CS4362 : Hardware Description Languages

Project Report

FPGA Based Insertion Sort Accelerator

Team Byte Hogs:

S. Kandeeban - 190296T

Nivinya Samarutilake - 190547P

Table of Contents

1. Introduction

Sorting algorithms are fundamental to the field of computer science and play a critical role in various applications such as data processing, information retrieval, and algorithmic efficiency. The essence of sorting lies in arranging data in a specific order, typically numerical or lexicographical, to facilitate easier access and processing. Classical sorting algorithms like Bubble Sort, Merge Sort, Quick Sort, and Heap Sort have been extensively studied and used in numerous applications. These algorithms exhibit a range of time complexities, from $O(n^2)$ in the case of simpler algorithms like Bubble Sort to O(nlogn) for more advanced algorithms like Merge Sort and Quicksort.

In recent years, the focus has shifted toward leveraging hardware acceleration to improve sorting performance, particularly for large datasets. Field-Programmable Gate Arrays (FPGAs) have emerged as a potent solution for this purpose, offering the flexibility of software with the performance benefits of hardware. FPGAs, with their parallel processing capabilities and configurable architecture, provide a unique opportunity to implement and accelerate sorting algorithms beyond the limitations of traditional, sequential CPU-based approaches.

This project proposes the development of a linear insertion sort accelerator using FPGA technology. The insertion sort algorithm, traditionally exhibiting a time complexity of $O(n^2)$, is a simple yet efficient algorithm for small datasets or nearly sorted arrays. However, its performance degrades with large datasets. By implementing a parallel version of the insertion sort algorithm on an FPGA, we aim to significantly enhance its performance.

The proposed parallel insertion sort algorithm operates by calculating a 'score' for each element in the dataset, based on comparisons with other elements. Unlike the traditional, sequential insertion sort, which iterates through the list, comparing and shifting elements one by one, the parallel approach allows simultaneous comparisons and calculations of scores for multiple elements. Once the scores are calculated, each element is placed directly into its corresponding score location in the output array, thus eliminating the need for sequential iteration and shifting. This approach leverages the inherent parallelism of FPGA architecture, resulting in a substantial reduction in sorting time, especially for large datasets.

By adopting this innovative approach, we envision not only enhancing the sorting efficiency but also demonstrating the practical utility of FPGAs in accelerating computational tasks traditionally handled by CPUs. This report will delve into the design, implementation, and performance evaluation of the linear insertion sort accelerator on FPGA, illustrating its advantages and potential applications in various data-intensive fields.

2. Traditional Method

This section presents a detailed examination of the naive approach to implementing the Insertion Sort algorithm in Chisel, a specialized language for constructing hardware in a Scala-based environment.

2.1. Naive Approach

The provided Chisel code defines a class InsertionSort, parameterized by width (bitwidth of the elements) and numElements (number of elements to sort). The core functionality of the Insertion Sort algorithm is encapsulated within this class, leveraging the powerful features of Chisel to model hardware behavior.

Key Components:

- I/O Definition : The module defines two primary I/O ports - an input vector in for the elements to be sorted, and an output vector out for the sorted elements.
- Internal State: Registers such as array, i, j, load, and sorted are used to manage the sorting process and control flow within the FPGA.
- Loading and Sorting Logic: The logic to load input data into the internal register array (array) and the core sorting mechanism are implemented using Chisel's 'when' construct.
- Output Assignment: The sorted data is assigned to the out port.

There are three processes defined in the implementation.

1. Loading the Input Data

```
// Load input into register array
when(load && !sorted) {
  array := io.in
  load := false.B
  i := 1.U
  j := 1.U
}
```

When load is true and sorted is false, the input data (io.in) is loaded into the array register. The flags and counters are then updated to begin the sorting process.

