Influence of FPGA implementation methods in High-Level Synthesis

Yusuke Watanabe
CRAFT WORK Co.,Ltd,
5F OS Bldg 3-5-15 Shibasaki-cho, Tachikawa, Tokyo, 190-0023, Japan

Graduate School of Life Science and Systems Engineering, Kyushu Institute of Technology,
2-4 Hibikino, Wakamatsu-ku, Kitakyushu, Fukuoka, 808-0196, Japan

Hakaru Tamukoh
Graduate School of Life Science and Systems Engineering, Kyushu Institute of Technology,
2-4 Hibikino, Wakamatsu-ku, Kitakyushu, Fukuoka, 808-0196, Japan
E-mail: watanabe.yusuke898@mail.kyutech.jp, tamukoh@brain.kyutech.jp
http://www.lsse.kyutech.ac.jp/english/

Abstract

We explain about how the difference of implementation methods written in C++ in High-Level Synthesis (HLS) influences on latency for tiny You only look once (YOLO) v2, a real-time object detection system to infer on an FPGA. To utilize features of FPGA, we need to implement hardware-oriented algorithms such as the weight binalization. We primarily focus on convolution in tiny YOLO v2 network and we report execution results on the Xilinx SDSoC development environment to know whether methods are appropriate or not.

Keywords: Convolution, FPGA, High-Level Synthesis, Hardware-Oriented Algorithm

1. Introduction

When a robot which power supply is generally limited detects objects, a deep neural network is often used. FPGAs are good choices to implement neural networks while reducing energy consumption. To reduce implementation time in FPGAs, High-Level Synthesis (HLS) has been used recently. It automatically creates digital hardware from C++ source codes and we don't need to use time-consuming hardware description languages. But typical C++ implementation is not hardware-oriented and therefore created hardware don't bring results as we would think.

In this paper, we experiment with some convolution operations in an object detection algorithm to utilize FPGAs better in HLS.

In conclusion, when we use HLS, hardware-oriented implementation is preferable and brings dramatically improved hardware.

2. Method

We explain about our implementation methods of convolution operations which occupy much of processing latency in object detection. As an object detection system, we use binarized tiny YOLO v2 which binarizes original tiny YOLO v2 input and weights data according to Ref. [1]. Convolution operations are executed in three dimensions of height, width and input channels. Due to input and weights data binarization, we can use bitwise operations which reduce FPGA resources utilization and latency and therefore can be regarded as hardware-oriented.

Eventually we use them as convolution operations and experiment with the following four methods.

(1) no bitwise operation
(2) width dimension
(3) channel dimension
(4) mix of width and channel dimension

The first method doesn't use bitwise operations. The second one uses bitwise operations along the width dimension and the third one along the channel. The last one is a mix of the second and the third. We apply bitwise operations to the dimension which has the largest value among height, width and input channel. We don't try along a height dimension because height values are always smaller than width ones.

3. Experiment

To experiment, we prepare for a specific input RGB image file which width and height are 384 and 288 pixels respectively. HLS was executed on 2018.3 release of Xilinx SDSoC Development Environment. Our targetFPGA board is a Zynq UltraScale+ MPSoC ZCU102 evaluation kit. We used automatically created synthesis reports to evaluate the performance of our implementation.

Table 1 shows the clock cycles to produce output. Differences between min and max come from the presence of conditional branches in source codes. Table 2 shows resources used to implement the binarized tiny YOLO v2 design in percentage.

Table 1. Latency comparison.

method		min	max
no bitwise op.		2,546,563,540	3,072,073,924
bitwise op.	width dimension	471,558,114	4,811,180,162
	channel dimension	159,132,162	3,695,079,650
	mix of width and channel dimension	115,836,258	4,634,795,522

Table 2. Resources usage comparison

method		BRAM	DSP	FF	LUT
no bitwise op.		1,364	67	31,355	54,777
bitwise op.	width dimension	328	22	22,388	42,965
	channel dimension	328	10	274,116	75,125
	mix of width and channel dimension	328	15	181,935	79,569
Available		1,824	2,520	548,160	274,080

4. Results

As for latency, if we use bitwise operations, the min clock cycles become lower and the actual execution time on the FPGA board also becomes shorter although max clock cycles become higher. This fact means that most processing doesn't take the max clock cycles paths. Further if we apply bitwise operations to the mix of width and channel dimension as shown in Table 1, clock cycles reduction becomes higher.

About resources, if we use bitwise operations, we can substantially reduce BRAM usage. If a network becomes deeper, we use more BRAM and BRAM shortage is likely to happen. It is important for BRAM not to get wasted.

5. Conclusion

We conclude that hardware-oriented algorithms like bitwise operations in this paper are essential even when we design hardware from C++ using HLS. This is because HLS environments doesn't automatically create the best hardware yet. There are challenges to exactly grasp which implementation brings better hardware. So we need to become used to how to write hardware-oriented codes in C++. Actually our current implementation is far from the best and we need to make it more hardware-oriented.

References

1. Courbariaux, Matthieu & Hubara, Itay & Soudry, Daniel & El-Yaniv, Ran & Bengio, Y.. (2016). Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1.
2. J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91.
3. Nakahara, Hiroki & Yonekawa, Haruyoshi & Iwamoto, Hisashi & Motomura, Masato. (2017). A Batch Normalization Free Binarized Convolutional Deep Neural Network on an FPGA (Abstract Only). 290-290. 10.1145/3020078.3021782.