Enabling Energy-Efficient DNN Training on Hybrid GPU-FPGA Accelerators

Xin He
xinhe@hnu.edu.cn
Hunan University
Changsha, China

Jiawen Liu
jliu265@ucmerced.edu
University of California, Merced
Merced, USA

Zhen Xie
zxie10@ucmerced.edu
University of California, Merced
Merced, USA

Hao Chen
haochen@hnu.edu.cn
Hunan University
Changsha, China

Guoyang Chen
g.chen@alibaba-inc.com
Alibaba Group US Inc.
San Francisco, USA

Weifeng Zhang
weifeng.z@alibaba-inc.com
Alibaba Group US Inc.
San Francisco, USA

Dong Li
dli35@ucmerced.edu
University of California, Merced
Merced, USA

Abstract

DNN training consumes orders of magnitude more energy than inference and requires innovative use of accelerators to improve energy-efficiency. However, despite having complementary features, GPUs and FPGAs have been mostly used independently for the entire training process, thus neglecting the opportunity in assigning individual but distinct operations to the most suitable hardware. In this paper, we take the initiative to explore new opportunities and viable solutions in enabling energy-efficient DNN training on hybrid accelerators. To overcome fundamental challenges including avoiding training throughput loss, enabling fast design space exploration, and efficient scheduling, we propose a comprehensive framework, *Hype-training*, that utilizes a combination of offline characterization, performance modeling, and online scheduling of individual operations. Experimental tests using NVIDIA V100 GPUs and Intel Stratix 10 FPGAs show that, Hype-training is able to exploit a mixture of GPUs and FPGAs at a fine granularity to achieve significant energy reduction, by 44.3% on average and up to 59.7%, *without* any loss in training throughput. Hype-training can also enforce power caps more effectively than state-of-the-art power management mechanisms on GPUs.

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460371

CCS Concepts: • **Hardware → Power and energy**; • **Computer systems organization → Architectures**.

Keywords: DNN training, Energy efficiency, Hybrid accelerators, GPU, FPGA

ACM Reference Format:
Xin He, Jiawen Liu, Zhen Xie, Hao Chen, Guoyang Chen, Weifeng Zhang, and Dong Li. 2021. Enabling Energy-Efficient DNN Training on Hybrid GPU-FPGA Accelerators. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3447818.3460371

1 Introduction

Deep neural network (DNN) training is quickly emerging as a common yet heavy workload in HPC data centers. However, training DNNs often involves large data sets, high computation power and long training time. As a result, DNN training can be extremely energy-consuming. For example, training BERT (a large natural language processing model) with eight NVIDIA V100 GPUs for 12 days takes nearly 2.5 billion Joules [3, 17]. Unfortunately, the situation only gets worse as more complex DNN models are being developed. Therefore, it is imperative to explore novel approaches to effectively reduce energy consumption of DNN training [15].

Indeed, the main reason for the large energy consumption of DNN training is the use of GPUs. Leveraging the massive thread-level parallelism, GPUs have become the predominant processing platform for DNN training. However, GPUs are very power hungry. NVIDIA GPUs, such as V100 and P100 with thermal design power (TDP) of 300 Watts and 250 Watts, respectively, can easily account for more than 90% of the total system power during DNN training [27]. Nevertheless, it is challenging to reduce GPU power and energy without impacting training throughput. The prevalent method of

scaling down GPU core frequency to save power [20, 37, 40, 59] may degrade training throughput considerably, e.g., over 30% as observed when scaling down the V100 core frequency from the available 1530 MHz level to the 1320 MHz level.

