Breaking the Interaction Wall: A DLPU-centric Deep Learning Computing System

Zidong Du, *Member, IEEE,* Qi Guo, *Member, IEEE,* Yongwei Zhao, Xi Zeng, Ling Li, Limin Cheng, Zhiwei Xu, *Senior Member, IEEE,* Ninghui Sun, *Member, IEEE,* and Yunji Chen, *Senior Member, IEEE*

Abstract—Due to the broad successes of deep learning, many CPU-centric artificial intelligent computing systems employ specialized devices such as GPUs, FPGAs, and ASICs, which can be named as Deep Learning Processing Units (DLPUs), for processing computation-intensive deep learning tasks. The separation between the scalar control operations mapped on CPUs and the vector computation operations mapped on DLPUs causes the frequent and costly interactions between CPUs and DLPUs, leading to the *Interaction Wall*. Moreover, the increasing algorithm complexity and DLPU computation speed would further aggravate the interaction wall substantially.

To break the interaction wall, we propose a novel DLPU-centric deep learning computing system consisting of an *exception-oriented programming (EOP) model* and the architectural support of *CPULESS DLPU*. The EOP model processes scalar control operations of a deep learning task as exception handlers to maximally avoid stalling the crucial and dominated vector computation operations. Together with the CPULESS DLPU which integrates a scalar processing unit (SPU) for scalar control operations and the parallel processing unit (PPU) for vector computation operations into a fused pipeline, the proposed DLPU-centric system can cost-effectively leverage the EOP model to execute the two kinds of operations simultaneously without disturbing each other. Compared with a state-of-the-art commodity CPU-centric system with discrete V100 GPU via PCIe bus, experimental results show that our DLPU-centric system achieves 10.30× better performance and 92.99% energy savings, respectively. Moreover, compared with a CPU-centric version of DLPU system where the SPU serves as the host with integrated PPU, the proposed DLPU-centric system still achieves 15.60% better performance from avoided interactions.

Index Terms—Neural net accelerators, system architectures, interaction wall.

1 Introduction

Due to the successful applications in various fields (including image/speech recognition [1], natural language processing [2], and game strategy [3]), deep learning (DL) has become an active field in both academia and industry. To improve the efficiency of processing DL algorithms, many DL computing systems have been proposed, which adopt CPUs as the central controller, and integrate specialized devices to accelerate the major computation, such as GPU, FPGA [4], [5], [6], and ASICs [7], [8], [9], [10], [11], etc. Such systems have been deployed in various scenarios from mobile devices to cloud servers. For example, Amazon released DeepLens, the world's first deep learning embedded video camera. Google announced TPU v3 based computing system and provides cloud computing services reaching 100 Petaflops peak performance. NVIDIA released DGX-1/DGX-2 server with 8/16 V100 GPUs for DL. IBM announced Summit, an AI supercomputer with 27648 NVIDIA V100 GPUs.

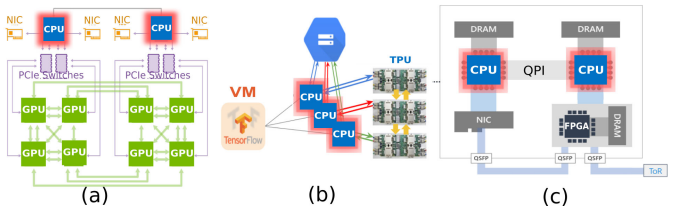


Fig. 1. Typical deep learning computing system architecture: (a) DGX-2; (b) TPU; (c) Brainwave [12].

As shown in Fig. 1, existing deep learning computing systems are essentially CPU-centric, where the host CPUs act as the masters for sending/receiving both control and data information to/from the DLPUs, and the DLPUs act as the slaves for vector computation accordingly. The separation between the execution of scalar control operations on CPU and the execution of vector computation operations on DLPU resulted in frequent and costly interactions between CPU and DLPU, which have become the key reason for

- *Zidong Du and Yongwei Zhao are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with Cambricon Technologies, Beijing, China. E-mail: {duzidong, zhaoyongwei}@ict.ac.cn.*
- *Qi Guo is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China. E-mail: guoqi@ict.ac.cn.*
- *Xi Zeng is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, the University of Chinese Academy of Sciences, Beijing 100049, China, and also with Cambricon Technologies, Beijing, China. E-mail: zengxi@ict.ac.cn.*
- *Ling Li and Limin Cheng are with Institute of Software, Chinese Academy of Sciences, Beijing 100190, China. E-mail: {liling, chenglimin}@iscas.ac.cn.*
- *Zhiwei Xu and Ninghui Sun are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China. E-mail: {zxu, snh}@ict.ac.cn.*
- *Yunji Chen is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China, and also with the Institute of Brain-Intelligence Technology, Zhangjiang Laboratory (BIT, ZJLab), Shanghai Research Center for Brain Science and Brain-Inspired Intelligence (Shanghai Brain—AI), CAS Center for Excellence in Brain Science and Intelligence Technology (CEBSIT), Beijing, China. E-mail: cyj@ict.ac.cn.*

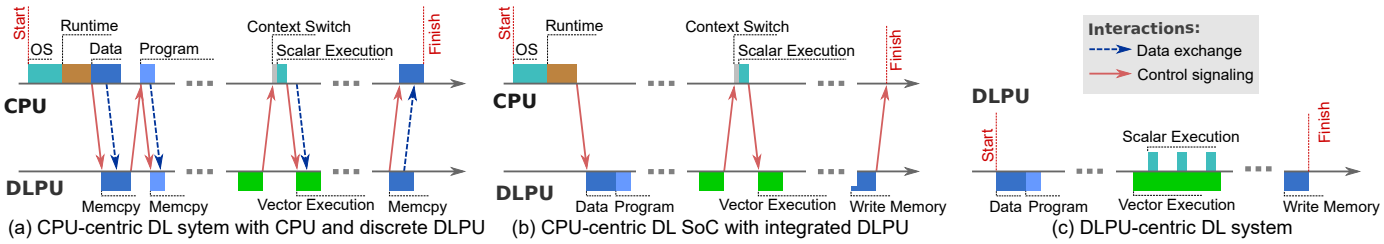


Fig. 2. (a) A typical processing flow in a CPU-centric DL computing system which consists of CPU and discrete DLPU: the CPU-DLPU interactions include *data exchange* and *control signaling*. (b) The processing flow in a CPU-centric DL computing system which integrates the DLPU: the CPU-DLPU interactions include *control signaling*. (c) The ideal processing flow in a DLPU-centric DL computing system: DLPU processes the DL tasks without interactions.

the system inefficiency. As illustrative examples on a CPU system with the K40 GPU, when running Faster R-CNN [13] for inferencing 100 images, only 43.9% of the total time is consumed by GPU computation, where the rest time is mainly consumed by the 1310 CPU-GPU interactions; when training GCN (Graph Convolutional Network) [14] with 200 iterations, only 4.8% of the total time is consumed by GPU computation, where the rest time is consumed by more than 92k CPU-GPU interactions.

Specifically, the CPU-DLPU interaction can be roughly classified into two categories: *data exchange* and *control signaling*. The exchanged data include algorithm inputs/outputs, intermediate results, etc; and the control signaling includes kernel launch, interrupt handling, etc. As shown in Fig. 2(a), to perform a DL task on a typical CPU-centric DL computing system, the host CPU needs to load the data and device program to the host memory first and then send to the DLPU for the major computation of vector operations. During the computation, the DLPU has to interact with the host CPU to perform the scalar control operations. After finishing the computation, the DLPU notifies the host CPU and sends the results to the host memory. Actually, the data exchange speed is typically limited by the bandwidth of IO subsystems (e.g., the PCIe bus for discrete DLPUs or the AXI bus for integrated DLPUs), and the control signaling speed is mainly restricted by the processing speed of relatively complicated software stack (OS kernel, driver, and runtime, etc.) on the host CPU. Apparently, there exists disparity between the processing rate of CPU-DLPU interaction and the computation speed of DLPU, which can be termed as *interaction wall*[1] in a way analogous to *memory wall*. In fact, the interaction wall would be further aggravated for two reasons. The first reason is that the increasing algorithm complexity incurs more and more CPU-DLPU interactions. The second reason is that the improvement of interaction speed between CPU and DLPU cannot keep pace with the computation speed of DLPUs.

To solve the ever-increasing challenge resulted from the interaction wall, an intuitive idea is to merge the DLPU into CPU (such as an additional SIMD functional unit) [15], [16], so as to eliminate CPU-DLPU *data exchange* interactions with shared memory space, see Fig. 2(b). However, it still suffers from the *control signaling* interactions, which fragmentize the vector computations on the DLPU, leading to significantly low DLPU utilization. Besides, CPU is not an ideal host for merging DLPU, which makes the merged SIMD-style architecture both cost-ineffective and energy-ineffective. The reason is because modern CPU lacks of architectural support for DL tasks, e.g., stream data path, programmer-visible buffer, dependency checking logic of vector operands, etc., but spends most area and energy consumption thriving for instruction-level parallelism improvement, e.g., branch predictor, out-of-order scheduler, renaming registers, complex memory hierarchy for random memory accesses, etc. [17], which are less useful for data-level paralleled DL tasks.

