Compiler and FPGA Overlay for Neural Network Inference Acceleration

Abstract
Overlays have shown significant promise for FPGAs. They allow fast development cycles, and remove many of the challenges found in traditional FPGA hardware design. However, this often comes with a significant performance burden resulting in very little adoption of overlays for practical applications. In this paper, we tailor an overlay to a specific application domain and show how we maintain its full programmability without paying a large performance burden traditionally associated with overlays. Specifically, we introduce an overlay targeted for deep neural network inference with only ~1% overhead to support the control and reprogramming logic using a lightweight very long instruction word (VLIW) network. Additionally, we implement a sophisticated domain-specific graph compiler that compiles deep learning languages such as Caffe* or TensorFlow* to easily target our overlay. We show how our graph compiler performs architecture-driven software optimizations that significantly boosts the performance of both convolutional neural network (CNN) and recurrent neural network (RNN). We demonstrate a 4X improvement on ResNet-101 and a 12X improvement for long short-term memory (LSTM) cells, compared to naïve implementations. Finally, we describe how we can tailor our hardware overlay, and use our graph compiler to achieve 900 frames per second (fps) on GoogLeNet on an Intel® Arria® 10 GT 1150 device—the fastest ever reported on similar FPGAs.

Introduction
Creating custom high-performance hardware designs on FPGAs is difficult and time consuming when compared to software programmable devices such as CPUs. A hardware designer must describe their system in a cycle-accurate manner, and worry about low-level hardware considerations such as timing closure to memory interfaces. Over the past decade, significant progress has been made to ease the use of FPGAs through high-level languages, such as OpenCL™, to implement high-performance designs [9]. However, even when using high-level design, one must still carefully describe an efficient parallel hardware architecture that leverages the FPGA’s capabilities such as the massive on-chip memory bandwidth or configurable multiplier blocks. Additionally, the designer must optimize both area and frequency through long compilations to realize performance gains versus other programmable platforms. Compared to writing a software algorithm targeting a CPU, designing for FPGAs is still drastically more difficult. Our goal is to present a software-programmable hardware overlay to demonstrate the ease of use of software programmability and the efficiency of a custom hardware design.

We introduce a domain-specific approach to overlays that leverages both software and hardware optimizations to achieve state-of-the-art performance on the FPGA for neural network acceleration. For hardware, we partition configurable parameters into run-time and compile-time parameters such that architecture performance can be tuned at compile time, and overlay programmed at runtime to accelerate different neural networks. We do this through a lightweight very long instruction word (VLIW) network that delivers full reprogrammability to our overlay without incurring any performance or efficiency overhead typically found in overlays [4].
Additionally, we create a flexible architecture where only the core functions required by a neural network are connected to a parameterizable interconnect (Xbar). This avoids the need to include all possible functions in our overlay during runtime; rather, we can pick from our library of optimized kernels based on the group of neural networks that are going to run on our system. Our approach is unlike previous work that created hardware that can run only a single specific neural network [1], [7-8].

On the software side, we introduce an architecture-aware graph compiler that efficiently maps an neural network to the overlay. This both maximizes the hardware efficiency when running the design and simplifies the usability of the end application; where users are only required to enter domain-specific deep learning languages, such as Caffe* or TensorFlow*, to program the overlay. Our compiler generates VLIW instructions that are loaded into the FPGA and used to reprogram the overlay in tens of clock cycles, incurring no performance overhead. Compared to fixed-function accelerators that can only execute one neural network per application run, our approach allows multiple neural networks to be run consecutively in a single application simply by reprogramming our overlay instead of recompiling or reconfiguring the FPGA.

