Argus: An End-to-End Framework for Accelerating CNNs on FPGAs

Yongming Shen, Tianchu Ji,
Michael Ferdman, and Peter Milder
Stony Brook University

***Abstract*—In this article, we present Argus, an end-to-end framework for accelerating convolutional neural networks (CNNs) on field-programmable gate arrays (FPGAs) with minimum user effort. Argus uses state-of-the-art methods to auto-generate highly optimized CNN accelerator designs for FPGAs, and includes software for running an FPGA-backed CNN inference microservice.**

The machine learning revolution enabled by deep neural networks has transformed the landscape of information technology in recent years. At the forefront of this are convolutional neural networks (CNNs),[1–4] deep learning networks that are primarily used to solve computer vision challenges.

Unfortunately, the success of CNNs is accompanied by immense computational costs. Central processing units (CPUs) are neither fast nor efficient enough for running modern CNNs. GPUs offer impressive performance, but their power-hungry nature limits their deployment. Application-specific integrated circuits (ASICs) have the potential to achieve the best performance and energy efficiency, but also have prohibitive development costs. Moreover, given the pace at which CNNs have evolved in recent years, ASIC development risks becoming obsolete before turning profitable. In contrast to these options, field-programmable gate arrays (FPGAs) offer a unique balance between performance, energy efficiency, and flexibility.

The main obstacle for using FPGAs to accelerate CNNs is the effort required to

Digital Object Identifier 10.1109/MM.2019.2930607
Date of publication 23 July 2019; date of current version 10 September 2019.

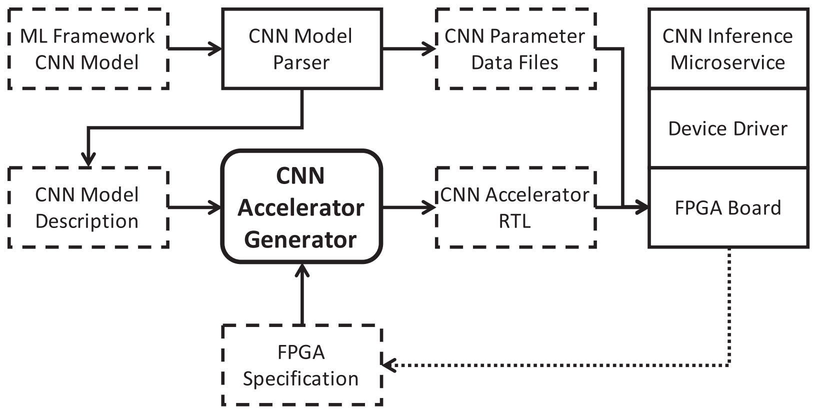

Figure 1. Overview of the Argus end-to-end CNN acceleration framework.

develop RTL designs to run on an FPGA. The majority of software developers and machine learning practitioners do not have the expertise necessary to program an FPGA, a skill set that takes years to acquire. Even for an experienced RTL designer, implementing a high-performance CNN accelerator on an FPGA can take from months to more than a year. Moreover, when the target CNN and/or FPGA changes, a major effort is needed to port the RTL design to the new target. To address this problem, we developed Argus, an end-to-end framework for accelerating CNNs on FPGAs. The end goal of Argus is for users with no RTL design knowledge to take advantage of FPGAs to accelerate CNNs. The key challenge in achieving this goal is the automatic generation of CNN accelerator RTL designs. For this, Argus follows the resource partitioning CNN accelerator optimization strategy[5] and takes advantage of the Medusa memory interconnect[6] to make sure that the autogenerated RTL designs make optimized use of an FPGA's compute, on-chip storage, and off-chip bandwidth resources.

Figure 1 presents the workflow of the Argus framework. Argus includes a CNN model parser, an accelerator generator, a Linux device driver, and a network-accessible CNN inference microservice. The model parser takes a target CNN model from a supported deep learning framework as input and produces a model description for the accelerator generator. Additionally, the model parser extracts model parameters (e.g., CNN filter weights) from the target CNN model and packs them into data files for the FPGA accelerator. The accelerator generator takes a CNN description and target FPGA specification as inputs, and generates the RTL design of a CNN accelerator as output. This is the core of the Argus framework, as it is what enables Argus to shield users from the complexity of RTL development, which is the key to a complete end-to-end experience. The device driver and the CNN inference microservice constitute the software stack, which provides end-user applications easy access to the FPGA-based autogenerated CNN accelerator.

