogram</td>
</tr>
</tbody>
</table>
Resolve: A Domain-Specific Framework

Sorting Architecture Generator

<table>
<thead>
<tr>
<th>Sorting Algorithms</th>
<th>Sorting Primitive Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>Merge unit, compare-swap</td>
</tr>
<tr>
<td>Radix sort</td>
<td>Prefix sum, Histogram</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>Compare-swap, Smart Cell</td>
</tr>
</tbody>
</table>
Resolve: A *Domain-Specific Framework*

### Sorting Architecture Generator

<table>
<thead>
<tr>
<th>Sorting Algorithms</th>
<th>Sorting Primitive Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>Merge unit, compare-swap</td>
</tr>
<tr>
<td>Radix sort</td>
<td>Prefix sum, Histogram</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>Compare-swap, Smart Cell</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>Compare-swap,</td>
</tr>
<tr>
<td>Selection sort</td>
<td>Compare-swap</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Compare-swap, Prefix sum</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>Prefix sum, Histogram</td>
</tr>
<tr>
<td>Rank Sort</td>
<td>Histogram, compare-swap</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>Compare-swap</td>
</tr>
<tr>
<td>Odd-even transposition sort</td>
<td>Compare-swap (Merge)</td>
</tr>
</tbody>
</table>
Resolve: A Domain-Specific Framework

**Sorting Architecture Generator**

- Insertion Sort
- Merge Sort
- Bitonic Sort
- Merge Unit
- Smart Cell
- Prefix Sum

Optimized Sorting Algorithms (Templates)
Resolve: A Domain-Specific Framework

**Sorting Architecture Generator**

- Insertion Sort
- Merge sort
- Bitonic Sort
- Merge Unit
- Smart-Cell
- Prefix sum

Optimized Sorting Algorithms (Templates)

Constraints (e.g., size)

Custom Sorter
Optimized Sorting Algorithms

- Optimized Sorting Algorithms is a library of HLS sorting code

- Optimized Sorting Algorithms = Restructured Code + directives
Optimized Sorting Algorithms

Why we do this way?
Insertion Sort
Insertion Sort
Insertion Sort
Insertion Sort
Insertion Sort
Insertion Sort
Insertion Sort
Insertion Sort

\[ n = 32 \]

```c
void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```
void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
Insertion Sort

void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}

n = 32
Insertion Sort

\(n = 32\)

```c
void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            //pragma HLS PIPELINE II=1
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```

![Graph showing throughput vs. area]
Insertion Sort

\[ n = 32 \]

```c
void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            #pragma HLS PIPELINE II=1
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```
Insertion Sort

$n = 32$

```c
void InsertionSort(DTYPE numbers[n])
{
    //pragma HLS PIPELINE II=1
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```
void InsertionSort(DTYPE numbers[n])
{
    #pragma HLS PIPELINE II=1
    #pragma HLS PARTITION numbers
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
Insertion Sort

\[ n = 32 \]

```c
void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            #pragma UNROLL FACTOR= 4
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```

![Graph showing throughput vs. area for Insertion Sort](chart.png)
**Insertion Sort**

\( n = 32 \)

```c
void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            #pragma UNROLL FACTOR= 4
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```

We want to be here
Insertion Sort

\[ n = 32 \]

```c
void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index))
        {
            #pragma UNROLL FACTOR= 4
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```

Throughput (Million samples/s)
Insertion Sort: Hardware
Insertion Sort: Hardware
Insertion Sort: Hardware
Insertion Sort: Hardware
Insertion Sort: Hardware
Insertion Sort: Hardware
void cell(hls::stream<DTYPE> &IN, hls::stream<DTYPE> &OUT){
    static DTYPE CURR_REG=0;
    DTYPE IN_A=IN.read();
    if(IN_A>CURR_REG) {
        OUT.write(CURR_REG);
        CURR_REG = IN_A;
    } else {
        OUT.write(IN_A);
    }
}
void cell(hls::stream<DTYPE>& IN, hls::stream<DTYPE>& OUT){
    static DTYPE CURR_REG = 0;
    DTYPE IN_A = IN.read();
    if(IN_A > CURR_REG) {
        OUT.write(CURR_REG);
        CURR_REG = IN_A;
    } else {
        OUT.write(IN_A);
    }
}

void InsertionSort(hls::stream<DTYPE>& INPUT, hls::stream<DTYPE>& OUT){

    #pragma HLS DATAFLOW
    hls::stream<DTYPE> out0("out0_stream");
    hls::stream<DTYPE> out1("out1_stream");

    // Function calls;
    cell0(INPUT, out1);
    cell1(out1, out2);
    ...
    cell31(out30, OUT);
}
Insertion Sort: Software vs. Restructured

void InsertionSort(DTYPE numbers[n])
{
    int i, j, index;
    for (i = 1; i < n; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j - 1] > index))
        {
            #pragma HLS PIPELINE II=1
            numbers[j] = numbers[j - 1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}

void InsertionSort(hls::stream<DTYPE> &INPUT, hls::stream<DTYPE> &OUT)
{
    #pragma HLS DATAFLOW
    hls::stream<DTYPE> out0("out0_stream");
    hls::stream<DTYPE> out1("out1_stream");
    // Function calls;
    cell0(INPUT, out1);
    cell1(out1, out2);
    cell31(out30, OUT);
}
Insertion Sort: Software vs. Restructured

Resolve library = optimized HLS sorting routines

void InsertionSort(hls::stream<DTYPE>& INPUT, hls::stream<DTYPE>& OUT){
    #pragma HLS DATAFLOW
    hls::stream<DTYPE> out0("out0_stream");
    hls::stream<DTYPE> out1("out1_stream");

    // Function calls;
    cell0(INPUT, out1);
    cell1(out1, out2);
    cell31(out0, out2);
}
Resolve: A *Domain-Specific Framework*

**Sorting Architecture Generator**

- Insertion Sort
- Merge sort
- Bitonic Sort

- Merge Unit
- Smart Cell
- Prefix sum

*Optimized Sorting Algorithms (Templates)*

Custom Sorter

Constraints (e.g., size)
Resolve: A *Domain-Specific Framework*

Optimized Sorting Algorithms (Templates)

(sort, sort) → new sort
Resolve: A Domain-Specific Framework

**Primitive ::=**
- Selection
- Rank
- Insertion
- Bubble
- Merge
- Quick
- Radix
- Odd-even
- Bitonic