The paper is organized as follows: We introduce our hardware architecture and describe how we can optimize our architecture to target specific neural networks using our compile-time parameters and Xbar interconnect. Importantly, we describe our lightweight VLIW network used for programming the overlay. We go on to describe our neural network graph compiler, and detail some of our architecture-driven optimizations that allow the efficient implementation of neural networks on architecture variants of different sizes. We elaborate on how our graph compiler and hardware overlay work together for efficient implementation of CNNs and RNNs. We walk through the hardware and software optimizations to implement both the ResNet and GoogLeNet CNNs, allowing us to achieve record-setting performance on GoogLeNet. Finally, we discuss the implementation of a long short-term memory (LSTM) cell simply by adding an additional kernel to our overlay, and relying on our graph compiler to mutate the LSTM cell graph to fit within our overlay. We refer to our system as a deep learning accelerator (DLA).

**Hardware Architecture**

Our domain-specific overlay aims to be general enough to implement any neural network, but remain customizable to be optimized for a specific neural network (Figure 1). At the core of the overlay is a 1D systolic processing element (PE) array that performs dot product operations to implement general matrix math such as convolutions or multiplications. We omit the discussion of numerics in this paper but we support different floating-point formats such as fp32/16/11-8 which have been shown to work well with inference [2]. These could easily be modified to support any nascent innovations in data precision such as bfloat [15], and other unique fixed or floating-point representations, due to the flexible FPGA fabric. Our Xbar interconnect augments the functionality of our overlay with different auxiliary functions or kernels. This section goes through the hardware architecture and highlights the built-in compile-time flexibility and run-time programmability of our overlay. We highlight our VLIW network and explain how we achieve very low programmability overhead. Finally, we describe how on-chip memory—one of the most powerful resources—is used to balance computation and memory bandwidth to accelerate the performance of specific neural networks.

**Figure 1.** System-Level Diagram of Our Neural Network Inference Accelerator.
VLIW Network

To implement a neural network on the DLA, our graph compiler breaks it into units called "subgraphs" that fit within the overlay's buffers and compute elements. For example, with CNNs, a subgraph is typically a single convolution with an optional pooling layer afterwards. We deliver new VLIW instructions for each subgraph to program the DLA correctly for subgraph execution.

Our novel VLIW network distributes instructions to each kernel (Figure 2). The VLIW reader continuously fetches instructions for the next subgraph from external memory and sends it down a 32 bit unidirectional ring network connected to all the kernels in the DLA. The VLIW instruction sequence is divided into different portions for each kernel. A header packet identifies the kernel, followed by a series of programming instructions destined for that kernel. "Transport" kernels parse the header packet and redirect instructions to the correct kernel.

![Figure 2. VLIW Network Distributes Instructions to Each Kernel.](Image)

Our instructions are actually counter end values and control flags loaded directly into registers within each kernel to govern its operation, avoiding the need for any instruction decode units. For example, the pool kernel receives approximately a dozen instructions: the image height/width/depth, the pool window size, and the type of pooling (maxpool or average pool). Before executing each subgraph, the pool kernel reads each of its 12 instructions serially, consuming 12 clock cycles, with no material impact on performance that typically takes thousands of cycles. However, it ensures the entire VLIW network remains only serially, consuming 12 clock cycles, with no material impact on performance that typically takes thousands of cycles.

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>FFs</th>
<th>ALMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW Reader</td>
<td>1,832</td>
<td>1,841</td>
<td>1,473</td>
</tr>
<tr>
<td>Transport</td>
<td>126</td>
<td>139</td>
<td>73</td>
</tr>
<tr>
<td>Total</td>
<td>3,092</td>
<td>3,231</td>
<td>2,046</td>
</tr>
</tbody>
</table>

Table 1. Area Overhead of VLIW Network with 10 Kernels at 450 MHz.

Xbar Interconnect

Machine learning is a fast-developing field. We increasingly see new functions implemented by the machine learning research community. New activation functions are constantly being evaluated such as Swish* [10]. A quick look at TensorFlow* shows more than 100 different layer types users can experiment with to build neural networks [15]. We aim to use the Xbar for DLA extensibility for users to easily add or remove functions to implement different neural networks.