BACKGROUND

To establish common terminology and notation for the remainder of this paper, we consider an image classification CNN (e.g., the work presented by Krizhevsky *et al.*[1]) that passes images through a sequence of convolutional *layers*. Each convolutional layer convolves input feature maps with filters to produce output feature maps. The filters of a convolutional layer contain weights that were previously learned using an algorithm such as stochastic gradient descent. Nonlinear layers, which typically perform computations such as subsampling or activation functions, interleave convolutional layers. In the end, the network includes one or more fully connected layers, each of which performs dot products across its entire input.

Listing 1 presents the pseudocode which computes a convolutional layer. The shape of a convolutional layer is defined by the number of input feature maps (N) and output feature maps (M), the output feature map height (R) and width (C), the filter size (K), and the convolution stride (S). A layer takes as input N input feature maps of dimensions $((R-1)\times S+K)\times((C-1)\times S+K)$ and convolves them with M sets of filters; by convolving one set of N filters ($N\times K\times K$ weights) with the input feature maps, one of the M output feature maps is obtained. Each of the M output feature maps is computed by repeating this process with each of the M sets of filters. Although omitted from Listing 1 for clarity, output feature maps are initialized with trained bias values.

Listing 1. Pseudocode of a convolutional layer

```
I[N][(R-1)*S+K][(C-1)*S+K] //input maps
O[M][R][C] //output maps
W[M][N][K][K] //weights
for (m=0; m<M; m++)
  for (n=0; n<N; n++)
    for (r=0; r<R; r++)
      for (c=0; c<C; c++)
        for (i=0; i<K; i++)
          for (j=0; j<K; j++)
            wx=W[m][n][i][j]
            ix=I[n][S*r+i][S*c+j]
            O[m][r][c]+=wx*ix
```

FPGA-Based CNN Acceleration

Among the many different options for accelerating CNNs, FPGAs stand out due to their energy efficiency and flexibility. The flexibility of FPGAs also naturally presents multiple different ways to accelerate CNNs.

The traditional approach to use an FPGA is to manually craft a specialized RTL design for a target CNN and FPGA pair. The advantage of this approach is that the design can be finely optimized for the target CNN and specific FPGA hardware. However, creating an efficient RTL design for a CNN requires advanced hardware design skills that most CNN end users do not have. Moreover, even after making the significant investment in time and effort to produce an RTL design, the result is extremely inflexible. Supporting a different CNN or targeting a different FPGA chip requires a major RTL redesign effort.

An alternative approach is to develop an ASIC-style RTL design of a specialized programmable CNN accelerator and implement it on an FPGA. Because the accelerator is programmable, the RTL design need not be changed to execute different CNNs on the same system. Although practical, this approach cannot take advantage of the greatest strength of FPGAs while paying all the costs of FPGA implementation. Instead of being maximally efficient from specializing for just one target CNN, the programmable accelerator must accommodate a large variety of CNNs, incurring performance and energy overheads. In some sense, the cost of programmability is paid twice: once in the FPGA fabric and again in the accelerator design. At the same time, compared to an ASIC implementation, the FPGA implementation pays a price in clock rate and compute density. Furthermore, the RTL design must still be manually updated and optimized for every target FPGA chip. Without such changes, either the accelerator design will not fit (if the new FPGA target is smaller) or resources will be wasted and performance sacrificed (if the new FPGA is larger).

Another potential approach relies on high-level synthesis (HLS) tools to generate a specialized RTL design for a target CNN and FPGA pair from a high-level specification (e.g., C++ or OpenCL). With the help of HLS tools, the effort required to update an RTL design as the target CNN and/or FPGA changes is smaller, and the process of modifying the code is more accessible to end users. However, based on our experience with the currently available HLS tools, achieving the performance of manually crafted RTL designs requires the user of the HLS tools to possess a high level of hardware design knowledge, and the process still consumes significant time and effort.