2. Insertion Sort Logic

```
// Insertion sort logic
when(!load) {
  when(i <= numElements.U) {
    when(array(j) < array(j-1.U) && j >= 1.U) {
      val temp = array(j-1.U)
      array(j-1.U) := array(j)
      array(j) := temp
      j := j - 1.U
    } .otherwise {
      i := i + 1.U
      j := i
    }
  } .otherwise {
    load := true.B
    sorted := true.B
  }
}
```

The insertion sort process iteratively selects an element and inserts it into its correct position in a sorted subsection of the array. This naive implementation follows the conventional approach, with comparisons and swaps occurring in a sequential manner, leading to the characteristic $O(N^2)$ time complexity for the worst and average Cases.

3. Output Assignment

```
// Assign sorted array to output
io.out := array
```

The sorted array is assigned to the out output port for external access.

While this naive approach effectively demonstrates the basic functionality of the Insertion Sort algorithm in a hardware description language, it also highlights the inherent limitations of traditional, sequential sorting methods, especially when scaling to large datasets. This limitation becomes a focal point for exploring more advanced techniques, such as parallel processing in FPGAs, to significantly enhance sorting efficiency. The subsequent section will explore how the parallel version of this algorithm, tailored for FPGA deployment, addresses these limitations by introducing a method of sorting through concurrent operations, thereby reducing the overall time complexity and increasing throughput.

3. Methodology

The aim of this project is to develop a hardware based solution to accelerate the sorting process. Our main focus is on the insertion sort algorithm, which produces a time complexity of $O(N^2)$ at the worst case scenario. The naive approach is quite simple to implement, however if the concept behind the insertion sort algorithm can be parallelized, it would drastically reduce the time complexity. Thus, we made an attempt to tweak the algorithm a little and enable parallelized operations. It was made possible by implementing a scoring system within the sorting algorithm. This section describes our implementation in detail - the parallel approach for sorting.

3.1. Modified Algorithm

In the traditional insertion sort method, the input array is rearranged into the sorted order by swapping the elements through comparison between pairs of values. In the parallel implementation, the initial input array is not rearranged and a separate array of memory is allocated to store the sorted values. In addition to the output array, another array is initialized to keep track of the 'scores' of each element.

There are 3 main stages in the accelerated sorting algorithm. First, the input data is read and stored in an array. The storing is done in a FIFO manner. The next stage is the comparison of each value with all the other values in the input array. The score for that particular element is updated alongside these comparisons. The initial score for all elements is 0. The scoring method is as follows:

Suppose the input array has 8 elements [*a, b, c, d, e, f, g, h*] and that the element being evaluated is *a.* The element *a* is compared with all the values in the array. When comparing with *b,*

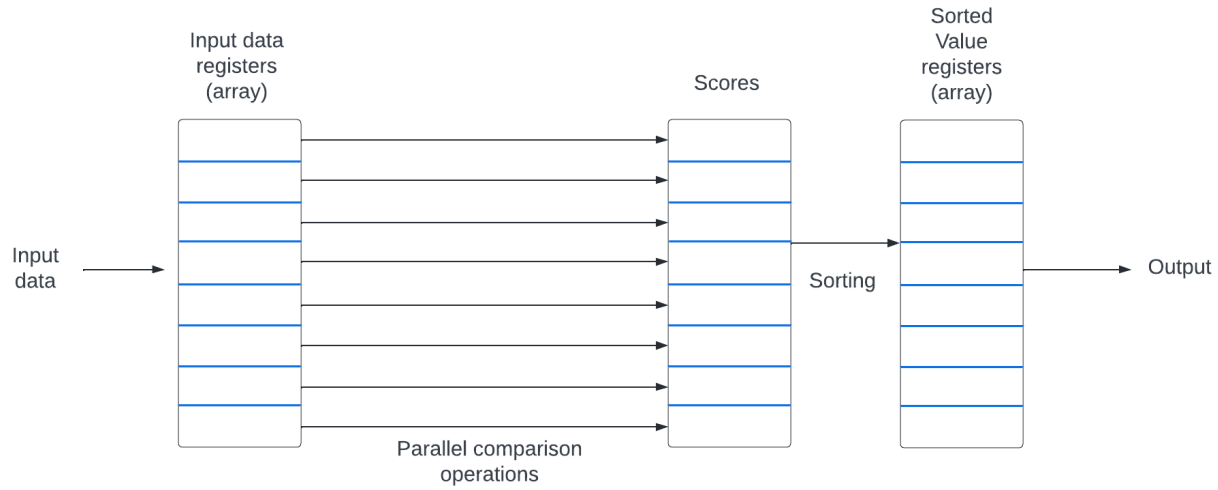
- If $b > a$, then the score is not updated
- If $b = a$, since the index of *a* is lower than *b*, the score is not updated. If the index of *a* was higher than *b*, then the score is increased by 1.
- If $b < a$, the score is incremented by 1