Meanwhile, Field-Programmable Gate Arrays (FPGAs) are recently deployed in data centers [52]. FPGAs are highly customizable and especially good at handling streaming workloads in a pipeline fashion. Much success has been achieved in using FPGAs to accelerate machine learning inference [35, 45, 48, 70, 72], although limited works have explored the use of FPGAs for training [21, 31]. Compared with GPUs, FPGAs have much lower power consumption (e.g., 90W in Intel S10 FPGA vs. 300W in Nvidia V100 GPU). However, FPGAs also have relatively lower operating frequency and memory bandwidth (e.g, 14.9 GB/s in S10 FPGA vs. 900 GB/s in V100), which hinders their effective use in DNN training.

Despite their complementary features, GPUs and FPGAs have only been used independently so far for DNN training, in the sense that the entire training process is assigned to run either on GPU or FPGA alone. This neglects the rich diversity of individual operations (e.g., `MatMul`) in DNN training, and squanders the tremendous opportunity for improvement. For instance, some operations with specific inputs do not demand large thread-level parallelism or high memory bandwidth (Section 3.2), thus can be assigned to FPGA for comparable and sometimes even better performance but significantly less power, while other operations remain to run on GPU. The diverse characteristics of operations prompt us to rethink the current architecture and combine the strengths of GPUs and FPGAs for DNN training.

Research objective: In this work, we take the initiative to explore new opportunities and viable solutions in enabling DNN training on hybrid GPU-FPGA accelerators. Such hybrid architectures provide the flexibility to execute individual training operations on high-performance (i.e., GPUs) or power-efficient hardware (i.e., FPGAs) based on operation characteristics and problem input. We investigate such a hybrid accelerator system featured with GPUs and FPGAs to demonstrate the usefulness in two common and important use cases: (1) saving energy as much as possible *without throughput loss*, and (2) meeting a strict power cap while maximizing throughput. Such a power cap is often imposed in data centers [27, 30, 44, 68, 69, 71] to reduce production cost and improve system reliability.

Fundamental challenges: As no work has studied fine-grained DNN training optimizations on hybrid GPU-FPGA accelerators, we have identified three key challenges that must be addressed:

(1) How to avoid training throughput loss. Since FPGAs typically have lower operating frequency and memory bandwidth than GPUs, directly offloading operations to FPGA may result in large performance loss. For example, in our hybrid system consisting of three S10 FPGAs and one V100 GPU, offloading the most common operation `MatMul` to FPGAs saves power by 51W or 20% of the total system power, but causes 16.2% throughput loss on average.

(2) How to schedule individual operations. A DNN training workload usually has thousands or even millions of training steps; each training step has hundreds of operations; each operation has different choices of executing on GPU or FPGA. For a specific operation, depending on the input, the operation may also manifest varying computation and memory access patterns, leading to different choice of execution. Given the large number of combined choices of executing operations, how to schedule operations to minimize energy consumption or meet an enforced power cap is challenging.

(3) Optimizing performance of an FPGA kernel is time-consuming, up to tens of hours, depending on kernel design complexity, as the optimizing process involves lengthy placement and routing cycles to assign the nets of the circuit to routing segments and turn on programmable switches [67]. Each operation kernel may use various FPGA-specific configurations, such as the number of compute units (CU), the width of SIMD (kernel vectorization), the work group size, etc. Therefore, it is not practical in terms of time to exhaustively evaluate all possible configurations in the design space on FPGA to find the optimal one.

Our work: We propose a novel and comprehensive framework named *Hybrid performance-aware energy-efficient training (Hype-training)*, that enables effective DNN training on hybrid GPU-FPGA architectures. Hype-training hides the underlying architecture complexities from users and automatically maps individual operations onto GPUs and FPGAs while meeting energy, power and performance goals.

Specifically, to address the first challenge on performance, we analyze and characterize DNN training at the per-operation level. Using memory bandwidth utilization and IPC of operations on GPUs as indicators, we identify those operations that can perform comparably or even better on FPGAs than on GPUs. We further analyze task dataflow graphs of DNN training workloads and identify operations that are not on the critical path of the execution. Those operations are offloaded to FPGAs, only if offloading to FPGA plus the involvement of data movement does not prolong the critical path. We also implement a set of techniques to optimize FPGA kernel performance and avoid throughput loss, such as local memory buffering, loop tiling and double buffering.