Noting the challenges and drawbacks of the above approaches, we take a different approach for Argus. Our goal is to retain peak efficiency of the resulting accelerators, provide rapid portability across CNN models and FPGA chips, and remove the burden of FPGA programming from the end user. We do this by developing an accelerator generator, which takes a CNN model and FPGA specification as its inputs and produces an accelerator RTL design specialized for the given CNN and target FPGA. If the target CNN and/or FPGA changes, the generator produces a new RTL design, yielding a highly specialized implementation that matches the performance of manually crafted designs, without manual RTL development or porting effort.

ARGUS OVERVIEW

Argus is an end-to-end framework for accelerating CNNs on FPGAs, centered around a flexible CNN accelerator generator (see Figure 1). The accelerator generator is implemented using a combination of Bluespec, C++, Python, and Verilog. The “Generating FPGA CNN Accelerators” section highlights some of the key aspects of the accelerator generator design.

To provide an end-to-end user experience, Argus includes a model parser to import CNN models from the PyTorch deep learning framework. Internally, Argus uses a framework-agnostic CNN representation that can be easily extended to support other deep learning frameworks. As a part of processing the target model description, Argus performs a sequence of transformations such as merging batch normalization layers into convolutional layer weights and biases, grouping chains of operations (e.g., convolution, max pooling, summation, ReLU) into tasks that can be assigned to a processing core in the accelerator, and rewriting the filter weights and biases into a form usable by the generated accelerator hardware.

To facilitate interaction with the CNN accelerators from a high-end host server system over PCIe, Argus provides a streamlined Linux device driver compatible with the generated CNN accelerators and a CNN inference microservice. Applications that use the accelerator send CNN inputs to the inference microservice and receive CNN outputs as replies. The inference microservice interacts with clients using the ZeroMQ library, servicing client requests over local IPC or over the network.

GENERATING FPGA CNN ACCELERATORS

The CNN accelerator generator forms the core of Argus. Internally, the accelerator generator includes a design optimizer and a library of handcrafted parameterized RTL components. The optimizer uses the requirements and constraints of the target CNN and FPGA platform to produce an accelerator RTL design by mixing, matching, and parameterizing the available library components.

To produce peak performance efficient designs, the accelerator generator must take full advantage of all available FPGA compute resources (DSP slices), usefully incorporating as many DSP slices into the generated RTL design as possible; we refer to this as *static utilization*. Just as critically, at runtime, the proportion of cycles during which DSP slices perform useful work must be maximized; we refer to this as *dynamic utilization*. In addition to the FPGA compute resources, the generator must optimize the use of the on-chip storage capacity and off-chip memory bandwidth to ensure that the system is not bottlenecked by off-chip data transfer. Simultaneously achieving high static and dynamic utilization requires Argus to explore a vast space of candidate designs that integrate the latest developments in CNN hardware acceleration.

Multi-CLP Optimizer

CNNs comprise multiple computation layers, whose inputs are arrays of various dimensions. A traditional way to implement a CNN in an RTL design is to build a single large processing core, a convolutional layer processor (CLP), which executes the CNN layers sequentially. Following this approach, an accelerator generator can combine a CLP template and an optimizer to select the CLP template parameters for the target CNN and FPGA combination.

Critically, CLP parameters that are optimal for one layer are often suboptimal for the other layers of a CNN. As such, jointly optimizing one CLP for all CNN layers leads to a dynamic underutilization of the FPGA resources, giving up performance. Figure 2 (top) illustrates this scenario. A single-CLP (white box) iteratively processes the three layers (blue boxes). The dimensions of the CLP and the layers are represented by the size and shape of the boxes. L1 is smaller than the CLP dimensions, resulting in low dynamic utilization because some hardware remains unused when computing this layer [see Figure 2(a)]. The dimensions of L2 exactly match the CLP and the layer is processed efficiently. However, L3 dimensions exceed the CLP dimensions, requiring the CLP to be used iteratively to compute different parts of L3 (first, its top portion, then, its bottom portion), again underutilizing the available hardware [see Figure 2(b)].