Therefore, from the perspective of processing efficiency, a DLPU-centric DL computing system where the DLPU takes over the scalar control operations can be much more effective. On one hand, a DLPU has all architectural supports to effectively execute vector computation operations of DL tasks. And on the other hand, it is cost-effective to add the *necessary* support for executing scalar control operations of DL tasks. However, beyond merely enhancing the DLPU with the scalar execution ability, the key challenge in constructing a highly efficient DLPU-centric DL computing system is ensuring the vector data flow to continuously execute without interruptions from the scalar operations for control and system management. As shown in Fig. 2(c), an ideal DLPU-centric DL computing system is able to process the scalar control operations in parallel with continuously vector data flow without interactions.

In this paper, we propose a novel DLPU-centric deep learning computing system consisting of the *exception-oriented programming* (EOP) model and corresponding architectural support of CPULESS microarchitecture. The EOP model is proposed to separate the vector data flow for continuous execution without inter-disturbance from the scalar control flow. Specially, in the EOP model, scalar control flow are organized as exceptions embedded into the vector data flow, which will be executed in parallel with the vector operations. Specifically, when the vector unit is busy, in-flight vector instructions block the pipeline for following instructions to issue (no matter the type of the following instructions are, scalar or vector). Then the EOP model allows the CPULESS DLPU to load a bunch of scalar operations as exception, utilizing the idle fetcher, decoder and scalar unit. The CPULESS microarchitecture is proposed to effectively support the EOP model. Particularly, CPULESS adopts a fused pipeline where a dedicated *scalar processing unit* (SPU) for scalar control flow and *parallel processing unit* (PPU) for vector data flow are integrated in a same pipeline with forwarding data paths to eliminate the interactions. To avoid the SPU disturbing the PPU, where

1. Note that the interaction wall is more than *IO bottleneck*, since interaction efficiency is not only determined by the native IO speed but also the software stack and processing ability of the host CPU.

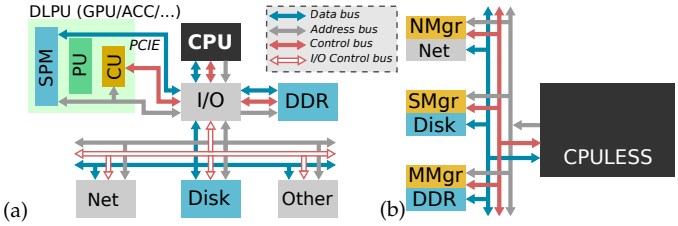


Fig. 3. The (a) CPU-centric (b) DLPU-centric architecture.

the interrupts/exceptions to SPU (e.g., control signaling) may frequently squash in-flight vector instructions in PPU (which may have executed for $\sim 10^5$ cycles), CPULESS puts issued in-flight vector instructions in a dedicated vector queue which is not affected by interrupts/exceptions to the main instruction pipeline and the SPU.

In this paper, we made the following contributions.

- We propose a novel DLPU architecture, which enables the DLPU-centric deep learning computing system to cost-effectively break the interaction wall caused by the separation of scalar control operations on CPU and vector computation operation on traditional DLPU.
- We propose an exception-oriented programming model to ensure the continuous execution of vector data flow. The EOP model enables a new execution model which execute scalar control operations of a DL task in the SPU as exception handlers. Whenever the pipeline is blocked by the PPU, the scalar operations can still execute on the SPU as exceptions without disturbance.
- We propose a highly efficient microarchitecture for CPULESS that can allow the simultaneous execution of scalar control operations on SPU and vector computation operations on PPU, which can significantly reduce the disturbance from the SPU to the PPU.
- We build the first DLPU-centric deep learning computing system based on the CPULESS DLPU and the EOP model, which is much more cost-effective than traditional CPU-centric deep learning systems. Experiments show that the DLPU-centric system achieves 10.30× better performance and 92.99% energy savings compared with the state-of-the-art commodity CPU-centric system with discrete V100 GPU via PCIe bus. Compared with a CPU-centric version of DLPU system where the SPU serves as the host with integrated PPU, the proposed DLPU-centric system still achieves 15.60% better performance from avoided interactions, which shows the effectiveness of EOP.

2 Interaction Wall in CPU-centric Systems

In this section, we first introduce the overall architecture of CPU-centric deep learning computing systems. Then we will introduce the interaction wall in CPU-centric systems, and use experimental results to demonstrate the non-negligible performance impact of the interaction wall.

2.1 CPU-centric systems

Currently, as shown in Fig. 3(a), mainstream DL computing systems are CPU-centric, where the host CPUs act as the masters for controlling the employed DLPU (and all other peripheral devices) through interfaces such as the PCIe bus (i.e., discrete DLPU) and AXI bus (i.e., integrated DLPU), see Table 1. In such CPU-centric DL computing systems, CPUs and DLPUs play different roles in processing DL tasks including not only vector computation operations but also scalar control operations. CPUs have a complex dynamic instruction pipeline and memory hierarchy developed for irregular scalar computation but have relatively low peak performance. Thus CPUs are normally only responsible for scalar control operations of DL tasks. Operating systems (OS), runtime and DL frameworks are also deployed on CPUs to provide controlling, programming, runtime support, and resource management for the system. On the other hand, DLPUs (especially ASIC [7], [9], [11]) have high vector computation speed but are incapable of dynamically controlled execution. Hence, DLPUs are responsible for only vector computation operations. Such separate operation mapping in CPU-centric DL systems inevitably introduces frequent and costly interactions between CPUs and DLPUs, which makes the interaction wall a severe problem.

2.2 Interaction wall: A Faster R-CNN example

We demonstrate the interaction wall on commodity mainstream CPU-centric systems (including CPU+discrete DLPU, CPU+integrate DLPU) with a representative state-of-the-art deep learning algorithms (i.e. Faster R-CNN). Fig. 4 shows the entire processing flow of Faster R-CNN. The Faster R-CNN algorithm processes an image first through a deep convolutional network (Deep ConvNet) and a region proposal network (RPN) to get the feature maps and several region-of-interest (ROI) proposals, respectively. For each ROI, a pooling layer is applied to extract a fixed-length feature vector, which is used to get two outputs, i.e., the softmax probability and the bounding box positions. In ROI pooling, instead of pooling over the entire feature map, pooling is performed on only a few windows of the entire feature map, where both the number and sizes of pooling windows are dynamically determined based on execution results. Thus the control flow of ROI pooling must be dynamically decided. Current CPU-centric systems have two possible solutions: either the CPU performs the entire ROI pooling, or the CPU performs the static control part of ROI pooling (window selection) and leaves only the vector computation part (pooling) on the DLPU. Both solutions incur massive interactions, leading to inefficiently processing of Faster R-CNN.

2.2.1 CPU-centric DL system with discrete DLPU

We profile the execution of CPU-centric DL system with NVPROF and NVVP. Fig. 5 (middle-left) shows the execution breakdown of the Faster R-CNN inference stage on the CPU-centric system with a discrete NVIDIA K40 GPU. In this example, Faster R-CNN is used for processing

TABLE 1
Examples of deep learning computing systems.

Example	Scale	Host CPU	DLPU	Interface
TPU cloud	Cloud	Intel Skylake	TPU accelerator	PCIe
Brainwave [18]	Cloud	Intel Xeon	Stratix 10 FPGA	PCIe
DGX-2	Server	Intel Xeon	V100 GPU	PCIe
Summit	Supercomputer	Power9	V100 GPU	PCIe
Tesla Autopilot	Work station	Arm A72	Drive PX-2 GPU	AXI
HUAWEI mate20	Mobile device	Kirin 980	NPU accelerator	AXI
DeepLen	IoT devices	Intel Atom	Intel Gen9 graphics	AXI

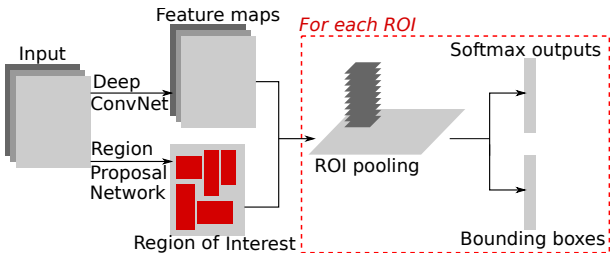


Fig. 4. The execution process of Faster R-CNN.

100 images with batch size as 1. Only 43.90% of the total execution time is spent on the DLPU computation, i.e., the DLPU utilization rate is only 43.90%. The 1310 CPU-DLPU interactions take 22.30% of total time, and GPU startup takes 32.22% of the total time, which mainly includes the execution of the high-level framework and GPU runtime. It is worth noting that the GPU startup is executed on the host CPU and can also be regarded as control signaling.

2.2.2 CPU-centric DL system with integrated DLPU

Fig. 5 (top-left) shows the execution breakdown of the Faster R-CNN inference state on the CPU-centric system with a integrated DLPU, i.e., the Jetson TX2 whose CPU SOC integrates a 256-core Pascal GPU. With the same configuration, it achieves a 47.40% DLPU utilization. Obviously, with an integrated DLPU, the *data exchange* interaction overhead can be largely removed: therefore, the DLPU utilization of Jetson TX2 is higher than CPU+K40 GPU. However, the 65398 CPU-DLPU interactions not only take a large proportion of the total execution time (40.72%) but also divide the entire GPU computation into over tens of thousands small fragments, which hampers the efficiency of GPU computation (69.16% utilization during working time).

2.3 Rising the *interaction wall*

Rather than addressed or alleviated, in fact, the *interaction wall* in CPU-centric DL computing systems is rising rapidly because of the fast developed DL algorithms and DLPUs.