To address the second challenge on scheduling, we develop a runtime system that schedules operations based on the above offline analysis and characterization of operations. The runtime system examines the runtime power consumption to determine which operations should be executed and in which order, such that the data movement and energy consumption are minimized while not violating user-specified power cap. Furthermore, Hype-training allows the co-existence of multiple FPGA kernels for a given operation,

each of which is optimized for one type of input. For a given input, the runtime system schedules a kernel that leads to the largest energy saving without performance loss. Such input-aware scheduling makes the best use of FPGAs for energy saving.

Last but not the least, to explore the large optimization space of a FPGA kernel, we propose performance models to decide which configuration combination leads to the best performance for a given operation. Our performance models are featured with being operation-specific and input-aware, which implicitly captures workload characteristics into models and enables lightweight yet accurate modeling. This distinguishes us from all the existing models [9, 60, 61, 72] that are generic but input-unaware (hence less accurate). With our performance models, users no longer need to evaluate each possible configuration directly on FPGA. This method reduces kernel optimization efforts from tens of hours to seconds, while achieving the same top-five accuracy.

To summarize, this is the first work that explores hybrid GPU-FPGA accelerators for energy-efficient DNN training. Our work demonstrates the great potential of using heterogeneous hardware resources in a fine-grained fashion for training acceleration. The main contributions of this paper are the following:

- We identify fundamental challenges in enabling effective DNN training on hybrid GPU-FPGA accelerators;
- We propose Hype-training, a software framework integrated with TensorFlow for operation scheduling and performance optimization on hybrid GPU-FPGA architectures. Hype-training is transparent to users and does not require modification of DNN models;
- Evaluation results show that Hype-training reduces energy consumption by 44.3% on average (up to 59.7%) without loss in training throughput, and being able to effectively enforce power caps with 10.1% performance improvement than without Hype-training.

2 Background

FPGA and OpenCL. FPGA is emerging as a promising accelerator for its energy efficiency, high performance, and customizability [35]. Intel's OpenCL SDK makes programming of FPGA much easier than the traditional methods. Users can employ the OpenCL SDK to develop and compile OpenCL kernel files to generate FPGA bitstreams. However, OpenCL compilation for FPGA is time-consuming (typically take hours for a kernel).

In OpenCL, work item is the basic unit of execution. A set of work items forms a work group, and a set of work groups are spawned when an OpenCL kernel is launched. Each OpenCL kernel can use different configurations to improve performance, such as the number of CU and SIMD width. Different configuration combinations consume different amount of FPGA resources (e.g., RAM blocks and number of DSPs), resulting in different power and performance.

DNN training. DNN training frameworks like TensoFlow use a dataflow graph where each node denotes a tensor operation (e.g., Conv2D); The graph specifies data and control dependencies between operations. A DNN model usually includes tens of kinds of operations. In each training step, an operation can be invoked by tens of times with different inputs sizes. Across training steps, the characteristics of many DNN training workloads typically remain stable and hence predictable [32, 33, 51], providing opportunities to use offline study to characterize operations for online scheduling. In the following, we use "operation" and "kernel" interchangeably.

We target common DNN models whose dataflow graphs do not exhibit data-dependent control, and each training step goes through exactly the same graph, which implies the input data of operations can be known before training. Such DNN models are very common and have been the targets of recent works [24, 32, 33, 54, 73].

3 Proposed Approach

3.1 Overview

We propose the Hype-training framework for systems with hybrid GPU-FPGA accelerators. As depicted in Fig. 1, there can be multiple GPUs and FPGAs attached to the system through the PCIe interface. CPUs are in charge of kernel allocation and scheduling operations to the associated GPUs and FPGAs at runtime. Power sensors are embedded in hardware components such as GPUs, CPUs and main memory, providing power information for runtime scheduling. These sensors widely exist in modern hardware [11, 18, 29].