To avoid this problem, Argus uses the multi-CLP methodology.[5] A multi-CLP design partitions FPGA resources among multiple CLPs, which concurrently operate on images in a pipelined fashion. We illustrate the operation of multi-CLP in Figure 2 (bottom), where the hardware resources are partitioned among two smaller CLPs. The two

CLPs are specialized and have different dimensions; this allows CLP1 to work well for L1 and L3, while the dimensions of CLP2 are compatible with L2. The different CLP dimensions match the layer requirements, maximizing dynamic utilization, with multi-CLP designs performing the same amount of work in less time compared to single-CLP [see Figure 2 (c)]. Using multiple CLPs, with each CLP specialized for a subset of layers instead of all the layers, a multi-CLP design outperforms a single-CLP design when using the same FPGA resources.

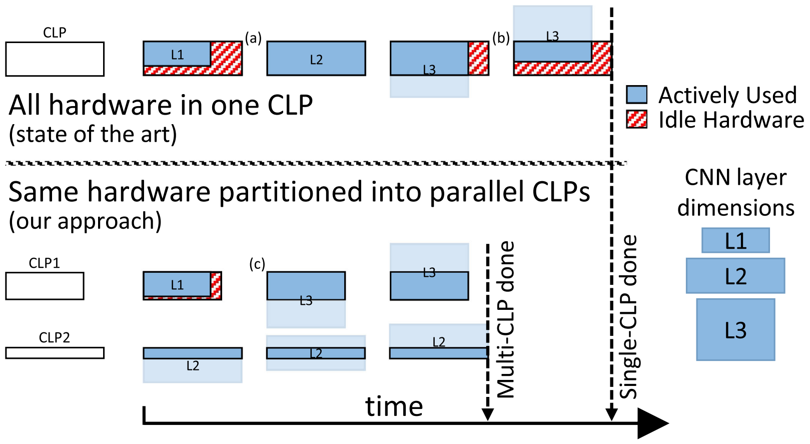

Figure 2. Operation of CLPs on a three-layer CNN. The multi-CLP approach uses the same total hardware resources as the single-CLP. However, the multi-CLP partitioned hardware closely matches the CNN layers, minimizing idle hardware and improving performance.

Table 1 compares the dynamic utilization of the DSP slices in single-CLP and multi-CLP systems across 32 different designs, spanning two target FPGAs (Xilinx Virtex-7 485T and 690T), two data types (32-bit floating point and 16-bit fixed point), and four target CNNs. All designs target an 80% static utilization of the FPGAs (by providing the accelerator generator with the corresponding budgets), as some FPGA resources must remain unused to allow for timing closure when using FPGA place-and-route tools. The results indicate that multi-CLP designs achieve better dynamic DSP slice utilization than single-CLP designs in all cases. The smallest improvement (1.01×) is observed when targeting VGGNet-E because the layers of VGGNet-E have regular dimensions, limiting opportunity for CLP specialization. The best improvement (3.8×) is observed when targeting AlexNet because the layers of AlexNet have significantly varying dimensions. Overall, Table 1 shows that multi-CLP adapts better to CNNs with irregular layer dimensions.

Virtex-7 485T and 690T are representative of midsized FPGAs. In data center environments, larger FPGAs are more common. Figure 3 compares how well single-CLP and multi-CLP designs scale as the size of the target FPGA increases (with DSP slice budgets ranging from 100 to 10,000). For this experiment, all designs target AlexNet and use 32-bit floating point as the data type. The *x*-axis shows the number of DSP slices used for each configuration. Dashed vertical lines illustrate the total number of DSP slices available on the Virtex-7 485T, Virtex-7 690T, Virtex UltraScale+ 9P, and Virtex UltraScale+ 11P FPGAs. As the number of available DSP slices increases, the throughput difference between the single- and multi-CLP designs grows. For example, going from 2240 to 9600 DSP

Table 1. Dynamic utilization of DSP slices in competing designs.

