DnnWeaver v2.0: From Tensors to FPGAs

Hardik Sharma[†§] Jongse Park[†§] Balavinayagam Samynathan[§] Behnam Robatmili[§]
Shahrzad Mirkhani[§] Hadi Esmaeilzadeh[‡§]

[†]Georgia Institute of Technology [§]Bigstream, Inc. [‡]University of California, San Diego

hsharma@gatech.edu {jongse, bala, behnam, shahrzad}@bigstream.co hadi@end.ucsd.edu

ABSTRACT

Deep Neural Networks (DNNs) has proven to be an effective tool in a wide range of domains [1,2,3,4,5], including emerging applications such as autonomous driving [6]. However, the use of DNNs to enable these emerging applications is predicated upon providing the hardware technologies that can both tame their high computation demands and adapt with algorithmic advances. As such, FPGAs are an attractive choice for DNNs since they represent an intermediate point between the efficiency of ASICs and the programmability of general purpose processors. However, obtaining both performance and energy efficiency with FPGAs is a laborious task even for expert hardware designers. Furthermore, the large memory footprint of DNNs, coupled with the FPGAs' limited on-chip storage makes DNN acceleration using FPGAs more challenging.

This work tackles these challenges by devising DnnWeaver, a framework that *automatically generates* a synthesizable accelerator for a given (DNN, FPGA) pair from a high-level specification in Tensorflow [7]. To achieve large benefits while preserving automation, DnnWeaver generates accelerators using hand-optimized design templates. First, DnnWeaver translates a given high-level DNN specification to a novel ISA that represents a macro dataflow graph of the DNN. The DnnWeaver compiler is equipped with our optimization algorithm that tiles, schedules, and batches DNN operations to maximize data reuse and best utilize target FPGA's memory and other resources. The final result is a custom synthesizable accelerator that best matches the needs of the DNN while providing performance and efficiency gains for the target FPGA.

We use DnnWeaver to generate accelerators for eight different deep networks targeted for three different FPGAs, Xilinx Zynq, Altera Stratix V, and Altera Arria 10. We rigorously compare the generated accelerators to multicore CPUs (ARM A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650Ti, and Tesla K40). Table 1 reports the results. These results show that DnnWeaver generated accelerators exceed CPUs in performance and in two of three cases (Zynq and Arria 10) deliver higher Performance-per-Watt than GPUs. To achieve these benefits, the programmer only defines the topology and layers of the DNN (< 300 lines of code) without dealing with hardware design or optimization. The relatively low programmer effort is particularly significant since the source code for our templates is over 10,000 lines of code and is optimized hardware by experts over the course of one year. As Figure 1 illustrates, the DnnWeaver generated accelerators for Zynq and Arria 10 lie on the Pareto frontier. The Tesla GPU represents the high-power high-performance Pareto optimal point. The results suggest that when power budget is limited, DnnWeaver enables FPGAs to operate as a platform of choice for deep networks.

We show the effectiveness of DnnWeaver, by performing a real-time object detection using the Yolo-v2-tiny DNN [8] using live video stream from a camera attached to a flying drone. To support the future efforts in the research community, we have open-sourced DnnWeaver[1], along with the software for the live drone demo[2]. This open-source initiative has already attracted adopters from academia [9, 10].

Table 1: Speedup and Performance-per-Watt comparison of DnnWeaver generated accelerators. Each cell represents the benefits of the FPGA in row-heading relative to the platform in column-heading.

FPGA	ARM A15	Xeon E3	Tegra K1	GTX 650Ti	Tesla K40
			Speedup Comparison		
Zynq	4.7×	0.59×	0.52×	0.15×	0.03×
Stratix V	22.39×	2.81×	2.43×	0.7×	0.15×
Arria 10	47.26×	5.94×	5.08×	1.48×	0.33×
			Performance-per-Watt comparison		
Zynq	11.5×	16.6×	1.7×	3.2×	1.6×
Stratix V	5.5×	7.9×	0.8×	1.5×	0.8×
Arria 10	9.6×	13.9×	4.8×	2.7×	1.3×

