HLS Optimization Tutorial

1 Introduction

This document includes a set of simple examples demonstrating how to transform the implementation of a polynomial multiplication ($a \times x^2 + b \times x + c$) in different approaches. There is minimal change to the C/C++ source code regarding the multiplication implementation in HLS. Most of optimizations (pragmas) are driven from the 'poly.tcl' TCL script for controlling Vivado HLS. Latency and Initiation Interval are two important metrics that represent the attribute of the hardware. We define II for the entire function as well as individual subloops. The constraints covered here include pipelining and II, array partitioning, streaming, and data-flow. With these four sets of constraints, you should be ready to reasonably optimize design tasks with HLS.

2 Definitions

Pipelining a loop with II="N" means that HLS should try to finish one iteration of the loop every N clock cycles. Normally you'd set II=1, which means that it finishes one loop iteration every clock cycle. Resource count often does not increase by much, and may even decrease - the same hardware is being used but it's being utilized more of the time. Instead of a RAM with the source data being read every fifth cycle (and the other four cycles being used for processing), it's being read every cycle while processing continues for the previous four elements. As a result, for many processing tasks pipelining is a very good approach. However, it has a limitation: you can't pipeline to less than one clock cycle per iteration, so the absolute minimum time taken for a loop with M iterations is M cycles (normally it's a little bit more as starting and stopping the pipeline takes time).

Unrolling a loop with factor="N" means that HLS should create N copies of the processing hardware and run them in parallel. This has the advantage that it is possible to finish lots of iterations in every single clock cycle - I've had a block that did 54 iterations of a loop in one cycle. The disadvantage is that it takes a lot of hardware. If you're going to process 30 elements from an array at once, you need to be able to read 30 elements at once. Since Xilinx block RAMs have a maximum of two ports, this implies that you're going to need at least 15 block RAMs in parallel. If you don't have that, then HLS will construct a huge state machine so that your processing hardware gets sequential access to the RAM, which essentially drops you back to the un-unrolled performance while using 30 times as much hardware. As a result, unrolling only makes sense when you have (or can make) data structures that work with it. Other key requirements are that the extra hardware it generates (N sets of the processing hardware) is justified by the performance gain.

3 Code Structure

- **simple**: a basic polynomial implementation. The TCL file only contains a clock period constraint. The resulting design has a Latency=41 and an II=41. No pipelining results in terrible performance.
- **loop**: the example is modified to read/write data from memory arrays and a loop in embedded in the function. The TCL file now contains an II + Pipelining constraint. This gives us a Latency=3 and II=1 for the loop, and a function Latency of 1K and II= 1K.
- **loop-unroll**: to get more parallelism from the problem, we replicate the hardware by splitting the loop across replicated hardware blocks. The TCL file simply provides an additional unroll constraint along with an array partitioning constraint to split the RAMs that supply x and y. The resulting Latency=8, II=1 but the function latency is 256 (1K/UNROLL FACTOR)
- **loop-unroll-stream**: we modify the design significantly to accept data from streams instead of RAMs. Streams that are AXI compatible allow design components to be stitched together. This is the hardware API for connecting things. We explicitly create/allocate memories and create separate reading/writing loops for input and output streams. The TCL file now contains constraints for enforcing II on all three loops, array partitioning + unrolling. The resulting loops are able to achieve an II=1, Latency=8, but the overall function Latency=2.3K cycles and II=2.3K cycles.
- **loop-unroll-stream-dataflow**: we add a constraint to parallelize the various loop bodies. This is done with a dataflow constraint + ping-pong hint. This decouples all the three loops into overlapping executions with dependencies handled using double-buffering optimization. This optimization simply improves function-level II to 1K, which is the maximum latency of the three loop bodies.

4 Conclusion

These experiments are run using Vivado HLS 2017.4 and your numbers may vary with the tool version you are using. We tabulate the overall results of the various design refinement in Table 1 (numbers are approximate).

Table 1: Result comparison.

Design	Fn. Latency	Fn. II	Loop Latency	Loop II
simple	41	41	N/A	N/A
loop	1K	1K	3	1
loop-unroll	256	256	3	1
loop-unroll-stream	2.3K	2.3K	8	1
loop-unroll-stream-dataflow	2.3K	1K	8	1