	AlexNet[1]	VGGNet-E[2]	SqueezeNet[3]	GoogLeNet[4]
485T (float)				
Single-CLP	74.1%	96.8%	78.0%	81.9%
Multi-CLP	95.4%	97.5%	95.8%	96.9%
690T (float)				
Single-CLP	65.4%	96.0%	76.4%	78.1%
Multi-CLP	99.0%	98.7%	96.7%	96.0%
485T (fixed)				
Single-CLP	31.0%	89.7%	51.1%	50.2%
Multi-CLP	93.9%	97.3%	93.6%	93.8%
690T (fixed)				
Single-CLP	23.7%	88.3%	42.0%	44.0%
Multi-CLP	90.6%	96.1%	93.1%	89.3%

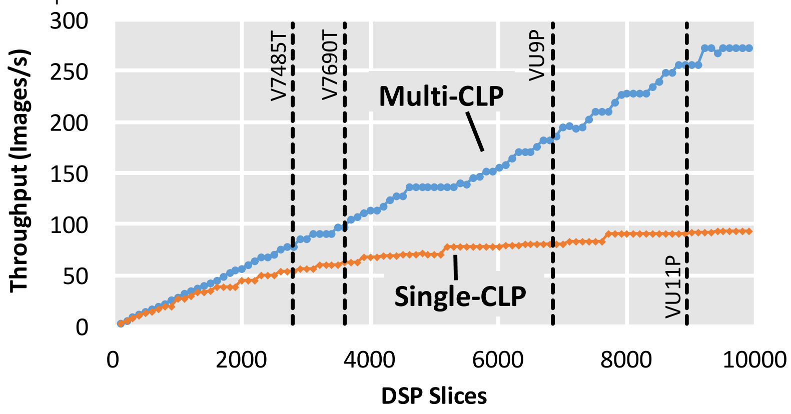

Figure 3. Throughput at 100 MHz for AlexNet on multi-CLP and single-CLP 32-bit floating point designs as a function of available DSP slices.

slices, the multi-CLP improvement over single-CLP designs increases from 1.3× to 3.3×.

The benefit of multi-CLP over single-CLP comes at the cost of design optimization complexity, with the multi-CLP design space being orders of magnitude larger than the single-CLP design space. For single-CLP, only the parameters of one CLP are determined by the optimizer. For multi-CLP, the optimizer must generate candidate partitioning schemes that decide on the number of CLPs, the distribution of layers among these CLPs, and the fraction of FPGA resources to assign to each CLP. Then, for each candidate partitioning, every CLP must go through the single-CLP optimization to determine the best CLP parameters for running the layers assigned to it. Fortunately, a combination of dynamic programming techniques and design space pruning heuristics bring the optimization time for multi-CLP to under an hour for even the largest designs (ResNet50[7] on Virtex UltraScale+ 9P).

Parameterized CLP Templates

The purpose of a CLP is to execute the convolutional layer computation presented in Listing 1. To leverage the parallelism within a convolutional layer and take advantage of the on-chip buffers to reduce off-chip data transfer, the Argus CLP template uses a computation and data transfer schedule based on the work of Zhang *et al.*,[8] which applies loop reordering, tiling, and unrolling to yield the pseudocode shown in Listing 2, pictorially depicted in Figure 4.

In this CLP design, the I_{buf}, O_{buf}, and W_{buf} arrays correspond to on-chip buffers for input, output, and weight data, respectively. Copying data in or out of these arrays corresponds to transferring data between the on-chip buffers and the off-chip memory. Double buffering is used to overlap data transfer with computation. The R, C, M, and N loops are tiled with factors T_r, T_c, T_m, and T_n, respectively. These loop tiling factors, which are determined by the