Improving the capability of DLPUs. While the computation speed of DLPUs keeps increasing dramatically, the *interaction wall* is exacerbated. The reason is because the interaction speed does not increase comparably with the DLPU computation speed. Therefore, according to the Amdhal's Law, the accelerated execution time T_2 can be formed as $T_2 = \alpha T_1 + (1-\alpha)T_1/p + T_{interaction}$ where T_1 is the original sequential execution time, α is the scalar execution ratio, p is the speedup achieved by parallel execution, and $T_{interaction}$ is the overhead of interaction. As the p can be improved drastically by improving the capability of DLPUs, the $T_{interaction}$ starts to dominate the total execution time. The disparity between the peak performance of GPUs and the IO speed (including the PCIe and QPI interconnection) continues to increase significantly. Particularly, we conduct experiments on such state-of-the-art GPU (NVIDIA V100), which has 3.66× higher peak performance than K40. As a result, with the same PCIe Gen3 interface, only 12.8% of the total time is spent on the GPU computation (vs. 43.9% on K40), while 55.8% is spent on GPU startup (vs. 32.22% on K40), as shown in Fig. 5 (bottom-left).

Increasing complexity of DL algorithms. Early deep learning algorithms (e.g., AlexNet [19] and VGG [20]) contain massive vector/matrix operations and quite few scalar control operations. However, state-of-the-art industrial deep learning algorithms (e.g., Faster R-CNN [13], Mask R-CNN [21], BigGAN [22], and BERT [23]) require much more dynamic controlling than their predecessors, where the scalar control operations play an important role. Similarly, according to the Amdhal's Law, the accelerated time T_2 starts to be dominated by the scalar part, i.e, the α is increased for these new DL algorithms. Particularly, we also conduct experiments on another state-of-the-art deep learning algorithm, i.e., Graph Convolution Networks (GCN) [14], which has been widely used in analyzing social networks, knowledge graphs, protein-interaction networks, and so on. As shown in Fig. 5(middle-right), when running on the K40 GPU system, there are 92k interactions in 200 epochs GCN training. Regarding the execution time, only 4.8% of the total time is spent on the DLPU computation (vs. 43.9% on Faster R-CNN), and 55.56% of the total time is spent on the GPU startup (vs. 32.22% on Faster R-CNN). The rest of the time includes data exchange between CPU and GPU, and overhead of CUDA runtime. The DLPU utilization rate of GCN is even much lower than that of Faster R-CNN since the memory access and computation on each graph node are dynamically controlled by graph edge information, which incurs massive interactions between the DLPUs and CPUs. Similar cases can be found on CPU+V100 and Jetson TX2, see Fig. 5(top-right) and (bottom-right).

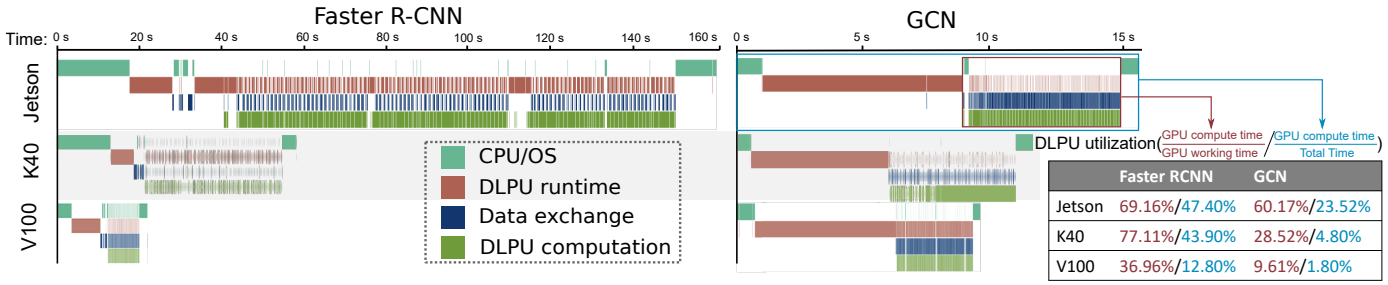
To further illustrate the impacts of the interaction wall, we also conduct experiments on many different alternative settings. For example, with max allocatable batch size on K40, the DLPU utilization rate is only 65.8%; with multiple DLPUs (i.e., four K40 GPUs), averagely only 49.4% of the total time is spent on GPU computation. We can confirm that the interaction wall is inevitable on existing CPU-centric deep learning computing systems, which always leading to a low DLPU utilization rate.

In summary, the interaction wall has significantly hampered the efficiency of CPU-centric deep learning computing systems, and will even more severely hamper the efficiency in the future. To break the interaction wall, society calls out for a new architecture for a deep learning computing system.

3 From CPU-centric to DLPU-centric

3.1 A DLPU-centric view

From the perspective of processing DL tasks, addressing the *interaction wall* at root is to ensure the high utilization of DLPU when processing the scalar executions in parallel with the major computations of DL tasks. Therefore, an ideal DL computing system should be able to ensure the continuous execution of vector data flow without inter-disturbance from the scalar control flow, i.e., addressing not only the *data exchange* but also *control signaling* interactions. Obviously, the CPU-centric system with integrated DLPU solution can only address the *data exchange* interactions by sharing the host memory between the host CPU and the DLPU (Fig. 5(middle)). However, such simple solution is unable to address the *control signaling* interactions and



	Faster RCNN	GCN
Jetson	69.16%/47.40%	60.17%/23.52%
K40	77.11%/43.90%	28.52%/4.80%
V100	36.96%/12.80%	9.61%/1.80%

Fig. 5. Execution timeline of deep learning algorithms on a CPU-centric system with a discrete K40 GPU, a discrete V100 GPU, and an integrated GPU (Jetson TX2). The CPU-DLPU interactions in CPU+K40 cost 22.30% and 94.24% of total execution time on Faster R-CNN and GCN, respectively; the numbers are 40.72% and 73.50% on Jetson TX2. GPU working time is the timespan from the starting of the first GPU activity to the ending of the last GPU activity.

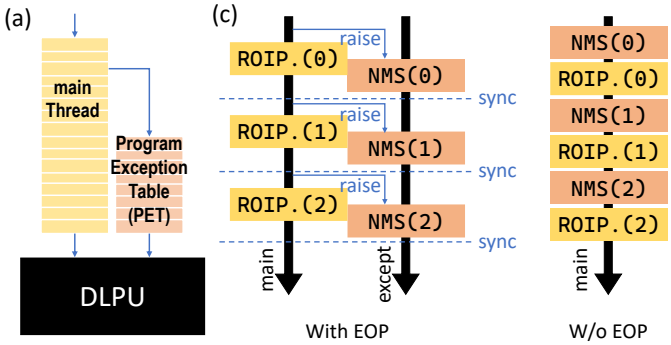
guarantee the high utilization of the DLPU, as the the scalar control flow on the host CPU may stall the vector operations on the DLPU. On the contrary, a DLPU-centric design principle where the DLPU takes over the scalar control operations as the master could lead to a high efficient DL computing system. A DLPU has all architectural supports to effectively execute vector computation operations of DL tasks, and it is cost-effective to add the necessary support for executing scalar control operations of DL tasks into a DLPU.

3.2 The EOP model

Current programming models on CPU-centric DL computing systems take the vector data flow on DLPUs as exceptions. Therefore, programmers have to manage the vector data flow as separated segments (e.g., `kernels` in GPU) where the `syscalls` are invoked by the scalar control flow to manage those segments. Even modern compilers try to assist with asynchronous execution method, the execution order is still maintained by the scalar control flow which introduces unnecessary interactions. As a result, to support the DLPU-centric design, the programming model should also be DLPU-centric.

We propose a novel exception-oriented programming (EOP) model to avoid the disturbance from scalar control flow. In the EOP model, scalar control operations in a DL task are written as exceptions, and the exception handlers are embedded into the vector data flow. Please note that the vector data flow may contain both vector instructions and scalar instructions, while the scalar control flow only contains scalar instructions. Therefore, the vector data flow is processed on the DLPU continuously while the exceptions of the scalar control flow are independent of vector operations and thus can be scheduled in parallel.

In Fig. 6, we provide the basic idea of EOP with the abstract view of vector data flow and scalar control exceptions, and an example code of ROI pooling. From the programmers' view, the programmers only learn the two threads of the DL task, i.e., the main thread which contains the major computations of the DL task and the exception thread which contains scalar computations packed as exceptions. In other words, the programmers are required to write a workflow mainly running on the PPU, and exceptions consisting of the exception handlers (e.g., `NMS()`) which will be performed on demands (when the PPU is busy while SPU is idle). The basic intuition behind EOP is to schedule those scalar exceptions to be executed in parallel with vector data flow as early as possible, so those exceptions, i.e., scalar control,



(b)

```
// input/output image of the roi-pooling layer.
extern vector[1][38][61][1024] stage3_unit23;
extern vector[300][14][14][1024] roi_result;
int roi[6000][5]; // the sorted 6k roi regions.
...
int suppressed_by_nms[6000] = {0};
for (i = 0; i < 6000; i++) {
  if (suppressed_by_nms[i]) continue;
  // defining an exception handler to process NMS.
  except NMS(i) {
    for (j = i + 1; j < 6000; j++) {
      // `JaccardOverlap` is pure scalar.
      float iou = JaccardOverlap(roi[i], roi[j]);
      if (iou > NMS_THRESHOLD)
        suppressed_by_nms[j] = true;
    }
  }

  //======== NMS+ROIPooling with EOP: ========
  raise NMS(i) as h; // raise NMS when VQ is full.
  // ROIPooling contains many vector instructions,
  // which often crams the VQ and blocks the pipeline.
  ROIPooling(roi_result[i], stage3_unit23, roi[i]);
  sync h; // The end of exception is synchronized here.

  //======== NMS+ROIPooling w/o EOP: =========
  NMS(i); // scalar operations executed solely.
  ROIPooling(roi_result[i], stage3_unit23, roi[i]);
  // issuing many vector inst.s blocking the pipeline,
  // thus vector operations executed almost solely.
}
```

Fig. 6. Exception-oriented programming model: (a) the abstract machine model for programmers, (b) an example code from the Faster R-CNN algorithms, (c) the programmer's view of the given example code. Note that the user-provided exception handler (i.e. NMS) will not be executed immediately. Instead, they will be registerred into the PET when loading executable.

will not stall the vector execution. The vector data flow and scalar control flow (i.e., exception thread) synchronize with each other by using explicit barrier synchronize instructions enforced by the programmers for ensuring correctness.

