Software–Hardware Codesign for Efficient Neural Network Acceleration

Kaiyuan Guo
Tsinghua University and DeePhi

Song Han
Stanford University and DeePhi

Song Yao
DeePhi

Yu Wang
Tsinghua University and DeePhi

Yuan Xie
University of California, Santa Barbara

Huazhong Yang
Tsinghua University

Designers making deep learning computing more efficient cannot rely solely on hardware. Incorporating software-optimization techniques such as model compression leads to significant power savings and performance improvement. This article provides an overview of DeePhi's technology flow, including compression, compilation, and hardware acceleration. Two accelerators achieve extremely high energy efficiency for both client and datacenter applications with convolutional and recurrent neural networks.

Current AI keywords include "deep learning"and "neural network." Deep learning is showing dominant performance in applications such as image classification[1] and speech recognition,[2] which makes it the top candidate for real-world AI applications. However, today's computational efficiency is still not enough, and the computational complexity of neural networks far exceeds traditional computer vision algorithms, so we cannot employ deep learning for many cases. To address this problem, researchers around the world have been working on customized hardware acceleration solutions.[3] There will be an unprecedented battle for deep learning hardware.

We believe that, to build an efficient system for deep learning, we must consider software–hardware codesign, because software and hardware are coupled in deep learning. Considering both optimization in software and hardware, we propose a new design flow (see Figure 1). Three factors affect how to efficiently compute deep learning algorithms: workload, peak performance, and efficiency.

A smaller workload with the same precision is always welcome. However, changing the workload can affect the hardware design. For example, replacing direct 2D convolution with a fast algorithm—for example, Winograd—in a convolutional neural network (CNN) changes the ratio between multiplication and addition and also changes the data access pattern. Furthermore, exploring the sparsity in neural networks changes even the data description format and the entire computing system—that is, from dense to sparse matrices.

A higher peak performance is always wanted. However, because peak performance is usually proportional to the computation

Published by the IEEE Computer Society

unit number and system frequency, a higher peak performance often results in higher cost and power. One way to increase the peak performance while lowering the cost is to simplify the operation—for example, by using fewer bits to represent data and weight in neural networks. The robustness of deep learning algorithms makes it possible to use 16-bit, 8-bit, and even fewer-bit fixed-point operations to replace 32-bit floating-point operations while introducing negligible accuracy loss. This tradeoff between peak performance and variable precision influences both algorithm and hardware design.

Efficiency reflects how well we use the computation units. An elegant memory system design to feed the computing units with enough data is the key to high efficiency. To achieve this, we need to tackle both on-chip memory and external memory system design. For the on-chip memory part, it is necessary to explore data locality and data reuse to make data stay in the cache as long as possible. For the external memory part, increasing the bandwidth helps increase the efficiency but also leads to higher cost and power. With the same theoretical bandwidth, we need to increase the burst length to fully utilize it—that is, organize data storage to match hardware requirements. The data simplification method also reduces the data-bit width and thus reduces the bandwidth requirement.

Taking all three factors into account helps us design a highly efficient deep learning system. Furthermore, because deep learning is evolving rapidly, taping out a certain design might not be a good choice for a commercial product. In this case, general-purpose processors or a specialized hardware with enough flexibility and change for reprogramming are preferable. Field-programmable gate arrays (FPGAs), with their inherent reconfigurability, let us explore all the three levels of the design and incorporate state-of-the-art deep learning techniques into a product within a short design time. Thus, FPGAs have the potential to become a mainstream deep learning processing platform.