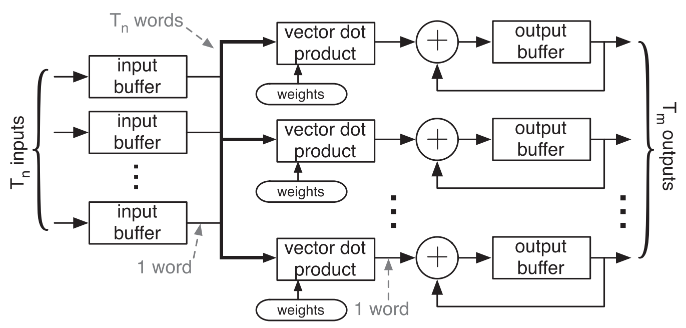

Figure 4. Diagram of a CLP datapath. Each dot-product unit takes T_n inputs and T_n weights and produces one output.

Listing 2. Pseudocode for loop tiling in a CLP

```
I[N][(R-1)*S+K][(C-1)*S+K] //input maps
O[M][R][C] //output maps
W[M][N][K][K] //weights
Ibuf[Tn][(Tr-1)*S+K][(Tc-1)*S+K]
Obuf[Tm][Tr][Tc]
Wbuf[Tm][Tn][K][K]
for (r=0; r<R; r+=Tr)
 for (c=0; c<C; c+=Tc)
  for (m=0; m<M; m+=Tm) {
   for (n=0; n<N; n+=Tn) {
   irx=r*S:(r+Tr-1)*S+K
   icx=c*S:(c+Tc-1)*S+K
   Ibuf=I[n:n+Tn][irx][icx]
   Wbuf=W[m:m+Tm][n:n+Tn]
   for (i=0; i<K; i++)
    for (j=0; j<K; j++)
     for (tr=0;tr+r<min(R,r+Tr);tr++)
      for (tc=0;tc+c<min(C,c+Tc);tc++)
       for (tm=0; tm<Tm; tm++) #UNROLL
        for (tn=0; tn<Tn; tn++) #UNROLL
         wx=Wbuf[tm][tn][i][j]
         ix=Ibuf[tn][S*tr+i][S*tc+j]
         Obuf[tm][tr][tc]+=wx*ix
  }
  O[m:m+Tm][r:r+Tr][c:c+Tc]=Obuf
 }
```

optimizer, control the amount of data transferred per buffer refill or write-out, and the order in which the data are transferred. The inner-most two loops are unrolled (based on T_m and T_n), which represents T_m vector dot-product units, each of width T_n, and each followed by an accumulator.

In addition to the convolutional layers, a CLP also needs to process a variety of other types of layers. For state-of-the-art CNNs, these include fully connected layers, max-pooling layers, summation layers, batch normalization layers, and activation (ReLU) layers. Supporting fully connected layers does not require hardware changes because a fully connected layer can be seen as a special case of a convolutional layer ($R = C = K = 1$). Batch normalization layers can also be supported without hardware changes because, for inference, a batch normalization layer can be absorbed into its adjacent convolutional layer. A ReLU activation layer changes negative numbers to zeros, which is trivially implemented as part of transferring results from the on-chip output buffers to off-chip memory. When max-pooling and summation layers occur in a CNN, they always follow convolutional layers. Correspondingly, Argus supports both with small hardware units placed adjacent to the CLPs. The T_m CLP output buffers in Figure 4 are connected directly to T_m comparators; the comparator results go to T_m adders, which write into the max-pooling or summation output buffers. T_m additional buffers are included for the summation layers to provide input data. Fusing convolutional layers with max-pooling and summation layers eliminates the off-chip data transfer of the intermediate data.

One noteworthy feature of Argus is that the multi-CLP optimization method does not dictate the design of the CLP template used. Additional CLP templates can be incorporated into Argus, selecting the best design based on the characteristics of the target CNN. For example, if the target CNN is bottlenecked by off-chip data transfer, a CLP template like Escher[9] can be used. In Listing 2, the output buffer represented by $Obuf$ is partitioned into T_m banks, where each bank holds data for one output channel. Escher generalizes this to let each output buffer bank hold data for more than one output channel. The data in the input buffers are reused to compute more output channels before being replaced, reducing the input retransfers from off-chip memory. Escher buffer management also includes support for batch processing, which further reduces the bandwidth of off-chip weight transfer. Given these additional degrees of freedom in buffer use, Escher can reduce the peak off-chip bandwidth requirements of a CLP by $10.5\times$ for some CNNs.