4 The CPULESS Microarchitecture

For the interaction-free architecture, we follow the proposed principles to design the CPULESS microarchitecture:

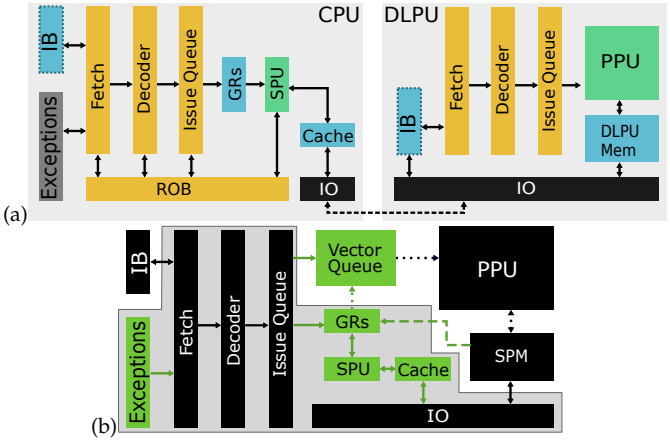


Fig. 7. (a) An intuitive microarchitecture of CPU and DLPU in CPU-centric systems. (b) The microarchitecture of CPULESS (*Black:* components also seen in traditional DLPU, *Green:* unique components, *Dotted Path:* data sharing from SPU to PPU, *Dashed Path:* data sharing from PPU to SPU, *Grey Area:* components can be affected by exceptions).

(1) it integrates the control and data flows into a fused pipeline to perform the functionalities of CPU and traditional DLPU, together with shared data paths to reduce the data exchanges. (2) it frees the vector computation on its parallel processing unit from the disturbance of interrupts and exceptions. (3) it uses programmer-managed scratchpad memory instead of cache and large vector register file for DLPU to provide a unified data space, avoiding the heavy costs of system software in context switch.

4.1 Overall microarchitecture.

CPULESS employs a fused pipeline to process both scalar control operations and vector computation operations simultaneously. Fig. 7 shows the microarchitecture of CPULESS, which consists of an instruction execution engine, functional units, on-chip buffer modules, IO module, and exception module. The instruction execution engine (IEE) contains an in-order multi-issue (e.g., 2-issue) pipeline, including instruction fetch unit, instruction decoder, issue queue (IQ) and vector queue (VQ). There are two different functional units: one is the highly parallel processing unit (PPU) for vector instructions[2], and the other is the scalar processing unit (SPU) mainly for the scalar branch, arithmetic, and memory operations. The on-chip buffer modules include the instruction buffer (IB), general-purpose registers (GRs), cache for the SPU to process scalar data, and scratchpad memory (SPM) for the PPU to process vector data.

As shown in Fig. 7 (b), most components of CPULESS are inherited from traditional DLPU design (in black) for vector computation operations, new components (in green) are the SPU, the GRs and cache, the VQ, and the exception module. By adding these new components, CPULESS is enabled to process not only the vector computation operations but also the scalar control operations, thus can process a whole DL task end-to-end without CPU.

4.2 Instruction set architecture

The instruction set architecture (ISA) is the key interface between hardware and software. In order to support scalar and vector operations in the same pipeline, the CPULESS adopts a mixed ISA design which contains both scalar and vector instructions. Similar to Cambricon [24], our ISA contains four types of instructions, including control, data transfer, computational, and logical instructions, see Table 2. Especially the data transfer and computational instructions are working on matrixes/vectors/scalars (note that matrix can be treated as a variation of vector), so as to leverage the data parallelism in hardware for high efficiency.

4.3 Key features

Comparing with traditional CPU-centric system with separate CPU and DLPU as shown in Fig. 7 (a), the CPULESS in Fig. 7 (b) has three significantly differences: a fused pipeline for the SPU and the PPU, data sharing paths between the SPU and the PPU, and a dedicated exception mechanism to allow the SPU to work in parallel with the PPU.

Fused pipeline. CPULESS adopts a pipeline architecture, including *six* stages, i.e., *fetch*, *decode*, *issue*, *execute*, *write back*, and *retire*. After fetching an instruction from IB/EB, CPULESS decodes the instruction based on its type, adds it to the IQ for future in-order issuing and registers it for future retiring. The IQ will issue the instruction to the SPU or VQ, once the instruction is the oldest ready-to-issue one in the IQ. For a scalar instruction, it will be executed in the SPU, then its result will be written back to the GRs; after writing back, it can retire from the IQ once it is the oldest issued one in the IQ. It is worth noting that for a vector instruction, once it is sent into the VQ, it can directly retire from the IQ without any effective computation, i.e., a retire-before-execution fashion. Thus, the following scalar instructions in the IQ can retire without waiting for previous vector instructions which could last over $\sim 10^5$ cycles. In such a way, CPULESS tightly fuses scalar control and vector computation in the same pipeline to cover the main functionalities of both CPUs and traditional DLPUs. It provides the foundation for eliminating the CPU-DLPU interaction in deep learning. As a comparison, Cambricon [24] also introduced multiple processing units but without letting the vector instructions retire in the IQ before their real executions. Therefore, scalar instructions will be stalled by vector instructions and vice versa, leading to inefficiency.

Sharing data paths. In CPULESS, we allow two paths to move data between the SPU and the PPU without involving the IO modules (as well as the external data buses between the CPU and the DLPU) as in the CPU-centric systems of Fig. 7(left). One is from the GRs to the PPU via the VQ for the cases that the PPU needs data from the scalar operations,

TABLE 2
The proposed ISA in CPULESS.

Type	-	Example
Control	Scalar	jump, conditional branch
Data transfer	Matrix	matrix load/store/move
	Vector	vector load/store/move
	Scalar	scalar load/store/move
Computational	Matrix	marix multiply vector, marix multiply matrix
	Vector	vector multiply vector, element-wise computation
	Scalar	scalar computations
Logical	Vector	vector comparison
	Scalar	scalar comparison

2. In this paper, we treat matrix instructions as a special case of vector instructions

c.f. the dotted path in Fig. 7. Thus, the VQ is able to access the GRs to fetch operands for vector instructions. The other is from the SPM to the SPU via the GRs for the cases that the ALU needs data from the vector operations, c.f. the dashed path in Fig. 7. Note that we do not allow the reverse path in the CPULESS for the sake of data consistency. Also, each time the SPM stores data to the main memory, the cache will check such address scope and mark out-of-date *cache lines* for later re-fetch.

Regarding dependency, write-after-write (WAW) and write-after-read (WAR) dependencies are avoided by explicitly using the MOVE instruction to move data between GRs and SPM along these two paths, thus only read-after-write (RAW) dependency happens between the SPU and the PPU in CPULESS. For the RAW dependency between a former scalar instruction and the following vector instruction, the VQ will record the index number of the IQ and wait for the scalar instruction to retire, so as to guarantee the vector instruction can read correct data. Otherwise, the VQ will record a zero-value and issue the vector instruction to the PPU in its turn (*dotted path*). In this way, CPULESS ensures the PPU execution not stalled by scalar instructions unnecessarily, as scalar instructions are finished in a few cycles.

For the RAW dependency between a former vector instruction and the following scalar instruction, a BARRIER instruction should be inserted by programmers before the scalar instruction, to ensure the correctness of execution. Since the vector instructions are retired-before-execution, the next scalar instruction will write its results, if without the BARRIER instruction. The BARRIER will be treated as a vector instruction but it only blocks the VQ until its emptiness, without real execution. Since VQ also works in an in-order way, the retire of BARRIER indicates the finishing of all previous vector instruction. Also, a MOVE instruction is included after the BARRIER to move data from SPM to GRs explicitly. Therefore, BARRIER resolves the RAW dependency.

Here we clarify the data sharing process with concrete examples.•**From SPU to PPU** The MOVE instruction from GR to SPM is treated as a vector computation instruction. The data travels from the GR to the VQ (when the MOVE instruction is issued into VQ) to the PPU (when the MOVE instruction is executed) and finally to the SPM (when the result of MOVE is written back). It costs 3 cycles if there are no stalls (i.e. when the VQ is empty). The moved data is immediately available to further vector instructions so there is no need for a BARRIER.• **From PPU to SPU** The MOVE instruction from SPM to GR is treated as a scalar instruction. Before the MOVE instruction, a BARRIER is required to ensure the RAW dependency is resolved. BARRIER is treated as a vector instruction. It finishes in 2 cycles in minimal (1 cycle issuing into VQ and 1 cycle executed by PPU if the VQ is empty). The following MOVE instruction takes 1 additional cycle so the whole process costs 3 cycles in total.

