From High-Level Deep Neural Models to FPGAs

Hardik Sharma Jongse Park Divya Mahajan Emmanuel Amaro
Joon Kyung Kim Chenkai Shao Asit Mishra† Hadi Esmaeilzadeh
Alternative Computing Technologies (ACT) Lab
School of Computer Science, Georgia Institute of Technology †Intel Corporation
{hsharma, jspark, divya_mahajan, amaro, jkkim, cshao}@gatech.edu asit.k.mishra@intel.com hadi@cc.gatech.edu

ABSTRACT

Deep Neural Networks (DNNs) are compute-intensive learning models with growing applicability in a wide range of domains. FPGAs are an attractive choice for DNNs since they offer a programmable substrate for acceleration and are becoming available across different market segments. 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 Caffe [1]. 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 its 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 high performance and efficiency gains for the target FPGA.

We use DNNWEAVER to generate accelerators for a set of eight different DNN models and three different FPGAs, Xilinx Zynq, Altera Stratix V, and Altera Arria 10. We use hardware measurements to compare the generated accelerators to both multicore CPUs (ARM Cortex A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650Ti, and Tesla K40). In comparison, the generated accelerators deliver superior performance and efficiency without requiring the programmers to participate in the arduous task of hardware design.

1 Introduction

Deep Neural Networks (DNNs) are rapidly gaining traction in a wide range of applications such as vision, robotics, video analytics, speech recognition, natural language processing, targeted advertising, and web search [2–9]. Although DNNs offer great prediction accuracy, they require a significant amount of computing power. With diminishing benefits from technology scaling [10, 11], the research community is increasingly turning to specialized accelerators for deep networks [12–15] and other workloads [16–22]. Even though ASICs provide significant gains in performance and efficiency for DNNs [12, 13, 23–26], they may not cope with the ever-evolving DNN models. Furthermore, ASICs and customized cores come at the price of high non-recurring engineering costs over long design periods. Since FPGAs represent an intermediate point between the efficiency of ASICs and the programmability of general purpose processors, they are an attractive alternative for accelerating DNNs. Nonetheless, FPGAs still require extensive hardware design expertise and long design cycles. In fact, several research works [14, 15, 27, 28] have made extensive efforts to provide FPGA accelerators for specific DNN models, or parts of DNN computation, targeted for a particular FPGA platform.

Using FPGAs as an acceleration platform for DNNs is challenging as they offer a limited preset on-chip memory and often possess limited off-chip bandwidth, both of which are critical for high performance. This restriction is particularly limiting for FPGAs since ASIC designs can circumvent this issue by optimally allocating die area to on-chip memory for a single or set of target DNNs. The FPGA's memory and bandwidth limitations are further exacerbated for DNNs due to their high memory footprint, as well as high variability in the number of operations and model sizes for different DNN models (Section 7.1). A rigid accelerator architecture for DNNs may not fully utilize the FPGA's limited resources for every DNN model. It is thus essential co-optimize both the accelerator architecture and the corresponding execution schedule to overcome the FPGA's limited on-chip memory for each DNN model. This work seeks to provide such a solution by developing DNNWEAVER, a framework that generates synthesizable accelerators for a variety of FPGA platforms, while completely disengaging the programmers from hardware design. DNNWEAVER provides a comprehensive and automated solution to make FPGAs available to a broader community of DNN application developers who use a wide range of DNN models and often lack any hardware design expertise.

This paper makes the following contributions to enable FPGA acceleration for a variety of DNNs:

(1) We develop a novel macro dataflow Instruction Set Architecture (ISA) for DNN accelerators. The ISA enables DNNWEAVER to expose a high-level programming interface. The programming interface is the same as Berkeley Caffe [1].

(2) Instead of just designing an accelerator for DNNs, we develop hand-optimized template designs that are scalable and highly customizable. The templates constitute a clustered hierarchical architecture that is contracted or expanded by DNNWEAVER to generate an accelerator that matches the needs of the DNN and the available resources on the FPGA.

(3) We provide a heuristic algorithm to co-optimize both the accelerator architecture and the corresponding execution schedule to minimize off-chip memory accesses and maximize performance. This algorithm strikes a balance between parallel operations and data reuse by slicing the computation and configuring the accelerator to best match the constraints of the FPGA.

Matching computation slice with the configuration of the accelerator is a unique challenge that needs to be addressed to create a framework that can generate highly efficient FPGA accelerators for DNNs. The aforementioned contributions enable DNNWEAVER to exploit the reconfigurability of the FPGAs while managing the large memory footprint of DNNs in the

978-1-5090-3508-3/16/$31.00

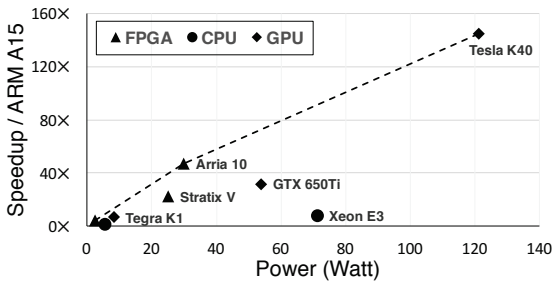


Figure 1: DnnWeaver generated accelerators for Zynq and Arria 10 lie on the Pareto frontier (the dashed line). Tesla K40 represents the other Pareto optimal point. These results suggest that for high power setting GPUs are better programmable accelerators while DnnWeaver makes FPGAs a compelling alternative when the power budget is limited.

```
layer {
 name: "conv1"
 type: CONVOLUTION
 bottom: "data"
 top: "conv1"
 convolution_param {
   num_output: 20
   stride: 1
   kernel_size: 5 } }
```

```
layer {
 name: "pool1"
 type: POOLING
 bottom: "conv1"
 top: "pool1"
 pooling_param {
   pool: MAX
   stride: 2
   kernel_size: 2 } }
```

Figure 2: DnnWeaver programming interface.

limited on-chip storage of FPGAs.

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 both multicore CPUs (ARM A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650Ti, and Tesla K40). The results are summarized as follows:

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×

These results show that DNNWEAVER generated accelerators outperform 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 K40 GPU represents the high-power high-performance Pareto optimal point. The results suggest that when power is limited, DNNWEAVER enables FPGAs to operate as a platform of choice for deep networks.

2 Overview of DNNWEAVER

This work seeks to alleviate the long design cycle necessary for using FPGAs to accelerate a wide variety of DNNs. We aim 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. To achieve these conflicting objectives, we develop DNNWEAVER which combines hand-optimized scalable *template* designs with an *automated workflow* that customizes the templates to match the specifications of a given (DNN, FPGA) pair. The foremost task required by DNNWEAVER is that the programmer specifies the DNN models using a high-level programming interface as discussed below.

Programming interface. The input to DNNWEAVER is a high-level specification of the DNN in Berkeley Caffe format [1]. Caffe is a widely used open-source deep learning framework that takes the DNN specification as input and computes the given model on CPUs and GPUs. The code snippet in Figure 2 shows how two DNN layers, convolution and pooling, are described and connected in Caffe. Section 3 describes the functionality of DNN layers in detail.

As Figure 3 illustrates, DNNWEAVER automatically transforms the programmer-provided DNN model to an accelerator by generating FPGA synthesizable verilog code. DNNWEAVER comprises four software components; (1) the Translator, (2) the Design Planner, (3) the Design Weaver, and (4) the Integrator.

① Translator. The Translator converts the DNN's specification to our macro dataflow instruction set architecture (ISA). Each instruction in the ISA represents a node in the macro dataflow graph of the DNN model. Note that the accelerator does not directly execute these instructions. DNNWEAVER compiler *statically* maps these instructions to control signals in the accelerator and creates an execution schedule. We choose this abstraction to provide a unified hardware-software interface and enable layer-specific optimizations in the accelerator microarchitecture without exposing them to the software. Hence, with this ISA, a variety of accelerator implementations which can be tuned to match the constraints of the target FPGA are possible. Furthermore, the ISA can be extended to support forthcoming layers or parameters. A one-time effort is required to develop the corresponding hardware templates. Section 4 describes this ISA in detail.

② Design Planner. Design Planner accepts the instructions representing the macro dataflow graph of the DNN and uses our Template Resource Optimization algorithm to optimize the hardware templates for the target FPGA platform. The Design Planner then partitions the computation of each layer to groups of operations that share and reuse data. We refer to each group's output as a slice. The slice is spilled to memory after computation and the accelerator proceeds to compute the next slice. Accelerating DNNs is particularly challenging due to their large memory footprint. By slicing its computations, DNNWEAVER manages this large footprint with the limited on-chip FPGA memory. Our Template Resource Optimization algorithm aims to strike a balance between parallel operations and data reuse by slicing computations and configuring the accelerator to best match the constraints of the FPGA (on-chip memory and external memory bandwidth). The Planner schedules slices of operations on the accelerator to generate a static execution schedule and the model layout in memory. Static scheduling simplifies the hardware and maximizes its efficiency and performance. Section 6 elaborates on the details of the Design Planner and our Template Resource Optimization algorithm.

