EE5332 : Mapping DSP Algorithms to Architectures Implementation and Analysis of Transpose form FIR filters

Bachotti Sai Krishna Shanmukh EE19B009
P N Neelesh Sumedh EE19B047

August 23, 2022

1 Introduction

Finite Impluse Response (FIR) filters are widely used in DSP applications like audio filtering and image processing. An FIR filter convolves input signal by the weights of the filter. Two of the well known FIR architectures are direct form and transpose form.
The critical path of **Direct form** FIR filter scales up with number of taps in the filter. Hence this implementation requires extra registers(delay elements) for pipe lining. The **Transpose form** FIR filter is an alternate way to implement an FIR filter. In this implementation, we can see that the number of delays is same as that of the direct form, but the critical path contains only one multiplier and one adder. Higher throughput can be achieved without adding much extra pipeline registers.

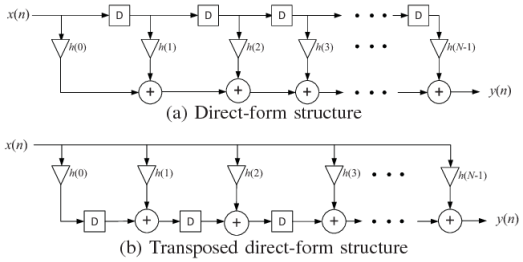


Fig. 1. Typical structures of sample-by-sample processed FIR filters

2 Literature Review

One of the transpose form FIR filter implementation is discussed in Mohanty et al[1]., which primarily focuses on Multiple constant multiplication alogrithm. One of the proven to be area-delay efficient algorithms is Multiple Constant Multiplication which can be implemented for Fixed Weight FIR Filters. The MCM scheme is more effective when a common operand is multiplied with more number of constants. Therefore, the MCM scheme is suitable for the implementation of large order FIR filters with fixed coefficients.

3 Baseline

We chose a simple C/C++ code with O(N*M) complexity where N is number of input samples and M is number of filter coefficients. When run on an general purpose x86 processor, the average execution

time was clocked at 6.3096 ms for an input size of 1,32,301 and a filter size of 16. The sequential nature of compute across all weights for a given input sample is a major drawback.

4 GPU Implementation

We then implemented a CUDA program with a kernel which does all multiplications and additions in parallel for a given input sample and is iterated over the samples. The number of threads used is equal to number of taps in the filter. For a filter size less than the warp-size (32), there is a wastage of hardware as the other threads in warp do no useful work. For a filter size greater than the warp size, there must be a synchronization barrier on a thread-block level for correct functionality which slows down the performance. For an input size of 1,32,301 and filter size of 16, a 12GB NVIDIA Tesla K80 GPU took 22.320641 ms for execution. Doing nvprof suggests that the kernel compute is active for 99.59 percent of total GPU time, suggesting that memcpys are not a bottleneck here, rather the computation in FIR filter.

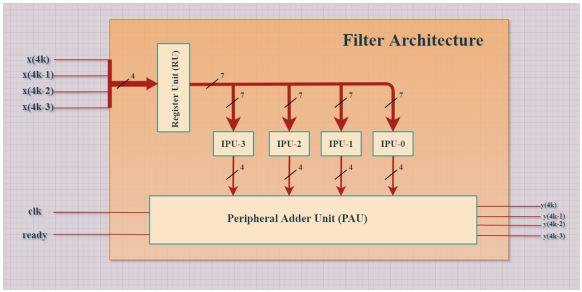
5 Vivado HLS

Using Vivado HLS, a resource and timing comparison was made between the direct form and transpose form implementations with no loop unrolling. For a target clock period of 10ns and filter size of 16, the direct form has a latency of 78 cycles and loop is not pipelined. Whereas, the transpose form has a latency of 36 cycles, with a pipelined loop whose initiation interval is 2. Most importantly, the FF estimate of transpose form (581 FFs) are less in compared to the direct form (606 FFs).
More optimizations are made to transpose form, such as loop unrolling which decreases the latency to 14 cycles. Array partitioning can be implemented to improve the access time of elements in array. The FF used can also be reduced by using BRAMs for acc array (80 FFs reduced in exchange for 2 BRAMs). Finally, the entire function is pipelined to II=1 and all computations in loop are unfolded. This results in best latency of 20ns with a resource estimate of 1335 FFs and 943 LUTs. Further implementing a simple AP HS interfacing increases the latency to 40ns and resources to 1337 FFs and 965 LUTs.

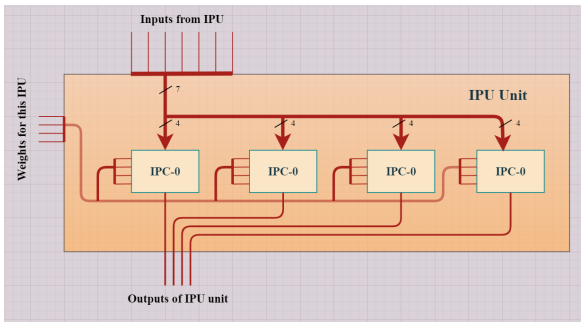
6 Verilog Implementation of MCM based approach

The verilog implementation involves the unrolled versions of transpose form of the FIR filter. We have implemented 2x, 3x and 4x unrolled versions similar to the block FIR filter mentioned in Mohanty et al[1]. The Architecture of the 4x unrolled version is shown in the figure below. The inputs of the architecture are x(4k-3), x(4k-2), x(4k-1), x(4k). These are the four consecutive inputs of the data which we are trying to filter. Off these inputs, x(4k-2), x(4k-1), x(4k) are stored in register to use them in the next iteration of computations. These stored values will become x(4k-4), x(4k-5), x(4k-6). These seven values are sent into units called Inner Product Unit (IPU). The IPU performs a matrix-vector multiplication between the inputs and weights. For a particular IPU, fixed weights are given. This matrix-vector multiplication is performed as four inner products between four input vectors and the weights. The architecture of IPU is given below.