Moreover, Argus can simultaneously incorporate multiple CLP templates in a multi-CLP design. For example, early convolutional layers of a CNN generate far more intermediate data than later layers. For CLPs responsible for the early layers, more sophisticated CLP designs such as layer fusion[10] can reduce off-chip intermediate data transfer. Due to the all-to-all connection among the input and output channels within the convolutional layers, naïvely chaining multiple convolutional layers together would be impractical, as it would require all intermediate data to fit on-chip. Instead, layer fusion considers a stack of convolutional layers to be a single nested loop and applies loop transformations to reorder the computation and data transfer, drastically reducing the on-chip storage requirements while entirely eliminating the off-chip data transfer of intermediate data.

Medusa Memory Interconnect

In addition to generating the CLPs, Argus must generate the interconnect between the CLPs and off-chip memory. For maximum efficiency in the allocation of memory bandwidth to each CLP and for flexibility in the choice of T_n and T_m for each CLP, multi-CLP designs require many independent ports of word-width size, with each port delivering data to one or more CLP input buffers, or draining data from one or more CLP output buffers. However, because FPGA clock frequencies are lower than the frequency of modern high-speed memory channels, FPGA memory controllers use a wide bus to expose the full DRAM bandwidth to the FPGA logic (e.g., a 12.8-GB/s DDR3 channel is connected with a 512-bit 200-MHz bus). As a result, the CNN accelerator interconnect must efficiently multiplex a wide memory channel across many independent narrow CLP ports, where the aggregate bandwidth of

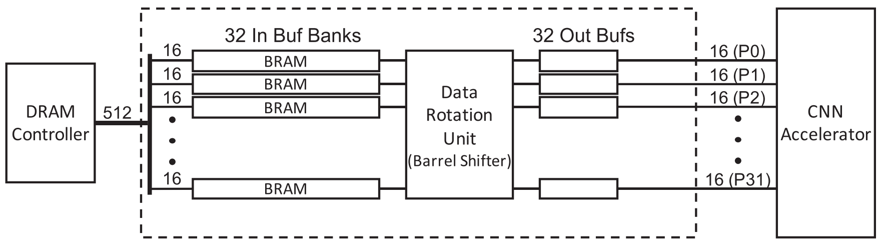

Figure 5. Medusa interconnect block diagram.

the narrow ports equals the available off-chip memory bandwidth.

Using a conventional approach, the FPGA memory interconnect alone can account for more than 20% of the LUTs and FFs of the entire accelerator. Worse yet, the interconnect dominates the critical path, severely limiting the overall accelerator clock frequency. To overcome these limitations, Argus includes Medusa,[6] a specialized on-chip network generator specifically tailored for high-performance interconnection between the wide off-chip memory interface of the FPGA and the large number of narrow input, weight, and output ports of the CLPs.

The efficiency of Medusa comes from its unique data transfer network design, illustrated in Figure 5. For memory reads, each 512-bit word from the DRAM controller is destined to one of the 32 narrow ports. Medusa buffers data from the DRAM controller in an input buffer which has the same width as the DRAM controller interface. The input buffer is divided into banks, with one bank per narrow output port. Each data word destined to a given output port is spread across all input buffer banks. On the output side, Medusa contains as many output buffers as the number of narrow ports, with each output buffer feeding data to one of the narrow CLP ports. Using a data rotation unit, Medusa transposes the data in transit from the input buffer to the output buffers. In this way, the data destined for an output port and spread across all the input buffer banks all ends up, in the correct order, in the output buffer of the destination port.

Although FPGAs are an excellent fit for accelerating CNNs from the performance and energy efficiency perspectives, their use is limited by the difficulty of FPGA programming. To address this problem, we developed Argus, and end-to-end framework for accelerating CNNs on FPGAs. The core of Argus is an accelerator generator which takes a CNN model and FPGA specification as inputs and produces highly optimized CNN accelerator RTL designs.