Therefore, the CPULESS reduces the data exchange between the SPU and PPU to negligible latency (3 cycles in minimal), while in traditional CPU-centric systems, the CPU and DLPU need hundreds even thousands of cycles to exchange data through IO modules and PCIe/AXI bus.

Dedicated exception. The CPULESS adopts a novel exception mechanism to execute scalar instructions and vector instructions in parallel, which enables highly-efficient scalar exception handler without blocking the execution of vector instructions. The CPULESS adopts accurate exception, which means that when an exception happens, the PC of the most recent retired instruction is recorded and all unretired instructions in the IQ are abandoned. However, all vector instructions in the VQ have already retired, thus can be continuously executed in the PPU unaffected by the exception. At the same time, the CPULESS can fetch scalar instructions from the entrance of exception handler and execute them in the SPU, in parallel with the execution of vector instructions in the PPU. When recovering from an exception, the CPULESS jumps to the next instruction of the recorded PC to continue its execution. Particularly, CPULESS has a special *VQ full* interrupt, which allows the SPU to execute *exception handlers* in the program exception table (PET), when the VQ is full of vector instructions or encountering the barrier instruction. Such special exception mechanism enables a new execution model: the scalar control operations of a deep learning task is executed in the SPU as an exception handler, while the data flow of a deep learning task is executed in the PPU in parallel.

4.4 Deploying the EOP model

For CPULESS, we employ the proposed EOP model where scalar control operations in a DL program are processed as exceptions to improve the parallelism between the PPU and the SPU. Those scalar control exceptions are independent of vector operations and thus can be executed on the SPU in parallel with them. During execution, they will be triggered by a *VQ full* interrupt, which means that the PPU has enough retired instructions (with respect to the IQ) to execute in following millions of cycles, thus the CPULESS can use the IEE and the SPU to process instructions from the scalar control flow without blocking the PPU (note that the exception in the CPULESS will not squash the retired vector instructions).

Fig. 6(b) shows an example from a real Faster R-CNN program. In this piece of code, the outmost for-loop is iterating over the 6k candidate RoI regions. For each RoI region, two operations are performed: • NMS - the Jaccard distance is computed between this region and any following regions, any regions highly overlapping with this region are suppressed. •RoI Pooling - the image in this region is bilinearly scaled to 14×14 and pooled. The NMS operation is fully scalar, while the RoI Pooling operation is composed with a large amount of vector instructions (i.e. avgpool), which will completely fill the VQ and blocks any further instructions to issue. As shown in Fig. 6(c), without the EOP model the execution of NMS and RoI Pooling is merely sequential, leading to a poor utilization of SPU and PPU. With EOP the NMS operation can be written as an exception handler which will not be executed immediately. Instead, the execution runs into RoI Pooling first, letting a bunch of avgpool instructions fills the VQ, then NMS is triggered by the *VQ full* interrupt. The PPU is busy working on the pooling operations and will not require IEE and SPU in a while, and NMS is loaded on the SPU under the help of IEE to keep the SPU utilized either.

The necessity of EOP model for DLPU-centric system is inherent in three aspects. First, in a tradition microarchitecture, the fullness of VQ must stall the entire pipeline, since

the next fetched instruction may still be a vector instruction, which will cause the overflow error of VQ. Instead of simply stalling the pipeline for the execution of vector instruction in the VQ (which may be $\sim 10^5$ cycles), the EOP model of CPULESS can schedule the exception thread to be fetched by IEE and executed in SPU, which can significantly improve the efficiency of the IEE and the SPU.

Second, EOP model can avoid the scalar instructions to grab the IEE sources required by the vector instructions, as the scalar operations are executed when the VQ full interrupt handler is triggered, where the execution of the vector instructions in VQ does not need IEE resource. As a comparison, traditional programming model which treats the scalar operations as a function can only call the scalar function at the fixed time determined at programming without dynamic information about the IEE resource requirement of the vector instructions.

Third, the EOP model enables coarse-grain instruction-level parallelism between the scalar operations and the vector operations, which gives the DLPU-centric system a large space to optimize the scheduling.

5 THE DLPU-CENTRIC SYSTEM

In this section we introduce the DLPU-centric system including the compiler toolchain and the runtime system.

5.1 Overall system architecture

As shown in Fig. 7(middle), the entire DLPU-centric system can be mainly divided into two parts, the central DLPU—CPULESS, and the IO subsystem. The IO subsystem connects all devices together through the bus. On each IO device, there exists a hardware manager for running a *monitor* on it that manages the related hardware component, as introduced in the *splitkernel* architecture [25]. The CPULESS integrates the control and data flows into a fused pipeline for eliminating the interaction wall. In Table 3, we show the comparison results of the software and hardware feature of the CPU-centric system and the DLPU-centric system.

Generally, the DLPU-centric system uses interaction-free architecture and lightweight runtime to deliver high efficiency for deep learning processing. Regarding the interaction-free architecture, the DLPU-centric system adopts the proposed CPULESS microarchitecture. Regarding the light-weight runtime, the DLPU-centric system employs a light-weight runtime as an alternative to the heavy-weight OS for system management, memory management, and processor management. To further reduce the complexity of runtime, DLPU-centric system takes advantages of distributed local hardware controllers to offload device-related functionalities to corresponding device modules, e.g., hard disk and network.

5.2 Compiling and linking

Fig. 8(a) illustrates the entire compilation process from the user program to the final machine executable. At first, the compiler, which consists of the vector and exception compiler, compiles the user program (*.eop) to the *vector object* (vec.o) and *exception object* (exp.o), respectively. Regarding the codes in the exception handlers, they are directly compiled to RISC-style scalar instructions. After compilation, the linker links the vector object and the exception object together for building the executable. The executable of EOP model is called *Exception Executable and Linking Format* (E^2LF), and the detailed format is shown in Fig. 8(b), which can be roughly divided into a header, header table, vector segment, and exception segment. The E^2LF executable can be parsed and loaded into the memory for execution, which relies on the underlying runtime system.

The programmers can define vector-typed variables either internal or external, which is statically allocated by the compiler on the scratchpad memory or the DDR memory respectively. Thus, programmers should be aware of the capacity of the scratchpad memory and lifecycles of the internal vector-typed variables, otherwise allocation failure will result in a compilation error. Assignments between internal and external variables result in load/store instructions. The vector operations can directly use the builtin intrinsic such as vadd and matmul for parallel computation, and expressions such as c = a + b (where a, b, c are vector-typed variables) are also supported for simple vector operations. The compiler can transform them into the corresponding vector instructions in the vector object.

Note that to support vector expressions, the value type of the expression a + b (where a or b are vector-typed) will be marked as *rv-value*, and *rv-value* must be assigned to an *l-value* internal vector variable to eliminate the *rv-value* property (e.g. c = a + b where c is a previously defined internal vector). The (intermediate) result of the expression is temporarily stored on the address of the assigned *l-value* (in this example, the address of c). Letting *rv-value* participate further computations will result in compilation error. The *rv-value* mechanism forbids the expression statements such as e = (a * b) + (c * d); or a + b;. The evaluation of such expressions must temporarily allocate for intermediate results, which messes up the memory management for the programmer. The above statements can be rearranged as e = (a = a * b) + (c = c * d); and c = a + b; respectively, to specify the destination of intermediate results explicitly and convert *rv-value*s into *l-value*s.

Built-in library calls are also provided for common operations, such as the computation of convolution or pooling layers. Calling built-in library will immediately invalidates all internal vector-typed variables to transfer the ownership of the whole scratchpad memory to the library. The code inside built-in library calls are generated by the compiler.

TABLE 3
Software and hardware features changed from a CPU-centic system to a DLPU-centric system.

	-	CPU-centric	DLPU-centric
OS functionality	System management	OS	Runtime
	Memory management	OS	Runtime
	Processor management	OS	Runtime
	I/O management	CPU+OS	Hardware controller/local service
	Device management	OS	Hardware controller/local service
	Internal transfer	Bus	Network transfer protocol
Hardware ft.	Computing modules	Two (CPU+DLPU)	One (DLPU)
	ISA	Two sets	One set (scalar+vector+matrix)
	System calls	CPU support	DLPU support
	Interrupts	CPU support	DLPU support

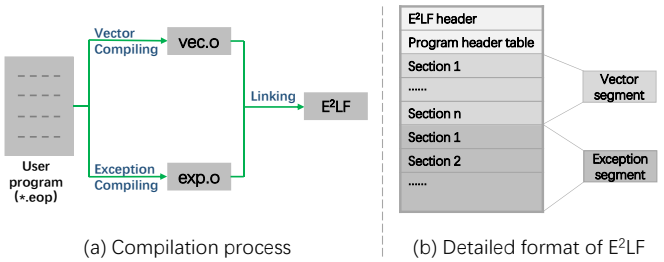


Fig. 8. The compilation process and the detailed format of the proposed E²LF executable.

5.3 Runtime system

The runtime system is crucial to the performance and programmability of the DLPU-centric system, and it mainly consists of three parts, i.e., DLPU management, memory management, and IO management.