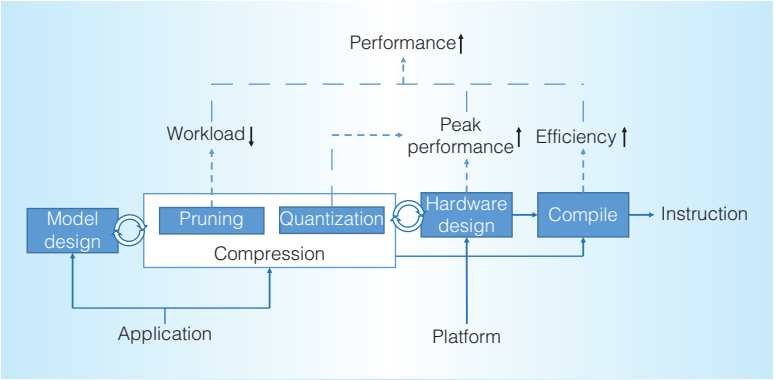

Figure 1. Our proposed design flow.

From Model to Instructions

No standard state-of-the-art neural network model exists. For CNNs, early models first applied several convolution (Conv) layers sequentially to the input image to generate low-dimension features, and then several fully connected layers as the classifier. Current networks, such as ResNet[1] and the inception module in GoogLeNet,[4] used different branches and parallel layers in the network to achieve multiscale sampling and avoid vanishing gradients. The model size ranges from fewer than 10 layers to more than 100 layers for different tasks. For recurrent neural networks (RNNs), there are also many variants, such as long short-term memory (LSTM), gated recurrent units (GRUs), bidirectional RNNs used in speech recognition,[2] and sequence-to-sequence learning used in neural machine translation (NMT).[5]

A system must be flexible enough to execute different neural network models. To achieve this, a flexible description is necessary. Caffe, TensorFlow, and other deep learning frameworks provide efficient interfaces on CPU and GPU platforms. However, for specialized systems, we need a tool and an intermediate representation to bridge these frameworks and the hardware. We design the customized hardware considering the patterns of neural network computation to achieve high efficiency while leaving the interface flexible. In this way, we can map different networks onto it. Meanwhile, algorithm researchers and hardware developers can work simultaneously, making the iteration of products fast and efficient.

We implement an instruction interface for our hardware. For CPUs or GPUs, the instructions are fine grained, usually with scalar- or vector-level operations. Fine-grained

instruction is highly flexible, but considering the specialty of neural networks, this might not be an efficient interface. For example, the neural network computation is usually full of loops, thus we try to partition the loops into small blocks such that each block can be done by hardware. For a CNN, each block can be a set of 2D convolutions, whereas for RNN, each block can be vector-matrix multiplications. The operations for each block can be represented by one instruction, which reduces the instruction size while maintaining the hardware efficiency. We also use instructions to describe data transfers between on-chip cache and off-chip memory, which lets the compiler do static scheduling to achieve a balance between the computation and I/O.

Consider the design flow shown in Figure 1. First, the deep learning algorithm is designed for the target application. For this design flow, the main target is to design the neural network model. Then, the model is optimized to be ready for hardware acceleration. This step usually includes model compression and data quantization to reduce the workload and increase the peak performance of the hardware design. Both of these steps are done by automatic tools, but developers need to choose the best decision, considering the accuracy loss and hardware performance gain. Next, the hardware is designed according to the optimization strategy used. These three steps are done iteratively to ensure that the target application's requirement is met. After hardware design, we use a customized compiler to convert the neural network model to instructions to be executed at runtime. Further optimization on scheduling is automatically done in the compiler to increase the hardware efficiency.

Aristotle: The CNN Accelerator

CNNs are widely used for image and video processing. One of the most popular a CNN applications is object detection. But a CNN's high computation complexity makes it an impractical choice for mobile platforms such as smartphones or drones. To solve this problem, we designed the Aristotle architecture for energy-efficient CNN acceleration.