Because all the buffers in Medusa are deep and narrow, they are efficiently implemented with BRAMs. The data rotation unit is implemented using a barrel shifter, which is resource efficient and scalable. For an example Argus CNN accelerator, Medusa multiplexes a 512-bit DRAM controller interface across 32 16-bit read ports and 32 16-bit write ports using 4.7× fewer LUTs and 6.0× fewer FFs than traditional interconnects, while improving clock frequency by 1.8×. For a similar accelerator with a 1024-bit DRAM controller interface, Medusa runs at 225 MHz, while routing congestion limits a traditional interconnect to under 25 MHz.

CONCLUSION

Although FPGAs are an excellent fit for accelerating CNNs from the performance and energy efficiency perspectives, their use is limited by the difficulty of FPGA programming. To address this problem, we developed Argus, and end-to-end framework for accelerating CNNs on FPGAs. The core of Argus is an accelerator generator which takes a CNN model and FPGA specification as inputs and produces highly optimized CNN accelerator RTL designs. For a complete end-to-end user experience, Argus includes a model parser to accept CNN models from popular deep learning frameworks, a Linux device driver for controlling the FPGA CNN accelerator, and a microservice server for handling CNN inference requests. The Argus accelerator generator leverages the multi-CLP[5] methodology to achieve both high static and dynamic utilization of FPGA compute resources. The modular design of the system enables incorporating advanced CLP designs such as Escher[9] and layer fusion[10] for some or all of the CLPs. To meet the memory access demands of the resulting high-performance accelerators, Argus includes the specialized Medusa[6] memory interconnect that efficiently multiplexes the wide FPGA memory interface to many narrow independent memory ports. By combining these techniques in an automated system, Argus enables users to efficiently run CNNs on FPGAs without the complexity of RTL development.

REFERENCES

1. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, vol. 1, pp. 1097–1105.
2. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.
3. F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size," *arXiv:1602.07360*.
4. C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.
5. Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 535–547.
6. Y. Shen, T. Ji, M. Ferdman, and P. Milder, "Medusa: A scalable memory interconnect for many-port DNN accelerators and wide DRAM controller interfaces," in *Proc. 28th Int. Conf. Field Programmable Logic Appl.*, 2018, pp. 101–105.
7. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
8. C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. 23rd ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 161–170.
9. Y. Shen, M. Ferdman, and P. A. Milder, "Escher: A CNN accelerator with flexible buffering to minimize off-chip transfer," in *Proc. 25th IEEE Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2017, pp. 93–100.
10. M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 22:1–22:12.

Yongming Shen is currently working toward a PhD in computer science at Stony Brook University. His research interests are in the area of computer architecture, with a focus on the design of hardware accelerators for deep learning applications. He has an MEng from the South China University of Technology. He is a Student Member of the IEEE. Contact him at yoshen@cs.stonybrook.edu.

Tianchu Ji is currently working toward a PhD in electrical and computer engineering at Stony Brook University. His research focuses on FPGA accelerators for trending deep learning algorithms such as CNN and RNN. He has a BEng from the Huazhong University of Science and Technology, China. He is a Student Member of the IEEE. Contact him at tianchu.ji@stonybrook.edu.

Michael Ferdman is an associate professor of computer science at Stony Brook University where he directs the Computer Architecture Stony Brook (COMPAS) Lab. His research interests are in the area of computer architecture, with emphasis on the design of server systems. He has a PhD from Carnegie Mellon University. His work has received multiple best paper awards and nominations, with three of his papers selected as IEEE Micro Top Picks in Computer Architecture. He is a Senior Member of the IEEE. Contact him at mferdman@cs.stonybrook.edu.

Peter Milder is an associate professor of electrical and computer engineering at Stony Brook University. His research focuses on domain-specific languages and tools for automatic hardware generation, FPGA accelerators, and hardware for machine learning and signal processing. He has a PhD in electrical and computer engineering from Carnegie Mellon University. He is a Senior Member of the IEEE. Contact him at peter.milder@stonybrook.edu.