GPU and FPGA, although both of them are used as accelerators, play different roles for DNN training. In particular, GPU is used to enable high performance execution of operations; FPGA is not used to provide superior performance over GPU; Instead, FPGA is used to run some operations traditionally executed on GPU to reduce energy and power.

Fig. 2 shows an overview of Hype-training. It includes two offline phases and a runtime scheduler. The two offline phases are used to determine the operation candidates that can be offloaded to FPGAs, and to optimize the performance of these operations. The first offline phase (Section 3.2) performs power and performance characterization of operations and makes FPGA-offloading decisions based on computation intensity and memory bandwidth utilization of the operations. The second offline phase (Section 3.3) optimizes performance of those FPGA kernels for offloading by choosing the optimal configurations based on performance models; The second offline phase also chooses small operations based on critical path analysis for FPGA offloading. The runtime scheduler (Section 3.4) schedules operations based on the

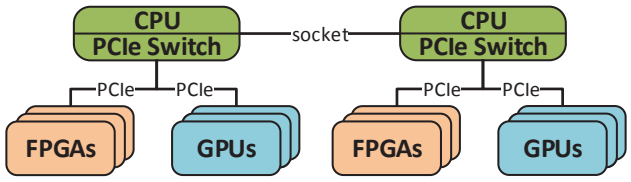

Figure 1. A heterogeneous system with GPUs and FPGAs.

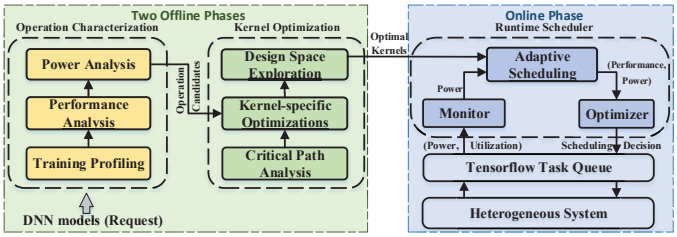

Figure 2. Overview of Hype-training.

offloading decisions and user requirement (either minimizing energy without reducing performance or maximizing performance under a power cap[a]).

3.2 Power and Performance Characterization

Table 1 lists the power and performance of seven operations that are predominant in DNN training. The first five operations can easily account for the majority of the total training time of a DNN model. For example, in Resnet50, the five operations takes more than 90% of the total execution time. Additionally, we study several small operations, such as `Transpose` and `Mul` listed in the table (others are profiled but not listed). Those small operations exist widely in DNN models. The characterization provides insights on how to offload operations to GPUs and FPGAs.

We study operations using tf-profiler [2] and nvprof [1]. These tools allow us to access hardware counters and correlate them with workload execution. The numbers presented in Table 1 are for one NVIDIA V100 GPU plus three Intel S10 FPGAs, but similar trends are observed for different number of GPU and FPGA combinations as discussed in evaluation. The memory bandwidth of a single V100 and S10 is 900 GB/s and 14.9 GB/s, respectively, and the peak operating frequency for V100 and S10 is 1530 MHz and 1000 MHz, respectively. A script is developed that runs each operation in the standalone mode in TensorFlow 1.8 [7]. The input to each operation is taken from six DNN models (Table 2). We use tensor cores in V100 to run an operation when the operation is eligible to use tensor cores in the six DNN models. We choose three inputs for each operation, representing cases of small, medium and large input sizes. Take MatMul with the dimension size (M, K, N) as an example. The small input size has 256<M<1024, 256<K<1024,256<N<1024; The medium one has 1024<=M<4096, 1024<=K<4096, 1024<=N<4096; The large one has any dimension size larger than 4096. We categorize the input size in the above way, based on detailed profiling on common machine-learning models and the impact of input size on execution time. When running operations on GPUs, the default configurations in TensorFlow are used. For a big operation such as `MatMul` or `Conv2D` with multi-dimensional inputs, we run it on three FPGAs for best performance, by partitioning the dimension size of input (i.e., rows and columns). For a small operation such as `Relu`, we run it as a whole on one FPGA, as opposed to three FPGAs, for best performance. Optimization techniques discussed in Section 3.3 are also applied. We use NVIDIA System Management Interface [46] and Powerplay Analyzer [12] to measure GPU and FPGA power, respectively. Each operation is run ten times on both FPGAs and GPUs to get average results.