DLPU management. The runtime system provides DLPU management during the entire execution of the binary, i.e., the loading, execution, and termination phase. During the loading phase, the E²LF binary is loaded and parsed through different sections. Note that the parsing of the exception segment is nontrivial as the program exception table (PET), where each entry stores the address to the corresponding exception handler binaries, should be configured. The entire process can be elaborated as follows. At first, both the number of sections and the total size of the exception segment can be retrieved from the header table for allocating the memory space for the PET and the entire exception binaries, respectively. After that, the offset of each section can be parsed from the E²LF file for attaining the starting address of the related section. Finally, the starting addresses of all sections will be filled into the PET. Once the entire PET is ready, additional PET register will be configured with the address of PET, and the program counter will also be configured for execution. During the execution phase, the __barrier intrinsic is exposed for the programmers to control the synchronization of the PPU and SPU. During the termination phase, the allocated resources such as memory should be reclaimed.

Memory management. Programmers are aware of both the main memory and the scratchpad memory of the PPUs, and the runtime system provides library functions for allocation/free of these memory spaces (i.e., malloc, free, spm_malloc, and spm_free). The memory copy functions are also provided for moving data within the main memory or between the main memory and scratchpad memory (all of them are using the memcpy function with different parameters). Moreover, the virtual/physical address mapping is also maintained.

IO management. As the proposed DLPU-centric system leverages the CPULESS DLPU with disaggregated hardware components, i.e., each peripheral device such as the network and disk has its own controller, the idea of disaggregated OS [25] can well fit in. In disaggregated OS, all the peripheral devices can employ a so-called *monitor* for serving various requests. Thus, in contrast to relatively heavy centric OS, the runtime system of disaggregated OS on the DLPU can be significantly simplified.

TABLE 4
Specifications of baseline and DLPU-centric systems.

-	Device	Name	Perf./Size@Freq.	Bandwidth	Die (mm^2)	Power
CPU-centric	CPU	Xeon 6130	2.10GHz	-	698 (14nm)	125W
	GPU	Tesla V100	125Tops@1312MHz	900GB/s	815 (12nm)	300W
	Bus	NVLink	-	300GB/s	-	-
	Memory	DDR4	512GB@4×2666MHz	85.32GB/s	-	20.12W
	Network	Ethernet	-	1Gb/s	-	4.4
	Disk	Hard Disk	2TB@7200rpm	207.92MB/s	-	5.15W
DLPU-centric	DLPU	CPULESS	2.08Tops@1GHz	64GB/s	16.94 (45nm)	2.27W
	Memory	DDR4	128GB@2×2133MHz	34.12GB/s	-	4.4W
	Network	InfiniBand	-	56Gb/s	-	7.18W
	Disk	Hard Disk	2TB@7200rpm	209.19MB/s	-	5.15W

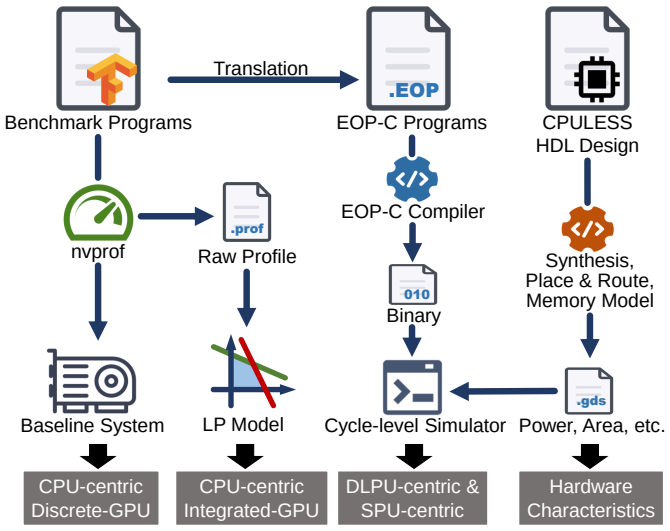


Fig. 9. The overall workflow of our evaluation process.

6 Methodology

In this section, we present the systems evaluated in this paper and how these systems are evaluated. The overall workflow of our evaluation is shown as Fig. 9.

CPU-centric systems. We select a commodity CPU+GPU system as one of our CPU-centric system baseline. The system adopts a Intel Xeon 6130 CPU as the control center, and adopts a discrete NVIDIA Tesla V100 GPU, which has 60× peak performance (125Tops/s) with respect to our DLPU-centric system, see Table 4. We deploy Tensorflow 1.6, with SIMD and CUDA 9.2 / cuDNN 7.2.1 support for CPU and GPU, respectively. We measure the GPU performance and

TABLE 5
Benchmarks.

Benchmark	Mode	Dataset	Batch Size
Natual Language Processing:			
Transformer [26]	Inference	WMT14	2
Reinforcement Learning:			
DQN [27]	Training	MsPacman	20
Graph Neural Network:			
GCN [14]	Training	Cora	2708
Image Classification:			
SENet [28]	Inference	CIFAR-10	1
ShuffleNet-V2 [29]	Inference	ImageNet	1
Object Detection:			
SSD (MobileNet-V1) [30]	Inference	VOC2007	1
Faster R-CNN (ResNet-101) [13]	Inference	VOC2007	1

energy cost using the official NVprof provided by NVIDIA with *persist mode* in order to avoid the impact of cold-start effect (power-up, about 10s). In each experiment, we only run a single DL task on the system to measure CPU and GPU without co-run affections. We use the Power_Gadget to measure the CPU, DRAM energy costs and use the pTop to measure the hard disk energy when executing benchmarks. We estimate the NIC energy costs based on its reported maximum power.

Besides the aforementioned real CPU-centric system with discrete GPU, we also develop an analytical model of a CPU-centric system with integrated GPU, which assumes that the CPU (Xeon 6130) and the GPU (V100) are integrated on the same chip and have zero memcpy cost. Technically, we obtained profile data from nvprof on the discrete GPU, which includes the starting and ending timestamp for each activity. We set the time of all memcpy activities to zero, and rearrange the timeline by solving the following linear programming problem: subject to the original partial order of CPU/GPU/Runtime/(zeroed)memcpy activities, minimize the total execution time. The reported data of the integrated GPU are based on the result given by the LP solver.

DLPU-centric system. The CPULESS DLPU in this paper is implemented in Verilog RTL. It has a peak performance of 2.08Tops/s with total 1.72MB on-chip memory and 128GB DDR4 main memory, see Table 4. It has a 34.12GB/s main memory bandwidth, which is 40% of the baseline system (85.32GB/s), and a 64GB/s inner bandwidth (between PPU and SPM), which is 7.11% of the baseline system. To better measure the CPULESS, we synthesis and place&route its RTL code with Synopsys toolchains under TSMC 45 nm technology to report the area cost and energy consumption. Besides the CPULESS DLPU, we build a cycle-level system simulator to obtain the traces of accesses to the hard disk and DDR4 memory. With these traces, we measure the concrete energy costs of hard disk and memory in a real system with the same configurations of the DLPU-centric system. The costs of Infiniband NIC is estimated with typical power. We implemented the EOP-C programming language and the EOP-C compiler toolchain as described in Section 5.2.

SPU-centric system. Moreover, we build a CPU-centric version of the proposed DLPU-centric system, where the SPU serves as the host with integrated PPU from CPULESS, i.e., CPU-centric system with integrated DLPU. To well illustrate the effectiveness of the proposed EOP model, such a system has the very same configurations (including OS functionalities and hardware features) as the measured DLPU-centric system, except the EOP model. For example, we use the simple SPU from CPULESS serving as the CPU in this system. Therefore, when processing the DL tasks, the only extra cost of this CPU-centric system is from the CPU-DLPU control signaling interactions, compared to the proposed DLPU-centric system.

Benchmarks. As listed in Table 5, we use seven representative DL algorithms from various domains as our benchmarks for evaluation. For CPU-centric systems, the benchmark programs are downloaded from public resources. We leave all settings in the benchmark programs as is. For the DLPU-centric system, the programs are faithfully translated into the EOP-C programming language and compiled with the EOP-C compiler toolchain.

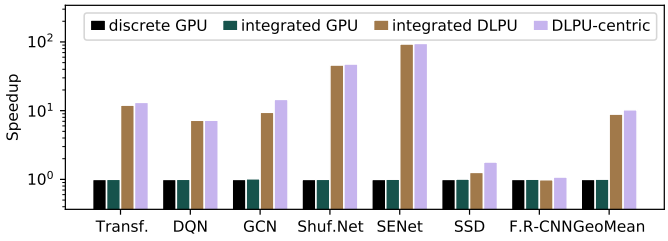


Fig. 10. Speedups of the DLPU-centric system when compared to the CPU-centric baseline systems.

7 Experimental Results

7.1 Characteristics of DLPU-centric system

In Table 6, we present the hardware parameters and layout characteristics of CPULESS, respectively. CPULESS has 32 VPEs where each VPE contains 32 PEs ($N_{VPE} = N_{PE} = 32$), a 32KB IB, a 32KB EB, a 8KB Cache, and a 1MB SPM. It is able to achieve 2.08T *bf16* operations per second while running at 1GHz. Overall, CPULESS has an area cost of 16.94mm^2, only 2.43% of the Intel Xeon Gold 6130 CPU (698mm^2 die area) and 2.08% of the V100 GPU (815mm^2 die area), respectively. The total power consumption of the DLPU-centric system is 19W, 4.18% of the CPU-centric system. Obviously, the DLPU-centric system is cost-effective, not only because the CPULESS is an efficient accelerator, but also because it saves the cost for the centric CPU.

7.2 Performance

Fig. 10 shows the performance of the proposed DLPU-centric system and three baseline system (CPU+discrete GPU, CPU+integrated GPU, and SPU+integrated DLPU), normalized to CPU+discrete GPU system. Overall, the DLPU-centric system achieves an average speedup of 10.30× when compared to CPU+discrete GPU, 10.22× to CPU+integrated GPU, and 1.16× to CPU+integrated DLPU, respectively.