```
A[]
```

- Insertion
- Radix Sort
- **....**
- Bitonic sort
Resolve: A *Domain-Specific Framework*

**Primitive :: =**
- Selection
- Rank
- Insertion
- Bubble
- Merge
- Quick
- Radix
- Odd-even
- Bitonic

**Sort :: =**
- Primitive
- Merge Sort Sort
Resolve: A Domain-Specific Framework

Primitive :: =
  | Selection
  | Rank
  | Insertion
  | Bubble
  | Merge
  | Quick
  | Radix
  | Odd-even
  | Bitonic

Sort :: =
  | Primitive
  | Merge Sort Sort
Resolve: A Domain-Specific Framework

**Primitive :: =**

- Selection
- Rank
- Insertion
- Bubble
- Merge
- Quick
- Radix
- Odd-even
- Bitonic

**Sort :: =**

- Primitive
- Merge Sort Sort
from components import RadixSort
from components import InsertionSort
....
conf=Configuration('xc7vx1140', '10',...)

# Call
data = [2,3,1]
HW=sort_fpga(input_data=data);
....

Resolve: A Domain-Specific Framework

Vivado HLS Tools

[1,2,3]
Resolve: Utilization vs. Performance
Resolve: Utilization vs. Performance

Input size = 16384

Number of slices

Throughput (MB/s)

BRAMs >2000
BRAMs >1000
BRAMs >500
BRAMs ~250
Resolve: Utilization vs. Performance

- Parameters:
  - Size
  - Pipeline or unpipeline
  - Clock period
Resolve: Utilization vs. Performance

- **Parameters:**
  - Size
  - Pipeline or unpipeline
  - Clock period

![Graphs showing utilization vs. performance with parameters size, achieved clock period, and throughput.](image.png)
Prototype of Resolve

- Tools
  - Vivado Suite and Vivado HLS

- Hardware platform:
  - VC709
  - Xc7vx690tffg1761-2
  - RIFFA 2.2

- Performance
  - ~0.5 GB/s at 125 MHz

<table>
<thead>
<tr>
<th></th>
<th>Number of elements</th>
<th>FF / LUT</th>
<th>BRAM</th>
<th>IP Core II</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIFFA</td>
<td>N / A</td>
<td>19472 / 16395</td>
<td>71</td>
<td>N / A</td>
</tr>
<tr>
<td>RIFFA + Resolve</td>
<td>16384</td>
<td>25118 / 20368</td>
<td>141</td>
<td>18434</td>
</tr>
<tr>
<td>RIFFA + Resolve</td>
<td>65536</td>
<td>26353 / 21707</td>
<td>333</td>
<td>73730</td>
</tr>
</tbody>
</table>
Conclusion & Future Work

- **Resolve makes it easy to generate and use flexible (customized) sorting accelerators**

- **Sorting accelerator can be integrated with RIFFA framework**

  - Release of Resolve
Backup slides
Comparison to Previous Work

- SN1 and SN3 are high performance with larger area
- SN2 and SN4 balance area and performance
- SN5 is optimized for area.

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>1024</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FF/LUT</td>
<td>BRAM</td>
<td>II</td>
</tr>
<tr>
<td>Spiral SN1</td>
<td>5866 / 1775</td>
<td>10</td>
<td>64</td>
</tr>
<tr>
<td>Spiral SN2.I</td>
<td>2209 / 880</td>
<td>5</td>
<td>397</td>
</tr>
<tr>
<td>Spiral SN2.S</td>
<td>5912 / 1803</td>
<td>10</td>
<td>64</td>
</tr>
<tr>
<td>Spiral SN5</td>
<td>9386 / 3023</td>
<td>18</td>
<td>64</td>
</tr>
<tr>
<td>Resolve</td>
<td>1560 / 1401</td>
<td>2</td>
<td>68</td>
</tr>
</tbody>
</table>

- FF/LUT: Flip-Flop/Look-Up Table
- BRAM: Block RAM
- II: Implementation Index
Insertion Sort: Hardware