Table 1 shows power consumption of an operation on GPU varies significantly, depending on the problem input. For example, the variance of peak power for the operation `BiasAddGrad` is up to 39%. Unlike GPU, the power variance due to input difference is much smaller on FPGA, e.g., less than 8% for `BiasAddGrad`. The table also shows that across operations, GPU has larger variance in power consumption. For example, the peak power of `MatMul` and `Transpose` differs by 68.4% on GPU but is only 6.1% on FPGA.

The above power variance across inputs and across operations can be attributed to three reasons. First, the power variance on GPU is closely related to the utilization of streaming multiprocessors (SMs). SMs are the most power-consuming elements on GPU. Different operations offer different thread-level parallelism and have different utilization of SMs. For example, a small operation such as `Mul` uses only one SM, causing it to use much less power than a larger operation such as `MatMul`. Besides, given an operation, different inputs have different SM utilization due to memory access patterns and usage of shared memory, which also leads to variance in power consumption. For example, `Conv2D` using all SMs has 53.3% variance in SM utilization when executed with different inputs, because of usage of shared memory.

Second, the power consumption of FPGA is dominated by that of RAM and DSP. With optimizations for kernels, the variance of dynamic power consumption of RAM and DSP for different operations with different inputs is relatively small, which leads to similar power consumption in FPGA.

Third, the difference between peak power and static power of FPGA is smaller than that on GPU. For the evaluated S10 FPGA and V100 GPU, the difference between peak and static powers is 33W and 261W respectively. Larger difference can trigger larger variance in power consumption.

Table 1 also reveals that regardless of the operation, running on FPGA always uses less power than on GPU. This observation is true for all problem inputs. Hence, without

[a]Unless indicated otherwise, "performance" refers to training throughput. Training accuracy is unaffected by any part of Hype-training, as we do not change task dataflow, training iterations, or FP precision.

Table 1. Profiling results of some of the representative operations in DNN training on FPGA (S10) and GPU (V100). The top five are large operations and the bottom two are small operations.

Operations	Input Size	FPGA			GPU				
		peak power (W)	average power (W)	execution time (ms)	peak power (W)	average power (W)	execution time (ms)	mem bw utilization (GB/s)	IPC
MatMul	Small	58	58	1.341	214	206	1.720	38.35	5.401
	Medium	65	65	3.551	274	270	4.137	48.05	5.549
	Large	75	75	7.932	302	300	7.144	57.72	5.764
Conv2D	Small	55	55	0.449	199	191	0.568	43.70	3.231
	Medium	60	60	1.825	272	262	2.344	60.31	3.342
	Large	63	63	4.663	296	292	4.088	71.93	3.357
Conv2DBackpropFilter	Small	70	70	2.137	202	199	1.314	214.70	3.982
	Medium	73	73	5.394	280	258	2.238	237.48	4.246
	Large	80	80	30.327	302	296	8.309	265.25	4.388
Conv2DBackpropInput	Small	83	83	2.227	210	206	0.874	245.04	3.891
	Medium	88	88	6.115	290	274	2.912	274.87	4.078
	Large	90	90	25.873	302	298	8.060	291.70	4.236
BiasAddGrad	Small	73	73	0.528	216	214	0.289	163.06	5.281
	Medium	78	78	1.413	266	256	0.556	176.69	5.724
	Large	80	80	5.905	300	292	1.779	185.57	6.007
Mul	Small	13	13	0.009	99	95	0.010	7.75	1.036
	Medium	15	15	0.011	105	99	0.014	8.76	1.117
	Large	18	18	0.023	113	109	0.022	9.62	1.239
Transpose	Small	13	13	0.006	83	81	0.006	7.15	1.254
	Medium	13	13	0.080	93	89	0.009	8.74	1.311
	Large	15	15	0.012	97	95	0.010	9.41	1.503

considering performance impact of FPGA, running operations on FPGA is always preferred.