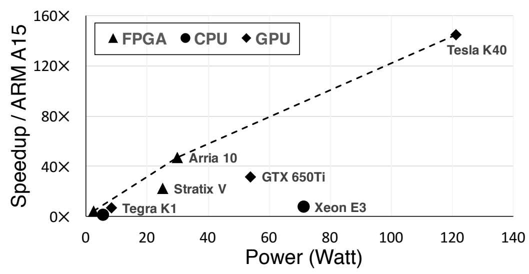

Figure 1: DnnWeaver generated accelerators for Zynq and Arria 10 lie on the Pareto frontier (the dashed line). These results suggest that DnnWeaver makes FPGAs a compelling alternative when the power budget is limited while for high power setting, GPUs are more effective.

1. OVERVIEW

This work aims to create an automated framework that (1) completely dissociates programmers from the details of hardware design and optimization; (2) deals with the limited availability of on-chip resources (e.g., on-chip memory); and (3) provides a scalable and reusable FPGA acceleration framework, which delivers high performance and large efficiency gains for continuously evolving DNN models on different FPGA platforms. The foremost task required by DnnWeaver workflow, shown in Figure 2, is that the programmer specifies the DNN using a high-level programming interface as discussed below.

Programming interface. Using the DnnWeaver API, the programmer specifies the dataflow graph of the DNN, wherein the edges are multi-dimensional arrays or tensors, and the nodes are transformations applied to the incoming tensors such as convolution. The code snippet in Figure 3 defines a

[1]`http://dnnweaver.org`
[2]`https://github.com/ardorem/dnnweaver2.drone`

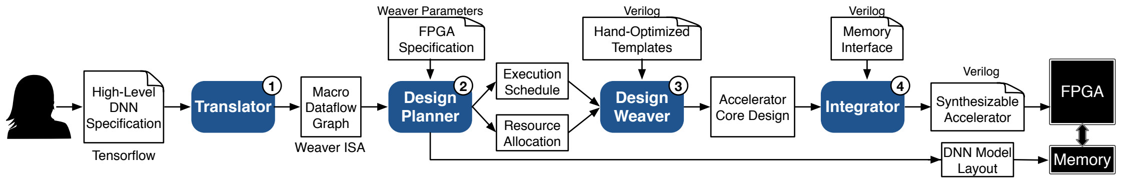

Figure 2: Overview of DNNWEAVER which takes a high-level specification of the DNN and the target FPGA to generate the accelerator design as synthesizable Verilog along with the accelerator execution schedule and the layout of the DNN model in the memory.

```
g = dnnweaver2.Graph('YOLO-v2-tiny')
with g.as_default():
   i = get_tensor(shape=(1,416,416,3),
                  name='input',
                  trainable=False)
   weights = get_tensor(shape=(16,3,3,3),
                        name='weights',
   biases = get_tensor(shape=(filters),
                       name='biases',
   conv0 = conv2D(tensor_in, weights,
                  biases, pad='SAME')
   ...
```

Figure 3: DNNWEAVER Programming Interface

graph called 'YOLO-v2-tiny' and then populates the graph with a single convolution node.

DNNWEAVER workflow. As Figure 2 illustrates, DNNWEAVER automatically transforms the programmer-provided DNN model to an accelerator by generating FPGA synthesizable verilog code. The first component of the DNNWEAVER workflow is the Translator, that converts the DNN's specification to our macro dataflow instruction set architecture (ISA). The next component of the workflow is the Design Planner that accepts the macro dataflow graph instructions and then co-optimize both the resource allocation to hardware templates for the target FPGA platform and the execution schedule for the DNN. Accelerating DNNs is particularly challenging due to their large memory footprint. The Design Planner component automatically *tiles* the computations in each DNN layer to generate an optimized execution schedule that minimizes off-chip data transfers subject to the limited on-chip memory (BRAMs) available on the target FPGA. Design Weaver is the penultimate component of DNNWEAVER which takes as input the resource allocation and the execution schedule determined by the planner to generate the accelerator core. The last component of DNNWEAVER is the Integrator, which adds the memory interface code to the accelerator code. As different FPGAs use different interfaces to the external DRAM, the Integrator contains a library of DRAM interfaces and adds the appropriate code for each target FPGA. DNNWEAVER currently includes the DRAM interface for three series of FPGAs (Xilinx's Zynq and KCU1500, and Altera's Stratix V and Arria 10) from the two major vendors. After the integration, the final Verilog code is ready to be synthesized on the target FPGA to accelerate the specified DNN.