Regarding the two CPU+GPU systems, the DLPU-centric system outperforms the most (95.2×/94.7×) on SENet due to the complex computation graph and limited parallelism among small computing blocks. The DLPU-centric system slightly outperforms the CPU-centric systems on Faster R-CNN and SSD with a speedup of 1.08×/1.08× and 1.79×/1.77× respectively. The reason is that the most time-consuming operations Faster R-CNN and SSD are regular vector operations, which can be well performed by GPUs. Moreover, as CPULESS takes over the control, the DLPU-centric system is able to avoid the cost on OS, driver,

TABLE 6
Hardware characteristics of CPULESS.

-	Area (mm^2)	(%)	Power (mW)	(%)
Whole Chip	16.94	100	2265.93	100
PPU	4.22	24.90	967.76	42.71
IEE	0.35	2.06	103.08	4.44
IB	0.36	2.12	25.28	1.12
Cache	0.05	0.27	3.16	0.14
Mem	2.98	17.59	312.67	13.80
DDR4 Controller&Phy	4.11	24.25	374.90	21.48
Other	4.88	28.80	104.18	5.97

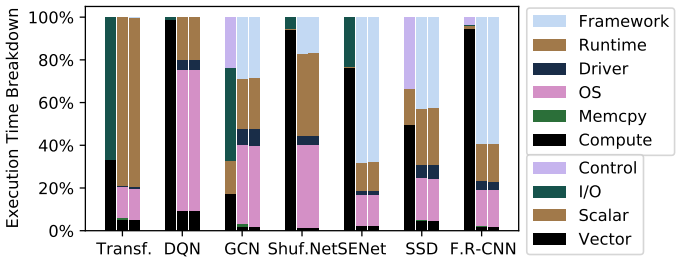


Fig. 11. Execution time breakdown. Left: DLPU-centric, Mid: CPU + discrete GPU, Right: CPU + integrated GPU

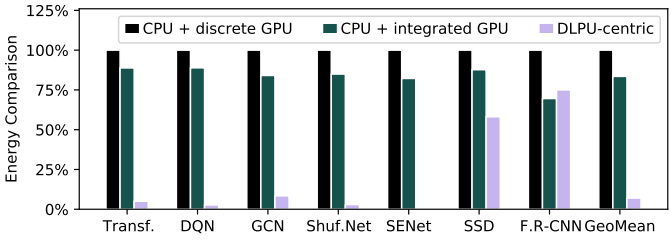


Fig. 12. Relative energy consumption comparison of the whole systems.

runtime and framework which are running on CPU in the CPU-centric systems, resulting in a 56.55% PPU utilization on average (vs. 2.87% and 2.89% in CPU+discrete GPU and CPU+integrated GPU, respectively), see Fig. 11 for the detailed execution breakdown. Particularly, for ShuffleNet-V2, almost all (99.5%) control operations on the DLPU-centric system can be hidden under the vector computation operation on its PPU with the EOP programming model, resulting in a PPU utilization rate as high as 94.11%, which is significantly higher than the GPU utilization rate of the CPU-centric systems (1.26%/1.27%).

Regarding the SPU+integrated PPU system, the DLPU-centric system could save the control signaling interactions between CPU and the integrated PPU, leading to an extra 15.60% improvement on average. Among all the benchmarks, the DLPU-centric system achieves the highest improvement (54.15%) on GCN from its 8.46 million interactions, the lowest improvement (0.13%) on DQN from its 701k interactions. Such results well demonstrate the essentiality and effectiveness of the EOP model in DLPU-centric systems.

7.3 Energy

Fig. 12 compares the total energy costs of the DLPU-centric system and the baseline CPU-centric systems. The DLPU-centric system can achieve an energy reduction of 92.99%/91.60% compared to baseline CPU-centric systems.

The energy benefit of the DLPU-centric system is mainly obtained from the following aspects. The DLPU-centric system eliminates the data exchange between CPU and DLPU in original CPU-centric systems, thus the energy is reduced by 17.40% on average. By reducing the control signaling energy cost, the DLPU-centric system is able to save 39.97% energy.

We also give out the system energy breakdowns in Fig. 13. It is interesting that the CPULESS DLPU, which occupies only 1.15% energy of the whole DLPU-centric system, takes over all the DL operations. As a comparison, to solve these operations, CPU and GPU need to consume 77.52%/92.85% energy of the CPU-centric systems. It well demonstrates that using CPULESS to replace traditional CPU/GPU is quite energy-efficient.

7.4 Discussion

CPU-centric CPU+accelerator system. We also evaluated a CPU-centric system with a DianNao [7] deep learning accelerator instead of a GPU. The DianNao under test has a peak performance of 0.52 TOP/s (1/4 of the DLPU-centric system) and a main memory bandwidth of 102.4 GB/s (3× higher than the DLPU-centric system). Specifically, the DLPU-centric system shows 5.23×/8.40× better performance over the CPU+accelerator system respectively on SSD and Faster R-CNN, because the DLPU-centric system eliminated the CPU-accelerator interactions.

SIMD-CPU-centric system. The intuitive idea of merging a SIMD functional unit into the CPU has the similar effect as the measured CPU-centric system with integrated PPU. However, the SIMD-CPU is not able to avoid the extra costs from OS, heavy CPU, context switching, and control signaling interactions. Especially, as a DLPU instruction may need to iteratively execute in the parallel processing unit for $\sim 10^5$ cycles, a squashed DLPU instruction must restart from the beginning after returning from the interrupt handler, resulting in a great loss of DLPU efficiency. As a piece of evidence, during the computation of Faster R-CNN on a CPU, it encounters averagely 260.36 interrupts per second. We develop a simulator for SIMD-CPU-centric system, whose SIMD-CPU has the same parallel processing unit, memory bandwidth and SPM size with CPULESS, and find that the frequent interrupts makes the SIMD-CPU-centric system 1.93× slower than the DLPU-centric system proposed in this paper.

CUDA by hand. Even implemented in a pure-GPU manner, i.e. avoid execution switch between CPU and GPU, the interaction wall still exists. In such approaches, the scalar control is performed by GPU with its scalar instructions (or masked SIMD instructions), which share the decode/issue unit with vector compute instructions. As a result, the scalar control flow and the vector data flow run alternately and sequentially on the GPU: the scalar control blocks the vector/matrix units (including CUDA cores and TensorCore). We implemented the GCN in CUDA by hand. The code only launches one single GPU call from the host, with the CUDA Dynamic Parallelism feature and the WMMA TensorCore API. The speedup DLPU-centric vs CPU-centric drops from 14.60× to 2.51×, but there is still a significant performance gap.

8 Related Work

DLPUs. Due to the success of deep learning, DLPU has become a hot topic in the computer architecture society. Many DLPUs have been proposed recently. GPUs are widely used in both academy studies and industrial products with the help of mature programming frameworks (e.g. Caffe, PyTorch, TensorFlow, MxNet) and libraries (CUDA, cuDNN). NVIDIA has released several different GPUs dedicated for

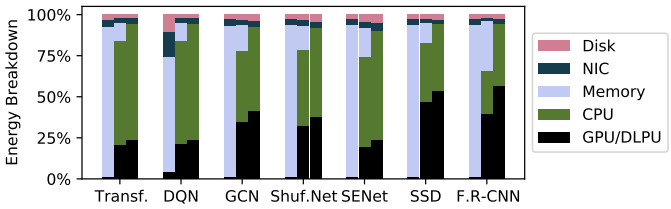


Fig. 13. Energy breakdown. Left: DLPU-centric, Mid: CPU + discrete GPU, Right: CPU + integrated GPU

deep learning, including the most recent Tesla V100. FPGAs are also used for implementing deep learning networks [4], [5], [6].

To avoid the inefficiency and improve performance, many customized accelerators are proposed as alternates to CPUs/GPUs/FPGAs. DianNao family [31], a series of vector-based high-performance accelerators, leverages the data reuse to minimize the data accesses in NNs for high efficiency. Cambricon [24] proposed a customized instruction set with scalar/vector/matrix operations to support different algorithm flexibly. Systolic architecture naturally fits for convolution operation while achieving internal data reuse with low input bandwidth requirement and thus been well exploit by researchers (ShiDianNao [9], Eyeriss [11], and TPU [10]).

Researchers also exploit hardware-oriented algorithms optimization and software/hardware co-design. To further improve the performance and energy efficiency, sparsity [32], [33], [34], [35], dynamic precision [36], [37], redundancy of inputs and weights [38], [39], sparsity irregularity [40] and structural compression [41] are successively addressed. However, the above accelerators still focus on improving the performance or energy-efficiency of DLPU as a coprocessor, without consideration of the interaction wall between CPU and DLPU. To the best of our knowledge, CPULESS is the first DLPU targeting to break the interaction wall.

Control-enabled accelerators. The earliest accelerator system with scalar control ability can be traced back to vector machines. Decoupled Vector Architecture [42] (e.g. Cray BlackWidow [43]) decouples scalar units and vector units to enable *run-ahead* execution, but the interference between scalar and vector flow is not studied. Many processors adopt a vector unit as a coprocessing core, including Tarantula [15], IBM Cell [16], Hwacha [44], but they do not implement a fused instruction pipeline, thus interactions between cores still exists. Recently accelerators coupled with a scalar control unit are proposed to enable end-to-end execution of the full workload without the support of CPU, e.g. Cambricon [24], Softbrain [45], but these accelerators did not focus on the interaction wall. More specifically, they do not have the exception-on-busy mechanism so the vector flow may blocks the execution of the scalar flow.