Regarding performance, Table 1 shows that the performance of some operations (particularly `Conv2D` and `MatMul`) on FPGA is very close to or even better than on GPU, although GPU has a higher memory bandwidth and operating frequency than FPGA. For example, for the operation `MatMul`, the performance with a large input on FPGA is only 11.1% worse than on GPU (7.932ms vs. 7.144ms), and the performance with a small input on FPGA is even 22.1% better than on GPU (1.341ms vs. 1.720ms).

To study the reason for the above observation, we measure memory bandwidth and IPC when running `Conv2D` and `MatMul` on GPU, shown in Table 1. Table 1 does not report memory bandwidth utilization for FPGA, because it is not measurable by any tool but bounded by the the peak memory bandwidth 14.9 GB/s. We find that the two operations consume relatively small memory bandwidth (much smaller than `Conv2DBackpropFilter` and `Conv2DBpropInput`, two most time-consuming operations). Also, the operation `Conv2D` has a relatively low IPC. This indicates that the performance of the two operations are not affected too much by the low memory bandwidth and operating frequency of FPGA. `MatMul` with the small input has worse performance on GPU, because this operation cannot offer many thread-level parallelism for GPU, while the customized pipeline execution scheme in FPGA is beneficial to run this operation.

Table 1 also shows that for some operations, FPGA leads to much worse performance than GPU. For example, for the operation `Conv2DBackpropInput`, FPGA is 110% worse than GPU. These operations are characterized with high memory bandwidth utilization. This utilization is 291.70 GB/s for `Conv2DBackpropInput`, which is much higher than that of `MatMul` (57.72 GB/s at most) where FPGA outperforms GPU.

We summarize the above **observations** as follows.

1. For an operation with different inputs, the variance in power is large on GPU but small on FPGA;
2. Different operations have different power consumption; Such difference is much bigger on GPU than on FPGA;
3. Running operations on FPGA always uses less power than on GPU;
4. Some operations on FPGA can lead to comparable or even better performance on GPU.

We repeat the tests on other GPUs and FPGAs, including NVIDIA K80 and Intel Arria 10, and vary the number of GPUs and FPGAs, all resulting in the same observations.

Implications of the observations. *Observation 1* indicates that we must consider the impact of input on performance. Based on this, our performance models in Section 3.3 include input information as model parameters.

Observation 2 motivates us to use operation-specific modeling to estimate performance instead of building a generic model for all operations. The operation-specific model easily introduces operation-specific input into the models. Moreover, since the number of power-consuming or time-consuming operations is small (less than ten) and repeatedly used in DNN models, we do not need to build many models, making operation-specific performance models viable.

Observation 3 provides us useful hints on runtime scheduling (Section 3.4). If there is no power cap, there is no need to consider the power consumption of operations on FPGA, because running operations on FPGA always saves power. We just need to focus on offloading an operation to the device that has the highest performance for that operation.

Observation 4 gives us hints on which operations to select to run on FPGA. In particular, the memory bandwidth utilization and IPC of an operation running on GPU can be good indicators of whether running the operation on FPGA can cause performance loss (compared with running on GPU).

Based on the above, we set up two thresholds, one for memory bandwidth (trd_{mem_bw}) and one for IPC (trd_{IPC}).