This scoring method is applied to all elements, and with how this scoring system has been constructed, there will be no elements with the same score. These scores are stored in a separate array.

The final stage of this approach is to sort the values into the output array. The final score for each element will be equivalent to the index of the position of that particular element at the sorted array (if the score is 3, the position of the sorted array is the third position). Therefore the values can be each placed at the sorted array without going through multiple swaps as in the traditional method.

3.2. Parallelism of the Modified Approach

The comparison stage for each element in the array can be executed in parallel as there are no dependencies between those operations. In the traditional approach, it is necessary to iterate through each element, one at a time, in order to do the swapping operations accurately and find the correct position in the final sorted array. This is a serial operation. In our method, there are no swappings happening in the comparison stage, thus the comparisons can be done in parallel (e.g: If there are 8 elements in the array, 8 threads can parallelly do the comparisons for each element). Thus, the modified approach has reduced time complexity theoretically to O(1). This is the idea behind the FPGA based acceleration for sorting algorithm.
Before the parallel operations, the input data are read in a sequential manner. After the comparisons, the writing of output data to the sorted array can also be parallelized. In our implementation, this part has not been parallelized.



3.3. Implementation in VHDL

We implemented the above described method using VHDL. This section provides an explanation of the VHDL code.

(For the ease of implementation, the number of input data was hardcoded as 8. The actual deployment should allow a dynamic number of inputs.)

There are 4 processes defined to perform the following functions:

1. *EnterData* - reads the inputs sequentially and stores them in the allocated location

```
-- "EnterData" is the process to read and store input values into inpt_list
-- When a rising edge on the write_inpt signal is detected,
-- a value will be read from the input medium and the data_count will be incremented

EnterData: process(clk, reset)
begin
    if (reset = '1') then
        inpt_list <= (others=> (others=>'0'));
        data_count <= 0;
    elsif (clk'event and clk = '1') then
        if (write_inpt = '1') then
            inpt_list(data_count) <= value;
            data_count <= data_count + 1;
        else
            data_count <= data_count;
            inpt_list <= inpt_list;
        end if;
    end if;
end process;
```

The data_count integer is updated with each entry. It is especially important when the number of inputs is unknown.
To store the array of 8-bit data, a custom type was declared called data_list.

```
architecture Behavioral of Sorter is

type data_list is array (0 to N-1) of std_logic_vector (7 downto 0);
signal inpt_list: data_list;    -- the array of input data (FIFO)
signal inpt_list_cpy: data_list; -- a copy of inpt_list
signal data_count: natural;     -- counter to keep track of the number of input data
```

2. *CompareData* - compares the data in the input array and calculates the score for each element

The comparison score is calculated according to the sorting ‘mode’ specified by the mode signal. The mode refers to the sorting order required - ascending or descending. These modes are represented by 0 and 1 respectively.
The scores are stored at the data_list called ‘scores’.

```
-- The "CompareData" process is where the parallel processing of data happens
-- This is a process with 3 states.

-- state 1 : the score and other variables are reset to 0
-- state 2 : the data in inpt_list_cpy are compared in parallel
-- state 3 : waits for 5 clock cycles to let the next process finish

CompareData:process(clk, reset)

variable score:data_list;

begin
    if (reset= '1') then
        score := (others=>(others=> '0'));
        inpt_list_cpy <= (others=>(others=>'0'));
        scores_list <= (others=>(others=>'0'));
        write_inpt_flag1 <= '0';
        write_inpt_flag2 <= '0';
        fin_compare <= '0';
        stage1 <= "000";
        wait_count <= 0;
```