CNNs mainly comprise several convolution layers. Within each layer, there are n input feature maps $N_i(x, y)$ and m output feature maps $M_j(x, y)$. Each feature map is a 2D image. Equation 1 describes the computation within one convolution layer. The * denotes 2D convolution operation, and W is the convolution kernel. The bias for each output feature map is b_j. Function f is a nonlinear function on each pixel (for example, ReLU or sigmoid).

$$M_j = f\left(\sum_{i=1}^{n} W_{ij} * N_i + b_j\right) \quad (1)$$

CNNs also use pooling layers for downsampling on the feature maps, usually max pooling or average pooling. With pooling layers, the size of feature maps is reduced. This helps increase the reception field of each neuron (pixel) in the feature map. Thus, larger features in the original image can be extracted.

Figure 2 shows the proposed architecture. We implement the architecture on a Xilinx XC7Z020 system on chip on a customized board. The board is 5 cm × 5 cm (see Figure 2a) with about 3 W runtime power consumption, and it can fit into small robots.

Figure 2b shows the system architecture. A common computation system includes a CPU and the external memory, which is the top white part. To accelerate the CNN, we implement the bottom gray part on the FPGA. Data and instruction communication between the CPU and the FPGA is achieved with a shared memory scheme. The FPGA-based accelerator accesses the external memory through the direct memory access module. The host CPU accesses the status registers of the FPGA accelerator and sends control signals through the general-purpose port by memory mapping.

At runtime, the accelerator sequentially reads all the instructions and executes them automatically. The host CPU does no scheduling work and waits only for the accelerator to finish. For software developers, calling the CNN accelerator is like establishing a new thread. In real applications, the CNN is usually a part of the algorithm. The CPU is used to schedule the flow of the algorithm and handles the non-CNN parts.

The basic unit for CNN computation is the processing element (PE). Figure 2c shows

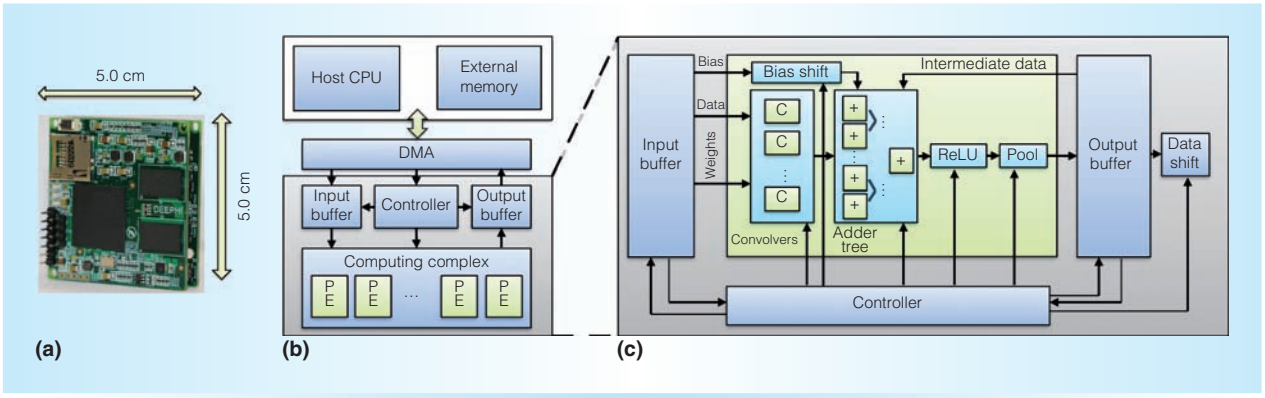

Figure 2. Aristotle architecture for the convolutional neural network (CNN) accelerator. (a) System board. (b) Overall system architecture. (c) Processing element (PE) architecture.