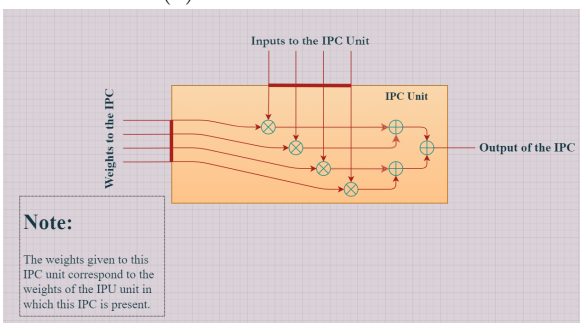
The Inner product in IPU is performed by another module called Inner Product Cell (IPC). This module performs four multiplications on the four inputs and adds these computed values in adder-tree fashion. The weight values are given as inputs to the IPC from the IPU module.
The outputs of the IPUs are then sent into the Pipelined Adder Unit (PAU). In this, we add these values in pipelined fashion such that, at a given instant of time, we add the correct input which is required for the output. The PAU for the 4x implementation has 12 registers or delay elements. The



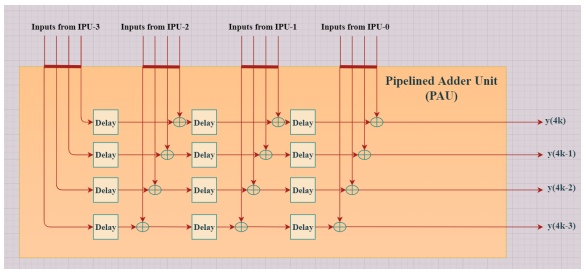
(a) Filter Architecture



(b) IPU Module



(c) IPC Module



(d) Pipelined Adder Unit

Figure 1: Architecture and Modules

PAU has 4 outputs, each corresponding to the filter outputs: y(4k-3), y(4k-2), y(4k-1), y(4k).

7 Comparision of the Implementations

Let us now compare the verilog implementations and also the HLS implementation.

Results					
Structure	LUTS	DSPS	FlipFlops	T_{cp}	Throughput
4x Unroll	3064	192	512	9.024	443.2 MHz
3x Unroll	1906	135	576	8.576	349.8 MHz
2x Unroll	1440	96	512	8.038	248.8 MHz
HLS Pipelined	943	0	1335	6.860	145.77 MHz

From the above results, we can infer that the HLS has clever optimization techniques which bypasses the usage of DSPs and instead uses for LUTs and FFs for the entire computation. Also, the critical path estimated is the least. However it uses too many FFs compared to the unfolded custom verilog implementations which explains the low value of critical path. In the custom verilog modules, as we increase the unroll factor the throughput also increases linearly. The DSPs used in 4x Unrolled module is exactly twice the DSPs used in 2x unrolled module. Since the FFs used are roughly same across the three unrolled versions, with increase in other hardware units like LUTs and DSPs, the critical path also increases.
The critical path of 4x implementation contains 2 multipliers and 2 adders where as the 3x implementation contains 2 multipliers and one adder. Thus we can see a decrease in the clock period in case of 3x implementation. In 2x implementation, the critical path contains only one multiplier and one adder, so this clock period is further less than that of 3x implementation.

```
Execution time for 1,32,301 samples:
HLS Loop Unrolled (Not overall pipelined) = 19.845ms
HLS Pipelined  = 1.323ms
```

```
4x Unrolled = 0.298 ms
3x Unrolled = 0.354 ms
2x Unrolled = 0.589 ms
```

The HLS Loop Unrolled non-pipelined version obviously takes much more time to execute than the pipelined version which is seen the execution time above. The 2x unrolled verilog version takes less execution time than half of the HLS pipelined version.
Now comparing between the 2x, 3x, 4x unrolled versions, we can see that the 4x unrolled version takes around 0.298ms which is around half the time taken by the 2x version. The 3x unrolled version takes 0.354ms which is 2/3rd of the time taken by the 2x unrolled version. As we can see here, the time taken by the unrolled versions are satisfying the expected observations.

8 Division of Work

- Literature Review - EE19B009 and EE19B047
- Gaussian Filter Weights Signal Data Quantization - EE19B047 and EE19B009
- Baseline Implementation - EE19B047
- GPU Implementation - EE19B009
- Optimizations in HLS, Synthesis, Co-simulation and Analysis of Timing Hardware Estimates - EE19B009
- 2x, 3x, 4x unrolled FIR Verilog Modules for Block Processing using MCM - EE19B047
- Testbenches and Debugging - EE19B009 and EE19B047
- Comparison and Analysis of FIR Implementations - EE19B009 and EE19B047

9 Code and Video Links

Code (GitHub Repo): GitHub Link Video Link: Link

10 References

[1]: A High-Performance FIR Filter Architecture for Fixed and Reconfigurable Applications

[2]: Vivado HLS Pragmas