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

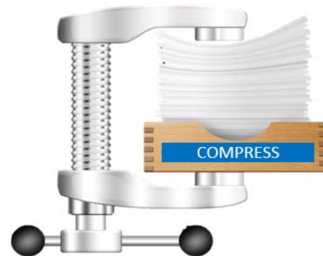
02/23/2016

UCSDCSE
Computer Science and Engineering

Motivation: Sorting is everywhere

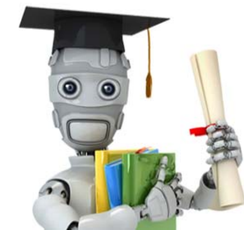
Database [1]

~31,500 records (UCSD)



Compression/ image processing [2, 3]

<1000 records



Map Reduce and Big Data[4,5]

> 1 Billion records (Facebook)

[1]Casper et al. "Hardware acceleration of database operations", FPGA''14

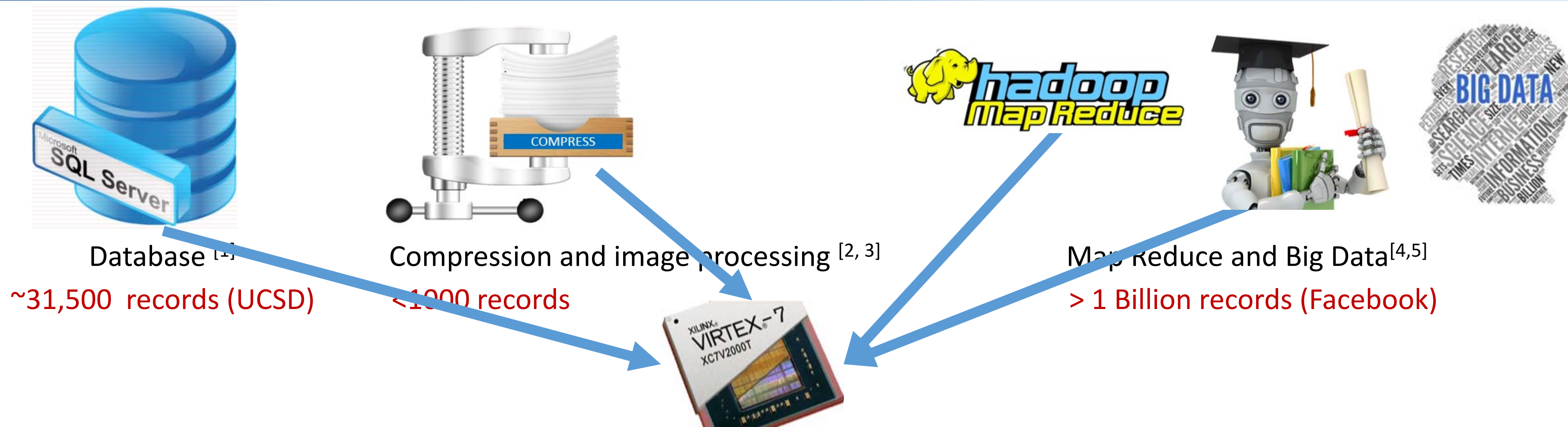
[2] Microsoft, "Xpress Compression Algorithm"

[3] Brajovic et al. "A Sorting Image Sensor", ICRA'96

[4] Dean et al. "MapReduce: simplified data processing on large clusters", Communications of ACM'08

[5] Herodotou et al. "Starfish: A Self-tuning System for Big Data Analytics", CIDR'11

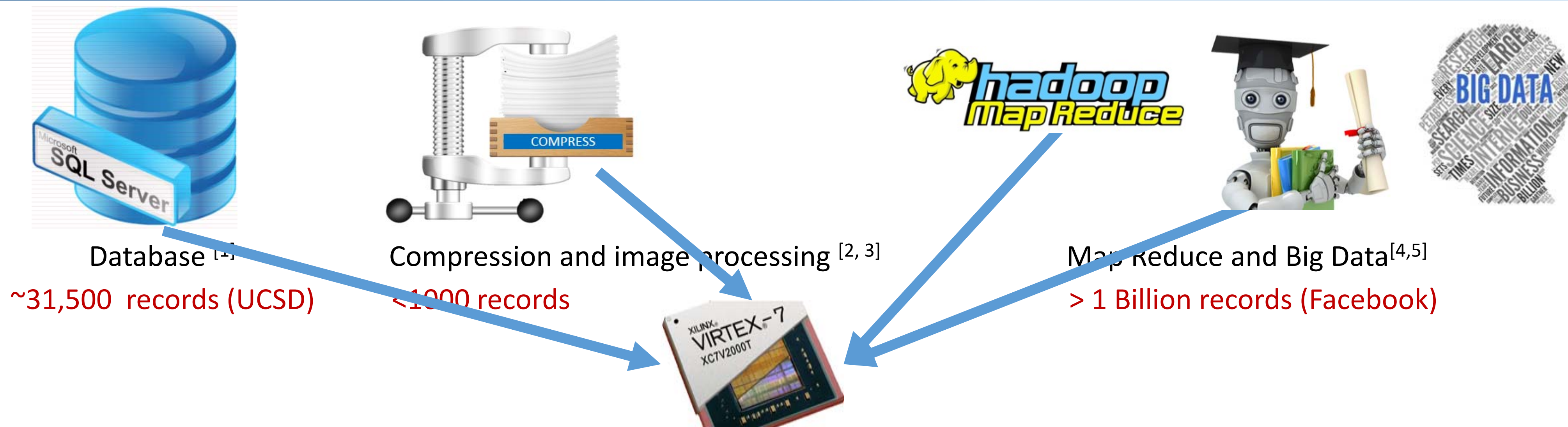
Motivation: Sorting is everywhere



- *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



- *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*

Constraints (e.g., size)

Sorting Architecture Generator

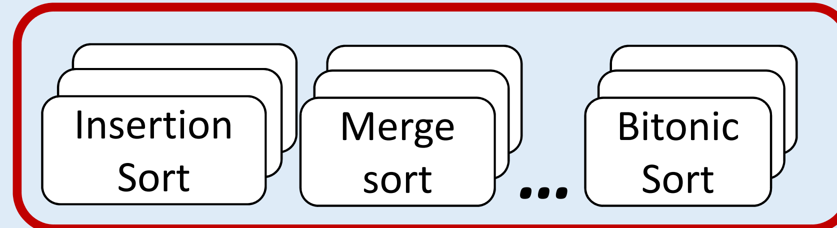
Custom Sorter

Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Custom Sorter

Sorting Architecture Generator



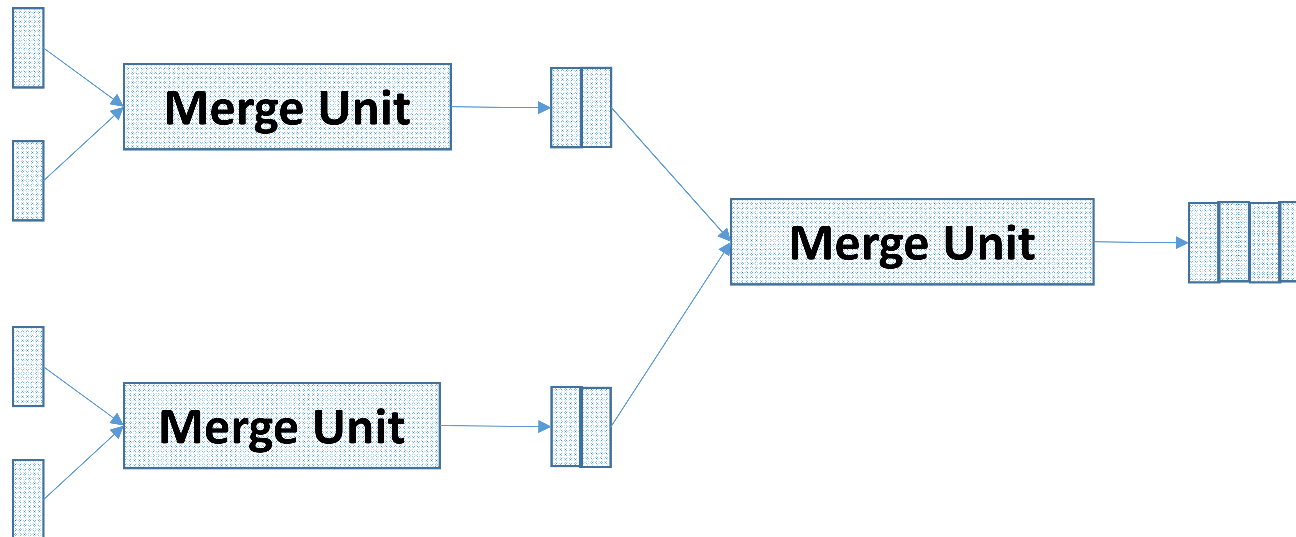
- *Based on function composition:*
  - ✓ *g([]) → [sorted]*
  - ✓ *f(g[],g[]) → [sorted]*

Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Sorting Architecture Generator

Custom Sorter



Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Sorting Architecture Generator

Sorting Algorithms	Sorting Primitive Kernels
Merge Sort	Merge unit, compare-swap

Custom Sorter

Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Sorting Architecture Generator

Sorting Algorithms	Sorting Primitive Kernels
Merge Sort	Merge unit, compare-swap
Radix sort	Prefix sum, Histogram

Custom Sorter

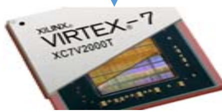
Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Sorting Architecture Generator

Sorting Algorithms	Sorting Primitive Kernels
Merge Sort	Merge unit, compare-swap
Radix sort	Prefix sum, Histogram
Insertion sort	Compare-swap, Smart Cell

Custom Sorter



Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Sorting Architecture Generator

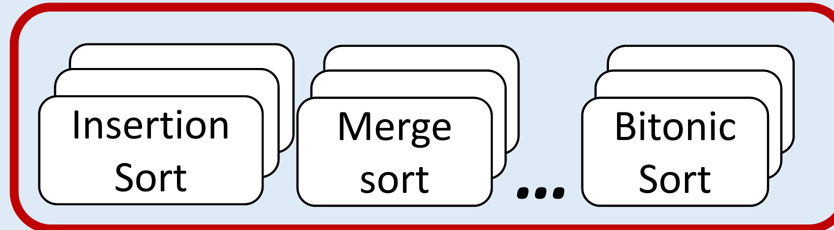
Sorting Algorithms	Sorting Primitive Kernels
Merge Sort	Merge unit, compare-swap
Radix sort	Prefix sum, Histogram
Insertion sort	Compare-swap, Smart Cell
Bubble sort	Compare-swap,
Selection sort	Compare-swap
Quick Sort	Compare-swap, Prefix sum
Counting Sort	Prefix sum, Histogram
Rank Sort	Histogram, compare-swap
Bitonic Sort	Compare-swap
Odd-even transposition sort	Compare-swap (Merge)

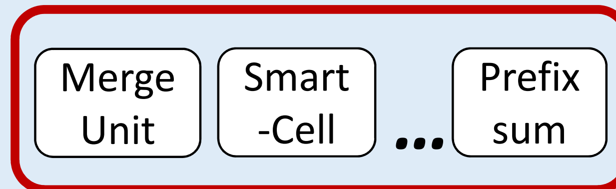
Custom Sorter

Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Sorting Architecture Generator





Optimized Sorting Algorithms (Templates)

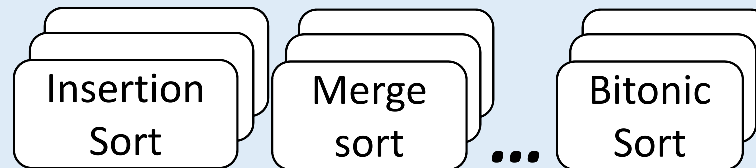
Custom Sorter

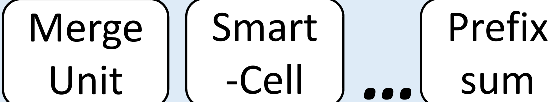
Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Custom Sorter

Sorting Architecture Generator





Optimized Sorting Algorithms (Templates)

Optimized Sorting Algorithms

- *Optimized Sorting Algorithms is a library of HLS sorting code*

- *Optimized Sorting Algorithms = Restructured Code + directives*

Optimized Sorting Algorithms

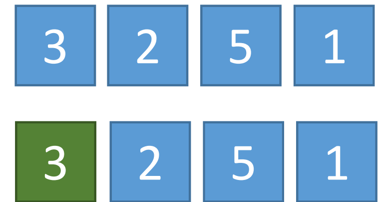
Why we do this way ?