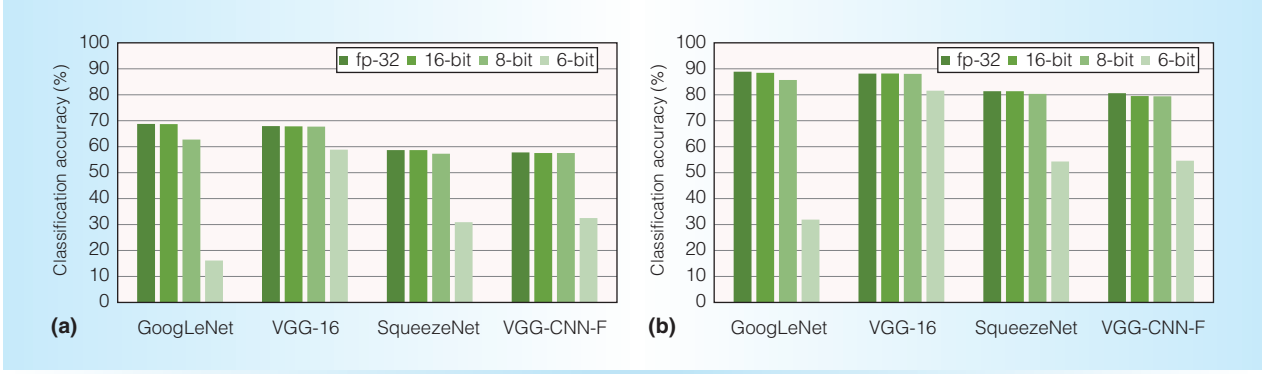

Figure 3. Quantization results for different CNN models. (a) Top-1 classification accuracy. (b) Top-5 classification accuracy.

the PE's architecture. As described by the equation of each output channel M_j, the convolution layer does the summation on 2D convolution results. So, we implement multiple 2D convolvers in a single PE and add the results together with an adder tree. We implement the line buffer design for the convolvers such that the 2D convolution can be processed in a pipelined manner, achieving the throughput of 1 pixel per cycle. Because the hardware resource is limited, we might not be able to do the summation of all the convolution results with the adder tree. So, the output buffer offers intermediate results back to the PE for accumulation. Nonlinear and pooling operations are integrated in the pipeline and can be bypassed if needed.

To fully utilize the data locality of CNN computation, the feature maps in the input buffer are shared by all the PEs. The same feature maps use different convolution kernels and biases to calculate different output feature maps. The address space of the input and output buffer is available in the instruction interface. For a certain convolution layer, the compiler specially manages the on-chip cache to minimize the external memory access.

Besides the hardware architecture design, we also do software-level optimization. We try to reduce the bit width of data in the CNN model, such that the limited logic and memory resource on the Zynq 7020 becomes relatively larger. To fully utilize the limited bit width, we use a fixed-point format but allow the radix point of data to vary among different layers. This strategy adjusts the data to different dynamic ranges in different layers and prevents overflow. Figure 3 shows our experimental results on state-of-the-art networks. We see that 8-bit quantization brings negligible performance loss for all these networks, so we adopt an 8-bit hardware design.

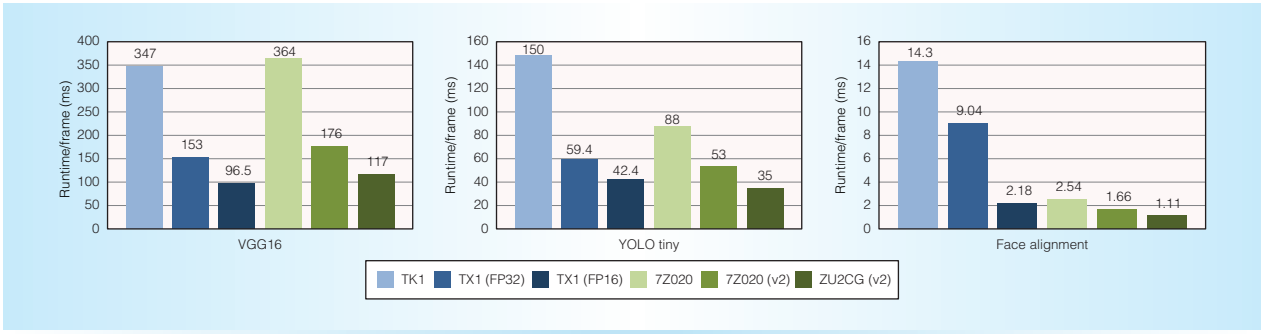

Figure 4. Performance comparison between mobile GPU and Aristotle on different FPGA platforms, measured by milliseconds per frame.

