Accelerating Merge Sort on FPGAs: Design and Implementation Using HLS

Claudio Di Salvo

Politecnico Di Milano
claudio.disalvo@mail.polimi.it

June 16, 2024

Abstract

Merge Sort is a classic comparison-based sorting algorithm known for its efficiency and stability, is optimized and adapted for hardware acceleration. Utilizing the Xilinx PYNQ-Z2 development board, which integrates a Zynq-7000 SoC (System on Chip) combining ARM Cortex-A9 processors and FPGA fabric, the project aims to leverage the parallel processing capabilities of FPGAs to enhance the performance of the Merge Sort algorithm.

The implementation focuses on creating and comparing two modular and reusable IP core, designed using High-Level Synthesis (HLS) tools to facilitate efficient translation of the algorithm into hardware description.

The results showcase the potential of hardware-accelerated sorting algorithms in improving computational efficiency, making this IP core a valuable addition for applications requiring high-speed data processing. The project concludes with insights into potential enhancements and future work to further optimize the IP core for various practical applications.

1 Introduction

Field Programmable Gate Arrays (FPGAs) stand out as one of the most critical accelerators in modern computing due to their reconfigurability, which allows them to be tailored to the specific needs of a system. Compared to other accelerators, such as Graphic Processing Units (GPUs), FPGAs consume significantly less energy, making them an ideal choice for low-power applications. While Application Specific Integrated Circuits (ASICs) offer efficiency on par with FPGAs, they are notably more complex to design and prohibitively expensive to produce.

In this context, FPGAs occupy a unique position between GPUs and ASICs, enabling the acceleration of various algorithms at the hardware level. This acceleration is achieved through programming an FPGA using VHDL/Verilog or through High-Level Synthesis (HLS).

2 The Merge Sort

Merge Sort is a highly efficient sorting algorithm based on the divide-and-conquer paradigm. While the recursive approach is commonly discussed, the iterative approach to Merge Sort is equally effective and can be implemented in an FPGA scenario. First, the data structure is divided into subarrays of one element each. At this stage, each subarray is inherently considered sorted since a single element does not require further sorting. Next, each sorted element (subarray of one) is compared and merged with another to produce an ordered sequence of double the original length. This process creates larger sorted subarrays from smaller ones. Finally, steps one and two are repeated iteratively. The algorithm continues to merge the resulting ordered subarrays, doubling the length of the sorted sequences with each iteration. This process is repeated until the entire data structure is sorted into one final, ordered sequence.

The subsequent sections will present two different approaches to implementing the iterative Merge Sort algorithm.

3 Splitting Approach

This part, such as the comparation one, are the same for each of the two approach.

Algorithm 1 Snippet of SplinterCell

```
 1 for (int width = 1; width < dim; width *= 2){
 2     for (int i = 0; i < dim; i += 2 * width){
 3         int left_end = std::min(i + width, dim);
 4         int right_end = std::min(i + 2 * width, dim);
 5         for (int j = i; j < right\_end; j++){
 6             #pragma HLS PIPELINE II=1
 7             if (j < left\_end){
 8                 left_stream.write(buffer[j]);
 9             }else{
10                 right_stream.write(buffer[j]);
11             }
12         }
13         merge_sort_primitive(left_stream, right_stream,temp_stream);
14         for (int j = i; j < right\_end; j++){
15             #pragma HLS PIPELINE II=1
16             buffer[j] = temp_stream.read();
17         }
18     }
19 }
```

This algorithm sorts an array (`buffer`) of size `dim` through a series of merging stages, where each stage progressively merges larger blocks of the array. The overall process can be divided into several key steps: defining the merging window, splitting the array into streams, performing the merge operation, and writing back the sorted elements.

The outer loop, starting at line 1, controls the size of the blocks that are merged together in each stage of the merge sort. Initially, the block size (`width`) is set to 1, and it doubles with each iteration of the loop. This exponential growth ensures that the entire array is eventually sorted. For each iteration of the outer loop, the inner loop iterates through the array in chunks of size `2 * width`, effectively determining the boundaries of the left and right blocks to be merged.

The boundaries of the blocks are defined using the `std::min` function, which ensures that the indices do not exceed the size of the array (`dim`). Specifically, `left_end` marks the end of the left block, while `right_end` marks the end of the right block. These boundaries are essential for splitting the current chunk of the array into two streams: `left_stream` and `right_stream`.

The subsequent nested loop, annotated with `#pragma HLS PIPELINE II=1`, iterates over the current chunk and splits the elements into the two streams. Elements from the left block are written to `left_stream`, while elements from the right block are written to `right_stream`. The HLS PIPELINE directive ensures that this loop is pipelined with an initiation interval of one cycle, optimizing the hardware implementation for parallel processing.

Following the stream splitting, the `merge_sort_primitive` function is called to merge the two streams into a temporary stream (`temp_stream`). Finally, another loop, also pipelined with an initiation interval of one cycle, reads the merged elements from `temp_stream` and writes them back to the original array (`buffer`). After the exit of the outer loop the `buffer` will be wrote into the `output_stream` passed as a parameter.

4 Single Stream Approach

In this approach, the assumption is that the input stream originates from a single source.

By employing the splitting method, we achieve optimal utilization of the available resources.

Resource	Utilization	Available	Utilization %
LUT	14043	53200	26.40
LUTRAM	250	17400	1.44
FF	20202	106400	18.99
BRAM	2.5	140	1.79
BUFG	1	32	3.13

Table 1: Resource Utilization for single stream

The optimization strategies were applied to a specific hardware implementation, with performance metrics recorded over 10 executions. The results revealed a remarkable speedup on the order of 50, demonstrating the substantial impact of the applied optimizations.

SW Execution Time	HW Execution Time	Speedup (SW / HW)
0.100081	0.001850	54.094072
0.099518	0.001846	53.907659
0.098053	0.001767	55.493591
0.102514	0.001718	59.669026
0.099518	0.001762	56.490729
0.098169	0.001705	57.571449
0.102723	0.001680	61.148453
0.099497	0.001955	50.905221
0.097792	0.001708	57.262460
0.102454	0.001730	59.231289

Table 2: Execution Times and Speedup for single stream

5 Two Streams Approach

In this approach we are assuming that the input is coming from two different resource.

The approach under discussion is fundamentally similar to the single stream approach. However, a significant issue arises when the two streams, once fully sorted, are merged. This merging process can lead to a deadlock situation caused by the busy waiting of one of the two FIFO queues in the `merge_sort_primitive` function. To circumvent this problem, a workaround is required. The proposed solution involves creating a temporary stream that holds both sorted streams and then performing the sorting operation on this combined stream. This method effectively prevents the deadlock and ensures that the merging process operates smoothly.

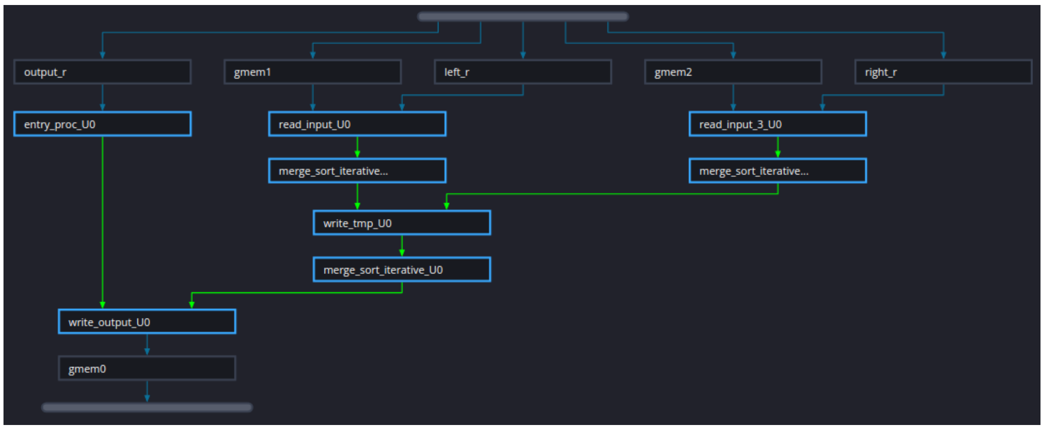

Figure 1: Dataflow-viewer of the two streams approach

However, this workaround, along with the substantial amount of additional computations required, significantly increases the resource utilization compared to the single stream approach. This higher resource demand is a critical consideration when evaluating the efficiency and practicality of this method.

Resource	Utilization	Available	Utilization %
LUT	38909	53200	73.14
LUTRAM	384	17400	2.21
FF	49974	106400	46.97
BRAM	6	140	4.29
BUFG	1	32	3.13

Table 3: Resource Utilization two streams approach

This approach involves a detailed refactoring of the original method, which resulted in an optimization for a single-stream process. This refactoring leveraged the reuse of certain components and the collapsing of loops, as demonstrated in 1. Despite its theoretical soundness, this solution seems to encounter issues due to the gmem configuration, even though it performs well in cosimulation.

6 Conclusion

This project explored the optimization of the Merge Sort algorithm through hardware acceleration using the Xilinx PYNQ-Z2 development board. By leveraging the parallel processing capabilities of FPGAs.

The Splitting Approach demonstrated effective utilization of FPGA resources by dividing the array into streams and performing merge operations in parallel, resulting in a significant speedup. The resource utilization metrics indicated a balanced use of the available hardware, ensuring optimal performance without overburdening the FPGA.

Unfortunaly the Two Streams Approach, while theoretically sound, encountered practical challenges due to increased resource demands and potential deadlock issues during the merging process. Despite successful cosimulation results, the higher resource utilization compared to the Single Stream Approach posed a significant limitation.

Potential future work includes refining the Two Streams Approach to mitigate resource utilization challenges and further optimizing the IP core for diverse practical applications.