Insertion Sort

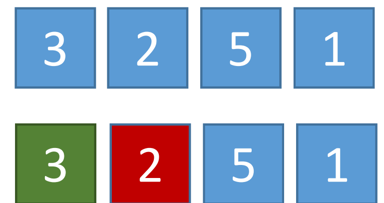
3	2	5	1

Insertion Sort

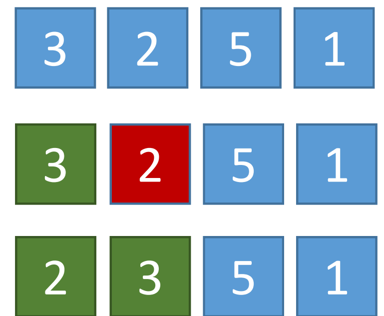
Insertion Sort



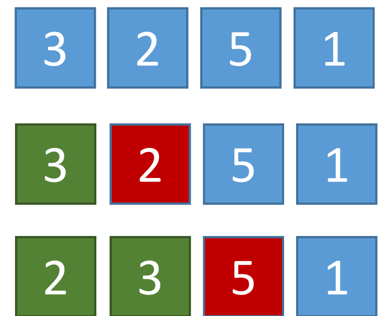
Insertion Sort



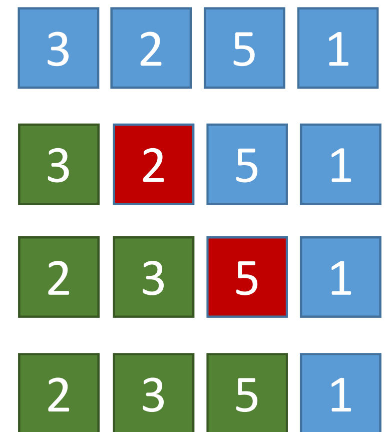
Insertion Sort



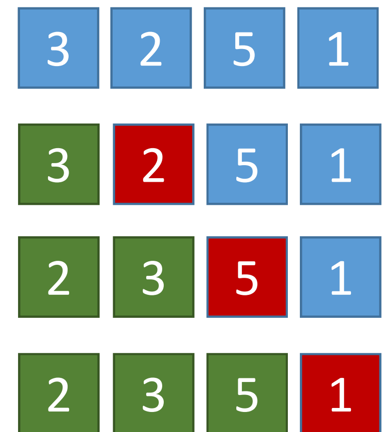
Insertion Sort



Insertion Sort



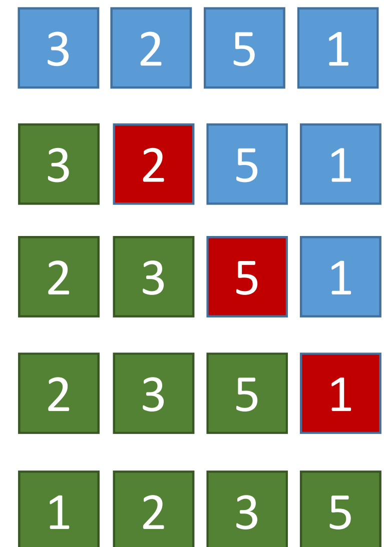
Insertion Sort



Insertion Sort

n = 32

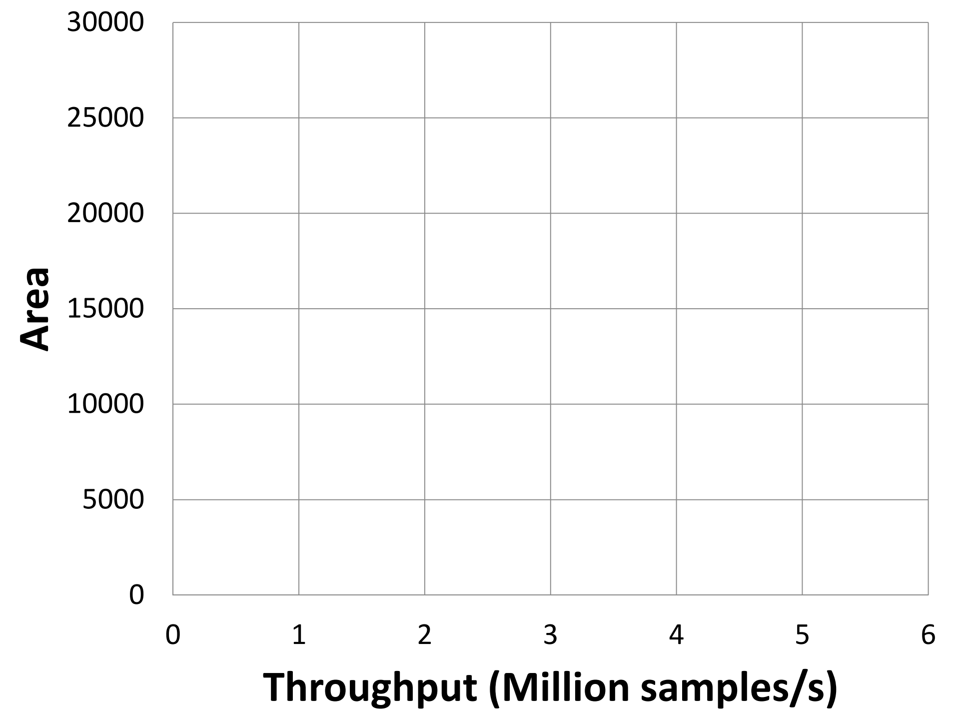
```
void InsertionSort(DTYPE numbers[n])
{

  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

n = 32

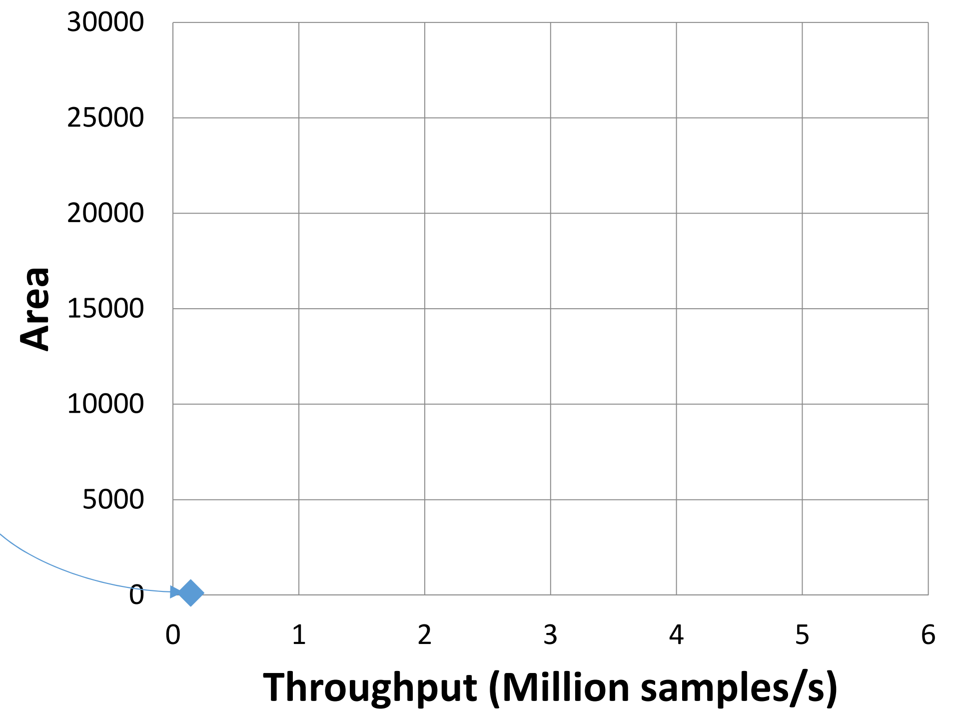
```
void InsertionSort(DTYPE numbers[n])
{

  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

n = 32

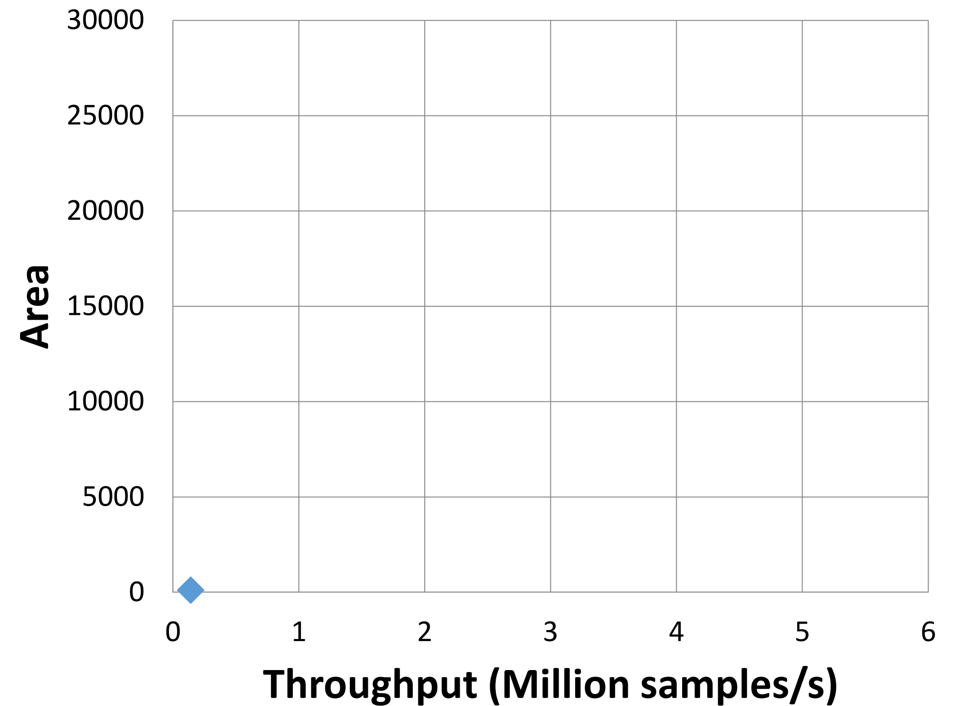
```
void InsertionSort(DTYPE numbers[n])
{

  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

n = 32

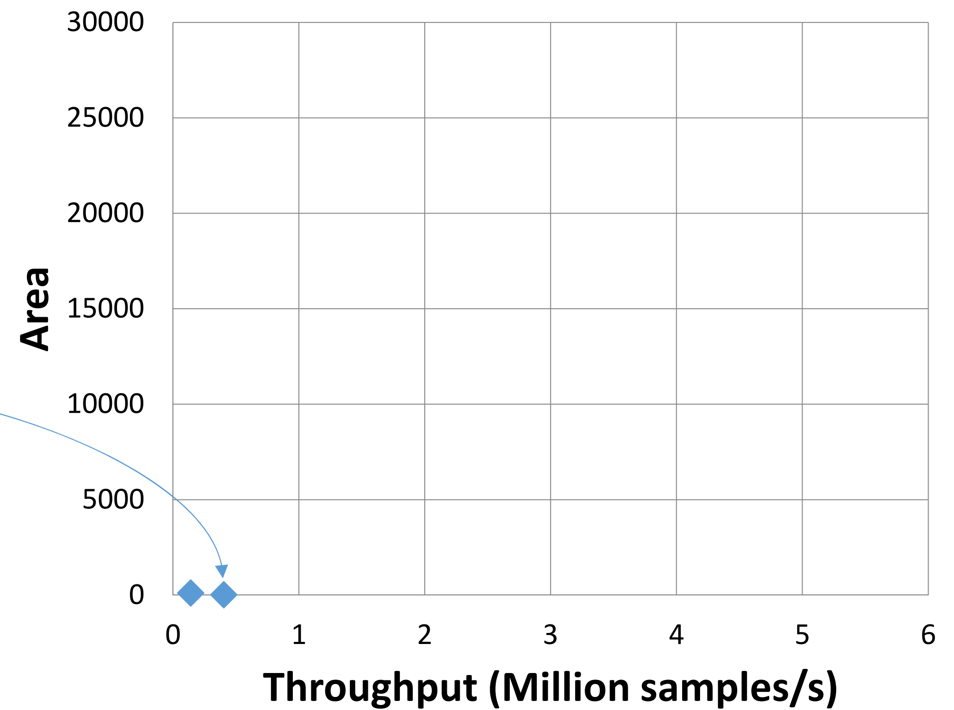
```
void InsertionSort(DTYPE numbers[n])
{