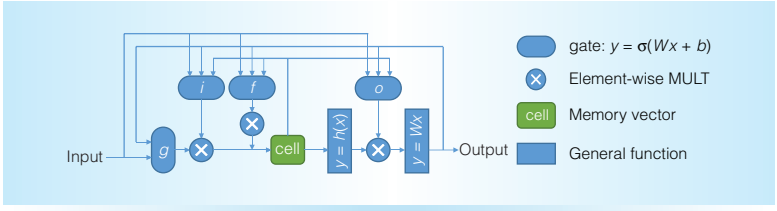

Figure 5. Computation graph of a single long short-term memory (LSTM) layer. σ denotes the sigmoid function on each element of a vector.

The implementation on XC7Z020 includes two PEs, each comprising 16 3-×-3 convolvers working at 214 MHz. This design uses 272-Kbyte on-chip block RAM, 198 digital signal processing (DSP) units, and about 27,000 lookup tables. The lookup table and DSP costs equal about 413,000 gates for an ASIC design. We use three applications to evaluate the system performance: face alignment, object detection using the YOLO (You Only Look Once) algorithm,[6] and image classification using the VGG (Visual Geometry Group) network.[7] The same applications are also realized on the Nvidia TK1 and TX1 platform with the latest cuDNN library. Figure 4 shows the performance comparison, including estimation of our next-generation Aristotle on XC7Z020 and ZU2CG. The proposed architecture on the FPGA can offer similar performance to the mobile GPU on these applications. However, the power consumption of the FPGA is about 3 W, versus 15 W for the GPU. Note that the peak performance is 326 Gflops for TK1 and 1 Tflops for TX1. However, Aristotle's peak performance is only 123 giga operations per second (GOPS). This shows that Aristotle is much more efficient than TK1 and TX1.

Descartes: The Sparse RNN/LSTM Accelerator

RNNs and LSTM are widely used in speech recognition, natural language processing, question answering, and machine translation.[2,5,8–10] An LSTM network accepts an input sequence $x = (x_1, \ldots, x_T)$ and computes an output sequence $y = (y_1, \ldots, y_T)$ by feeding the input vectors sequentially to the computation graph shown in Figure 5. In each time step t, a memory vector c_t is calculated and used in the $t + 1$ time step. Two main types of computation are involved in this graph: matrix vector multiplication and element-wise operation.

State-of-the-art RNN and LSTM models are both computationally and memory intensive, making them power hungry and increasing a datacenter's total cost of ownership. Accessing memory is more than two orders of magnitude more energy consuming than arithmetic logic unit (ALU) operations, so it's critical to reduce memory reference.

To address this problem, we first present an effective model compression algorithm for LSTM, which consists of pruning and quantization. The basic idea of the pruning strategy is to zero out the weights with the smallest absolute values. The loss of accuracy can be compensated by retraining the remaining network with back propagation. Note that we do the pruning in a load-balance-aware way, which balances the number of nonzero weights in different submatrices. So, when different submatrices are assigned to different processing elements in hardware, the workload is