2. DNNWEAVER DEMO

To demonstrate the effectiveness of DNNWEAVER and show its practicality, we perform a live demo, which uses a DNNWEAVER-generated accelerator to execute a real-time object detection algorithm using an off-the-shelf FPGA and a camera-attached drone. We use a state-of-the-art real-time object detection algorithm, Yolo-v2-tiny [8], which requires 2.7 billion MAC operations. For this demo, we use Xilinx KCU1500 FPGA development board and a DJI Tello drone [11] that has an in-built camera to obtain a live video feed.

Using the DNNWEAVER programming interface, we wrote a model specification of Yolo-v2-tiny DNN, of which number of lines of code is only 116. Taking this model specification as input, DNNWEAVER automatically generates a synthesizable Verilog code for the accelerator on the KCU1500 FPGA board. We synthesize the accelerator Verilog code on the Xilinx's synthesis tool, Vivado (v2018.2), and program the output bitstream file on the board. The DNNWEAVER compiler also schedules the optimized execution for the Yolo-v2-tiny network, and then compiles the Yolo-v2-tiny model into the accelerator instructions. The scheduled instructions are serialized into a binary file, which needs to be loaded on the accelerator before the DNNWEAVER runtime invokes the execution to the accelerator. While the DNNWEAVER accelerator programmed on FPGA is ready to perform the DNN execution, it needs to be explicitly invoked by the host application. DNNWEAVER comes with a set of runtime APIs to (1) trigger the acceleration computation, (2) read/write to the FPGA DRAM for data communication, and (3) get notified by the accelerator for the execution completion. These APIs are all developed in Python to ease the use of broader community. Using these APIs, the host application can interface with the FPGA.

As mentioned before, our demo takes the live video stream from a flying drone. The host machine equipped with the FPGA board and in turn the DNNWEAVER-generated accelerator, communicates with the drone through Wifi connection. The host application interfaces with the drone and obtains the live video stream from the drone. Then, the host application decodes the video stream into video frames, each of which becomes the input tensor for the Yolo-v2-tiny model. Finally, the host application sends the input, trigger the DNN execution, gets the output tensor, and produces the bounding boxes around the objects detected by the algorithm. The bounding boxes are drawn on the video frames presented on the window, which can be visually displayed on the monitor.

The entire code for the DNNWEAVER stack, including the accelerator template architecture code, written in Verilog, the design planner/weaver, the integrator, the compiler, the runtime APIs, the drone demo code as an example use of DNNWEAVER, is open-sourced at http://dnnweaver.org.

3. CONCLUSION

Our results shows that FPGAs can be a Pareto optimal choice when power is constraining. DNNWEAVER is an initial step in providing an open-source automated acceleration solution that makes FPGAs accessible to the broader community of DNN developers that often do not possess hardware design expertise. Community engagement and contribution are vital for providing a general platform for DNN acceleration. To facilitate community engagement, DNNWEAVER has been made publicly available at `http://dnnweaver.org`.

4. REFERENCES

[1] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. 2015.

[2] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, 2015.

[3] Alex Graves, A-R Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *ICASSP*, 2013.

[4] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[5] Adam Coates, Adam Coates, Brody Huval, Tao Wang, David J. Wu, and Andrew Y." Ng. Deep learning with cots hpc systems.

[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[7] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467 [cs]*, 2016.

[8] Joseph Redmon et al. You only look once: Unified, real-time object detection. *arXiv:1506.02640*, 2015.

[9] Dnnweaver tutorial. `http://venividiwiki.ee.virginia.edu/mediawiki/index.php/DNNWeaver`.

[10] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, and Michael Wei. Sharing, protection and compatibility for reconfigurable fabric with amorphos. 2018.

[11] Dji tello drone. `https://store.dji.com/shop/tello-series`.