  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

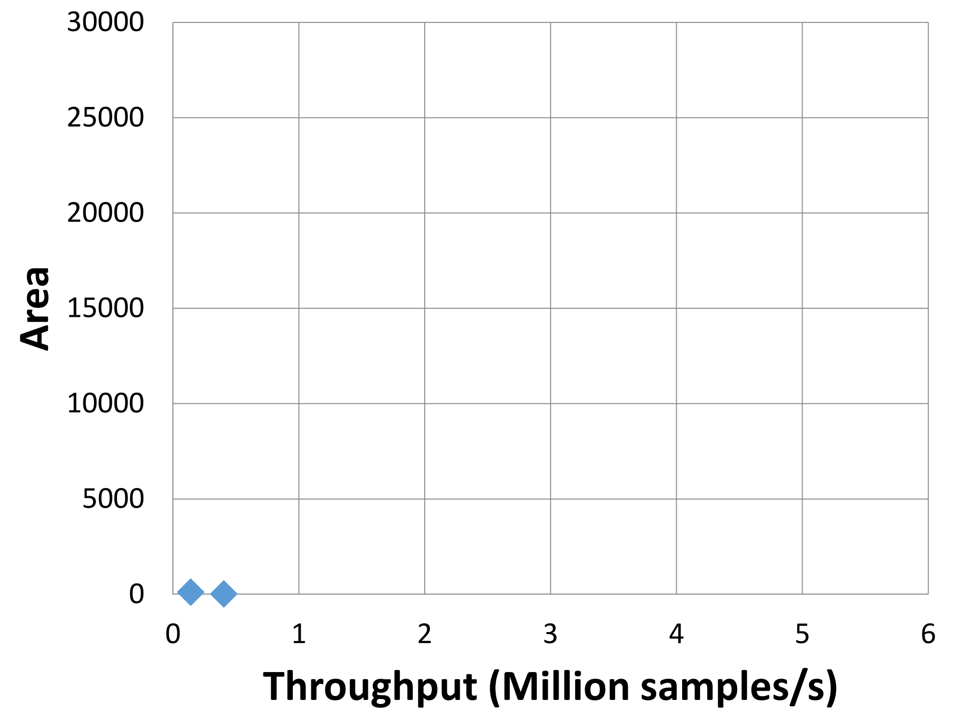
```
void InsertionSort(DTYPE numbers[n])
{

  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

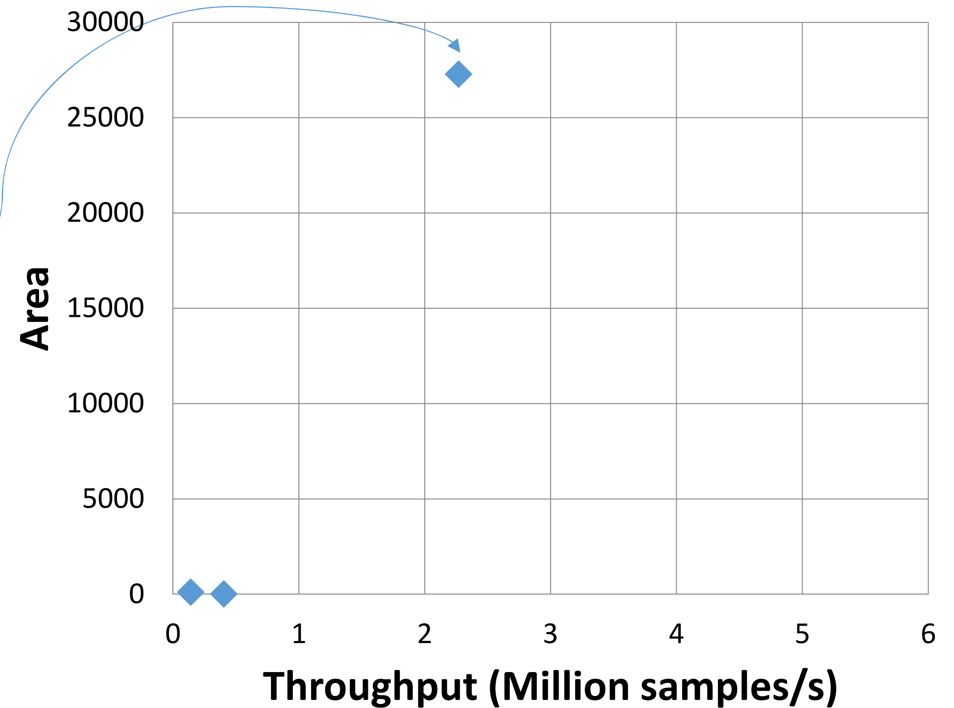
```
void InsertionSort(DTYPE numbers[n])
{
 #pragma HLS PIPELINE II=1
  int i, j, index;
  for (i = 1; i < n; i++)
  {
    index = numbers[i];
    j = i;
    while ((j > 0) && (numbers[j-1] > index))
    {


         numbers[j] = numbers[j-1];
         j = j - 1;
    }
    numbers[j] = index;
  }
}
```



Insertion Sort

n = 32

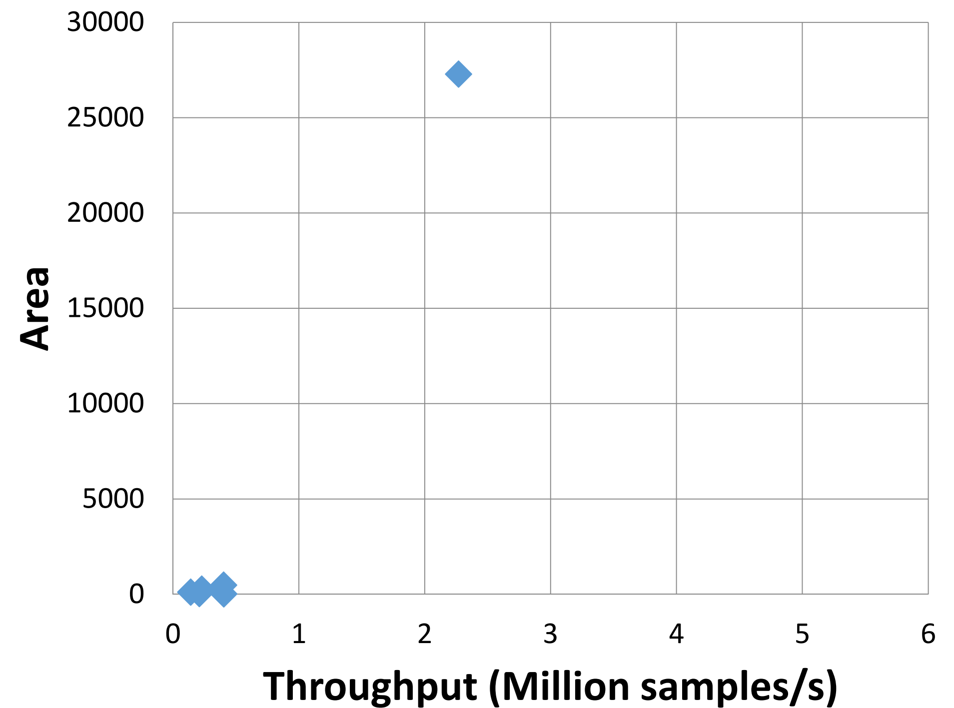
```
void InsertionSort(DTYPE numbers[n])
{ #pragma HLS PIPELINE II=1
  #pragma HLS PARTITION numbers
  for (i = 1; i < n; i++)
  {
    index = numbers[i];
    j = i;
    while ((j > 0) && (numbers[j-1] > index))
    {


        numbers[j] = numbers[j-1];
        j = j - 1;
    }
    numbers[j] = index;
  }
}
```



Insertion Sort

n = 32

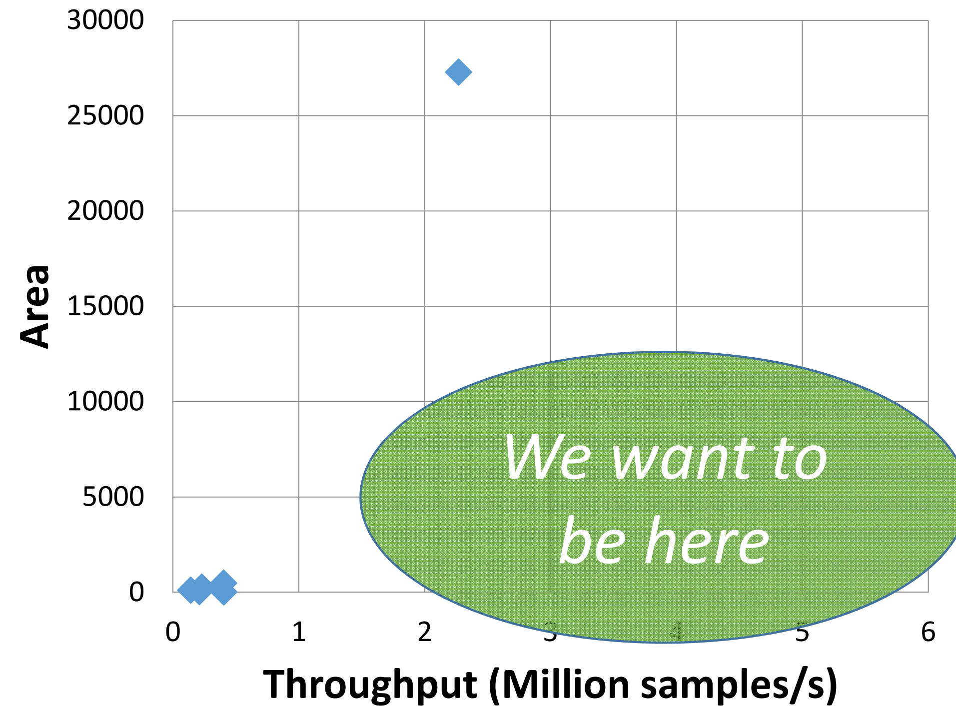
```
void InsertionSort(DTYPE numbers[n])
{