③ Design Weaver. Design Weaver is the penultimate

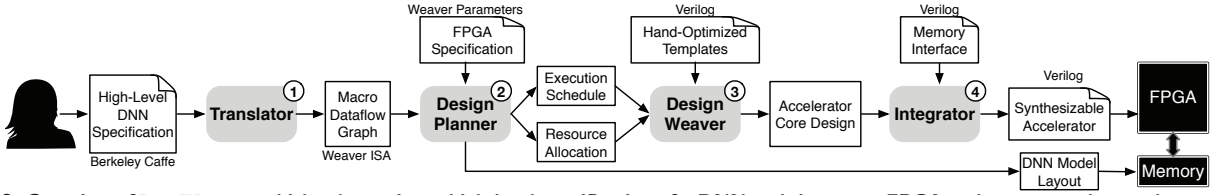


Figure 3: Overview of DNNWEAVER which takes as input high-level specification of a DNN and the target FPGA and generates the accelerator design as synthesizable Verilog along with the accelerator execution schedule and the layout of the DNN model in the memory.

component of DNNWEAVER which takes as input the resource allocation and the execution schedule determined by the planner to generate the accelerator core. The Design Weaver uses a series of hand-optimized design templates and customizes them in accordance to the resource allocation and hardware organization provided by the planner. These templates provide a highly customizable, modular, and scalable implementation for the Design Weaver that automatically specializes the templates to accommodate a variety of DNN that are translated to our macro dataflow ISA. Furthermore, the Design Weaver converts the planner-provided execution schedule into state machines and microcodes, embedded within the hardware modules. Section 5 details the template designs and Section 6 discusses how the Design Weaver specializes the templates.

④ **Integrator.** 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 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.

3 Background: Deep Neural Networks

The advent of deep learning, or more precisely, deep structured learning, can be traced back to Convolutional Neural Networks [29]. Convolutional Neural Networks are commonly used deep learning models, and hence are the focus of our work. As follows, a typical DNN consists of several back-to-back *layers* that represent increasingly abstract representations of the input.

Convolution layer. A convolution operation generates its output by sliding a window of parameters referred to as filters or kernels, over its inputs. A convolution layer is a set of these convolution operations that combine multiple input features and kernels to generate a single or multiple output feature maps. The initial layers of DNN are generally these convolution layers.

Pooling layer. A pooling layer down-samples its input to reduce its size. This layer subsamples each window of the input feature maps to a single pooled output, which is usually the average, maximum, or minimum of the features in the window.

Inner product layer. This layer computes the inner product of an input vector and a weight matrix. Before computing this inner product, the previous layer output that might be multidimensional is arranged as a single dimensional vector.

Activation layer. An activation layer is a dimensionality preserving operation that applies an element-wise transfer function on its input feature map. Typical transfer functions are non-linear, (e.g., *sigmoid* , *tanh*), or piece-wise linear functions (e.g., *rectified linear*, *absolute value*).

Normalization layer. A normalization layer performs local inhibition by sliding a window over its input feature map. The normalization operation first produces the square-sum of the elements in the sliding window and then applies a non-linear function to the square-sum, which is multiplied with the input element being normalized to generate the output.

The programmer specifies the DNN using the layers described above. DNNWEAVER then converts this specification into a macro dataflow ISA that implements the operations of these layers as an abstraction for hardware acceleration. The details of this ISA are discussed in the next section.

Bits	63 – 60	59 – 56	55 – 32	31	30 – 24	23 – 17	16	15 – 10	9 – 4	3 – 0
input	Opcode = 0	Function	Destination Instruction ID	Fixed-Point/ Floating Point	Value Bitwidth	Fraction Bits/ Exponent Bits	1	# of Dimensions		
conv	Opcode = 1						Use the Next Word	Window Width	Window Height	Window Stride
pool	Opcode = 2							# of Neurons		
norm	Opcode = 3									
ip	Opcode = 4									
act	Opcode = 5									
fanout	Opcode = 6		# Destinations				0	Reserved		
output	Opcode = 7		0							

(a) IWORD1

Bits	63 – 48	47 - 32	31 – 16	15 – 0
input	Length of Dimension 0		Length of Dimension 1	
conv	Reserved	Window Width	Window Height	Window Stride
pool				
norm				
ip		# of Neurons		
act	Not Used			
fanout				
output				

(b) IWORD2 (Optional)

Figure 4: Instructions of the macro dataflow ISA.

4 Instruction Set Architecture Design

DNNWEAVER provides a macro dataflow ISA to (1) abstract away the details of accelerator design from the software; (2) enable layer-specific optimizations; (3) facilitate portability across different FPGA platforms; and (4) allow static execution scheduling at compile time. We chose a dataflow architecture to alleviate the von Neumann overhead of general-purpose architectures such as instruction fetch, decode, etc. The accelerator is not expected to execute these instructions. The compiler *statically* translates these instructions to microcodes and state machines. This dataflow architecture does not have explicit registers, which enables DNNWEAVER to impose significantly fewer restrictions on the accelerator architecture and allows portability across different FPGAs. Using an explicit dataflow architecture also allows DNNWEAVER to perform static optimizations at compile time and avoid data dependencies (e.g., register renaming) at runtime. Additionally, the coarse-grained nature of the ISA enables the microarchitecture to incorporate layer-specific optimizations without exposing them through the software stack.

Figure 4 shows the eight instructions of our ISA. These instructions are variable-sized and are designed to be able to express a large variety of deep neural networks. These instructions are further translated to state machines and microcodes at compile time. We use 64-bits to encode each word of the instruction. Since the ISA is a dataflow architecture,

each instruction is assigned a unique 24-bit static ID and none of the instructions include source operands. Instead, a part of instruction opcode encodes the unique static ID of the destination instruction that will receive the results. Below, we discuss each instruction type in our macro dataflow architecture.

Instruction input. The input instruction reads a DNN input (e.g., an entire image) from memory and sends it to another instruction for processing. As Figure 4(a) shows, bits 63–60 in the IWORD1 contain the opcode, which in this case is 0. Bits 59–56 are the function bits and are not used for this instruction. Bits 55–32 contain the unique ID of the destination instruction that will consume the inputs. Bit 31 indicates whether the generated values by this instruction are fixed-point or floating point. Bits 30–24 specify the total bit width of the generated values (e.g., 32). The bits 23–17 encode the number of fraction bits or the exponent, if the generated values are fixed point or floating point, respectively. Bit 16 indicates whether the next 64-bit word is part of the instruction. Bits 15–0 encode the number of dimensions in the DNN input (e.g., two for an input image.) The next instruction word, IWORD2, specifies the size of each dimension as presented in Figure 4(b). If the number of dimensions exceeds two, then more words are added to specify the size of dimensions. After specifying the size of all the dimensions, the next 64-bit value contains the address of the input in the memory.

Instruction conv. This instruction type performs the convolution operation by sliding a window over its inputs. The dimensions of the window and the sliding stride are encoded in bits 15-0, with six bits for the width and the height of the window, and four bits for the sliding stride. If these bits are not enough for specifying the window dimensions, bit 16 is set to 1 and IWORD2 is used to specify the structure of the window. The other fields in the IWORD1 are similar to the input instruction. After the IWORDs, an array of immediate values is used to specify the weights for the convolution operation.

Instruction pool. This instruction performs pooling on its inputs. Similar to conv, the structure of the pooling window and its stride is either specified in bits 15–0 or in IWORD2. The function field specifies the pooling type, such as max, min, or average.

Instruction norm. This instruction performs normalization and its window specification is similar to previous two instructions. The parameters are listed as immediate values after the IWORDs.

Instruction ip. This instruction corresponds to an inner product layer. Bits 15–0 specify the number of neurons in this layer, up to a maximum of 65536. For a larger number of neurons, IWORD2 is used. The immediate value after the IWORD is the address of the inner product weights in the memory.

Instruction act. This instruction corresponds to an activation layer and takes only one IWORD. The function field encodes the type of the activation function (e.g., sigmoid).

Instruction fanout. Since our ISA is dataflow and each instruction only encodes one destination, we provide a fanout instruction. This instruction is single IWORD and the bits 55–32 encode the number of destinations. The following immediate values after the IWORD encode the ID of the destination instructions.

Instruction output. The output instruction does not have a destination instruction. It writes the outputs of the DNN to the memory address specified in the immediate values.

As discussed above, the instructions are translated to state machine and microcodes at compile time. Translation from Caffe to this ISA is straightforward since the instructions match the DNN layers. The Design Planner uses this ISA to customize the pre-designed templates and generate a static schedule for the operations within this macro dataflow, that is best suited for a given (DNN, target FPGA) pair. The next section discusses the hand-optimized template.

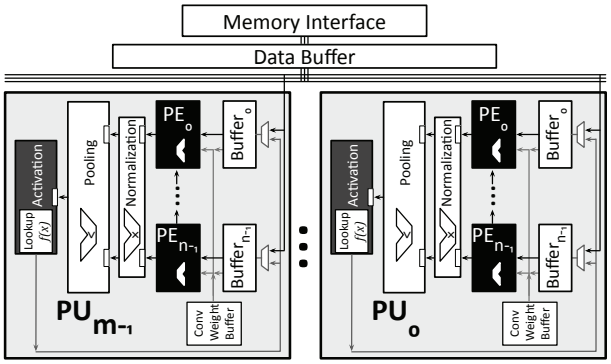


Figure 5: Overview of a clustered hierarchical template design. The template accelerator is divided into Processing Units (PUs) that are comprised of multiple smaller Processing Engines (PEs).

5 Template Accelerator Architecture

The template designs are highly *customizable* and *scalable*. The scalable architecture enables the Design Planner to shrink or expand the accelerator based on the requirements of the DNN and the resource constrains of the target FPGA. Templates are also designed to be *general*. That is, the templates include exchangeable components that realize different layers of DNNs. If a DNN does not require a certain layer (e.g., normalization), the corresponding component is excluded to free resources for other layers.

5.1 Overall Organization

Figure 5 illustrates the template architecture that provides these necessary characteristics. As depicted, the template architecture is clustered with two levels of hierarchy; a collection of self-contained Processing Units (PUs) that comprise a set of smaller Processing Engines (PEs). The PEs and the buffers in the template PU architecture provide compute capabilities for convolution and inner product layers. The customizable normalization, pooling, and activation modules provide support for the other possible layers in DNNs. This clustered architecture provides scalability via modularity and by making the data traffic local to PUs and utilizing a unified bussing fabric across them. These features allow the Design Weaver to generate a concrete accelerator with any number of PUs and PEs-per-PU. Furthermore, the Design Weaver tunes the parameters of the hardware modules; all of which are extensively parameterized.

Specializing the design for a target FPGA. Each FPGA offers a certain number of hard blocks including DSP slices (ALUs) and Block RAMs (on-chip SRAM units, called BRAMs). Using these hard blocks is essential for exploiting the compute capabilities of the FPGA and achieving reasonably high frequency. Thus, the template architecture in Figure 5 maps the PU Buffers to the BRAMs and the ALUs to the DSP slices. The availability of these resources determines the maximum possible number of PEs and PUs for a given FPGA. However, as described in Section 6, the Design Planner determines the composition of the PU based on the size of the feature maps produced by the convolution/pooling/normalization/inner product layers to maximize resource utilization and the overall computation throughput. The next resource is the available off-chip bandwidth which determines the parameters of the Data Buffer that is connected to the memory interface as shown in Figure 5. The Design Planner performs static data marshaling and determines the layout of the

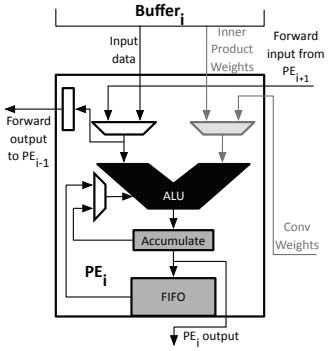


Figure 6: Processing Engine PE$_i$.

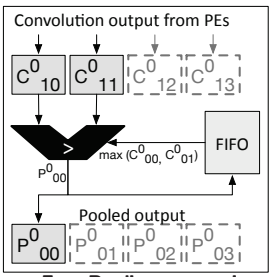


Figure 7: Pooling operations to compute P^0_{00}.

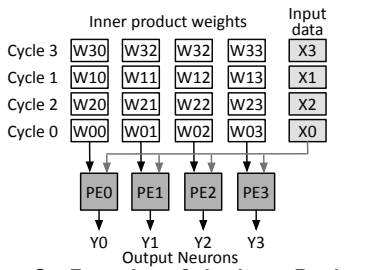


Figure 8: Execution of the Inner Product layer using MACC operations in PEs.

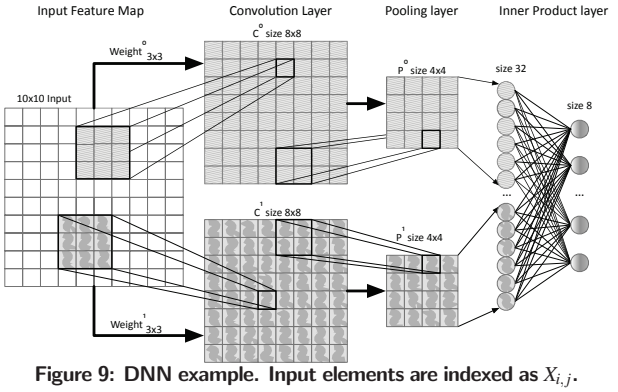


Figure 9: DNN example. Input elements are indexed as $X_{i,j}$.

DNN weights and parameters to streamline transfer of parameters from the memory in contiguous chunks; maximizing the bandwidth utilization. The Design Planner also generates a static schedule for the Data Buffer to fetch data from the external memory and feed the PUs through the inter-PU bus. Static scheduling avoids contention on the bus and alleviates the need for PUs to perform complex handshaking. This approach, in turn, improves the scalability and efficiency of the template architecture.

Processing engines. PEs are the basic compute units that perform convolution, inner product, and parts of normalization. As shown in Figure 6, each PE contains a hard ALU that supports Multiply, and Multiply-Add operations. Neighboring PEs have a unidirectional link that forwards input data from a PE with higher index (PE$_{i+1}$) to the adjacent PE with lower index (PE$_i$). This forwarding link is used to reuse data across the adjacent PEs to minimize data transfer from memory. As depicted in Figure 5, each PE in a PU is associated with a dedicated buffer that feeds inner product weights and input data to the PE. Inner product weights typically require larger storage. These weights are streamed in the dedicated buffers that are mapped to the hard BRAM blocks.

Below, we use a running example (Figure 9) to demonstrate the operations and scheduling of different DNN layers.

5.2 Accelerating Layers of DNN

The first layer of the DNN in Figure 9 contains convolution kernels with a window size of 3×3 that produce two 8×8 outputs. Each convolution output is sub-sampled with a 2×2 max-pooling operation. Outputs from the pooling layer are arranged in a 1×32 vector, as the input to the inner product layer. The inner product layer has eight output neurons and requires a weight matrix of size 32×8. The example accelerator contains one PU with four PEs. For the running example, we assume that the model parameters are contained within the FPGA's on-chip storage. In Section 6, we relax this assumption and account for external memory accesses required when the data cannot fit on-chip.

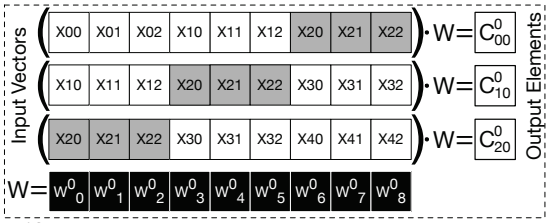


Figure 10: Convolution operations. X_{i0}, X_{i1}, and X_{i2} are input elements in the i^{th} row of the input feature map.

5.2.1 Convolution Layer

As shown in Figure 9, the first layer convolves the input using two set of weights ($Weight^0$ and $Weight^1$) and produces two output feature maps (C^0 and C^1). The convolution operation can be expressed as a vector dot product between input elements and corresponding weights. The following operations are performed to generate the $\{0,0\}^{th}$ element of output feature map C^0.

$$C^0_{00} = Input_{00} \,.\, Weight^0$$
$$Input_{00} = [X_{00}, X_{01}, X_{02}, X_{10}, X_{11}, X_{12}, X_{20}, X_{21}, X_{22}]$$
$$Weight^0 = [W^0_0, W^0_1, W^0_2, W^0_3, W^0_4, W^0_5, W^0_6, W^0_7, W^0_8]$$

Dedicated buffer for convolution weights. To produce each output element in C^0, we require $Weight^0$. We minimize the overhead of accessing weights from the memory by using a convolution weight buffer in the PU that stores all the required weights and is shared across the PEs of the PU.

Parallelism across output elements. Convolution operations within an output feature map have no data dependencies on each other, and can be executed in parallel (e.g. C^0_{01}, C^0_{00}, etc.). These parallel calculations are performed by the PEs within a PU.

Saving partial results to minimize data communication. As shown in Figure 10, the convolution operations that generate the three outputs C^0_{00}, C^0_{10}, and C^0_{20} require accesses to the same inputs three times. These input elements $\{X_{20}, X_{21}, X_{22}\}$ are highlighted in gray in Figure 10. To reduce these redundant accesses, the PEs read the input row by row and generate partial results. The PEs then store the partial results in a local FIFO as depicted in Figure 6. When the PEs read the next set of the input elements, they also dequeue the partial results from the previous row and calculate the next set of partial results.

Data forwarding. Figure 11 shows another optimization that reduces remote data transfer through re-use. Convolution windows that produce adjacent outputs share input elements. Therefore, PEs computing adjacent output elements use partially shared data. We add a dedicated unidirectional link between adjacent PEs to forward these shared input elements. The arrows in Figure 11 show this data forwarding to re-use data for convolution. The unique data read accesses for each PE are highlighted in gray.

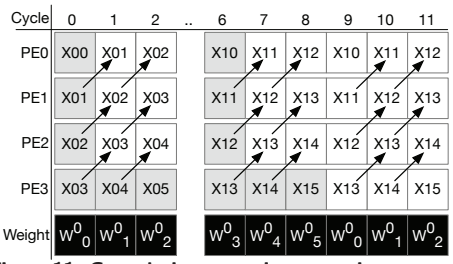


Figure 11: Convolution operation execution pattern.

Reusing data across convolution kernels. Using the sequence of operations in Figure 11, the four PEs compute four adjacent output elements using the first kernel ($Weight^0$).

5.2.2 *Pooling Layer*

The example uses a window of size 2×2 and a stride of two for max pooling. The input feature map for the pooling layer is divided into 16 non-overlapping windows of size 2×2, each corresponding to an output element. As Figure 7 illustrates, to compute pooling output element P^0_{00}, the unit require C^0_{00},C^0_{01},C^0_{10},C^0_{11}. Since the convolution layer produces adjacent elements in a row, the unit first compute max(C^0_{00}, C^0_{01}) for the first row and push it to the FIFO. When the second row is available, the FIFO is popped and we compute max(max(C^0_{00}, C^0_{01}), C^0_{10}, C^0_{11}).

Hiding latency. To hide execution latency, the pooling module overlaps its operations with the convolution operations in PEs. Since the kernel size for convolution in the previous layer is 3×3, the four PEs in the PU generate four output elements every nine cycles. As shown in Figure 7, the convolved output elements are sent to the shared pooling module with a single 3:1 comparator.

5.2.3 *Inner Product layer*

Inner product layer can be expressed as a vector-matrix multiplication. In the running example, the pooling layer produces two outputs, P^0 and P^1 of size 4×4 each or 32 elements in total. P^0 and P^1 are flattened and concatenated to generate a 1×32 input vector that is multiplied with the weight matrix of size 32×8 to generate the output vector of size 1×8, shown in Figure 9.

Parallelism across output elements. In inner product layer, each output neuron Y_j is generated as $Y_j = \sum_i X_i * W_{ij}$, where X_is are the inputs to the layer and W_{ij}s are the weights. To exploit the parallelism between output computations, each output neuron Y_j is assigned and computed on a single PE using a series of multiply-accumulate operations. With four PEs, the PU simultaneously calculates outputs Y_j in groups of four starting from $\{Y_0, Y_1, Y_2, Y_3\}$ as shown in Figure 8.

5.2.4 *Normalization and Activation*

As Figure 5 depicts, a part of normalization (sum of squares) uses the convolution hardware and the other part (scaling) is implemented as a separate unit. The activation transfer functions are implemented using lookup tables in each PU.

The Design Planner exploits the FPGA's reconfigurability by customizing the described template architecture for the target FPGA and DNN model as discussed in the following section.

6 Design Planner

The template architecture described in the previous section serves as a scalable template for the accelerator's microarchitecture. DNNWEAVER takes advantage of the FPGA's reconfigurability using Template Resource Optimization, a heuristic search algorithm, that co-optimizes both the accelerator architecture and the corresponding execution schedule to minimize off-chip accesses and maximize performance. The two key factors affecting performance are: (1) the allocation of compute and memory resources to the various components in the template architecture, which determines the degrees of parallelism in the accelerator; and (2) the schedule of operations on the accelerator, which determines the required external memory accesses. This algorithm aims to strike a balance between parallel operations and data reuse and configuring the accelerator to best match the constraints of the FPGA (on-chip memory and memory interface bandwidth). As the memory requirement of DNNs is typically much higher than the on-chip storage available on FPGAs, we divide the output feature map of each layer into slices. A slice is a portion of the output feature map that is spilled to memory after computation. Template Resource Optimization maximizes performance by varying (1) PEs-per-PU and (2) slice dimensions. Below, we discuss the two variables in further detail.

Variable (1) Number of PEs-per-PU. The template architecture exposes two levels of parallelism: (1) parallelism between PEs in a PU that generate adjacent output elements, and (2) parallelism between PUs generating independent output feature maps. Due to a fixed number of resources on the FPGA, increasing the number of PEs-per-PU would decrease the total number of PUs and vice versa. Our Template Resource Optimization algorithm aims to find a PEs-per-PU configuration that strikes a balance between the two degrees of parallelism and provides the best performance for the (DNN, FPGA) pair.

Variable (2) Output slice. The next variable is the slice of the output feature map that is computed within each epoch of the PU execution. This slice is the fraction of the output feature map that fits in the on-chip storage of the PU. The amount of this storage depends on the number of PEs[1] in each PU. The slicing dictates the number of external memory accesses and determines the degree of reuse in the computation. Convolution-like layers operate on overlapping input elements. Reusing these overlapping elements can only happen within a slice but not across slices. Therefore, the slice determines the degree of data reuse. The algorithm first tries to fit a row of the output in the on-chip memory of the PU. The stride is based on the number of PEs-per-PU to match the outputs with the PEs. If extra storage is still available, it tries to store more output rows. With this approach, the Design Planner picks a slice that maximizes data reuse and minimizes external memory accesses for each PU. By doing so, all the output slice computations can be done with local information in the PU. Another aspect of this approach is that the next layer can start operating on the slice before spilling it to memory. This optimization further reduces the off-chip memory accesses.

Template Resource Optimization search algorithm. Algorithm 1 illustrates our heuristic search which solves the following optimization objective:

$$arg\ \min\ executionCycles = \sum_l f(nPEperPU\ ,\ sliceSize_l)$$

$$\text{subject to}\quad 1 \le nPEperPU \le \text{FPGA.max\_PEs}$$
$$sliceSize_l \le min(BRAM \times nPEperPU\ ,\ output_{size})$$

Here, $nPEperPU$ is the number of PEs-per-PU and $sliceSize_l$ is the dimensions of the slice in layer l, FPGA.max_PEs is the maximum PEs that can the accommodated on the FPGA, and $output_{size}$ is size of a single channel in the output feature map.

[1]Note that each PE is assigned a BRAM.

Algorithm 1: Template Resource Optimization search.

Inputs : D: DNN Macro Dataflow Graph
F: FPGA Constraints
Output : $nPEperPU$: number of PEs per PU
$sliceSize_l$: Slice dimension in each layer
arg min : eec: $execution\_cycles$
Function `findSliceSize`(pe, F)
  Initialize $width_r = pe$ Initialize $height_c = 1$
  while $(width_r \times height_c \leq min\,(F.BRAM \times pe, output_{size})$ **do**
    $width_r = width_r + pe$
  end
  while $(width_r \times height_c \leq min\,(F.BRAM \times pe, output_{size})$ **do**
    $height_c = height_c + 1$
  end
  return $width_r$, $height_c$
end
Initialize $eec = \infty$; Initialize $l = D.numLayers$
Initialize $nPEperPU = 1$; Initialize $\forall\ sliceSize_l$ in $l = \phi$
for pe *in range* $(1, F.max_{PE})$ **do**
  Initialize $\forall\ ss_l$ in $l = \phi$
  for $\forall\ l$ *in* l **do**
    $ss_l =$ `findSliceSize`(pe, F)
  end
  $cycles = g(D, F, pe, ss_l)$
  if $(cycles < eec)$ **then**
    $eec = cycles$
    $nPEperPU = pe$
    $\forall\ l$ in L $sliceSize_l = ss_l$
  end
end

The algorithm takes in as input the DNN macro dataflow graph (D) and the constraints of the FPGA platform (F). The FPGA constraints (F) provide the maximum number of PEs and the capacity of the BRAM in each PE. The algorithm finally outputs the $nPEperPU$ and the $sliceSize_l$ by taking the following steps:

(1) Initialize. Initialize the number $nPEperPU$ and $sliceSize_l$ (for each layer). Initialize the estimated[2] execution cycles (eec) to ∞. The eec is an estimation of the number of cycles to execute a particular DNN with an organization that complies with $nPEperPU$ and $sliceSize$.

(2) Increment $nPEperPU$**.** Vary the $nPEperPU$ iteratively starting from 1 to the maximum number of PEs that can be synthesized on the FPGA platform.

(3) Calculate $sliceSize_l$**.** For the current choice of the $nPEperPU$, calculate the dimensions of the slice that fits in the PU. This calculation is done for each layer of the DNN.

(4) Estimate execution cycles. Estimate the execution cycles given the current choices of $nPEperPU$ and $sliceSize_l$.

(5) Reiterate or terminate. If the $cycles$ is less than eec, record the choices. Terminate if $nPEperPU$ has reached the maximum value, otherwise reiterate from step (2).

Figure 12 illustrates the result of search for Altera's Arria10 FPGA for AlexNet [30] and Overfeat [31]. We use Xeon E3 as the baseline to better visualize the trends. Performance for a layer in a DNN is highest when the output feature map size is a multiple of PEs-per-PU. Thus, the PEs-per-PU configuration that achieves best performance varies for the layers in a DNN, resulting in multiple peaks as shown in Figure 12. The peak performance occurs at 14 PEs-per-PU for AlexNet and 12 PEs-per-PU for Overfeat. The $sliceSize_l$s are different for each point of the graph.

[2]We have built a mathematical model g to estimate the execution cycles.

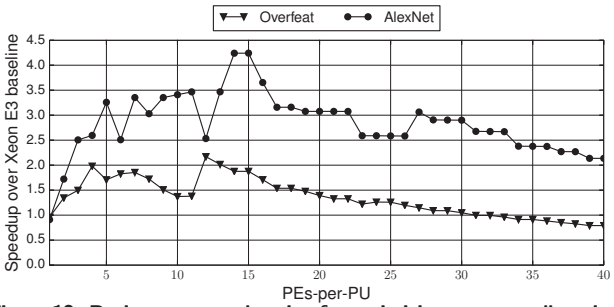


Figure 12: Design space exploration for optimizing resource allocation.

Table 2: FPGA Platform Details.

	Xilinx Zynq ZC702	Altera Stratix V SGSD5	Altera Arria 10 GX115
FPGA Capacity	53K LUTs	172K LUTs	427K LUTS
	106K Flip-Flops	690K Flip-Flops	1708K Flip-Flops
Peak Frequency	250 MHz	800 MHz	800 MHz
BRAM	630 KB	5035 KB	6782 KB
MACC Count	220	1590	1518
Evaluation Kit Price	$895	$6,995	$4495
Technology	TSMC 28nm	TSMC 28nm	TSMC 20nm

Table 3: Benchmark DNNs and their input datasets. The model size provides the size in Mega Bytes of the weights required for the network.

Network	Data set	Domain	Model Size (MegaBytes)
CIFAR-10 Full	CIFAR-10	Object Recognition	0.17 MB
LeNet	MNIST	Handwritten Digit Recognition	0.82 MB
NiN	ImageNet	Object detection and classification	14.50 MB
Djinn ASR	Kaldi	Speech-to-text decoder	48.40 MB
AlexNet	ImageNet	Object detection and classification	116.26 MB
VGG-CNN-S	ImageNet	Object detection and classification	196.26 MB
Overfeat	ImageNet	Object detection and classification	278.30 MB
VGG-16	ImageNet	Object detection and classification	323.87 MB

Table 4: Evaluated CPUs and GPUs.

Platform	Cores	Clock freq (GHz)	Memory (GB)	TDP (W)	Technology (nm)	Cost
ARM Cortex A15	4+1	2.300	2 (shared)	5	28	$191
Intel Xeon E3-1246 v3	4	3.600	16	84	22	$290
Tegra K1 GPU	192	0.852	2 (shared)	5	28	$191
NVIDIA GTX650Ti	768	0.928	1	110	28	$150
Tesla K40	2880	0.875	12	235	28	$2,599

7 Evaluation

To evaluate the effectiveness of DNNWEAVER, we use three off-the-shelf FPGA platforms, Xilinx Zynq ZC702, Altera Stratix V GS D5, and Altera Arria 10 GX-AX115. Table 2 summarizes their specifications. Henceforth, we refer to DNNWEAVER generated accelerator for Zynq, Stratix V, and Arria 10 as DW-Zynq, DW-Stratix, and DW-Arria, respectively.

7.1 Methodology

7.1.1 Benchmark DNNs and Their Input Datasets

Table 3 shows our benchmark DNN models, their input datasets, size of model parameters, and the number of multiply-accumulate operations required. The selected DNNs are used for various applications ranging from handwritten digit recognition, object recognition, to speech-to-text decoders. Among these, CIFAR-10 Full targets object detection in the CIFAR-10 thumbnail dataset [32]. LeNet targets the MNIST handwritten digit recognition dataset [33]. The DjiNN ASR network is a DNN speech-to-text decoder obtained from the DjiNN and Tonic benchmark suite [7]. NiN [34], AlexNet [30], Overfeat [31], VGG-CNN-S [35], and VGG-16 [35] target the ILSVRC ImageNet dataset [36].

7.1.2 CPU and GPU Execution

Table 4 lists the five evaluated CPU and GPU platforms.

Runtime measurements. We compare the execution time

of DNNWEAVER generated accelerators to the execution time on CPUs and GPUs using Berkeley Caffe. The CPU and GPU baselines are compiled with GCC 4.8 and NVCC 6.5, respectively. We obtain the baseline timings by using the timing feature of Caffe. For Arria 10 and Stratix V, we synthesize the accelerator using Qaurtus II v14.1 tool and use a cycle estimator to measure performance for the synthesized accelerator. Across all other platforms, we run each DNN 100 times and use the average.

Multi-threaded vectorized CPU execution. We use OpenBLAS for the BLAS backend required by Caffe to produce CPU-specific optimized binaries. Hence, we used Haswell-specific optimization for the Xeon E3 CPU, and the A15 optimization for the ARM A15 processor (Jetson TK1). The Haswell version of OpenBLAS uses AVX2 and Fused Multiply-Add (FMA) instructions whereas the A15 version uses the NEON SIMD engine. When evaluating the Xeon E3 CPU we used 4 threads, as we empirically found that this provided the best performance – enabling SMT affected the performance negatively. For the ARM A15 CPU we used 4 threads as well since it does not have SMT support.

Optimized GPU execution with cuDNN. For fastest GPU execution, Caffe can be configured to use the NVIDIA cuDNN library. We use the latest cuDNNv5 for GTX 650Ti and Tesla K40, and cuDNNv2 for Tegra K1, separately compiled for each GPU with architecture-specific compiler optimization. Tegra K1 does not support newer versions of cuDNN.

FPGA platforms details. In the Zynq board, the interface between DRAM and programmable logic is a standard AXI bus. In the Arria 10 and Stratix V boards, we used Altera's Avalon interface IP for interfacing the DRAM with the programmable logic. We implement a custom controller on the programmable logic to interface with the main memory. We synthesize the hardware with 64-bit Vivado v2015.1 for the Zynq board and Qaurtus II v14.1 for the Stratix V and Arria 10 boards. We use the synthesis tools to generate the area utilization numbers presented in Table 5. The frequency of operation of the accelerator on the Zynq board is 150 MHz and the frequency of operation on the Stratix V and Arria 10 FPGAs is 200 MHz.

Power measurements using vendor libraries. We employ a variety of strategies in order to gather power measurements for most tested platforms. We use the NVIDIA Management Library (NVML) to obtain the average power of Tesla K40. Given that GTX 650Ti does not support the NVML library, and since the GTX 650Ti and Tesla K40 share the same microarchitecture, we make a conservative estimation of the GTX 650Ti power by scaling the Tesla K40 measurements using the two chips' TDPs. For each DNN, we calculated the ratio of measured power in Tesla K40 over its TDP. We multiply this ratio with the GTX 650Ti TDP and use the 95% of this number.

We utilize the Intel Running Average Power Limit (RAPL) energy consumption counters available in the Linux kernel for power measurements on the Xeon E3.

Power measurements in hardware. The ARM A15 CPU and the Tegra K1 GPU are a part of the Jetson TK1 development board. Jetson TK1 does not provide software means to gather power readings. Therefore, we use the Keysight E3649A Programmable DC Power Supply to get the power consumption of the Jetson TK1 board. To do so, we subtract the idle average power 3.12W from the power reading we obtain during benchmark execution. For Arria 10 and Stratix V, we use the TDP as a measure of power consumed during execution. Finally, we use a GPIO to USB adapter to read the power directly from the power controllers in the Zynq board.

7.2 Comparison to High Level Synthesis

As an alternative to DNNWEAVER, HLS can also generate hardware implementations for DNNs, where the programmer uses HLS's C-like syntax to express layers of the DNN model. Although HLS provides a high level abstraction to programmers, optimizing the hardware implementation for a DNN model on a target FPGA requires expertise in both hardware design and the specific programming tool used for HLS. In our experiments, two Masters students with moderate amount of experience in hardware design spent one month to optimize a Vivado HLS implementation of the LeNet Benchmark for the Xilinx Zynq ZC702 FPGA. The resulting implementation ran at 100 MHz and provided a slow-down of 19.7× compared to DNNWEAVER generated accelerator for the same FPGA platform. The benefits of the template approach is more evident when considering a recent work [14] that uses commercial HLS tool and yet spends significant effort to implement the convolution layers of just one DNN, AlexNet. Moreover, another recent work [37] shows that using dataflow templates as an intermediate compilation target for high-level synthesis of C/C++ programs delivers 9× higher performance than the state-of-the-art HLS tools. Recall that for none of the FPGA acceleration, DNNWEAVER requires anything beyond just expressing the DNN in Caffe format.

7.3 Experimental Results

7.3.1 Performance Comparison with CPUs

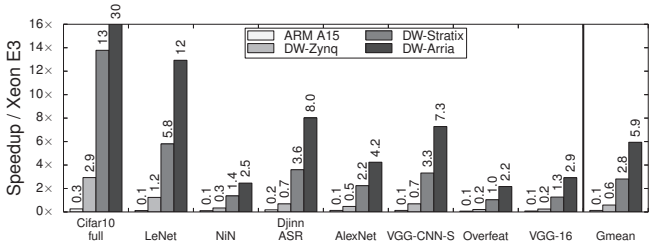
Speedup compared to Xeon E3. Figure 13a illustrates the performance benefits when the DW-Zynq, DW-Stratix, and DW-Arria are used to compute the models under evaluation. The performance of Xeon E3 for the eight DNN models using Caffe's framework is used as a baseline for comparison. The average speedup for DW-Zynq, DW-Stratix, and DW-Arria is 0.59×, 2.81×, and 5.94×, respectively; thus, DW-Arria provides 10× more speedup than DW-Zynq. Among the evaluated models, Cifar-10 Full sees the highest speedup of (2.9×, 13.8×, and 30.9×) while Overfeat shows the lowest speedup of (0.2×, 1.0×, and 2.2×) over DW-Zynq, DW-Stratix, and DW-Arria, respectively. The significant gap in performance benefits comes from the disparity in the model topology, some layers are more favorable to the DNNWEAVER generated accelerators than the others. We will further discuss the difference in per-layer computation efficiency later in Section 7.3.3.

Speedup compared to ARM A15. Figure 13a also shows the performance comparison with a low-end processor, ARM A15. ARM A15 exhibits an 8× slowdown with respect to Xeon E3. Compared to the low-end ARM A15 processor, DW-Zynq, DW-Stratix, and DW-Arria provide 4.7×, 22.39×, and 47.26× speedup respectively. As expected, the low power ARM A15, which is not intended for high performance computing is significantly outperformed by the Xeon E3 server class processor.

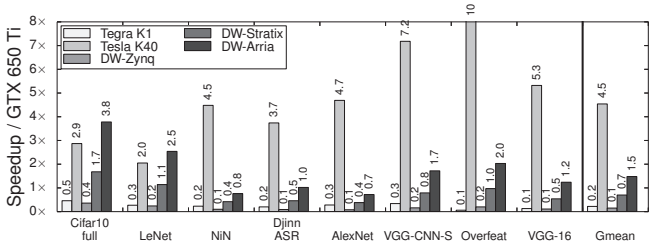
These results demonstrate the performance benefits provided by DNNWEAVER generated accelerators over both low-end and high-end CPUs, as well as their scalability over various FPGAs.

7.3.2 Performance Comparison with GPUs

Speedup compared to GTX 650Ti. We compare our accelerators with GPU platforms including GTX 650Ti, Tegra K1, and Tesla K40 in Figure 13b. The baseline is GTX 650Ti, a middle-tier GPU. DW-Zynq and DW-Stratix provide 6.60× and 1.4× slowdown compared to GTX 650Ti, while DW-Arria shows a 1.48× speedup. Tesla K40 provides a speedup of 4.5× over the baseline. For the three FPGAs, (DW-Zynq, DW-Stratix, and DW-Arria), maximum speedup



(a) Speedup and comparison with CPUs (baseline=Xeon E3)



(b) Speedup and comparison with GPUs (baseline=GTX 650Ti).

Figure 13: Speedup of DnnWeaver generated accelerators in comparison to a range of CPU and GPU platforms.

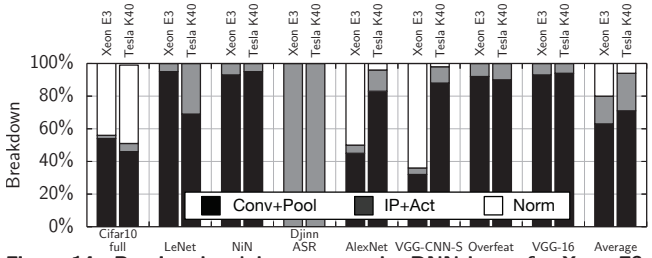


Figure 14: Runtime breakdown across the DNN layers for Xeon E3 and Tesla K40. (Conv: Convolution, Pool: Pooling, IP: Inner Product, Act: Activation, Norm: Normalization).

of (0.4×, 1.7×, and 3.8×) is observed from Cifar-10 Full, whereas AlexNet shows the minimum speedup of (0.1×, 0.4×, and 0.7×).

Speedup compared to Tegra K1 and Tesla K40. In Figure 13b, we also show a comparison with a low-end GPU, Tegra K1, and a high-end GPU, Tesla K40. The low-end Tegra K1 offers a 0.2× average speedup over the baseline. In contrast, the high-end GPU Tesla K40 presents a 4.5× speedup. In comparison with Tesla K40, (DW-Zynq, DW-Stratix, and DW-Arria) show an average speedup of (0.03×, 0.15×, and 0.33×), respectively.

7.3.3 Per-Layer Performance Benefits

To understand the per-layer efficiency of the DnnWeaver generated accelerators, we examine the runtime breakdown across the model layers and the speedup for individual layers.

Runtime breakdown. Figure 14 shows the runtime breakdown of the models computed from the Caffe framework using two baselines platforms, Xeon E3 and Tesla K40. For convenience, we combine (1) Convolution and Pooling layers (Conv+Pool), (2) Inner Product and Activation layers (IP+Act), and (3) Normalization layer (Norm). On average, Conv+Pool occupies 68% and 74% of the computation runtime, while the IP+Act occupies 16% and 21% when run on Xeon E3 and Tesla K40, respectively. The larger proportion of execution time is spent on Conv+Pool than the other layers as the convolution layer has significantly higher number of operations than the inner product layer.

The composition of runtime varies between the network models depending on the network topology and layer sizes. Cifar10 Full, AlexNet, and VGG-CNN-S are the only networks that include a normalization layer, which is executed for 16% and 5% of the runtime on Xeon E3 and Tesla K40, respectively. With the exception of Djinn ASR, which consists of just IP+Act, most of the time in the rest of the benchmarks is spent on Conv+Pool.

Per-layer speedup. Figure 15 shows the per-layer speedup of the DnnWeaver generated accelerators compared to the baseline software execution with Xeon E3 (Figure 15a) and Tesla K40 (Figure 15b). As shown in Figure 15a, for the set of (DW-Zynq, DW-Stratix, DW-Arria), the average speedup for Conv+Pool, IP+Act, and Norm in comparison with Xeon E3 is (0.5×, 2.2×, and 4.6×), (2.3×, 12.5×, and 28.7×), and (68.5×, 139×, and 393×), respectively. Norm shows high speedup, particularly over CPUs, as the non-linear operations within normalization are implemented efficiently in FPGAs using lookup tables. Norm is a significant portion of the runtime for Cifar10 Full and VGG-CNN-S, leading to a high speedup for these models. Overfeat's runtime is dominated by Conv+Pool, which presents the lowest speedup. Similarly, Figure 15b shows the per-layer speedup with the baseline of GTX 650Ti. For Norm and IP+Act, DW-Stratix outperforms the baseline, while it closely follows the baseline performance for Conv+Pool. DW-Arria also follows the same trend and the speedup for IP+Act is higher than that for Norm.

