Optimizing FFT Resource Efficiency on FPGA using High-level Synthesis

Guanglin Xu, Tze Meng Low, James C. Hoe, Franz Franchetti
Department of Electrical and
Computer Engineering
Carnegie Mellon University
Email: {guanglinxu, lowt, jhoe, franzf}@cmu.edu

***Abstract*—The practical use of High-level Synthesis (HLS) for FPGAs may compromise resource efficiency, which is a common design goal in FPGA programming. Specifically, when designing the iterative datapath for computing a FFT, combining naive HLS code with compiler optimizations can fail to saturate the BRAM ports and incur significant penalty in clock frequency. To solve this problem, we suggest that the target datapath should be explicit in HLS code. Using this solution, our FFT cores reduce the FFT latency by N cycles for the radix-2 N-point FFT comparing with previous HLS-based work and the *Xilinx LogiCORE FFT*. Meanwhile, our designs achieve comparable peak frequency and consume similar resources to *Xilinx*.**

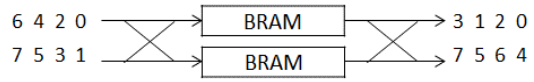
1. Introduction

High-level Synthesis (HLS) is gaining wider acceptance in FPGA programming because of its higher productivity. However, improved productivity using HLS often sacrifices the efficiency of using resources. Specifically, Vivado HLS cannot saturate the BRAM ports and incur significant penalty in peak frequency when the iterative datapath for a FFT is described using naive code.

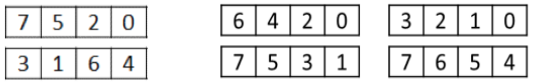
A motivating example is the bit reversal permutation which is a building block of FFT. Figure 1(a) shows an example of 8-point permutation where the data points stream into the datapath in sequential order and stream out in permuted order. The following code shows a high level specification for the permutation hardware that is synthesizable in Vivado HLS. The code does not detail the desired BRAM partition scheme for implementing the array as shown in Figure 1(b) for saturating the BRAM ports. Vivado HLS cannot find the desired scheme either because it only supports the two schemes shown in Figure 1(c).

```
1  void bit_rev(stream &X, stream &Y)
2  {
3   for (j=0; j<N/2; j++) {
4      in = X.get();
5      buf[2*i+0] = in[0];   buf[2*i+1] = in[1];
6   }
7   for (j=0; j<N/2; j++) {
8      out[0] = buf[reverse_bit(2*j+0)];
9      out[1] = buf[reverse_bit(2*j+1)];
10     Y.put(out);
11  }
12 }
```

Previous work has relied on other HLS compilers to saturate the BRAM ports for other building blocks in radix-2 FFT, but the peak clock frequency cannot exceed 150 MHz [5]. To solve this problem, we argue that the target datapath has to be made explicit in HLS code. Using our solution, we reduce the FFT latency proportionally to the problem size while achieving similar peak frequency and consuming similar resources to the *Xilinx LogiCORE FFT*.



(a) A desired datapath containing BRAMs for 8-point bit reversal permutation



(b) A desired BRAM partition scheme (c) Two available BRAM partition schemes in Vivado HLS

Figure 1: The lack of desired memory partition scheme for bit reversal permutation in Vivado HLS.

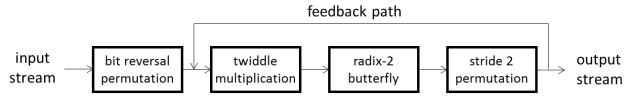


Figure 2: The block diagram of the iterative datapath for radix-2 FFT. It features a few blocks and a feedback path.

2. Iterative Datapath for FFT

This study implements the iterative datapath [2] for various radix FFT as shown in Figure 2 (radix-2 is used for easier explanation). It features a feedback path and four streaming building blocks: the bit reversal permutation, the twiddle multiplication, the radix-2 butterfly and the stride 2 permutation. Building blocks within the feedback path are used multiple times and the latency of each block can be hidden when given enough independent butterfly operations.