The comparison logic for ascending mode is given below.

```
  when ("001") =>
      L1: for i in 0 to 7 loop
          L2: for j in 0 to 7 loop
              next L1 when j=data_count;
              if (mode = '0') then
                  if (inpt_list_cpy(i) < inpt_list_cpy(j)) then
                      score(i) := score(i) + 0;
                  elsif (inpt_list_cpy(i) = inpt_list_cpy(j)) then
                      if (i < j) then
                          score(i) := score(i) + 0;
                      else
                          score(i) := score(i) + 1;
                      end if;
                  else
                      score(i) := score(i) + 1;
                  end if;
```

3. *SortData* - puts the data on the sorted array according to the given score

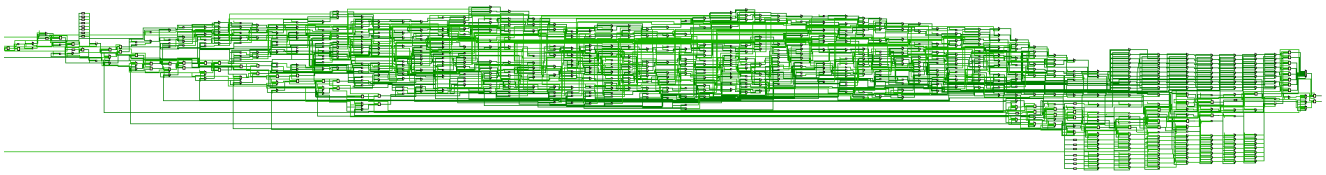
This process checks whether the sorting is completed at each clock cycle. There are flag signals defined in order to detect the end of the comparison process.
When the flags show that the comparison is completed, the sorting will take place by positioning the input values into the array_sorted list, according to the scores_list given by the previous process.

```
elsif (clk'event and clk = '1') then
    fin_comp_flag1 <= fin_compare;
    fin_comp_flag2 <= fin_comp_flag1;
    if (fin_comp_flag2 = '0' and fin_comp_flag1 = '1') then
        for k in 0 to 7 loop
            addr := conv_integer (scores_list(k));
            array_sorted(addr-1) <= inpt_list_cpy(k);
        end loop;
        fin_sort <= '1';
    else
        array_sorted <= array_sorted;
        fin_sort <= '0';
    end if;
```

4. *OutputData* - outputs the sorted data sequentially

This is the process to output the sorted array values, one by one.

The RTL design generated for the above implementation is given below.



3.4. Hardware Implementation

We were provided with a Digilent Basys 3 Artix-7 FPGA board for the hardware implementation of the project.

As the implementation requires serial data input to the board, we tried to use UART communication to input and output data to/from the board. A separate UART transmitter module was used for this purpose [1]. However, at the time of writing this report, there were some errors with this UART transmitter module. Thus, we were unable to confirm the functionality of the accelerator with board deployment.

The I/O port mapping is as follows.

I/O :

- V17-W13 → Input pins
- V2-R2 → Output pins

Communication :

- A18 → TX
- B18 → RX
- U18 → tx_enable
- W19 → rx_enable

- T18 → Reset
- W5 → Clk
- T17 → Mode

4. Testing

4.2. Behavioral Simulation

To test the functionality of the above design logic, a simple testbench was written and the behavioral simulation was run. It produced satisfactory results.
As the implementation was designed for a set of 8 values, the testbench included 8 values (<255). The values were sorted in both ascending and descending modes.