A custom interconnect Xbar example, connecting pool/local response normalization (LRN) kernels for CNNs (Figure 1), is built around exactly what is needed to connect the auxiliary kernels. The SqueezeNet graph has no LRN layers, so that kernel can be removed completely. From a prototxt architecture description, the Xbar (including width adaptation) is automatically created to connect auxiliary kernels. We use width adapters to control the throughput of each auxiliary kernel and can decrease the width of infrequent kernels, such as LRN, to conserve logic resources. The interconnect pattern within the Xbar is also customizable based on the order of the auxiliary operations. The AlexNet* graph has both MaxPool and LRN layers, but LRN always comes first; whereas the GoogLeNet* graph has some layers in which MaxPool precedes LRN, which is supported in Xbar by adding more multiplexing logic.

To demonstrate the power of our extensible architecture, and compiler, we add a single kernel to the Xbar that extends our architecture to implement LSTM cells alongside CNNs. This allows implementing video-based RNNs commonly used for gesture recognition [16].

Vectorization

To ensure our overlay can be customized to different neural network models and FPGAs, we support vectorization, or degree of parallelism, across difference axes. The degrees of parallelism available in the accelerator, configurable via vectorization, are shown in Figure 1. Q_VEC and P_VEC refer to parallelism in the width and height dimensions, while C_VEC and K_VEC refer to the input/output depth parallelism, respectively. Every clock cycle, the product of Q_VEC, P_VEC, C_VEC, and K_VEC feature values are processed in parallel.

Initially, our design was scaled by increasing K_VEC. However, this method of scaling saw diminishing returns, since quantization inefficiencies can become more pronounced as vectorization dimensions increase. For example, if the output depth (K) of a layer is 96, and K_VEC is 64, this requires 2 complete iterations through the PE array, with only 96 of 128 (75%) useful computations. On the other hand, if K_VEC is 32, the output depth divides perfectly into three iterations for 100% efficiency. To mitigate this quantization effect, it is possible to balance the scaling of the design across multiple different dimensions besides just K_VEC (e.g., P_VEC, Q_VEC, C_VEC). The optimal balance of vectorization depends on the graph layer dimensions, and can be seen by comparing the throughput of two architectures with similar area for different graphs (Figure 3). The optimal balance of scaling between P_VEC and K_VEC varies based on the neural network topology used and is an example of how we tune our overlay to achieve the highest possible performance for a specific neural network.
Stream Buffer and Filter Caches

A single Intel Arria 10 FPGA contains ~4 terabytes per second (TBps) on-chip memory bandwidth, interspersed within the FPGA in configurable 20 kilobit (Kb) memory blocks. This powerful resource is pivotal in determining the performance of compute operations. The DLA leverages these block RAMs to buffer both the activation and filter tensors. The filters are stored in a double-buffered “filter cache” contained in each PE (Figure 1), allowing the PEs to compute data while filters are pre-loaded from external memory for the next subgraph. The “stream buffer” is a flexible scratchpad used to store intermediate tensors on-chip. Many graph compiler passes are dedicated to the efficient use of this stream buffer.

When presented with an intermediate tensor larger than the stream buffer or filter caches, our graph compiler slices the tensor into multiple pieces that fit within the on-chip caches. Any remaining pieces are stored in slower off-chip memory, that require higher latency to fetch and compute. To limit slicing, we can increase the size of the stream buffer and/or filter caches, but this decreases the number of RAM blocks available for PE array vectorization. Therefore, a memory vs. compute trade-off exists between the size of the caches and the number of PEs. An optimal trade off for one neural network can cause as much as 40% or more performance degradation in another (Figure 4).

Slicing

To achieve the highest possible throughput, it is desirable to fit the entire input feature and filter tensors in the stream buffer and filter caches. However, as the image resolution increases and neural network graph topologies become deeper, on-chip allocation of these tensors may not be feasible. To overcome this constraint, slices of the input tensor are fetched from external memory into the stream buffer and processed independently.