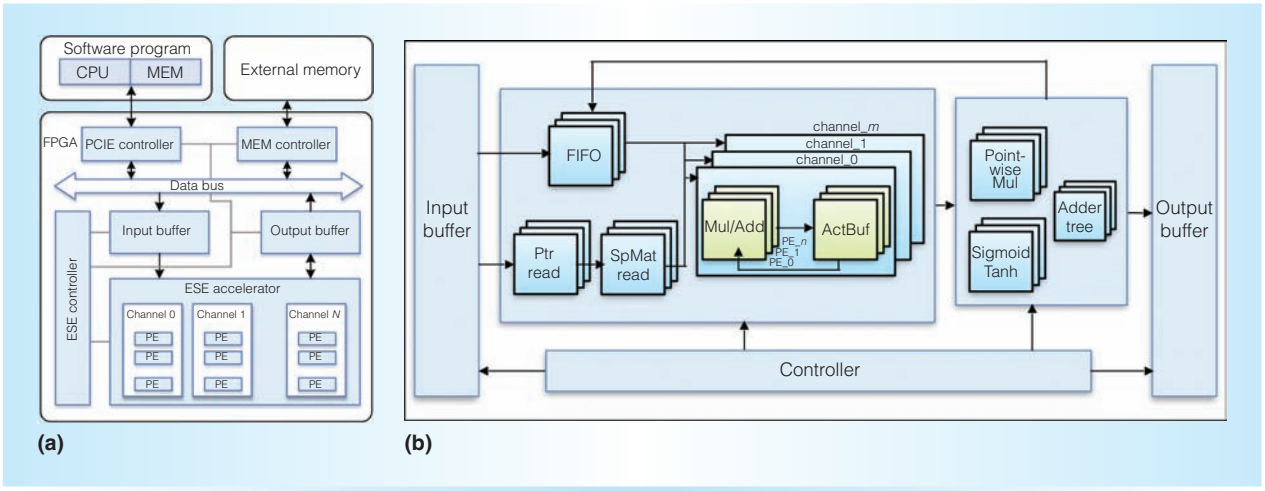

Figure 6. Descartes architecture. (a) Overall system architecture. (b) Processing element architecture.

balanced. Thus, the synchronization overhead among different processing elements is kept low.

Quantization is the second step for LSTM compression. We encode the weights with a 12-bit fixed-point format and the index with 4 bits; thus, a single weight takes 2 bytes. In a real-world speech dataset, such as the one used in our experiments, we pruned away 90 percent parameters and quantized the model to 12 bits, resulting in negligible loss of accuracy. The overall compression ratio is $2 \times 10 = 20$ times.

We designed a scheduler that can effectively schedule the complex LSTM operations using sparse matrix vector multiplication as the basic building block, with memory reference fully overlapped with computation. Because our design targets server applications, we fix this schedule strategy in hardware to offer the best efficiency.

The irregular computation pattern after compression poses challenges for the hardware design. Therefore, we designed a hardware architecture called Descartes that can work directly on the compressed model.[11] It is composed of the components shown in Figure 6. The input vector is buffered in the first-in, first-out, and the matrix is encoded with a compressed sparse column format, which contains the pointer, matrix value, and index value. Descartes first reads out the pointer, then the index and weight. The decoded weights are fed to the ALU to get multiplied with the input vector and added to the activation buffer; the location is specified by the index. This completes the matrix vector multiplication. On the right side is the element-wise unit, which contains a point-wise multiplier, adder tree, and Sigmoid and Tanh units. These are not on the critical path compared with matrix vector multiplication units. Sigmoid and Tanh units are implemented with a lookup table.

Descartes achieves high efficiency by load balancing and partitioning both the computation and storage. Descartes also supports processing multiple speech data concurrently, shown as different channels in Figure 6.

Implemented on a Xilinx XCKU060 FPGA running at 200 MHz, the Descartes architecture has a processing power of 282 GOPS per second, working directly on a compressed LSTM network, corresponding to 2.52 tera operations per second (TOPS) per second on an uncompressed network in which each operation is an add or multiplication. The LSTM model is pruned to 10 percent nonzeros; taking padding zeros into account, it's 12.2 percent nonzeros. With 32 PEs, Descartes can process a speech recognition LSTM with 1,024 hidden memory cells at 82.7 microseconds. We evaluate the same LSTM on the Core i7-5930k CPU and Pascal TitanX GPU. On the CPU and GPU, the four LSTM gates are merged into a single matrix, which improves the CPU/GPU computation resource utilization. We used MKL CBLAS and MKL SPBLAS for the CPU and cuBLAS and cuSPARSE for the GPU.

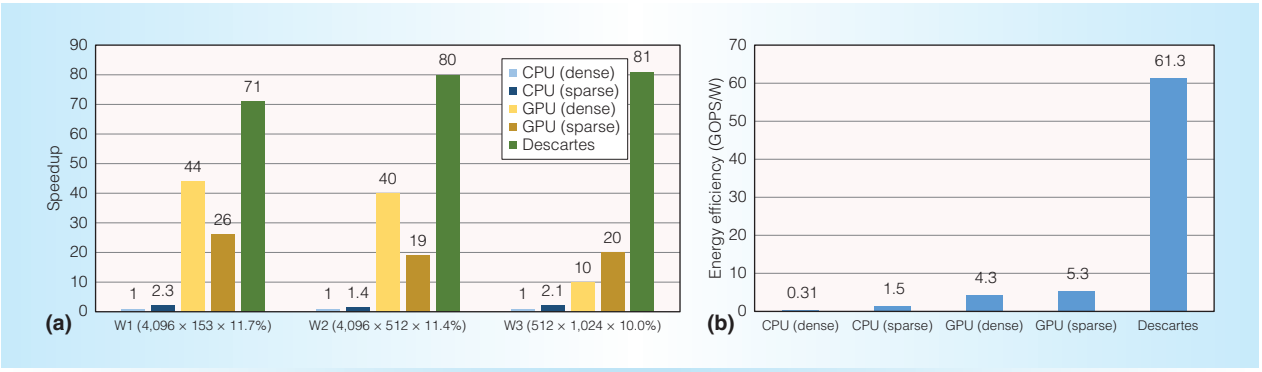

Figure 7. Performance comparison between Descartes and CPU/GPU. (a) Matrix multiplication speed, normalized with the speed of dense matrix multiplication on the CPU. W1, W2, and W3 are the three main matrices in an LSTM layer, expressed as (row × col × sparsity). (b) Power efficiency. (GOPS: giga operations per second.)

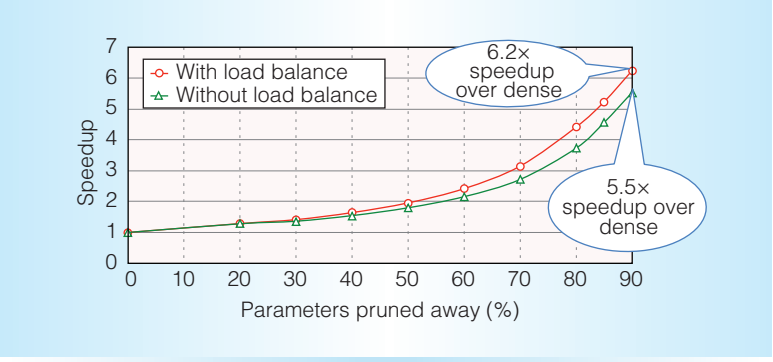

Figure 8. Running the sparse LSTM model is 6.2 times faster than running the dense model.

Figure 7a shows the speedup over CPU/GPU dense/sparse implementations. Descartes even achieves about 2 times speedup over the latest Pascal TitanX GPU.

The Descartes architecture can run both the sparse model and the dense model. Figure 8 shows the speedup with sparsity. Working on the compressed model with 90 percent of the parameters pruned away, Descartes is 6.2 times faster than the baseline uncompressed dense model.

Descartes costs 294,000 lookup tables, 1,505 DSPs, and 4.18 Mbytes of SRAM on an FPGA. The lookup table and DSP costs equal about 5,080,000 gates for an ASIC design. The power consumption of Descartes is 41 W. The power consumption of the CPU is 111 W for dense implementation and 38 W for sparse implementation. The GPU consumes 202 W for dense matrix multiplication and 136 W for sparse implementation. Figure 7b shows the energy-efficiency comparison. Descartes achieves 10 to 200 times better energy efficiency over the other platforms.