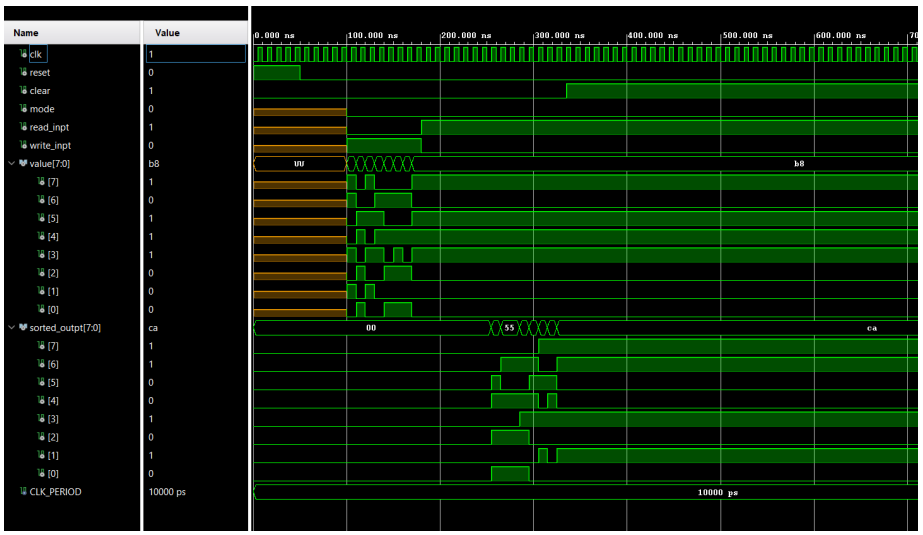
```
entity testbench is
end testbench;

architecture Behavioral of testbench is
    constant CLK_PERIOD : time := 10 ns;
    signal clk, reset, clear, mode, read_inpt, write_inpt : std_logic;
    signal value, sorted_outpt : std_logic_vector(7 downto 0);

begin
-- Instantiate the Sorter module
    UUT: entity work.Sorter
      generic map (N => 8)
      port map (
        clk => clk,
        reset => reset,
        clear => clear,
        read_inpt => read_inpt,
        write_inpt => write_inpt,
        value => value,
        sorted_outpt => sorted_outpt,
        mode => mode
      );

    -- Clock generation process
    clk_gen: process
    begin
      while now < 1000 ns loop
        clk <= '0';
        wait for CLK_PERIOD / 2;
        clk <= '1';
        wait for CLK_PERIOD / 2;
      end loop;
      wait;
    end process;
```

```
-- Stimulus process
stimulus: process
begin
  reset <= '1';
  wait for 5 * CLK_PERIOD;
  reset <= '0';
  wait for 5 * CLK_PERIOD;

  -- Test set
  write_inpt <= '1';
  read_inpt <= '0';
  mode <= '0';

  value <= "11001010";
  wait for CLK_PERIOD;
  report "Input given " & integer'image(conv_integer(value));

  value <= "00110101";
  wait for CLK_PERIOD;
  report "Input given " & integer'image(conv_integer(value));
```

Likewise, the values 202, 53, 170, 120, 85, 93, 85, 184 were input.

Results for ascending mode:

```
Note: Sorting will begin
Time: 195 ns  Iteration: 1  Process: /testbench/UUT/CompareData
Note: sorted 53
Time: 255 ns  Iteration: 1  Process: /testbench/UUT/OutputData
Note: sorted 85
Time: 265 ns  Iteration: 1  Process: /testbench/UUT/OutputData
Note: sorted 85
Time: 275 ns  Iteration: 1  Process: /testbench/UUT/OutputData
Note: sorted 93
Time: 285 ns  Iteration: 1  Process: /testbench/UUT/OutputData
Note: sorted 120
Time: 295 ns  Iteration: 1  Process: /testbench/UUT/OutputData
Note: sorted 170
Time: 305 ns  Iteration: 1  Process: /testbench/UUT/OutputData
Note: sorted 184
Time: 315 ns  Iteration: 1  Process: /testbench/UUT/OutputData
Note: sorted 202
Time: 325 ns  Iteration: 1  Process: /testbench/UUT/OutputData
```



The descending mode produced accurate results, similar to the above.

4.2. FPGA Testing

We decided to follow a comparison strategy to evaluate the efficiency of the accelerator FPGA.

1. 3 test cases are considered - arrays of 128, 256 and 1024 values
2. Each array is sorted with the traditional method on the Basys3 FPGA board and execution time will be noted in nanoseconds
3. Each array is sorted with the accelerator FPGA and execution times will be noted
4. Conclude the efficiency of the accelerator by comparing the execution times

Test case	Traditional Insertion Sort	Accelerator
128	N/A	N/A
256	N/A	N/A
1024	N/A	N/A

(The testing has not been performed yet)

To monitor the serial inputs/outputs, the TeraTerm tool can be used. It is an open-source, free, software implemented, terminal emulator (communications) program.

4.3. Pending Unresolved Issues

- The testing for accelerator FPGA has yet to be completed. There are issues with the communication protocol between the PC and the board. Although UART was selected as the communication method due to having serial inputs and outputs, it can be highly inefficient. However, it is the best option available as it does not require additional hardware.
- UART module has to be debugged as it gives erroneous inputs and outputs.

5. Further Improvements

In this implementation, the read/write operations of data were done sequentially. The accelerator would be much more efficient if these operations can be parallelized. This shortcoming should be addressed through any future improvements of this FPGA based accelerator.

If the accelerator can be paired up with a csv file reader module, the data can be input and read easily as a csv file. Then large numbers of data can be input and sorted efficiently with the composite accelerator. This is an improvement that can be considered in the future.

Other than Basys3 FPGA, the module should be tested on other FPGA boards as well. This will help to gain a better understanding of the effectiveness of the developed architecture. More comparison tests must be done in order to have an insight into the possibilities of further improvements.

6. Conclusion

This project was aimed to enhance the traditional Insertion Sort algorithm's efficiency through parallelization, particularly tailored for FPGA implementation. Theoretically, this approach promised a significant reduction in execution time, potentially lowering the complexity from $O(N^2)$ in the naive approach to an impressive O(N) in the parallel version. This can be further reduced with the parallelization of read/write operations, which are intrinsic to FPGA capabilities, allowing simultaneous processing of multiple data elements.

However, while the theoretical framework is promising, practical limitations inherent to FPGA architectures must be acknowledged. One such limitation is the dynamic data count constraint. Given the finite resources of an FPGA, it is impractical to sort an unlimited number of data points. This reality necessitates a careful consideration of the hardware's limitations when implementing the algorithm, ensuring that the sorting process remains efficient and feasible within the available resources.

Looking forward, exploring merge mechanisms [2] presents a potential pathway to handle larger batches of data. By dividing a large dataset into smaller subsets, sorting these individually in parallel, and subsequently merging them, the algorithm can be scaled to handle larger data sets more effectively. This approach aligns with the divide-and-conquer strategies commonly employed in efficient sorting algorithms like Merge Sort and can be adapted to leverage the parallel processing capabilities of FPGAs.

References

1. A. Sudbin, "UART Interface in VHDL for Basys3 Board,"*Hackster.io*. ,2020. [Online]. Available: https://www.hackster.io/alexey-sudbin/uart-interface-in-vhdl-for-basys3-board-eef170#schematics. [Accessed December 2023].

2. P. Papaphilippou, C. Brooks, and W. Luk, "An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics," in *Proc. of the 2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020.

3. Elektrotehnika, “caravel-linsorter” *GitHub*, 2020. Available: https://github.com/elektrotehnika/caravel_linsorter/tree/master [Accessed Dec. 27, 2023].

4. K. Manev and D. Koch, “Large Utility Sorting on FPGAs,” in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018.

GitHub Link for this project :

https://github.com/NivinyaSamarutilake/HDL-Linsorter.git