DL processing system. In existing DL processing systems, DLPUs are usually hosted by a central CPU as coprocessors. Amazon's DeepLens, a deep learning embedded video camera, has a central Intel Atom CPU. Google's TPU cloud is hosted by two Intel Skylake CPUs for every four TPU chips. NVIDIA's DGX-1/DGX-2 use two Intel Xeon E5 CPUs to host 8/16 V100 GPUs. Summit, a DL supercomputer, still has 9216 Power9 CPUs to host 27648 NVIDIA V100 GPUs. Microsoft's Brainwave uses Intel Xeon CPU to host deployed FPGAs in cloud [12]. Huawei integrated Neural Processing Unit in its Kirin 970/980 soc chips in its phones. All these real products, from sever-end to IoT devices, integrate centric host CPUs. Thus, they have to suffer from inefficient interactions between CPU and DLPU. As a comparison, we propose the first DLPU-centric DL computing system, which can be freed from the interaction wall.

9 Conclusion

In this paper, we focus on the major inefficiency source, i.e., *interaction wall* in CPU-centric DL processing systems, which includes *data exchange* and *control signaling*. To break the interaction wall, we proposed a DLPU-centric system which features an exception-oriented programming (EOP) model and the architectural support of CPULESS DLPU. Overall, the proposed DLPU-centric system achieves at most 10.30× better performance and 92.99% energy saving, respectively, than a CPU-centric deep learning computing system. Compared with a CPU-centric version of DLPU system where the SPU serves as the host with integrated PPU, the proposed DLPU- centric system still achieves 15.60% better performance from avoided interactions.

Acknowledgments

The authors thank Zhenxing Zhang, Yifan Hao and Zhe Fan from Institute of Computing Technology, Chinese Academy of Sciences and Guichun Wang from University of Science and Technology of China for their contribution to the experiments. This work is partially supported by the National Key Research and Development Program of China (under Grant 2017YFB1003101, 2018AAA0103300, 2017YFA0700900, 2017YFA0700902, 2017YFA0700901), the NSF of China (under Grants 61732007, 61672491, 61732002, 61702478, 61732020), Beijing Natural Science Foundation (JQ18013), Beijing "New Generation Artificial Intelligence Technology Cultivation Projects" (Z181100008918020), National Science and Technology Major Project (2018ZX01031102), the Transformation and Transfer of Scientific and Technological Achievements of Chinese Academy of Sciences (KFJ-HGZX-013), Key Research Projects in Frontier Science of Chinese Academy of Sciences (QYZDB-SSW-JSC001), Strategic Priority Research Program of Chinese Academy of Science (XDB32050200, XDC01020000, XDC08040102), Standardization Research Project of Chinese Academy of Sciences (BZ201800001), Beijing Academy of Artificial Intelligence (BAAI) and Beijing Nova Program of Science and Technology (Z191100001119093), Guangdong Science and Technology Program (2019B090909005) and Xplore Prize.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[2] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1–32, 1997.

[3] T. Mahlmann, A. Drachen, J. Togelius, A. Canossa, and G. N. Yannakakis, "Predicting player behavior in Tomb Raider: Underworld," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, Aug 2010, pp. 178–185.

[4] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "ESE: Efficient Speech Recognition Engine with Compressed LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016, pp. 3–8.

[5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*, pp. 161–170, 2015.

[6] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *Field-Programmable Logic and Applications*, 2016, pp. 26–35.

[7] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, Salt Lake City, UT, USA, 2014, pp. 269–284.

[8] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, 2015, pp. 609–622.

[9] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 92–104.

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-L. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. Mackean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, 2017, pp. 1–17.

[11] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[12] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. Van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[13] S. Ren, K. He, and R. Girshick, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in *Advances in neural information processing systems*, 2015, pp. 1–9.

[14] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[15] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec, "Tarantula: A vector extension to the Alpha architecture," *Conference Proceedings - Annual International Symposium on Computer Architecture, ISCA*, pp. 281–292, 2002.

[16] D. C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa, "Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, pp. 179–196, 2006.

[17] E. Fayneh, M. Yuffe, E. Knoll, M. Zelikson, M. Abozaed, Y. Talker, Z. Shmuely, and S. A. Rahme, "4.1 14nm 6th-generation core processor soc with low power consumption and improved performance," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, Jan 2016, pp. 72–73.

[18] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. Reinhardt, A. Caulfield, E. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[19] A. Krizhevsky, G. E. Hinton, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

[20] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recoginition," in *arXiv preprint*, 2015, pp. 1–14.

[21] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017, pp. 2980–2988.

[22] A. Brock, J. Donahue, and K. Simonyan, "Large scale GAN training for high fidelity natural image synthesis," 2018.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018.

[24] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An Instruction Set Architecture for Neural Networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 393–405.

[25] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," 2018.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[28] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[29] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 116–131.

[30] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[31] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "DianNao family: Energy-efficient hardware accelerators for machine learning," *Commun. ACM*, vol. 59, no. 11, pp. 105–112, Oct. 2016.

[32] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, vol. 16, 2016, pp. 243–254.

[33] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X : An Accelerator for Sparse Neural Networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[34] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 1–13.

[35] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN : An Accelerator for Compressed-sparse Convolutional Neural Networks," in *The 44th International Symposium on Computer Architecture (ISCA)*, 2017.

[36] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-Level Dynamically Com-

posable Architecture for Accelerating Deep Neural Networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2017.
[37] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-Serial Deep Neural Network Computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 6056, no. c, 2016, pp. 1–1.
[38] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: a Dejavu-Free Differential Deep Neural Network Accelerator," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.
[39] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.
[40] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-S : Addressing Irregularity in Sparse Neural Networks through A Cooperative Software / Hardware Approach," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.
[41] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-CirculantWeight Matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.
[42] R. Espasa and M. Valero, "A simulation study of decoupled vector architectures," *J. Supercomput.*, vol. 14, no. 2, pp. 124–152, 1999.
[43] D. Abts, A. Bataineh, S. Scott, G. Faanes, J. Schwarzmeier, E. Lundberg, T. Johnson, M. Bye, and G. Schwoerer, "The cray blackwidow: a highly scalable vector multiprocessor," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–12.
[44] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators," in *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, 2014, pp. 199–202.
[45] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 416–429.

Zidong Du (Member, IEEE) recieved the bachelor's degree in engineering from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2011, and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2016, with the guidance from prof. Yunji Chen, prof. Olivier Temam, and prof. Chengyong Wu. He is currently an associate professor at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

Qi Guo (Member, IEEE) received the B.E. degree in computer science from Tongji University, Shanghai, China, in 2007, and the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2012.,From 2012 to 2014, he was a Staff Researcher at IBM Research, Beijing. From 2014 to 2015, he was a Postdoctoral Researcher with Carnegie Mellon University, Pittsburgh, PA, USA. He is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include computer architecture, programming models, and machine learning.

Yongwei Zhao recieved the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2020, and the bachelor's degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2015. He is currently an assistant professor at the Institute of Computing Technology, Chinese Academy of Sciences.

Xi Zeng received the bachelor's degree in computer science and technology from Central South University, Changsha, China, in 2015. Currently she is working toward the PhD degree in the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, and the University of Chinese Academy of Science, Beijing, China.

Ling Li is currently a professor at the Institute of Software, Chinese Academy of Sciences, Beijing, China. She joined the Godson (Loongson) project, in 2009. She was the chief architect of Godson video decoding IP. She has authored or coauthored papers on journals (including the IEEE Transactions on Image Processing, IEEE Transactions on Parallel and Distributed Systems, IET Image Processing) and conferences (including DCC, SPAA, and ICASSP). Her research interests include intelligent computing and video processing.

Zhiwei Xu (Senior Member, IEEE) received the B.S. degree from the University of Electronic Science and Technology of China, Chengdu, China, in 1982, the M.S. degree from Purdue University, West Lafayette, IN, USA, in 1984, and the Ph.D. degree from the University of Southern California, Los Angeles, CA, USA, in 1987.,He is currently a Professor and the Chief Technology Officer (CTO) with the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. His prior industrial experience includes the Chief Engineer of Dawning Corporation (now Sugon as listed in Shanghai Stock Exchange), a leading high-performance computer vendor in Beijing, China. He currently leads the Cloud-Sea Computing Systems, a strategic priority research project of the CAS that aims at developing billion-thread computers with elastic processors.

Ninghui Sun (Member, IEEE) received the BS degree from Peking University in 1989, and the MS and PhD degrees both in computer science from the Institute of Computing Technology, Chinese Academy of Sciences in 1992 and 1999, respectively. He is a professor in the Institute of Computing Technology, Chinese Academy of Sciences. He is the architect of the Dawning series high-performance computers. His research interests include computer architecture, operating system, and parallel algorithm.

Yunji Chen (Senior Member, IEEE) graduated from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, China, in 2002, and recieved the PhD degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, China, in 2007. Currently, he is a full professor at Institute of Computing Technology, Chinese Academy of Sciences. He leads his lab to develop neural network processors. Before that, he participated in the Godson/Loongson project for more than ten years, and was a chief architect of Godson-3 microprocessor. Yunji Chen has authored or coauthored 2 books and over 90 papers. He was a recipient of ASPLOS'14 and MICRO'14 best paper awards for advances in neural network processors.