3. HLS Implementations

Combinational blocks: Dataflow intensive combinational blocks can be efficiently generated from HLS by using naive code. These blocks include the twiddle multiplication, the radix-2 butterfly and the switch inside a permutation.

Permutation blocks: The datapath for a permutation block is derived using the method from [1], which is composed of two switches and BRAMs. Control signals to switches and addresses for BRAM access are generated externally and hard-coded in the program, ensuring no port conflict in each BRAM access. As shown in the following

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

pseduo code, the datapath is implemented with a write stage and then a read stage.

```
void write_stage(type buf[N], type in[2]) {
    switch_in(in, in_tmp, control);
    bram_write(buf, in_tmp, address);
}
void read_stage(type buf[N], type out[2]) {
    bram_read(buf, out_tmp, address);
    switch_out(out_tmp, out, control);
}
```

Feedback path as HLS loops: A feedback path can be synthesized from a loop with loop-carried dependencies. Such a loop requires the loop body to start with reading from a buffer and end with writing back to the buffer. However, the streaming blocks shown in Figure 2 are not surrounded by buffer accesses. To solve this problem, we exploit a feature of the permutation blocks that they are built with a write stage and a read stage. These two different stages, after a slight modification to the datapath, can become two different ends of the feedback path so that it can be coded in HLS explicitly. As shown in the following code, the modification requires a prolog write stage of the bit reversal permutation in the first loop. Then the second loop nest where a feedback path is to be synthesized features a mux for selecting a read stage of either the bit reversal permutation or the stride 2 permutation determined by the FFT stage. Finally, an epilog read stage of the stride 2 permutation is used in the third loop.

```
void pease_fft(type X[N], type Y[N])
{
#pragma HLS INTERFACE axis port=X,Y
#pragma HLS ARRAY_PARTITION variable=X,Y cyclic \
        factor=2
 for (j=0; j<N/2; j++) {
#pragma HLS PIPELINE
    in[0] = X[2*j+0];  in[1] = X[2*j+1];
    write_stage<bitrev>(buf, in);
 }
 for (i=0; i<LOG2N; i++) {
  for (j=0; j<N/2; j++) {
#pragma HLS DEPENDENCE variable=buf inter false
#pragma HLS PIPELINE
     if (i==0) read_stage<bitrev>(buf, in);
     else read_stage<stride2>(buf, in);
     twid_mult(in, twiddled, i);
     radix2_butterfly(twiddled, out);
     write_stage<stride2>(buf, out);
  }
 }
 for (j=0; j<N/2; j++) {
#pragma HLS PIPELINE
    read_stage<stride2>(buf, out);
    Y[2*j+0] = out[0]; Y[2*j+1] = out[1];
 }
}
```

In the code, several *#pragma*s are used to specify the characteristics of the datapath. The PIPELINE pragma will initiate loop iterations at every cycle. Because all three loops are all fully pipelined, the latency of one FFT can be calculated through iteration counts, which is $N + \frac{N \log_2 N}{2}$ (N for the problem size) for radix-2 FFT. The INTERFACE and PARTITION pragmas specify a streaming interface and the data width. The DEPENDENCE pragma is used to disabled the conservative dependence analysis.

4. Evaluation

We compare our results to a previous HLS-based work [5] and the Xilinx LogiCORE FFT [4]. These three solutions are named as *HLS*, *PICO* and *Xilinx*, respectively.