  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;
  }
}
```



Insertion Sort

n = 32

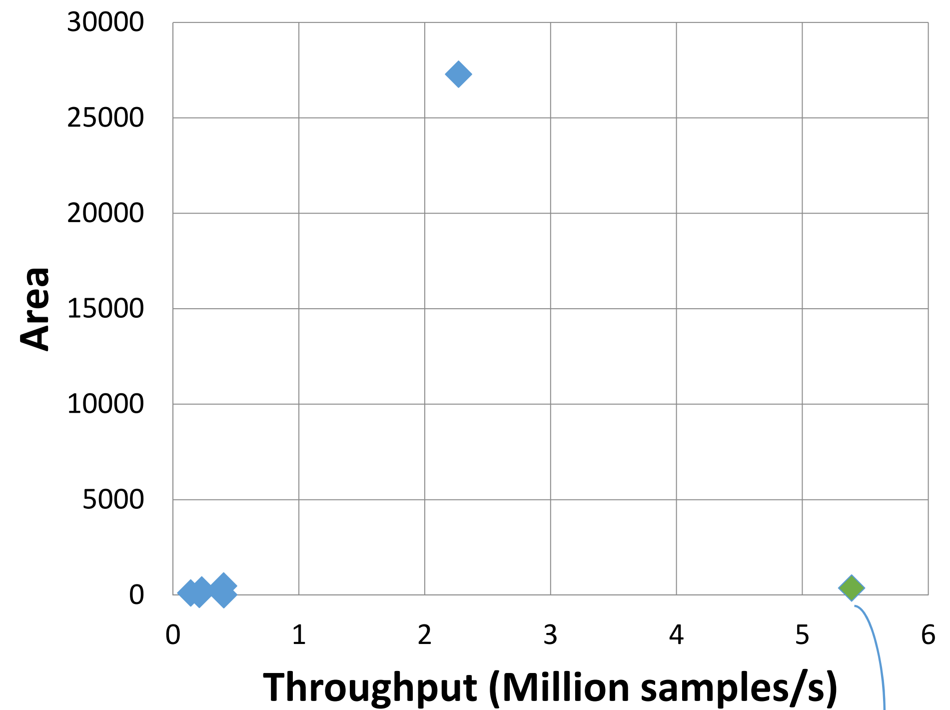
```
void InsertionSort(DTYPE numbers[n])
{

  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;
  }
}
```



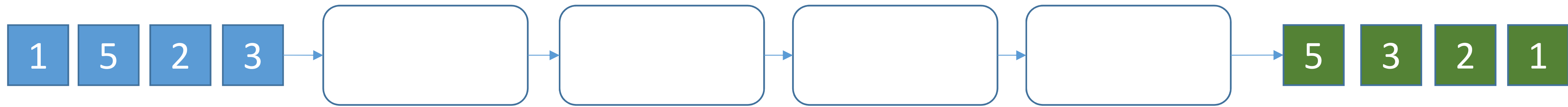
Insertion Sort

n = 32

```
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;
  }
}
```



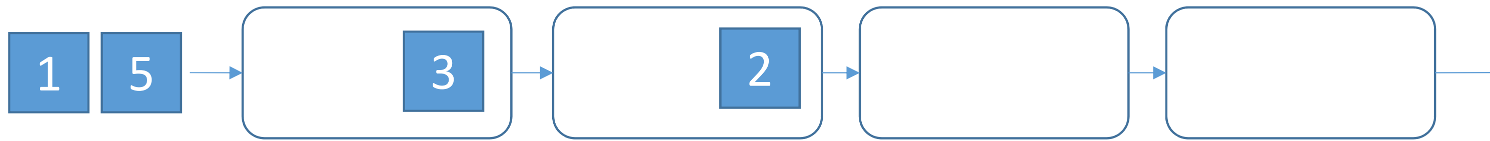
Insertion Sort: Hardware



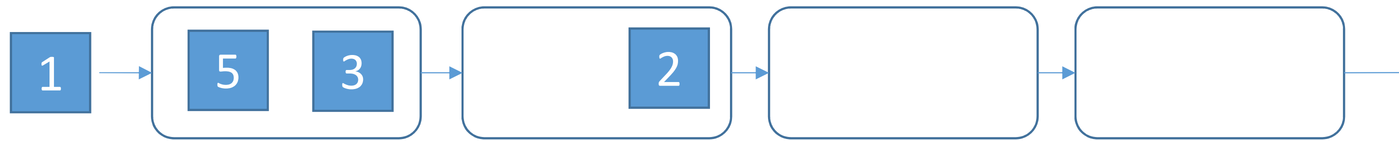
Insertion Sort: Hardware

Insertion Sort: Hardware

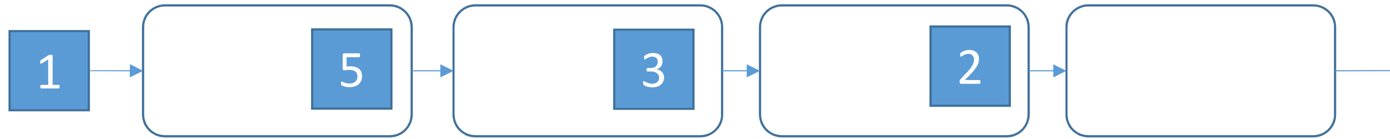
Insertion Sort: Hardware



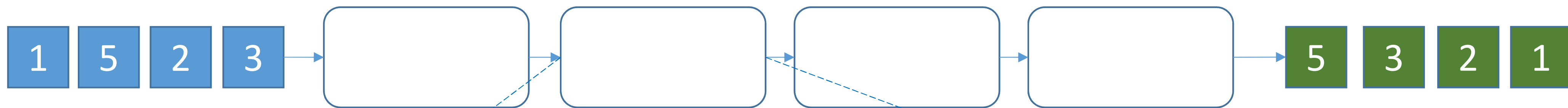
Insertion Sort: Hardware



Insertion Sort: Hardware

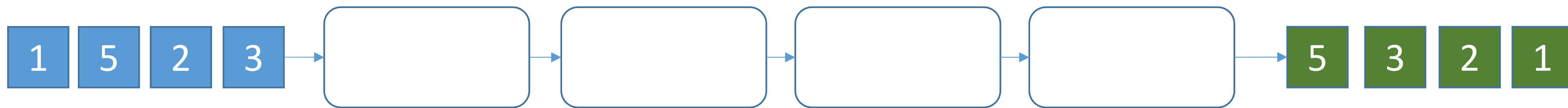


Insertion Sort: Hardware




```
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);
        }
}
```

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 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])
{
                                              )
    numbers[j] = index;
  }
}
```

$>>n^2$

```
void InsertionSort(hls::stream<DTYPE>
&I
                                                    ;
                                                    ;
ce
};
```

$2n$

Insertion Sort: Software vs. Restructured

❖ *Resolve library = optimized HLS sorting routines*

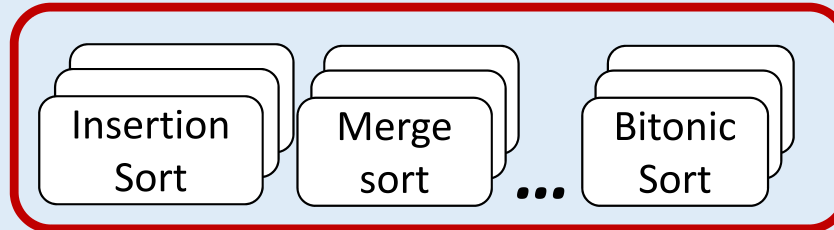
Sorting Algorithms
Merge Sort
Radix sort
Insertion sort
Bubble sort
Selection sort
Quick Sort
Counting Sort
Rank Sort
Bitonic Sort
Odd-even transposition sort

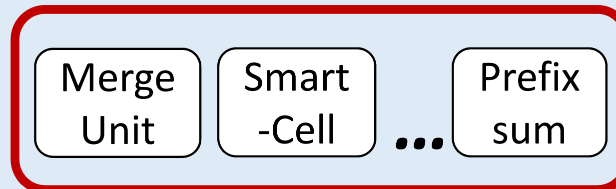
```
void InsertionSort(hls::stream<DTYPE>
&INPUT, hls::stream<DTYPE> &OUT){

#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, out2);
};
```

Resolve: *A Domain-Specific Framework*

Constraints (e.g., size)

Sorting Architecture Generator

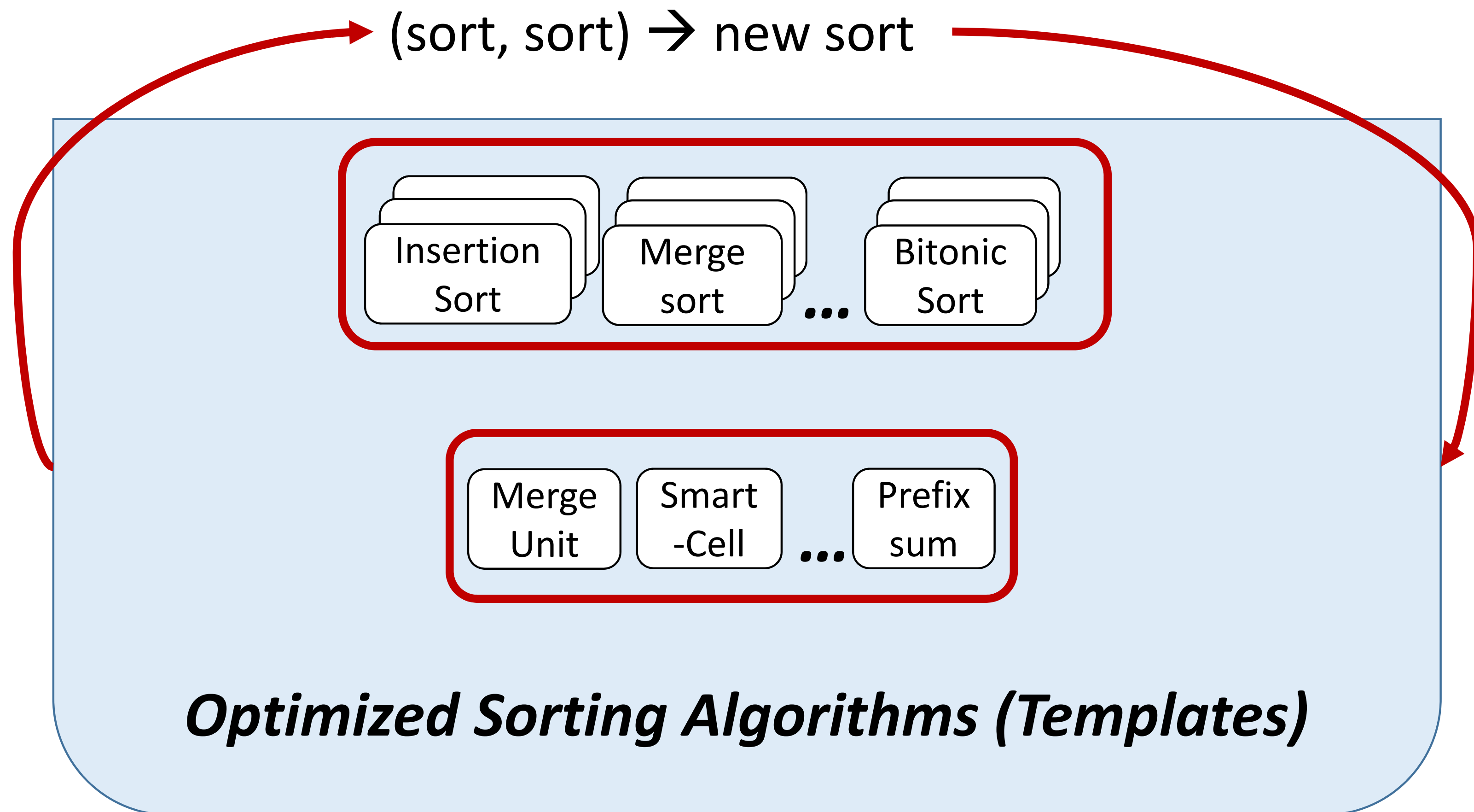




Optimized Sorting Algorithms (Templates)

Custom Sorter

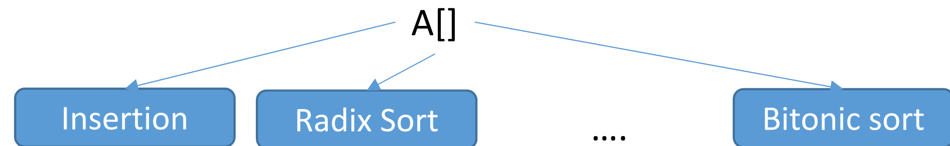
Resolve: *A Domain-Specific Framework*



Resolve: *A Domain-Specific Framework*

Primitive *:: =*

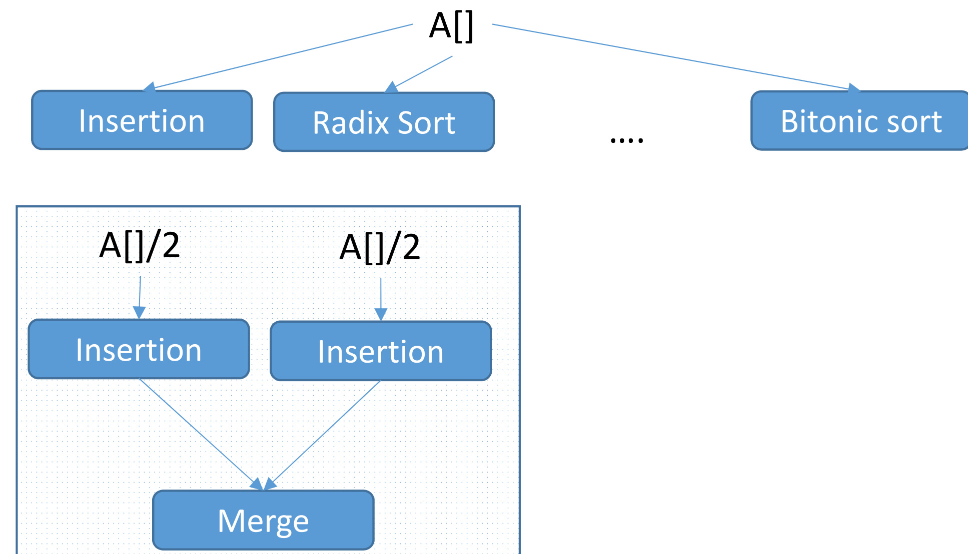
| Selection
| Rank
| Insertion
| Bubble
| Merge
| Quick
| Radix
| Odd-even
| Bitonic



Resolve: *A Domain-Specific Framework*

Primitive *:: =*

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



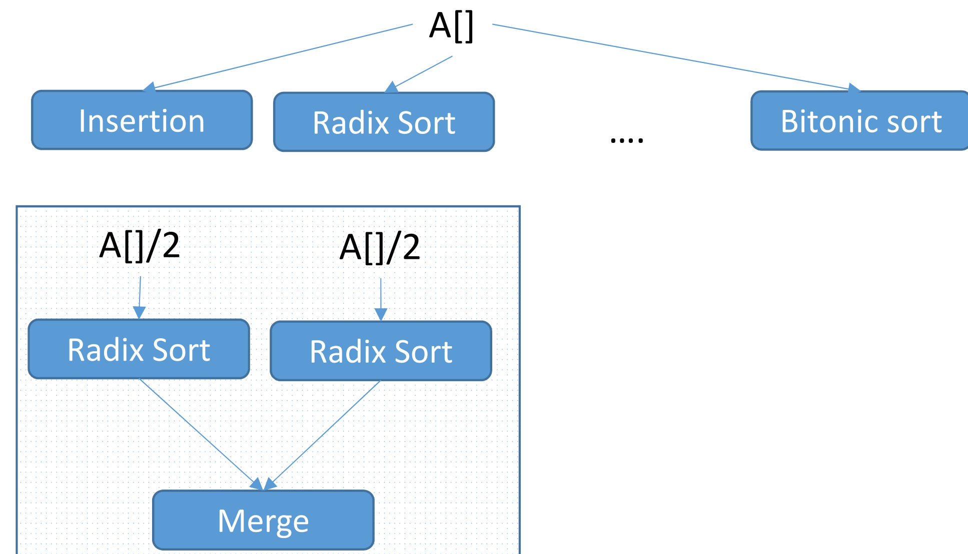
Sort *:: =*

| 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

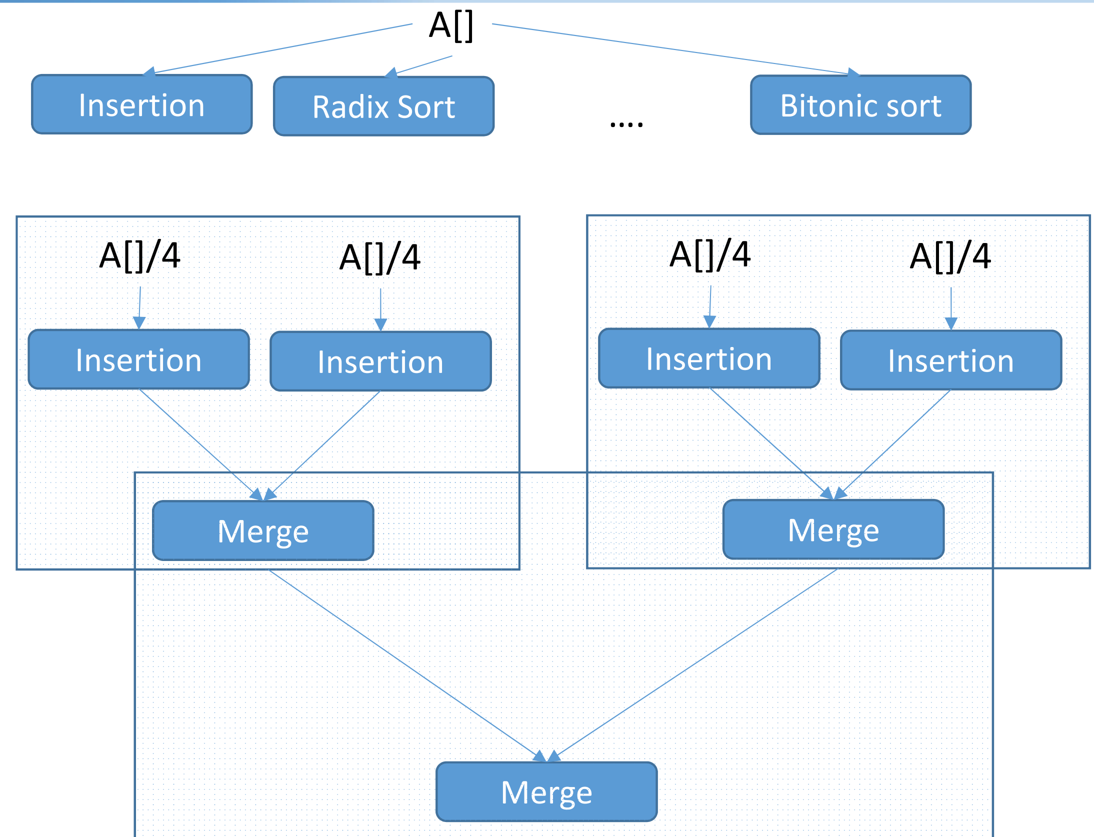
Resolve: *A Domain-Specific Framework*

Primitive *:: =*

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

Sort *:: =*

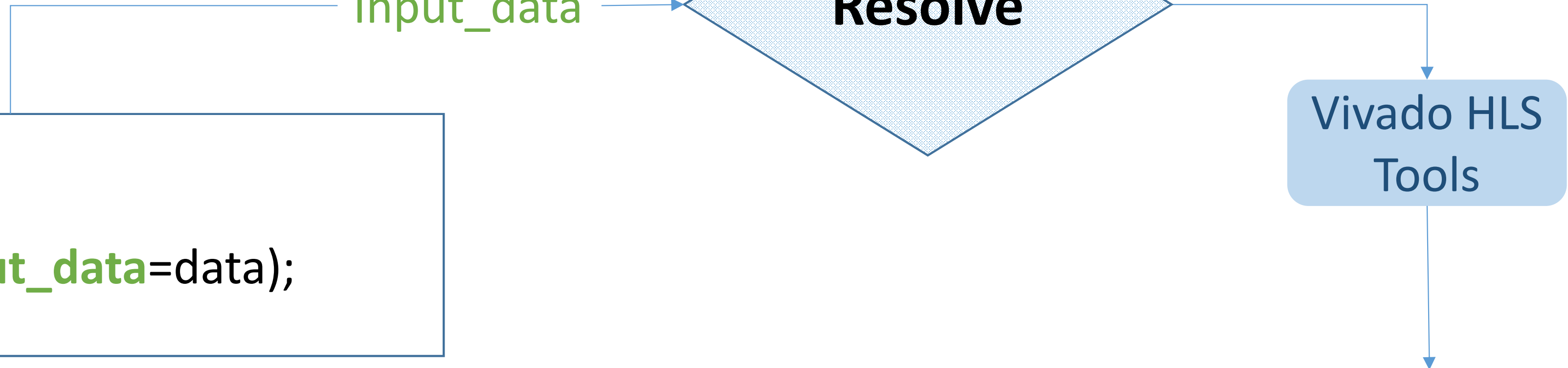
| Primitive
| Merge Sort Sort



Resolve: *A Domain-Specific Framework*

```
from components import RadixSort
from components import InsertionSort
....
```

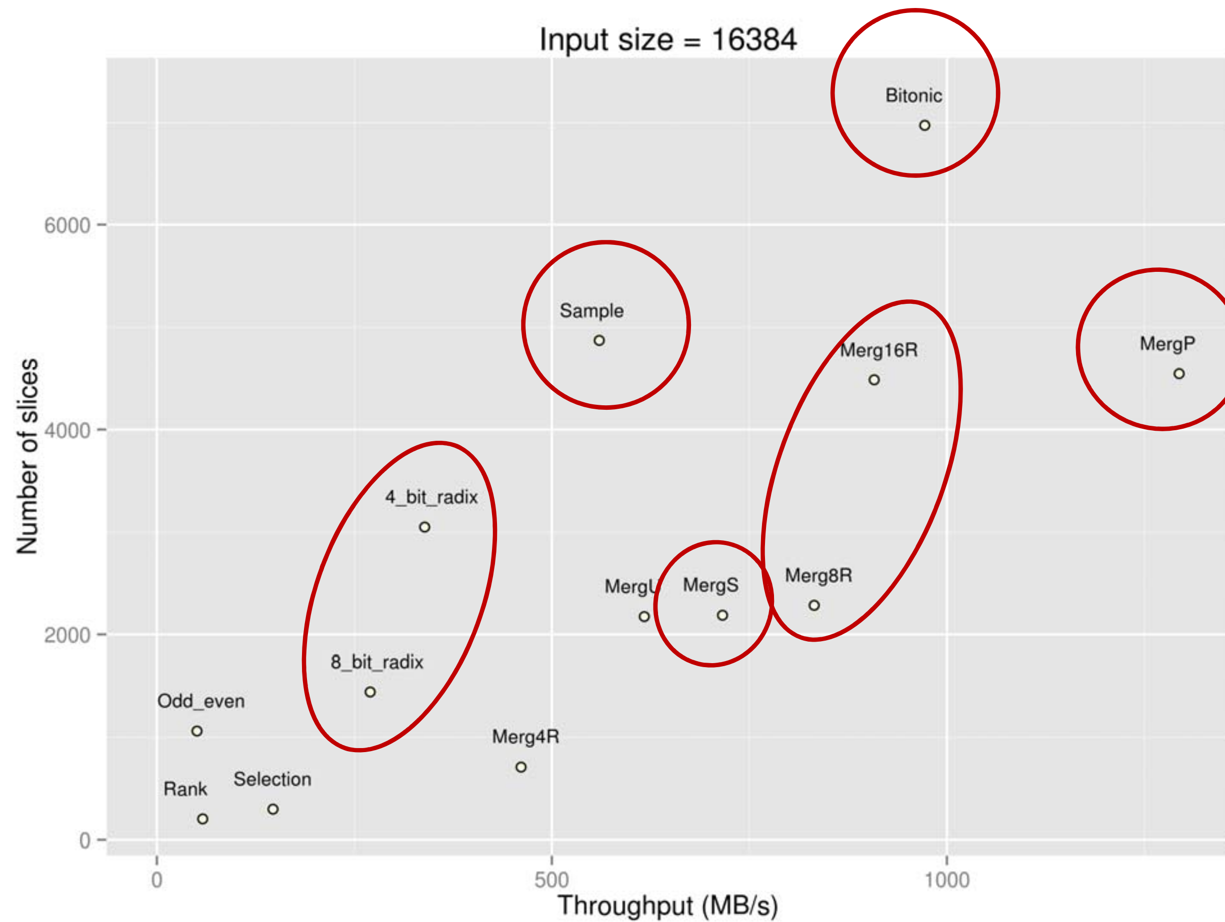
```
conf=Configuration('xc7vx1140', '10',...)
```



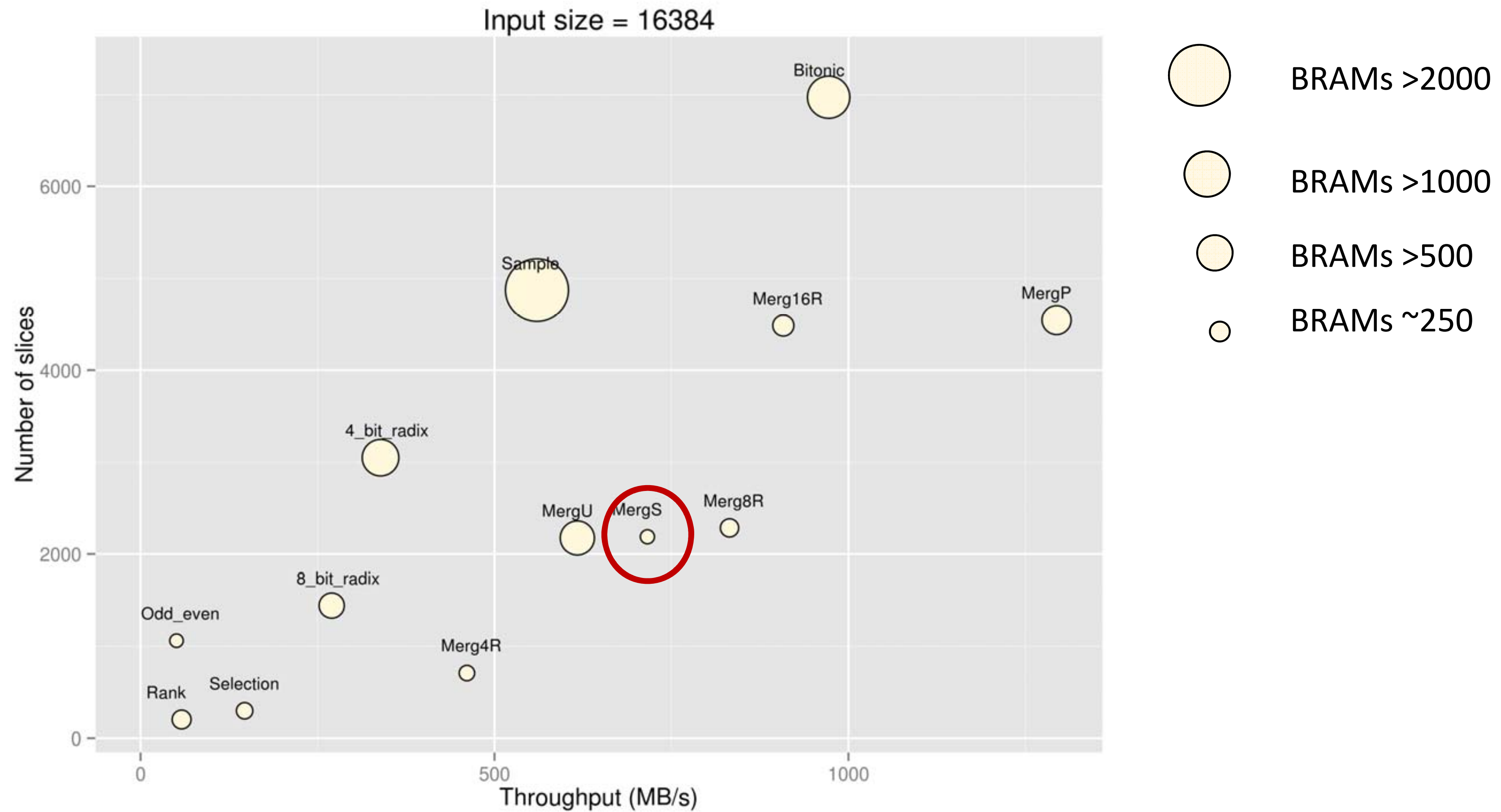

```
# Call
data =[2,3,1]
HW=sort_fpga(input_data=data);
....
```

[1,2,3]

Resolve: Utilization vs. Performance

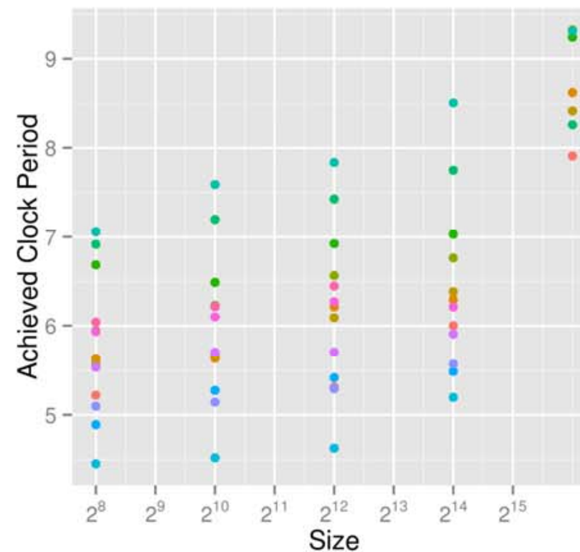


Resolve: Utilization vs. Performance



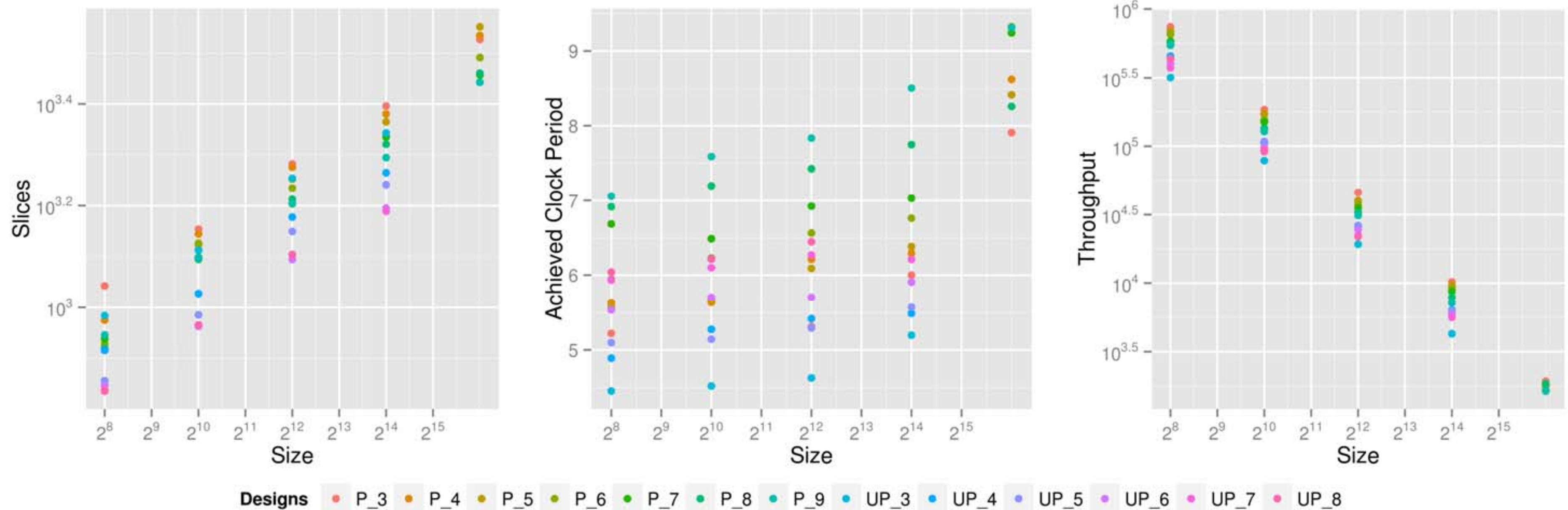
Resolve: Utilization vs. Performance

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



Resolve: Utilization vs. Performance

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



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

	Number of elements	FF / LUT	BRAM	IP Core II
RIFFA	N / A	19472 / 16395	71	N / A
RIFFA + Resolve	16384	25118 / 20368	141	18434
RIFFA + Resolve	65536	26353 / 21707	333	73730

Conclusion & Future Work

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

❖ *Sorting accelerator can be integrated with RIFFA framework*

❖ *Testing with http://sortbenchmark.org/*

❖ *Release of Resolve*

Backup slides

Comparison to Previous Work

	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

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

Insertion Sort: Hardware