This article discusses efficient methods and hardware for deep learning. As shown with our experiments, model optimization such as quantization and sparsification greatly benefits hardware design. To achieve the best energy efficiency, ASIC is always the optimal choice. But now, it's a bad idea to tape out certain networks considering the fast iteration of deep learning algorithms. The CPU and GPU can always support deep learning, but they are not that energy efficient. FPGA just reaches the balance point. The proposed Aristotle and Descartes architectures are the beginning of the deep learning era. We will try to explore more efficient methods for software–hardware codesign for deep learning in the future. MICRO

Acknowledgments

This work represents the combined efforts of many talented full-time and intern engineers led by Dongliang Xie, Hong Luo, and Lingzhi Sui at DeePhi.

References

1. K. He et al., "Deep Residual Learning for Image Recognition," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

2. D.A. et al., "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin," *Proc. 33rd Int'l Conf. Machine Learning*, 2016, pp. 173–182.
3. J. Qiu et al., "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," *Proc. ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays*, 2016, pp. 26–35.
4. C. Szegedy et al., "Going Deeper with Convolutions," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2015, pp. 1–9.
5. I. Sutskever et al., "Sequence to Sequence Learning with Neural Networks," *Advances in Neural Information Processing Systems*, 2014, pp. 3104–3112.
6. J. Redmon et al., "You Only Look Once: Unified, Real-Time Object Detection," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2016, pp. 779–788.
7. K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv preprint arXiv:1409.1556, 2014.
8. A. Hannun et al., "Deep Speech: Scaling Up End-to-End Speech Recognition," arXiv, preprint arXiv:1412.5567, 2014.
9. A. Karpathy and L. Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2015, pp. 3128–3137.
10. S. Antol et al., "VQA: Visual Question Answering," *Proc. IEEE Int'l Conf. Computer Vision*, 2015, pp. 2425–2433.
11. S. Han et al., "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," *Proc. ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays*, 2017, pp. 75–84.

Kaiyuan Guo is a PhD candidate in the Department of Electronic Engineering at Tsinghua University and an intern at DeePhi. His research interests include hardware acceleration of deep learning and SLAM. Guo received a BS in electronic engineering from Tsinghua University. Contact him at gky15@mails.tsinghua.edu.cn.

Song Han is a founder of DeePhi and a PhD candidate in the Department of Electrical Engineering at Stanford University. His research interests include deep learning model compression and hardware acceleration. Han received a BS in electronic engineering from Tsinghua University. Contact him at songhan@stanford.edu.

Song Yao is the CEO and a founder of DeePhi. His research interests include 3D IC, compiler designs, and hardware acceleration of deep learning. Yao received a BS in electronic engineering from Tsinghua University. Contact him at songyao@deephi.tech.

Yu Wang is a founder of DeePhi and an associate professor in the Department of Electronic Engineering at Tsinghua University. His research interests include brain-inspired computing system design with both CMOS and emerging devices. Wang received a PhD in electronic engineering from Tsinghua University. Contact him at yu-wang@mail.tsinghua.edu.cn.

Yuan Xie is a professor leading the Scalable and Energy-Efficient Architecture Lab (SEAL) at the University of California, Santa Barbara. His research interests include computer architecture, electronics design automation (EDA), VLSI design, and embedded systems design. Xie received a PhD in electrical engineering from Princeton University. He is an IEEE Fellow. Contact him at yuanxie@ece.ucsb.edu.

Huazhong Yang is a professor in the Department of Electronic Engineering at Tsinghua University. His research interests include mixed-signal circuit design, EDA, VLSI design, and IoT applications. Yang received a PhD in electronic engineering from Tsinghua University. Contact him at yanghz@tsinghua.edu.cn.