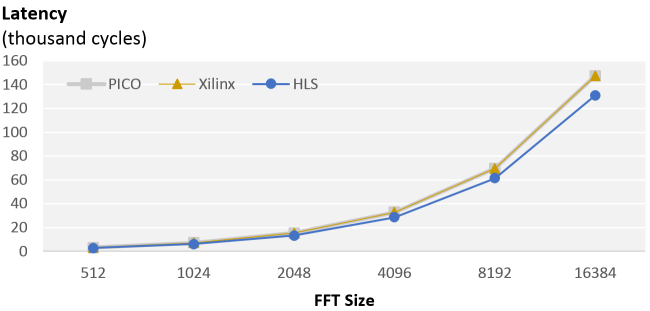


Figure 3: Latency of radix-2 FFT. Lower is better.

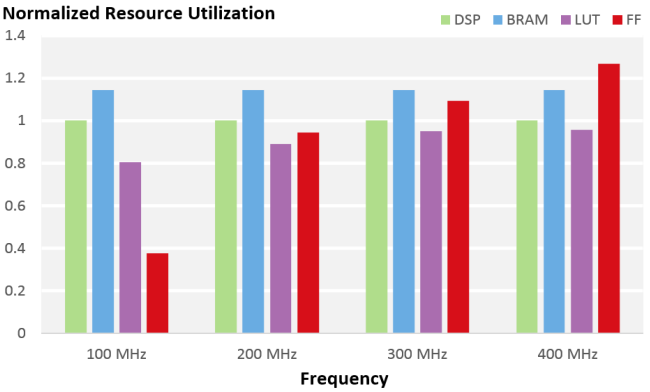


Figure 4: Resource utilization of radix-4 FFT-1k normalized to *Xilinx*.

Firstly, the radix-2 FFT latency is evaluated because radix-2 is the only option where *PICO* works. *PICO* formulates its latency as $2N + \frac{N log_2 N}{2}$ which is N larger compared to our results. The latency of *Xilinx* is acquired from RTL simulation and is similar to those of *PICO*. This can be because they do not saturate the BRAM ports for data loading/unloading. Figure 3 shows that growing the problem sizes leads to more latency savings in our results.

Secondly, the resource utilization across various frequency targets of radix-4 FFT-1k are compared to *Xilinx* (*PICO* data for our FPGA devices are not available). Both results share a similar peak frequency around 400 MHz. The results are collected after place-and-route on Xilinx ZC706. Figure 4 shows that our results consume resources similarly with *Xilinx* for a wide range of frequency targets. Our results use 14% more BRAMs because we do not implement extra BRAM optimization for twiddle factors that exists in *Xilinx*. When a low frequency target is set, our solutions use fewer FFs and LUTs because Vivado HLS optimizes resource with respect to the frequency target more aggressively than *Xilinx*.

5. Conclusion

To saturate the BRAM ports and achieve high peak frequency for FFT, we make the iterative datapath explicit in HLS code. Our solution reduces FFT latency, sustains in high frequency and consumes comparable resources to *Xilinx* expert implementation. Since permutation and iterative datapath are not unique for FFT, our solution could potentially be applied to other problems as well.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

References

[1] M. Püschel, P. Milder and J. Hoe, ”Permuting streaming data using RAMs”. *Journal of the ACM*, vol. 56, no. 2, pp. 1-34, 2009.

[2] P. Milder, F. Franchetti, J. Hoe and Ma. Püschel, ”Formal Datapath Representation and Manipulation for Implementing DSP Transforms”. *2008 45th ACM/IEEE Design Automation Conference*, Anaheim, CA, 2008, pp. 385-390.

[3] ——, ”Vivado Design Suite User Guide: High-Level Synthesis (UG902 v2016.1)”, 2016. [Online].

[4] ——, ”Fast Fourier Transform v9.0, LogiCORE IP Product”, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ipdocumentation/xfft/v90/pg109-xfft.pdf. [Accessed: 26- May- 2017].

[5] E. Oruklu, R. Hanley, S. Aslan, C. Desmouliers, F. Vallina and J. Saniie, ”System-on-Chip Design Using High-Level Synthesis Tools”, *Circuits and Systems*, vol. 03, no. 01, pp. 1-9, 2012.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE