TensorFlow to Cloud FPGAs: Tradeoffs for Accelerating Deep Neural Networks

Stefan Hadjis and Kunle Olukotun
Department of Computer Science, Stanford University, Stanford, CA, USA
Email: shadjis@stanford.edu

***Abstract*—We present the first open-source TensorFlow to FPGA tool capable of running state-of-the-art DNNs. Running TensorFlow on the Amazon cloud FPGA instances, we provide competitive performance and higher accuracy compared to a proprietary tool, thus providing a public framework for research exploration in the DNN inference space. We also detail the optimizations needed to map modern DNN frameworks to FPGAs, provide novel analysis of design tradeoffs for FPGA DNN accelerators and present experiments across a range of DNNs.**

I. Introduction

Deep Neural Networks (DNNs) provide state-of-the-art results in many industries [1]. Many companies have turned to custom hardware accelerators for DNN processing to achieve improved throughput, latency and power compared to GPUs and CPUs [2], [3]. In particular, programmable accelerators like FPGAs are useful because computations vary across DNNs and algorithms often change. However, designing programmable accelerators requires a long development process, and DNN accelerators pose unique challenges due to the size of their design space and the complexity of modern DNNs.

The first challenge is that designing an accelerator for a given DNN requires a number of choices, ranging from accelerator architecture to memory management. In addition, design decisions change based on properties of the DNN, such as layer types and data structure sizes. Exploring architectures to optimize for these considerations involves a large design space and complex implementation process.

In addition, due to their complexity, DNN applications are almost always developed using high-level frameworks. TensorFlow [4] from Google is the most popular. As of 2019 it is the top machine learning framework on GitHub.com. This leads to a second challenge, because efficiently running a DNN expressed at a high-level on a low-level programmable hardware target requires optimization at many levels of abstraction.

To solve these problems, we develop an end-to-end toolchain to go from high-level DNN models to low-level hardware. The input is a TensorFlow model and the output is an optimized FPGA design. Formats from other frameworks also can be swapped in. Our toolchain supports FPGAs from multiple vendors, but here we focus on the recently announced, publicly available Amazon cloud FPGAs [5], [6].

This solves both problems above: it allows easily experimenting with architectures and algorithms to explore large design spaces, and it performs the required optimizations at each level of the stack so DNNs expressed in a high-level framework can be efficiently deployed to hardware. We hope that our toolchain, which allows DNNs to be expressed in modern formats and targets public hardware, can help the FPGA community research DNN accelerators. The code is linked below[1]. We make the following contributions:

1) We study the optimizations needed to start from a modern DNN framework and efficiently map to FPGAs. This includes DNN optimizations overlooked in the FPGA literature and recommendations to improve existing tools.

2) We provide novel experimental analysis of tradeoffs in the FPGA DNN accelerator design space, providing insights into generating efficient DNN FPGA hardware.

3) We provide an open, end-to-end toolchain to accelerate TensorFlow DNNs on FPGAs. It is capable of running on publicly-available cloud FPGAs and compiles DNNs achieving state-of-the-art accuracy (the first open-source TensorFlow to FPGA compiler to do either of these). It also automatically performs state-of-the-art optimizations from the literature, allowing it to be used as a research tool for design exploration.

Next we describe the space when designing an FPGA accelerator for a given DNN, and how design choices are impacted by various factors.

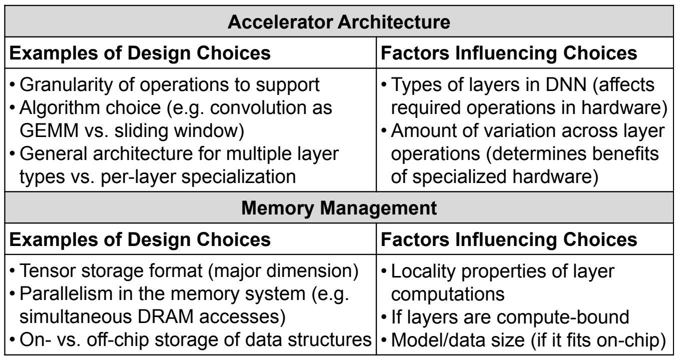
II. Tradeoff Space for DNN Accelerators

Given a DNN to accelerate, this section discusses accelerator design choices and factors influencing them. DNNs are composed of layers arranged in a dataflow graph. The number of layers can range from a dozen to hundreds. Example layers include convolution and pooling [1]. Each layer transforms an input tensor (multi-dimensional data array) to an output tensor. These tensors of data are also called “activations”. Some layers also contain model parameters (sometimes called weights or kernels) which are read-only, e.g. convolution filters.

For clarity we partition the design space into three components: architecture, memory management and accuracy tradeoffs. In this work we focus on the first two. Accuracy tradeoffs include approximations like lowering precision or changing the DNN model, which may give speedups but also impact correctness. We discuss precision in Section 5, but otherwise this work focuses on accelerating a DNN as-given.

Fig. 1 describes the design space related to architecture and memory management, as well as factors which impact design choices. Architecture design determines the coarse-grained computations the architecture must support (e.g. which DNN layers) as well as core computational units of the accelerator’s processing elements, for instance whether convolutions

[1]https://github.com/stanford-ppl/spatial-multiverse



Accelerator Architecture	
Examples of Design Choices	**Factors Influencing Choices**
• Granularity of operations to support • Algorithm choice (e.g. convolution as GEMM vs. sliding window) • General architecture for multiple layer types vs. per-layer specialization	• Types of layers in DNN (affects required operations in hardware) • Amount of variation across layer operations (determines benefits of specialized hardware)
Memory Management	
Examples of Design Choices	**Factors Influencing Choices**
• Tensor storage format (major dimension) • Parallelism in the memory system (e.g. simultaneous DRAM accesses) • On- vs. off-chip storage of data structures	• Locality properties of layer computations • If layers are compute-bound • Model/data size (if it fits on-chip)

Fig. 1. An overview of common choices when designing DNN accelerators as well as factors influencing these choices.

should be performed as dot products or a sliding window (stencil). Another decision is whether different types of DNN layers should be implemented by specialized processors or whether multiple layer types should share a single, more general processor. Whereas an ASIC DNN accelerator must support a variety of DNNs [7], the reconfigurability of FPGAs allows accelerator designs to be specialized for a specific DNN. Therefore the number of DNN layers, the types of computations they perform and the degree of layer variability impact architecture design choices.

Memory management choices include where data structures should be stored in memory, the formatting of these data structures in memory and the amount of parallelism in the memory system to eliminate memory bottlenecks. Different DNN layers have different locality properties and arithmetic intensity [2],therefore the types of DNN layers as well as the sizes of their data structures also impact design choices.

The space is large, and our next step is to design an end-to-end toolchain which can explore this design space starting from a high level of abstraction.

III. Overview of Compiler

Our compiler allows DNNs developed in high-level frameworks to be efficiently deployed to FPGA hardware, and performs the required optimizations at each level of the stack. It contains three Optimization Levels, shown in Fig. 2.

A. Level 1: DNN-Specific Optimizations

Level 1 processes the DNN graph to perform optimizations that will later help hardware generation. DNN frameworks represent a DNN as a data-flow graph (DFG), where nodes are operations (e.g. Convolution) and edges are data tensors passed between them. TensorFlow and other modern frameworks (e.g. ONNX) contain utility tools to perform graph optimizations, e.g. constant folding. However, it is often not obvious to the DNN framework that certain optimizations apply and the burden is placed on the user to perform graph processing.

Table I shows an example of how various optimizations simplify the TensorFlow DFG for the popular ResNet-50 [8], a state-of-the-art CNN. Each row corresponds to an optimization run by our script. The first row is the downloaded model after using TensorFlow's API to create the inference graph, which freezes variables to constants and removes training nodes.

TABLE I
Optimizations on the ResNet-50 TensorFlow graph

Description	# DFG Nodes	Needs RSqrt	Needs Scale
Freeze for Inference	460	Yes	Yes
Fold Constants	354	No	Yes
Fold BatchNorms	301	No	No

Next we run optimization passes. These are not automatically performed by TensorFlow (or ONNX) when freezing for inference so they need to be run separately. In the ResNet example, two optimizations help for custom hardware. Both are related to constant folding and regard the layer pattern of Convolution followed by Batch Normalization (BN), which occurs 53×. When converting variables to constants, folding opportunities arise for operations which differ in the forward pass of training vs. inference, such as Dropout or BN.

The first optimization is noted by prior FPGA work [9], [10]. It eliminates reciprocal square roots and other operations on tensors of constants: BN computes statistics on the data but during inference these are constants and can fold into a linear function. The result is shown in the second row of Table I.

The second optimization however is overlooked by prior FPGA work. It eliminates the tensor scaling operation of BN by folding the constant scaling factors of BN into the Convolution before the BN. This updates the already-trained Convolution weights to incorporate the BN scaling factors, and replaces BN with a simpler bias addition which requires no multiplication. The third row shows this final result. Prior FPGA work computes both the scaling and addition components of BN, e.g. recent work still remarks how BN needs "multipliers that are expensive" [11]. We therefore suggest FPGA tools move to modern frameworks to get these benefits.

Finally, optimization utilities can fail for DNNs with control paths, e.g. when selecting operations that differ from training to inference. In such cases our scripts also perform a traversal to first fold constant branch inputs through control nodes (Switch, Merge) and eliminate unused subgraphs.

B. Level 2: Optimizations for DNN Hardware Accelerators

Level 2 contains the majority of the optimization and is the focus of Section 4. It converts the optimized DNN graph from Level 1 into a Hardware IR describing the circuit. This specifies the exact architecture layout of the DNN on-chip.

C. Level 3: DNN-agnostic, Target-specific Optimizations

Level 3 compiles the Hardware IR into synthesizable Verilog and makes optimizations for the target FPGA. Our compiler uses the open-source High-Level Design language Spatial [12] as the Hardware IR. It provides hardware-specific abstractions, i.e. it is more like a high-level HDL than a high-level synthesis (HLS) tool. We chose Spatial because these abstractions allow fast prototyping without sacrificing knowledge of the synthesized circuit, e.g. by allowing users to explicitly control memory hierarchy (DRAM, SRAM, FFs). While a strength of HLS is that it allows software engineers to make fast progress, for our toolchain the user's entry point is already TensorFlow. We therefore opted for an IR which gives more control over the underlying circuit.

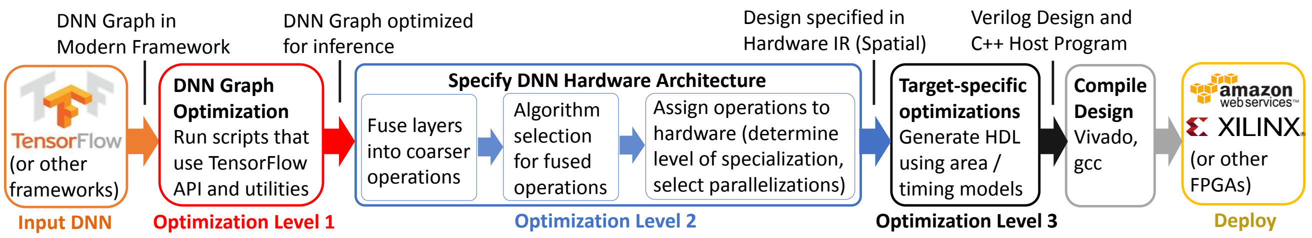

DNN Graph in Modern Framework
DNN Graph optimized for inference
Design specified in Hardware IR (Spatial)
Verilog Design and C++ Host Program
TensorFlow (or other frameworks)
DNN Graph Optimization
Run scripts that use TensorFlow API and utilities
Specify DNN Hardware Architecture
Fuse layers into coarser operations
Algorithm selection for fused operations
Assign operations to hardware (determine level of specialization, select parallelizations)
Target-specific optimizations
Generate HDL using area / timing models
Compile Design
Vivado, gcc
amazon web services
XILINX
(or other FPGAs)
Input DNN
Optimization Level 1
Optimization Level 2
Optimization Level 3
Deploy

Fig. 2. A detailed view of the compiler flow, showing optimizations at each level of the stack.

At the same time, because the goal is to explore algorithms and architectures, describing hardware at a higher level than Verilog is helpful. Spatial facilitates this by performing optimizations like pipeline scheduling to exploit parallelism and memory banking to reduce on-chip memory usage. It also performs automated tuning of parameters such as operator latencies based on FPGA-specific timing models. Finally, it is open-source and supports FPGAs from multiple vendors, including being one of few tools to support the Amazon cloud FPGAs. The output of Spatial is a Verilog design for the FPGA and a C++ program which runs the design from the host CPU.

Our DNNs were larger than previous Spatial applications however, so modifications were needed to improve performance. First, we modified Spatial's interface between the application and the Amazon-provided shell to delete unused IPs, reducing LUT and BRAM utilization by 8% and 19% of the total available. Second, we wrote an analysis pass to assign SRAM data structures with size $>$ 1024 words to UltraRAMs (URAMs), which are deeper block RAMs on Xilinx UltraScale+ FPGAs that were unused by Spatial. Third, a challenge in designing CNN hardware is that irregular size parameters are ubiquitous (e.g. 3×3 convolution kernels, 7×7 feature maps). These odd numbers lead to complex access patterns when performing sliding window computations which Spatial was not optimized for. This resulted in large LUT overheads after SRAM banking because banking schemes using non-powers of 2 were selected, which used expensive integer division and modulus to calculate addressing. Our DNNs drove improvements to Spatial's banking, including (1) banking support for more general access patterns to eliminate unnecessary crossbars and address calculation, and (2) modifications to Spatial's banking algorithm to automatically bank multi-dimensional SRAMs with parallel accesses. These led to a more than $2\times$ reduction in LUTs for some applications and are now part of Spatial's official repository.

In Summary, compiler Levels 1 and 3 leverage and improve third-party, open-source tools to improve DNN hardware performance. Level 2 is entirely within our framework and converts the optimized TensorFlow DFG to a program in the Spatial Language. We now describe Level 2 in more detail.

IV. DNN-Hardware Optimizations and Design Exploration

Optimization Level 2 converts the DNN DFG to Hardware IR. It involves three steps, shown in Fig. 2 (center). We now describes these, with emphasis on exploring design tradeoffs. As a running example we consider the ResNet-50 CNN [8].

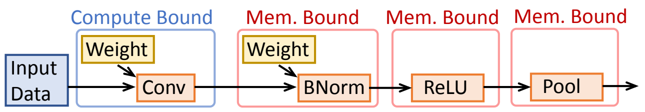


Fig. 3. An example of a frequently occurring layer pattern in CNNs. It is more efficient to fuse this into a single coarse-grained operation.

Layer	Count
Conv 7x7, Stride 2	1
Conv 1x1, Stride 1	30
Conv 3x3, Stride 1	16
Conv 1x1, Stride 2	6
Pool (Avg, Max)	2
Batch Norm	53
ReLU	49
Tensor Add	16
Fully-Connected	1

Operation	Count
Fused Conv 7x7, Stride 2	1
Fused Conv 1x1, Stride 1	30
Fused Conv 3x3, Stride 1	16
Fused Conv 1x1, Stride 2	6
Tensor Add	16
Fully-Connected	1

Fig. 4. ResNet-50 layers (left) and the coarse-grained operations they are grouped into after fusion (right).

A. *Identify coarse-grained operations (fusion)*

The first step selects the granularity of hardware operations to compose the DNN. Often, DNN layers occur in repeating patterns. Fig. 3 shows a common example in CNNs: Convolution/BN/ReLU/Pool. This pattern appears twice in ResNet-50 and the smaller pattern of Conv/BN/ReLU appears $53\times$.

Fusion is a state-of-the-art technique for DNN accelerators which minimizes DRAM bandwidth [13], [14]. As Fig. 3 shows, all but the Convolutional layer contain little computation and are bound by the time to access DRAM. It is therefore beneficial to fuse these layers into a single, more coarse-grained *operation*. This loads the data from DRAM once, performs all the layers in the operation without separately materializing the output of each, then stores the result after only the final layer. In this way, fused layers communicate using SRAM and registers to minimize DRAM bandwidth.

Our compiler performs fusion by matching for common patterns within the graph. Once a layer pattern is matched, the layers are replaced with the fused operation. Fig. 4 shows how the operations change in ResNet-50 after fusion. With these fusions applied, the arithmetic intensity of ResNet-50 inference, measured in Ops/byte (as in the Roofline models used by [2], [15]), improves by 2.55x.

B. *Algorithm-Level Transformations*

The next step considers algorithm-level transformations on fused operations to identify efficient hardware for each. Revisiting the example from Fig. 4, note there are two major types of operations: 1x1 and 3x3 convolutions (Tensor Add is computationally inexpensive and also can be fused into other layers). "1x1" and "3x3" refer to the filter size (rows$\times$columns). Other modern DNNs also primarily use smaller convolutions, and these sizes (1x1, 3x3) have become most common.

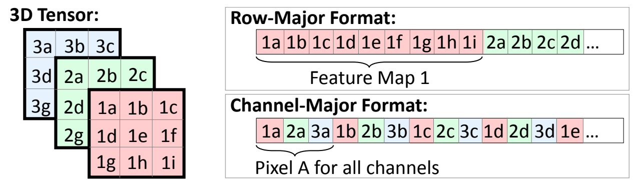


Fig. 5. Two popular storage formats for 3D data tensors in CNNs.

1x1 convolutions have lower arithmetic intensity and some locality in the 2D spatial dimension (rows×columns). 3x3 convolutions have higher arithmetic intensity and more locality in the 2D spatial dimension. Recall that convolutions in CNNs operate on 3D tensors of data, which consist of a series of 2D "channels" or "feature maps". The convolution involves sliding a series of filters across the rows/columns of each channel, creating 2D locality. Moreover, at each step of the sliding window, a 2D dot product (e.g. 3x3) is performed, further increasing locality. The convolved results of each channel are then added together to produce a single channel of output (output feature map). This sum now creates locality in the channel dimension. Finally, the process is repeated for multiple sets of filters to produce multiple output channels.

While the loops comprising this algorithm have been studied for DNN accelerators [16], [17], prior work often overlooks the storage format of these 3D tensors. Fig. 5 shows the two popular data storage formats in DRAM used by DNN frameworks and accelerators: row-major and channel-major.

Each format is better suited to a different algorithm for convolution. Row-major (rows contiguous in DRAM) is better suited to convolution as sliding window, due to locality in the rows/columns. Here the inner loops implement the 2D stencil sliding across the rows/columns, and then the outer loops sum each 2D result. Sliding window was used by [10], [14]. Channel-major (channels contiguous in DRAM) is also useful, due to locality in the channels (convolution results from each channel are summed). In this case the outer loops implement the 2D sliding window, while the inner loop performs a 1D dot product along each activation and kernel channel [17].

Fig. 6 shows the architecture for row-major format, which uses sliding window convolution. Each PE loads input feature maps from DRAM and performs convolution with a block of kernels. The core computation (dot product) is each 2D window reduction. The partial output feature maps from each PE are then summed across channels to produce a block of output feature maps. The architecture for channel-major is very similar, except that the dot products are 1D, and performed along the channels of the kernels and activations. As a result the accumulation across channels is performed within each PE as part of the dot product rather than in the outer-loop.

Fig. 7 shows experiments with the different data formats for both major operation types in ResNet-50: 1×1 (left) and 3×3 (right) convolution. The performance requirement is set to 1ms for 1×1 and 2ms for 3×3, and since tensor dimensions change along the depth of a DNN, designs must meet this latency for sizes ranging from (rows, columns, in_channels, out_channels) = (56, 56, 64, 64) to (7, 7, 512, 512), which is common for DNNs. For each layer type, we compare the two data formats

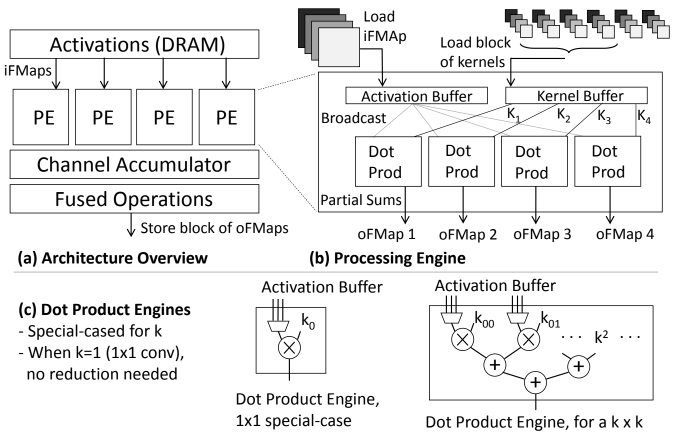


Fig. 6. Architecture overview for tensors stored in row-major format.

and their associated algorithm.

For 1x1 convolution, both formats use similar resources as there is locality both in the channels as well as rows/columns.

For 3x3 convolution, now row-major benefits from additional locality in the rows and columns. With channel-major, overlapping regions of the sliding window need to be loaded multiple times from DRAM, increasing bandwidth requirements. To meet the same performance, larger buffers were needed for channel-major to store more kernels on-chip at once. This reduced the number of times the data is re-loaded from DRAM, but increased logic and memory utilization. Overall, due to locality, row-major is more efficient for larger convolution sizes. Choosing this incorrectly results in a $> 2\times$ increase in BRAM/URAM end-to-end for ResNet.

There is some prior work which also finds the data formats best-suited to their architecture [9], [18], but we are the first to quantify the costs of choosing incorrectly by comparing formats experimentally. Also, [9] focused on small DNNs with small input images (32×32). Such smaller data structures may not reflect real-world applications and could change the locality analysis. Our work compares data structure formats experimentally for larger inputs used by modern DNNs, and we are the first to measure how format impacts different algorithms and convolution sizes based on locality, in particular for convolutions of the sizes used by ResNet-like DNNs.

Based on these experiments, our compiler selects default settings for data format and algorithm. While we described one algorithm tradeoff related to memory format, our toolchain can be used to explore others, e.g. FFT or Winograd convolutions. We also observed while optimizing for each format that sometimes algorithms ran into memory bottlenecks, yet were not at the intrinsic DRAM bandwidth. This was because bottlenecks were in memory access logic, e.g. generating addresses and dequeuing from FIFOs. The customizability of FPGAs allowed us to solve this by introducing parallelism into the memory system, e.g. creating parallel channels to the DRAM controller and allowing PEs to issue simultaneous DRAM commands to hide memory access latencies. This eliminated memory bottlenecks and made execution compute-bound, allowing us to use a static model for runtime and resources to select parallelizations. We describe this in the following subsection.

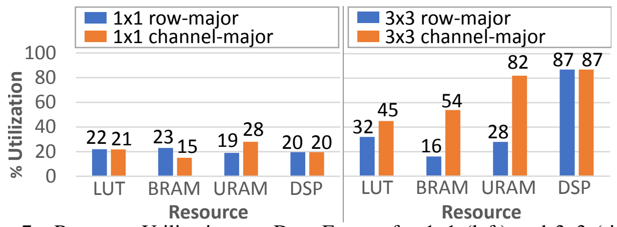


Fig. 7. Resource Utilization vs. Data Format for 1x1 (left) and 3x3 (right) convolutional layers

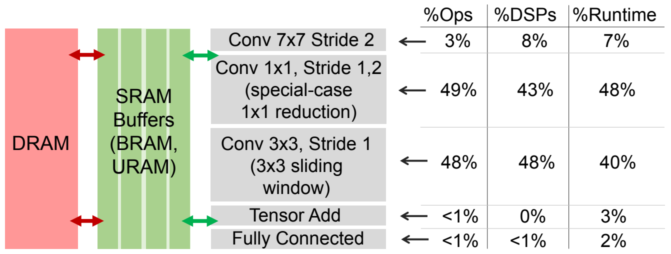
C. Assign Operations to Hardware

The final step assigns operations to hardware and schedules their execution. This involves determining the right degree of specialized hardware per DNN operation. We build on the strategy from [17], which takes advantage of the FPGA's flexibility to instantiate a specialized processor for each operation type. They show that specializing portions of the FPGA for different convolutions increases performance more than 2× over a single, more general architecture shared by all layers, and that this gap grows for larger FPGAs (like on the Amazon cloud). A limitation of [17] however was that the approach was not detailed for the deeper, state-of-the-art networks like ResNet. Moreover, it considered only convolutional layers simulated in isolation and not entire DNNs running end-to-end in hardware. We therefore extend their analysis and explore to what degree this strategy applies to entire, modern DNNs.

Returning again to the ResNet-50 example of Fig. 4, note that 6/36 of the 1x1 convolutions have stride=2. The hardware of Fig. 6 is the same regardless of stride as it just skips computations in the sliding window, therefore all 1x1 operations map to a single processor with no overhead. Stride was not discussed by [17], and so we here report that different strides can map to the same processor without penalty.

We also mapped the 1x1 and 3x3 convolutions to a single processor, but there was no net area reduction from merging these two. While merging did instantiate one less processor, it added extra overhead due to the differences in control logic (Fig. 6c) needed when performing the window reduction for 3x3 and 1x1. Separate processors also helped routability across stacked dies of the device. This extends the finding of [17] that kernel specialization is still beneficial in modern DNNs.

Our compiler therefore distinguishes operations by their fused layer types, treating convolutions of different kernel sizes as separate types. It then instantiate one processor per operation type, each like Fig. 6 and specialized to its operation, and creates an FSM which executes the fused operations sequentially on their associated processor, writing intermediate data to DRAM. The architecture for the ResNet example is shown in Fig. 8. Our compiler statically calculates the number of total multiply-accumulates (MACs) contributed by each operation type, and allocates DSP resources proportional to that (rounded to a power of 2). We show these details next to each processor, and for reference include the percentage of runtime for a single inference. The totals, described further in section 5, are the total MACs in ResNet-50, total DSPs used by the design and total inference time. The result is that the runtime is balanced between two processors, 1x1 and 3x3, as



	%Ops	%DSPs	%Runtime
Conv 7x7 Stride 2	3%	8%	7%
Conv 1x1, Stride 1,2 (special-case 1x1 reduction)	49%	43%	48%
Conv 3x3, Stride 1 (3x3 sliding window)	48%	48%	40%
Tensor Add	<1%	0%	3%
Fully Connected	<1%	<1%	2%

Fig. 8. Overview of architecture for ResNet example.

expected from Fig. 4. These can be used simultaneously by processing two inputs at once, as described in [17].

This section described the optimizations needed to transform a DNN DFG into a synthesizable circuit. We performed experimental analysis of the design choices summarized in Fig. 1, including operation granularity (fusion), hardware specialization, algorithm transformations, data format, and memory-level parallelism. Next, we measure the performance of various DNNs mapped to hardware using this toolchain.

V. Experiments

We consider 4 benchmark TensorFlow DNNs from various multimedia applications in Table II. These were chosen to exercise a wide variation in the design space. ResNet-50 is a complex graph containing a wide variety of layers as Section 4 showed. Djinn-ASR is a speech-to-text cloud service MLP [19] which contains fully-connected (FC) layers. Unlike CNNs which process images, Djinn-ASR has small activations but a large model. TensorFlow CIFAR is the CNN used by TensorFlow's tutorial for the CIFAR-10 dataset. LeNet is a small CNN chosen because its weights and activations can fit entirely on-chip. These DNNs and datasets are commonly evaluated in FPGA DNN papers [9], [14], [20].

We ran these DNNs on the Amazon EC2 F1.2xlarge FPGA instance, which contains a Xilinx UltraScale+ 16nm VU9P FPGA. The only other tool available which ran DNNs on the F1 was Xilinx's ML-Suite (v1.3) [21]. As we will discuss, our toolchain can compile TensorFlow models that gave ML-Suite errors. First however, we compare speed performance.

Table II compares single-input (batch-1) latency of these DNNs. Microsoft argues batch-1 latency to be the most valuable metric for DNN data-center workloads [3]. We measure performance of these DNNs for both our tool, which is an academic research tool, and Xilinx's ML-Suite, which is a commercial tool. ML-Suite is state-of-the-art, achieving same or superior performance to GPUs [22] and running certain memory-bound layers on the CPU using optimized BLAS [21].

We match ML-Suite on all but ResNet, for which we are currently 5.7x slower. This is because of two major differences: (1) ML-Suite runs DSPs at 500 MHz, while we currently target the default 125 MHz for the F1, and (2) we use 32-bit fixed point (10/22) and match the exact output of TensorFlow, while ML-Suite uses 8-bit precision. Implementing these optimizations in our toolchain would result in 4x and 7.4x speedup respectively (discussed further below).

Low precision causes problems however for ML-Suite. The only reasonable accuracy we could achieve from it was using

TABLE II
PERFORMANCE AND RESOURCE UTILIZATION OF VARIOUS TENSORFLOW DNNS RUNNING ON VU9P OF EC2 F1.2XLARGE.

TensorFlow Benchmarks				**Xilinx ML-Suite**	**Ours**					
DNN	**Dataset**	**#Weights**	**#MAC**	**Latency (ms)**	**Latency (ms)**	**LUT**	**Reg**	**DSP**	**BRAM**	**URAM**
ResNet-50	ImageNet	25.5M	3.9G	38.1	216 (8bit: 28.9)	51.3%	21.9%	87.8%	51.9%	41.4%
Djinn-ASR	Kaldi	25.4M	25.4M	10.7	10.0	15.1%	6.54%	14.1%	16.3%	0.00%
TF CIFAR	CIFAR-10	1.06M	19.5M	1.20	1.39	25.4%	10.9%	44.6%	37.0%	28.5%
LeNet	MNIST	430k	2.29M	1.01	0.656	16.3%	8.44%	26.7%	18.9%	3.13%

their pre-compiled ResNet-50 Caffe model (Caffe is an older framework). When trying to compile TensorFlow through ML-Suite, we followed the documentation and exhaustively tried settings for both TensorFlow's official ResNets as well as a Xilinx-provided ResNet converted to TensorFlow from Caffe, and the highest Top-5 ImageNet validation accuracy we could achieve was 12.1%, which indicates a bug in the TensorFlow compilation. Though our tool uses higher precision, it matches the output of a CPU or GPU (93.2%). Overall, only our tool could reasonably run the TensorFlow model on the F1. ML-Suite could achieve acceptable accuracy (91.5%) with their pre-compiled ResNet example made from Caffe, although for this we also had to search configurations and only one (8-bit, "medium" kernel configuration) worked. The rest also produced unusable accuracy. The speed of this pre-compiled ML-Suite Caffe example is 12.8 ms. This is faster than the TensorFlow ResNet compiled through ML-Suite, and now has a gap of 16.8x with our tool. With the speedups mentioned above from frequency/precision, our ResNet would have competitive performance. We note though that ML-Suite's pre-compiled Caffe would still fail e.g. the recent DAWNBench inference acceleration benchmarks [23], which require 93% Top-5 ImageNet validation accuracy.

Table II also shows utilization for each benchmark. For ResNet, the limiting resource is DSPs and ResNet is compute-bound. We experiment with changing the datatype to 8-bit and DSP utilization decreases by 7.47x. From this we project an 8-bit latency of $\sim$28.9ms if quantization were applied, which is a common technique [3], [21], [24]. Prior work reports similar scaling when lowering precision [15], [18], [25]. We also run at the default 125 MHz, but higher frequencies are available for the F1. While Spatial does not yet support this for general applications due to routing across stacked dies of the VU9P, our DNN-specific toolchain could achieve higher frequencies by constraining layer processors to separate 2D logic regions, which is also a known optimization [21]. ML-Suite's higher frequency comes at the cost of higher power consumption however. We ran continuous batch-1 inference as in [3], [25], and measured power using the F1's power metrics. At 125 MHz we consume 17W. With batch size set to 1, ML-Suite's ResNet example consumes 42W. As a reference we compare to a 12nm NVIDIA V100 GPU on the EC2 P3 instances (recommended by Amazon for deep learning), which cost \$3.06/hr (vs. \$1.65/hr for F1). Its continuous TensorFlow ResNet-50 batch-1 inference latency is 5.54ms and it consumes 110W, measured using `nvidia-smi`.

Other benchmarks do not hit resource limits and can be further parallelized. Djinn-ASR is bandwidth-bound, so more parallelization would not help. Its consists of 7 FC layers and the design contained a single FC/Bias/Sigmoid fused operation processor. TF CIFAR contains 2 Conv layers (5x5) and 3 FC layers, with Conv followed by pooling. 5x5 Conv and pooling increase locality in rows/columns and data was stored in row-major. The design contained two fused operation processors, Conv/Bias/ReLU/Pool and FC/Bias/ReLU. LeNet contains 2 Conv/Bias/ReLU/Pool followed by 2 FC/Bias/ReLU operations and used the same processor types as TF CIFAR.

VI. RELATED WORK AND CONCLUSIONS

Ours is the first open tool which uses a modern DNN framework like TensorFlow (TF) as a starting point and either (1) targets public hardware like Amazon F1 or (2) compiles DNNs reaching state-of-the-art accuracy on an FPGA (cloud or not). [26] presents a survey of tools targeting FPGAs from DNN frameworks, noting that very few are open source. Moreover, nearly all use older frameworks like Caffe [20], [25], [27]–[29], which are rarely still used in the DNN community. [18], [30] discuss TF but are closed-source. [31] is open-source and uses TF, but runs only small MLPs and convolutions. ML-Suite [21] is closed-source and had incorrect TF results. Often tools also do not report accuracy, even when using low precision, and many support only older DNNs like VGG16. [32] is open-source and used MXNet as an input, but the DNN presented is ResNet-18 which has low accuracy like VGG16 [8]. DNN tools for cloud FPGAs include ML-Suite and [15], [33]. ML-Suite compiles TF while [15], [33] use Caffe. [15] mentions TF but not in its release, and unlike ML-Suite neither of [15], [33] had complete instructions or examples to compile and run models on F1. [33] used low precision but accuracy was omitted, and ML-Suite's Caffe ResNet is faster on the F1. [15] achieves high performance on the F1 using 1-bit precision, but runs small DNNs.

By performing the optimizations outlined in this work, we enable FPGA designs to be explored from high-level DNNs. We studied design tradeoffs that showed the importance of specialized hardware and memory-system flexibility, highlighting where FPGAs excel. We hope our toolchain can be useful for researchers to explore FPGA accelerators for DNNs so that FPGAs can continue playing a role in this exciting domain.

ACKNOWLEDGMENT

This material is based on research sponsored by DARPA under agreement number FA8750-17-2-0095. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, no. 521, pp. 436–444, 2015.

[2] N. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *In Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture, ISCA '17*. ACM, 2017, pp. 1–12.

[3] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu *et al.*, "A configurable cloud-scale DNN processor for real-time AI," in *In Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture, ISCA '18*. IEEE, 2018, pp. 1–14.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis *et al.*, "TensorFlow: A system for large-scale machine learning," in *In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation, OSDI'16*. ACM, 2016, pp. 265–283.

[5] Amazon, *EC2 F1 Instances with FPGAs - Now Generally Available.*, 2017. [Online]. Available: aws.amazon.com/blogs/aws/

[6] Xilinx, *Custom Hardware Acceleration in the AWS Cloud.*, 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/acceleration-zone/aws.html

[7] V. Sze, T.-J. Yang, Y.-H. Chen, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE, Volume 105 Issue 12*, pp. 2295–2329, December 2017.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR'16*. IEEE, 2016, pp. 770–778.

[9] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA'17*. ACM, 2017, pp. 65–74.

[10] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin *et al.*, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA'17*. ACM, 2017, pp. 15–24.

[11] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 26 Issue 7*, pp. 1354–1367, July 2018.

[12] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis *et al.*, "Spatial: A language and compiler for application accelerators," in *In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'18*. ACM, 2018, pp. 296–311.

[13] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO'16*. IEEE, 2016.

[14] S. I. Venieris and C.-S. Bouganis, "Latency-driven design for FPGA-based convolutional neural networks," in *In Proceedings of the 27th International Conference on Field Programmable Logic and Applications, FPL'17*. IEEE, 2017.

[15] M. Blott, T. B. Preusser, N. J. Fraser, G. Gambardella, K. Obrien *et al.*, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks." *ACM Transactions on Reconfigurable Technology and Systems (TRETS) - Special Issue on Deep learning on FPGAs, Volume 11 Issue 3, Article No. 16*, December 2018.

[16] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA'15*. ACM, 2015, pp. 161–170.

[17] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *In Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture, ISCA '17*. ACM, 2017, pp. 535–547.

[18] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *In Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM'17*. IEEE, 2017, pp. 152–159.

[19] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li *et al.*, "Djinn and tonic: DNN as a service and its implications for future warehouse scale computers," in *In Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture, ISCA '15*. IEEE, 2015, pp. 27–40.

[20] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim *et al.*, "From high-level deep neural models to FPGAs," in *In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO'16*. IEEE, 2016.

[21] Xilinx, "Xilinx DNN processor: An inference engine, network compiler and runtime for Xilinx FPGAs." in *Hot Chips*, 2018.

[22] xilinx, *ML Suite Frequently Asked Questions.*, 2019. [Online]. Available: https://github.com/Xilinx/ml-suite/blob/master/docs/faq.md#how-does-fpga-compare-to-cpu-and-gpu-acceleration

[23] C. Coleman, D. Narayanan, D. Kang *et al.*, "DAWNBench: An end-to-end deep learning benchmark and competition." in *SysML*, 2018.

[24] Intel, *OpenVINO(TM) Toolkit.*, 2018. [Online]. Available: https://software.intel.com/en-us/openvino-toolkit

[25] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks." in *In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD'16*. IEEE, 2016.

[26] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions," *ACM Computing Surveys (CSUR) Volume 51 Issue 3, Article No. 56*, July 2018.

[27] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated FPGAs: FPGA framework for convolutional neural networks." in *In Proceedings of the International Conference on Field-Programmable Technology, FPT'16*. IEEE, 2016, pp. 265–268.

[28] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs." in *In Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM'16*. IEEE, 2016, pp. 40–47.

[29] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *In Proceedings of the 27th International Conference on Field Programmable Logic and Applications, FPL'17*. IEEE, 2017.

[30] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong *et al.*, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *In Proceedings of the International Conference on Computer-Aided Design, ICCAD'18, Article No. 56*. ACM, 2018.

[31] D. H. Noronha, B. Salehpour, and S. J. E. Wilton, "LeFlow: Enabling flexible FPGA high-level synthesis of Tensorflow deep neural networks," in *In Proceedings of the Fifth International Workshop on FPGAs for Software Programmers, FSP'18*, 2018.

[32] T. Moreau, T. Chen, and L. Ceze, "Leveraging the VTA-TVM hardware-software stack for FPGA acceleration of 8-bit ResNet-18 inference," in *In Proceedings of the 1st Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning, ReQuEST'18, Article No. 5*. ACM, 2018.

[33] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs," in *In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA'19*. ACM, 2019, pp. 73–82.