The 3D input feature tensor can be sliced along the height, width, or depth to fit in the on-chip stream buffer. When slicing along the width and height, the slices must overlap if the filter window size is greater than 1x1. The graph compiler tries to pick slices to minimize the overall overlapped computation for the sliced tensor. Alternatively, slicing across the depth does not require overlapped computations, but does require an operation to add the results of the depth-wise slices.

To boost performance and minimize DDR4 SDRAM spillover, we enhance our slicing algorithm to slice multiple sequential convolutions together—called "Group Slicing". Instead of completing all slices within a layer, we compute several sequential convolutions with a single slice using the stream buffer before moving on to the next slice. Our group slicing algorithm reduced the number of external memory spillovers (Figure 5), and improved throughput by 19% over simple slicing for a Resnet-101 neural network with 1080p high definition (HD) image resolution.

**Graph Compiler**

The previous section focused on the hardware overlay architecture and how to configure it at compile time to maximize specific neural network graph performance. This section describes our neural network graph compiler that takes advantage of the overlay VLIW instructions to decompose, optimize, and run a neural network model on the overlay. The graph compiler breaks a neural network into subgraphs, schedules subgraph execution, and importantly, allocates explicit cache buffers to optimize the use of the stream buffer and filter caches. We will go through our core compiler “passes” (slicing, scheduling and allocation), showing examples of how smart graph compilation allows more efficient hardware implementations. Besides these general core passes, our compiler is easily extensible with the addition of more specific algorithms that target and optimize specific neural network patterns.

---

**Figure 3.** Throughput/Area on Two Architectures With Different P_VEC and K_VEC Vectorization.

**Figure 4.** Impact of Stream Buffer Memory vs. Compute Tradeoff on AlexNet, GoogleNet and ResNet-101.

**Figure 5.** Group Slicing Minimizes External Memory Spillover by Computing Multiple Sequential Convolutions for Each Slice.
Allocation

The allocation pass manages reading and writing from the stream buffer. Allocation calculates the addresses for each slice, and computes the total stream buffer memory used by a graph. One of the main goals is to reduce fragmentation – gaps between allocated memory blocks in the stream buffer. In its most simple operation, the stream buffer is used as a double buffer to store both the input and output of a subgraph. To achieve this double-buffering while reducing fragmentation, the input buffer starts at address 0 and counts up, while the output buffer starts at the end of the stream buffer and counts down (Figure 6).

Figure 6. Double-Buffering in the Stream Buffer.

This leaves a contiguous space in the middle used to allocate more data slices in the stream buffer; this is especially useful for graphs with multiple branches, demonstrated by the GoogLeNet example in the next section. Note the allocation pass must keep track of the lifetime of each buffer to be able to free/overwrite its memory once it is no longer used. Additionally, the allocation pass assigns addresses in external memory when the stream buffer is not large enough, but memory size is not a problem so it is simply done left-to-right, in the first available space.

Scheduling

The DLA compiler partitions neural networks into subgraphs; a list of functions that can be implemented on the DLA without writing to a buffer, except at the very end of subgraph execution – scheduling decides when each subgraph is executed. In early CNN models, such as AlexNet [5] or VGG-16 [17], there was no need for a scheduler as there are no decisions to be made on which subgraph to execute next. When considering CNNs with branching nodes such as GoogLeNet [6], ResNet [14], or graphs that require slicing, the order of subgraph execution heavily influences the stream buffer size required for a given graph to avoid external memory spillover.

Figure 7a illustrates an example of an inception module from GoogLeNet, partitioned into DLA subgraphs with the relative output sizes of each subgraph. We show the stream buffer allocation corresponding to two possible schedules of the inception module. Both are depth-first schedules, Figure 7b starts with the leftmost branch, while Figure 7c starts with the rightmost branch. This simple change in schedule results in a 30% reduction of the required stream buffer size for this inception module. When considering large graphs with many branching nodes that either converge to a single output, such as GoogLeNet, or graphs that diverge to several outputs, such as those used for single-shot multibox detection [12], an exhaustive search of all possible schedules may not be feasible without incurring large compile-time penalties.