7.3.4 Sensitivity to on-chip storage

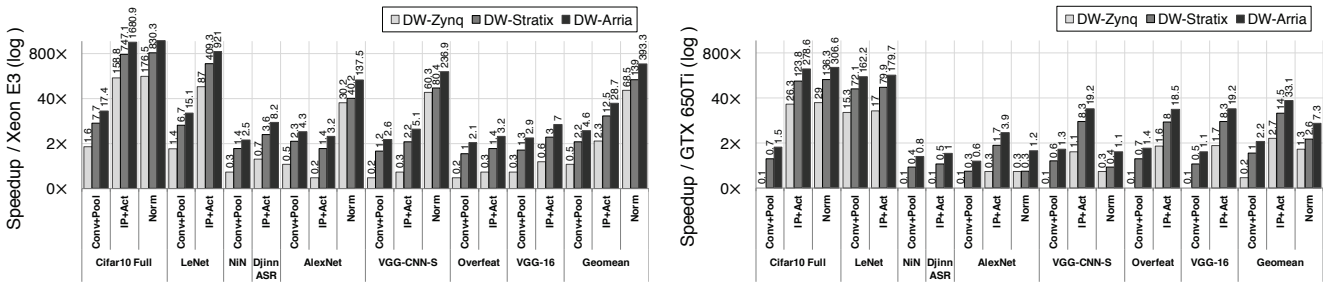
Figure 16 illustrates the impact of limited on-chip storage over performance on the Arria 10 board. We use a cycle accurate simulator, which we validate against hardware measurements, to generate the figure. The figure compares the performance of DnnWeaver generated accelerator to Xeon E3 CPU baseline when varying the size of on-chip BRAM resource from 1× to 256× the available. The impact of memory size is most prominent for inner product layers, since the inner product weights are significantly large, and seldom fit in on-chip memory. Storing inner product weights off-chip reduces the accelerator performance due to the overhead of external memory accesses. As illustrated in Figure 16, the amount of on-chip storage required to store the inner product weights and overcome this memory wall is more than 16× the available storage.

We reduce this overhead by sharing inner product weights over a batch of inputs. Figure 16 compares the performance of the generated accelerator with and without batching. The speedup from batching is most prominent in Djinn ASR, where the model consists of back-to-back inner product and activation layers. On the other hand, the performance of NiN is unaffected by batching as it does not contain an inner product layer. The benchmarks AlexNet, Overfeat, and VGG-16 observe a 2.2× increase in performance through batching. Benchmarks VGG-CNN-S, LeNet and Cifar10 Full observe a similar trend.

7.3.5 Performance-per-Watt Comparison with CPUs

As shown in the speedup results, the performance benefits from diverse CPU, GPU, and FPGA platforms vary substantially. In fact, these hardware platforms occupy different design points in the underlying performance vs. energy efficiency tradeoff. To understand the performance benefits with the fixed energy efficiency, we measure the power consumption and evaluate the performance-per-Watt for each hardware platform.

Comparison with Xeon E3. Figure 17a delineates the comparison of performance-per-Watt for ARM A15, DW-Zynq, and DW-Arria with the baseline of Xeon E3. On average, DW-Zynq shows 16.6×, DW-Stratix shows 7.9×, and DW-Arria shows



(a) Speedup for each DNN layer with the baseline of Xeon E3.

(b) Speedup for each DNN layer with the baseline of GTX 650Ti.

Figure 15: Per-layer speedup when accelerated with DW-Zynq and DW-Arria.

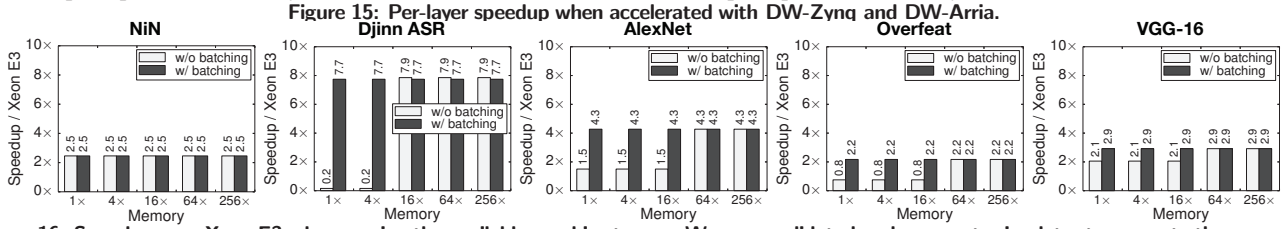
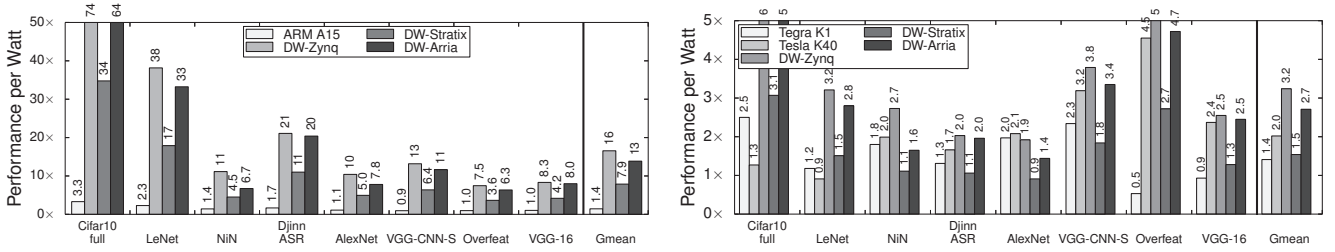


Figure 16: Speedup over Xeon E3 when varying the available on-chip storage. We use a validated cycle-accurate simulator to generate these results.



(a) CPU Performance-per-Watt Comparison (Baseline=Xeon E3)

(b) GPU Performance-per-Watt Comparison (Baseline=GTX 650Ti)

Figure 17: Performance-per-Watt of the DnnWeaver generated accelerators in comparison to a range of CPU and GPU platforms.

13.9× higher performance-per-Watt than the baseline. Note that although DW-Stratix provides about 10× higher speedup, the increased power consumption by DW-Stratix (2W vs. 25W) leads to the lower performance-per-Watt than DW-Zynq. However, DW-Arria provides higher performance that DW-Stratix, without a commensurate increase in power consumption, leading to higher performance-per-Watt. This trend is observed for all the evaluated DNN models.

Comparison with ARM A15. Low-end processors such as ARM A15 are commonly used in mobile devices and are known to have high energy-efficiency. We also compare the ARM A15 processor with our accelerators and Xeon E3. The ARM A15 processor shows 1.4× higher performance-per-Watt compared to Xeon E3. When compared with ARM A15, DW-Zynq, DW-Stratix, and DW-Arria show 11.5×, 5.5×, and 9.6× higher performance-per-Watt, which demonstrates the energy efficiency of the DnnWeaver generated accelerators.

7.3.6 Performance-per-Watt Comparison with GPUs

Comparison with GTX 650Ti. Figure 17b shows the performance-per-Watt in comparison of Tegra K1, Tesla K40, DW-Zynq, and DW-Arria with the baseline, GTX 650Ti. The pair of (DW-Zynq, DW-Stratix, and DW-Arria) provides (3.2×, 1.5×, and 2.7×) higher performance-per-Watt than the baseline. Although DW-Arria outperforms DW-Zynq with the speedup of 10× shown in Figure 13b, DW-Zynq offers a 1.2× higher performance-per-Watt compared to DW-Arria.

Comparison with Tegra K1 and Tesla K40. Figure 17b also compares the performance-per-Watt of Tegra K1 and Tesla K40 with the baseline. On average, Tegra K1 and Tesla K40 have 1.4× and 2.0× higher performance-per-Watt than GTX 650Ti.

7.3.7 Area and FPGA Utilization

Table 6 shows the framework determined number of PUs and the number of PEs-per-PU for DW-Zynq, DW-Stratix, and DW-Arria. The resource utilization in DW-Stratix is limited by the LUTs available on chip, and the resource utilization in DW-Zynq is bounded by the number of BRAM blocks available on-chip. Table 5 shows the resource utilization to generate the DnnWeaver accelerators for each DNN model.

8 Related Work

There have been several proposed and realized hardware designs that accelerate machine learning algorithms and DNNs. However, this work differs from other efforts in that DnnWeaver is not an accelerator, but an accelerator generator. Our work produces an optimized design for a given (DNN, FPGA) pair. Furthermore, DnnWeaver provides a novel ISA to unify DNN accelerators across different FPGA platforms. In this section we discuss the most related work in the area of FPGA implementations and ASIC accelerators for DNNs.

FPGA implementations for machine learning. Tabla [22] provides an FPGA accelerator generator for the training phase of statistical machine learning algorithms. However, DnnWeaver focuses on inference with DNNs. In addition, Tabla uses stochastic gradient descent as the abstraction between hardware and software, and has no notion of ISA or Deep Neural Networks. Tabla provides its own mathematical language, while DnnWeaver uses Berkeley Caffe for model specification.

The work by Chen, et al. [14] focuses on using an analytical

Table 5: Resource utilization on the three FPGA platforms for each benchmark DNN.

Benchmark DNN	Xilinx Zynq ZC702				Altera Stratix V SGSD5				Altera Arria 10 GX115			
	LUTs (Total: 53200)	BRAM (Bytes) (Total: 630KB)	Flip-Flops (Total: 106400)	DSP Slices (Total : 220)	LUTs (Total:	BRAM (Bytes) (Total: 5035KB)	Flip-Flops (Total: 690000)	DSP Slices (Total: 1590)	LUTs (Total:	BRAM (Bytes) (Total: 6782KB)	Flip-Flops (Total: 1708800)	DSP Slices (Total: 1518)
	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization	Utilization
Cifar-10 Full	61.44%	98.57%	30.77%	61.82%	85.29%	95.33%	46.90%	37.74%	84.99%	84.92%	45.85%	94.86%
Djinn ASR	42.43%	100.00%	18.25%	63.64%	53.98%	85.80%	28.12%	36.23%	68.96%	94.36%	39.27%	98.81%
LeNet	47.57%	100.00%	21.90%	61.82%	66.53%	80.44%	32.83%	33.96%	84.99%	84.92%	45.85%	94.86%
VGG_CNN_S	62.22%	97.14%	29.73%	61.82%	88.07%	78.50%	50.81%	37.04%	84.64%	89.64%	47.59%	88.54%
VGG_16	65.92%	100.00%	31.23%	63.64%	87.65%	78.20%	50.68%	37.42%	84.64%	89.64%	47.59%	88.54%
AlexNet	64.56%	100.00%	30.78%	63.64%	86.70%	77.16%	50.04%	37.23%	82.19%	86.69%	46.24%	88.54%
NiN	68.59%	100.00%	34.62%	63.64%	86.70%	77.16%	50.04%	37.23%	84.64%	89.64%	47.59%	88.54%
Overfeat	61.52%	94.29%	29.28%	60.00%	84.68%	75.07%	48.79%	36.98%	85.57%	86.25%	48.19%	88.93%

Table 6: Total number of PUs and the number of PEs per PU built on the three FPGA platforms for each benchmark DNN.

Benchmark	Xilinx Zynq ZC702		Altera Stratix V SGSD5		Altera Arria 10 GX115	
	# of PUs	# of PEs	# of PUs	# of PEs	# of PUs	# of PEs
Cifar10 full	8	17	8	60	8	135
Djinn ASR	7	20	23	24	23	54
MNIST LeNet	8	17	8	60	8	135
VGG_CNN_S	17	8	17	31	19	64
VGG_16	10	14	15	35	19	64
AlexNet	14	10	14	37	14	84
NiN	7	20	14	37	19	64
Overfeat	12	11	12	42	13	90

design scheme based on the roofline model to find the fastest design for a particular DNN for FPGA acceleration. However, their design does not support some DNN layers such as pooling and normalization. The work by Farabet, et al. [15, 28] develops an FPGA accelerator for a specific DNN. Gokhale, et al. [27] propose a mobile co-processor for DNNs and evaluate it on a Zynq board. Chakradhar, et al. [38] present a VLIW co-processor for DNNs and emulate it on a Virtex 5 FPGA. They propose a special switch that allows to dynamically group the convolution engines in different ways. The design has a low-level VLIW ISA but the paper does not include any details about its design. Unlike DNNWEAVER, they do not generate Verilog code for FPGA accelerators. The works by Qiu et al. and Suda et al. [39, 40] present implementations of accelerators for particular DNN models. Neither of these works support generation of accelerators for arbitrary DNN topologies.

DNNWEAVER makes FPGAs accessible to the machine learning community by automatically generating an optimized accelerator from high level DNN specifications. On the other hand, previous works come short of providing at least one of the following features: optimized accelerator generation, ISA support, a workflow starting from high level abstractions.

ASIC accelerators for DNNs. Recent research efforts present low-power deep learning ASICs. For example, (Da)Diannao [12, 13] provide DNN accelerators with a low-level fine-grained ISA, yet they do not define an ISA to unify DNN accelerators. In contrast, DNNWEAVER uses a ISA for deep neural networks (DNN) representing high-level operations (layers) that provides the flexibility necessary to optimize the accelerator microarchitecture for the FPGA platform and DNN model. Sim, et al. [24] showcase a DNN ASIC for IoT devices. However, the article doesn't make a reference to classification layer support. Qadeer, et al. [25] propose Convolution Engine which reduces the number of operations required in convolution layers. Conti, et al. [26] develop convolution cores designed to integrate with a shared-memory cluster of RISC processors. PuDianNao [20] is an ASIC accelerator for machine learning algorithms but lacks deep convolutional networks support.

All of these previous efforts require ASIC design, not FPGA realization, which is the focus of our work.

Concurrent submissions. Hardware implementation for DNNs is a thriving and active area of research. The following efforts have been published concurrently to our work. Wang, et al. [41] use a library of fixed-function blocks to accelerate DNNs on Xilinx Z7020 and Z7045 FPGAs. Unlike the PEs in DNNWEAVER, the architecture in their work lack explicit data sharing. Liu, et al. [42] propose an ISA for neural networks optimized for high code density over vector and matrix operations. Chen, et al. [23, 43] develop an ASIC design with a 2D spatial array of PEs for Convolutional Neural Networks. Song, et al. [44] propose an ASIC implementation with adaptive data-level parallelism for DNN accelerators. EIE [45], Minerva [46], and Cnvlutin [47] propose ASIC accelerators that use operation pruning and quantization in DNNs for power and performance benefits.

9 Conclusion

Deep Neural Networks are gaining increasing applicability and are amongst the most important workloads that can significantly benefit from acceleration. However, DNNs are in a state of flux and new disruptive advances require hardware solutions that can adapt to these changes. DNNWEAVER is an initial step in providing such solutions that support a wide variety of DNN models and can be further extended for more advanced models. While GPUs serve as an attractive platform for DNNs, our results shows that FPGAs can be a Pareto optimal choice when power is constraining. Nonetheless, reducing the programmer involvement in hardware design is imperative to the adoption of FPGAs in this domain. To this end, DNNWEAVER converts high-level specification of DNNs into highly efficient accelerators that operate within a limited power budget and on-chip memory of the FPGA. The conversion is made possible by a novel dataflow ISA and a heuristic search algorithm that generates high performance accelerator by customizing the hand-optimized template designs for a given (DNN, FPGA) pair. DNNWEAVER takes an effective step in making FPGAs available to a broader community of DNN developers who often do not possess hardware design expertise. Community engagement and contribution are vital for providing a general platform for DNN acceleration. To facilitate such engagement, DNNWEAVER has been made publicly available at `http://act-lab.org/artifacts/dnnweaver`.

10 Acknowledgements

We thank the anonymous reviewers for their insightful comments and feedback. We thank Bradley Thwaites, Manan Chugh, Sushant Kumar Singh, and Payal Bagga. This work was supported in part by NSF awards CCF #1553192 and CNS #1526211, Semiconductor Research Corporation contract #2015-TS-2636, and gifts from Google, Qualcomm, and Microsoft.

References

[1] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[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] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[7] 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.

[8] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876*, 2015.

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

[10] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–Aug. 2011.

[12] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[13] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[14] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.

[15] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *CVPRW*, 2011.

[16] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.

[17] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA*, 2011.

[18] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO*, 2011.

[19] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, 2014.

[20] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ASPLOS*, 2015.

[21] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *ISCA*, 2015.

[22] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, pages 14–26. IEEE, 2016.

[23] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *ISSCC*, 2016.

[24] J. Sim, J. S. Park, M. Kim, D. Bae, Y. Choi, and L. S. Kim. 14.6 a 1.42tops/w deep convolutional neural network recognition processor for intelligent ioe systems. In *ISSCC*, 2016.

[25] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 24–35. ACM, 2013.

[26] Francesco Conti and Luca Benini. A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters. In *DATE*, 2015.

[27] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *CVPRW*, 2014.

[28] Clément Farabet, Berin Martini, Polina Akselrod, Selçuk Talay, Yann LeCun, and Eugenio Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *ISCAS*, 2010.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[30] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[31] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

[32] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.

[33] Yann LeCun and Corinna Cortes. MNIST handwritten digit database, 2010.

[34] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013.

[35] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. In *British Machine Vision Conference*, 2014.

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 115(3):211–252, 2015.

[37] Shaoyi Cheng and John Wawrzynek. High level synthesis with a dataflow architectural template. *CoRR*, abs/1606.06451, 2016.

[38] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 247–257. ACM, 2010.

[39] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *FPGA*, 2016.

[40] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*, 2016.

[41] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 110:1–110:6, New York, NY, USA, 2016. ACM.

[42] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *ISCA*, 2016.

[43] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. 2016.

[44] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 123:1–123:6, New York, NY, USA, 2016. ACM.

[45] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528, 2016.

[46] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose Miguel Hernandez-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. *CoRR*, abs/1602.01528, 2016.

[47] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. 2016.