Our scheduling is conducted using a priority queue-based approach, where the cost of executing a given node is determined by the ratio of its output size to its effective input size (the input size multiplied by the number of users of the input tensor). This approach achieved the stream buffer savings of Figure 7c, with minimal impact on compiler run time.

Figure 7. Scheduling One of the GoogLeNet [6] Inception Modules.
**CNN Implementation**

This section focuses on two popular CNNs: ResNet [14] and GoogLeNet [6]. We explain different hardware/software co-optimizations possible with our run-time reconfigurable and software programmable overlay, allowing us to significantly boost CNN performance with little effort.

**ResNet Convolution Merging**

ResNet-101 is a large graph targeted for high-definition image resolutions, creating intermediate tensors that require significant slicing to run on the Intel Arria 10 FPGA DLA overlay. ResNet is composed of three types of resmodules (Figure 8). Each type has two convolution branches merged through an element-wise addition operation (eltwise).

We present a resmodule optimization that eliminates the eltwise operation by merging it with the preceding convolutions. This reduces the number of arithmetic operations in the DLA, and more importantly, decreases the number of slices and DDR4 SDRAM spillover. Instead of storing intermediate tensors between the convolution and the eltwise addition operations, we combine them in a single convolution operation where tensor size is at least half as big as the eltwise input.

Consider the computation that produces every output element of the eltwise in a Type 1 resmodule (Figure 8a)—it is the sum of the corresponding output elements of convolution A and B3. In Figure 9a, this sequence of operations is equivalent to a single convolution after input A and B3 and the corresponding filter A and B3 are merged depth-wise. This effectively absorbs the eltwise addition operation into the dot product operation of the preceding convolutions. Figure 9b shows the Type 1 resmodule after convolution A and B3 are merged with the eltwise layer. Since this optimization converts the explicit eltwise operations into a convolution, outputs A and B3, which usually reside in DDR4 SDRAM or on-chip memory, become intermediate results for GoogLeNet. Given the extremely low frequency of these non-convolution layers (e.g., 2 of 147 for ResNet-101), it is best to map them to convolutions. In this way, we can reuse the powerful convolution engine (PE array) instead of adding non-convolution layers (e.g., 2 of 147 for ResNet-101), it is best to map them to convolutions. In this way, we can reuse the powerful convolution engine (PE array) instead of adding dedicated auxiliary kernels that would, over time, be under-utilized.

An FC layer performs a multiplication between a vector (input) and a matrix (weights). It is mapped as follows: (1) the 1D FC input of length N is mapped to a 3D convolution input of shape 1 x 1 x N, and (2) the 2D FC weight matrix of shape N x M is mapped to M 3D convolution filters of shape 1 x 1 x N. With this mapping, the computation of each FC output is assigned to a PE.

Average pooling of window H x W on a 2D image is equivalent to a 2D convolution with a filter of size H x W. Each filter element is of value 1/(H x W). For a 3D input of depth D, average pooling is applied to each 2D input surface, producing the corresponding output surface. In this case, the equivalent convolution filter for the output surface at depth d is of shape H x W x D, with all zero filter values except the surface at depth d being the average pooling filter.
Sparse Filter Shortening

Even though they save area, the identity and average pooling convolutions introduced in the previous optimizations could come at a high cost to throughput due to the large but sparse filters involved. For an identity convolution of input and output shape $H \times W \times D$, there are $D$ filters, each of shape $1 \times 1 \times D$. Since each filter is responsible for copying input surface at depth $d$ to the output surface at the same depth, the values of this filter are all zeros except 1 at depth $d$. Figure 11 illustrates the identity and average pooling convolution filters, how to leverage their sparsity to conserve DLA operations, and improve performance by skipping computations with filter entries filled with zeros. Since the PEs process $K\_VEC$ filters one at a time, we trim the filters size $K\_VEC$ to fit perfectly in the PE array. This effectively reduces the filter depth from $D$ to $K\_VEC$, saving both compute time and filter data load time. We call this optimization sparse filter shortening, which can also be applied to average pooling convolution (Figure 11), due to the same filter sparsity.

1x1 Filters Optimization

To efficiently compute convolutions using $3x3$ filters, the DLA architecture is often tuned to be vectorized in the filter width dimension by setting $S\_VEC=3$. Increasing the filter width vectorization increases PE throughput as well as filter prefetch bandwidth for large (e.g., $3x3$) filters. However, many of the latest CNNs have a mix of $3x3$ and $1x1$ filters. Convolutions using $1x1$ filters do not benefit from filter width vectorization, and achieve low digital signal processing (DSP) efficiency and filter prefetch bandwidth. To avoid this, the DLA architecture has been optimized for $1x1$ filters in two ways. First, DSPs that would have been used in a $3x3$-filter convolution to process the second and third filter values in the filter width direction are instead used to calculate two additional output pixels in a $1x1$-filter convolution. This allows the PEs to maintain the same DSP efficiency for both $3x3$ and $1x1$ filters. Second, the filter prefetch bandwidth added to load a 3-wide filter is used to load more 1-wide filters in parallel. Overall, these two optimizations allow the DLA to achieve high throughput through vectorization for $3x3$-filter convolutions without suffering any additional quantization loss for $1x1$-filter convolutions.

Optimization Impact on ResNet

Table 2 summarizes the impact of each optimization on ResNet-101 throughput for 1080p image resolution. The number in each row is the normalized throughput after applying all optimizations listed up to this row. Here, we apply the mapping of GAP and FC layers to convolution unconditionally (i.e., in the baseline). The huge speedup of sparse filter shortening comes from the filters of the identity convolutions introduced by convolution merging optimization on Type 3 resmodules which account for 87% of all resmodules in ResNet-101.

<table>
<thead>
<tr>
<th>OPTIMIZATION</th>
<th>RELATIVE THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1.0</td>
</tr>
<tr>
<td>1x1 Filter Opt</td>
<td>1.2</td>
</tr>
<tr>
<td>Conv. merging (Type 1)</td>
<td>1.3</td>
</tr>
<tr>
<td>Conv. merging (Type 2)</td>
<td>1.6</td>
</tr>
<tr>
<td>Conv. merging (Type 3)</td>
<td>1.9</td>
</tr>
<tr>
<td>Sparse filter shortening</td>
<td>3.5</td>
</tr>
<tr>
<td>Group slicing</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Table 2. Optimization Impact on ResNet-101.

Optimization Impact on GoogLeNet

Two of the CNN optimizations described are used to improve GoogLeNet throughput: (1) the $1x1$ filter optimizations, and (2) the average pool mapped to convolution optimization—this allowed the DLA to fit a larger PE array instead of wasting dedicated resources on an average-pooling kernel. The GoogLeNet throughput improvement from these two optimizations was 17% (Table 3). The next row shows the improvement from increasing the PE array vectorization from $\{P\_VEC,K\_VEC\} = \{1,48\}$ to $\{2,32\}$. Finally, the last row points to an accurate model of external memory optimizations that allowed the design to achieve ~900 fps on GoogLeNet on an Intel Arria 10 GT 1150 device, which to our knowledge, is the most efficient acceleration of GoogLeNet on an FPGA.†

<table>
<thead>
<tr>
<th>OPTIMIZATION</th>
<th>RELATIVE THROUGHPUT</th>
<th>RAW THROUGHPUT (INTEL ARRIA 10 GT 1150)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1.0</td>
<td>469 fps</td>
</tr>
<tr>
<td>1x1 Filter Opt</td>
<td>1.1</td>
<td>506 fps</td>
</tr>
<tr>
<td>Avg Pool Mapped to Conv</td>
<td>1.2</td>
<td>550 fps</td>
</tr>
<tr>
<td>Additional Vectorization</td>
<td>1.7</td>
<td>777 fps</td>
</tr>
<tr>
<td>External Memory Opt</td>
<td>1.9</td>
<td>900 fps</td>
</tr>
</tbody>
</table>

Table 3. Optimization Impact on GoogLeNet.

† This optimization is a work in progress and will be added to the paper if completed before the deadline.
LSTM Cell Implementation

LSTM cells are a widely-used variant of RNNs, commonly used in speech recognition [3], translation [18] and motion detection [16]. The DLA is designed to be a flexible neural network accelerator for all relevant deep learning workloads, including LSTM-based networks. This section discusses how the graph compiler mutates an LSTM cell to map it to the DLA overlay yet achieve high performance.

Mapping an LSTM Cell to DLA

Most of the computation in an LSTM cell occurs in eight matrix multiplications to compute the three LSTM gates (input/forget/output) [11]. We combine those eight matrices into one large matrix (Figure 12), reducing the DLA execution time from 12 subgraphs to a single subgraph at least \( \approx 12 \times \) faster.

We end up with two large matrix multiplications, one matrix multiplication sharing the same input vector.

Depending on LSTM cell size, our compiler may later decide to slice the matrix if it does not fit on the FPGA. We choose to combine matrices, only to slice them again if they are too large, because the slicing algorithm is architecturally aware, and optimized to the DLA vectorization parameters and buffer sizes, whereas the separate matrices may not map well to the existing architecture.

With the combined matrix, each of the LSTM gates are computed one after another since matrix rows are computed in order. However, this is not FPGA friendly, as the input/forget/output gate values need to be buffered in either expensive on-chip RAM or slow external memory, to be combined in the second half of the LSTM cell. However, by interleaving the rows of the large matrix-the first row contains filters for the input gate, the second for the 'g' gate, the third for the forget gate, and the fourth for the output gate-we can compute one output from each gate in each time step (Figure 13). This removes the need for buffering large intermediate gate outputs [13], and allows us to directly stream the gate values into the dedicated LSTM hardware block (Figure 14).

This demonstrates the flexibility of the DLA overlay, and the power of our graph compiler in implementing different neural networks. By simply attaching the LSTM kernel to the Xbar, we can leverage our powerful multi-precision PE array to compute the matrix multiplication of the LSTM cell, then stream the data directly into the dedicated LSTM block.
External-Memory-Bound RNNs

Non-convolutional neural networks are effectively a matrixvector multiplication computed with batch=1. Most applications using RNNs are real-time applications such as speech/gesture recognition or translation; therefore, they require low-batch and low-latency processing that is ideal for FPGAs. However, external memory bandwidth is often a bottleneck, since a large matrix has to be fetched from external memory, only to be multiplied with one vector. Compute time is lower than memory fetch time so it is impossible to hide memory fetch latency. Intel Stratix® 10 devices have 2-HBM2 devices integrated on some boards, providing up to 500 gigabytes per second (GBps) peak memory bandwidth, 20X higher than DDR4 SDRAM 2400.† Looking forward, we modeled the performance of a 4-layer stacked LSTM neural network with input=output=hidden=2048 used for speech recognition. Adding more external memory bandwidth, going from DDR4 SDRAM to HBM2, decreased speech processing latency by more than 5X (Figure 15).†

Conclusion

We presented a methodology that achieves software ease of use with hardware efficiency by implementing a domain-specific customizable overlay architecture. We described the hardware trade-offs involved with neural network acceleration, and delved into our graph compiler that maps neural networks to our overlay. Using our hardware and software methodology we achieved a 4X improvement on ResNet-101 HD, 12X on LSTM cells, and 900 fps on GoogLeNet on Intel Arria 10 FPGAs. We will continue to develop the DLA to encompass more use cases such as deployment of multi-FPGAs [2]. In the future, we aim to implement similar overlays for application domains, such as genomics, packet processing, compression, and encryption making FPGAs more accessible for high-throughput computation.†

Figure 15. Latency of an LSTM Neural Network When Varying External Memory Bandwidth.
References


[13] Y. Guan et al. FPGA-Based Accelerator for Long Short-Term Memory Recurrent Neural Networks. ASP-DAC. 2017