An operation whose memory bandwidth is smaller than trd_{mem_bw} or IPC smaller than trd_{IPC} on GPU is a candidate to be offloaded to FPGA. This candidate is then optimized for performance (Section 3.3) and potentially scheduled by the runtime (Section 3.4). Using the above threshold-based method, we can save the effort on FPGA kernel optimization by eliminating unpromising operations early on. The two thresholds are obtained by running several common operations with a range of problem inputs and studying when the performance on FPGA is better than on GPU. We found that using two operations `MatMul` and `Conv2D` with various problem inputs is sufficient to represent various memory access intensity and computation parallelism, hence effectively building a filter to offload operations. Each threshold is the maximum value of the thresholds from the two operations. We use *maximum*, which is conservative to offload operations to FPGA in order to avoid performance loss. However, our experiment results suggest that the case of missed promising operations due to this conservative selection is rare (less than 1%) and that the missed operations tend to be small so do not contribute much to the overall training energy consumption.

3.3 Offline FPGA Kernel Optimization

After determining which operations to offload to FPGA based on workload characterization, we implement them in OpenCL and optimize their performance through design space exploration and critical path analysis.

Enabling fast design space exploration. To optimize the performance of an FPGA kernel, we need to find the best combination of the following four configuration parameters that have the largest influence on kernel performance. Each parameter itself represents a challenging trade-off:

The number of CU is critical to the construction of kernel pipeline and FPGA hardware utilization. The kernel pipeline can be replicated multiple times to generate multiple CUs and achieve higher throughput. However, more CUs require more hardware resources, which tends to reduce FPGA operating frequency that in turn hurts throughput.

The SIMD width determines how many work-items are executed in a single instruction. Larger SIMD width increases data processing efficiency due to potential memory coalescing. Meanwhile, wider SIMD puts greater pressure on memory bandwidth and may also reduce performance due to increased chance of control path divergence.

The degree of loop unrolling can be challenging to decide, as a larger degree of loop unrolling exposes more computation concurrency but also requires more FPGA fabric resources (e.g., DSP and RAM) that are shared by other components.

The work group size determines the number of work-items per group. A larger work group size allows compilers to apply aggressive optimization to tap hardware resource without using excess logic, but is limited by available shared FPGA resources and the maximum supported work group size.

Identifying the best combination of the above four parameters is time consuming. Taking the operation `MatMul` for example, compiling and running each combination takes around 8 to 15 hours, and the number of combination is 20,480 on our FPGA (S10). To avoid the time-consuming process of evaluating each combination, we introduce performance models to estimate and compare the performance of various combinations indirectly. Specially, given operation input and configuration parameters, the performance is evaluated via a new metric that we refer to as the latency indicator (LI). LI is operation specific. We use `MatMul` as an example below, and the modeling of other operations follows the same method. The LI for `MatMul` ($A \times B$) is defined as:

$$\texttt{LI} = \frac{A_{height} \times A_{width} \times B_{width}}{work\_group\_size \times F_{\max}(cu\_num, simd\_w, ul)} \quad (1)$$

In Equation 1, A_{height} and A_{width} ($A_{width} = B_{height}$) are the dimension sizes of the 2D input A, and B_{width} is of the 2D input B, so the numerator in the equation quantifies the total computation in the operation. F_{max} is FPGA operating frequency, which is a function of the number of CU (cu_num), width of SIMD ($simd\_w$), and degree of loop unrolling (ul). A smaller `LI` indicates better performance.

In essence, the performance of an kernel is related to the operating frequency F_{max} and input size. Smaller input and higher operating frequency lead to better performance, both of which are captured in the latency indicator. F_{max} works as a bridge to connect the three configuration parameters with the performance. F_{max} reflects the trade-off between hardware resource constraints and the desire of using more CUs, wider SIMD and higher degree of loop unrolling for better performance. The equation also considers the impact of work group size on performance: larger work group sizes lead to higher performance, subject to hardware constraints.

Given an operation and its input size, we use Equation 1 to determine the optimal configuration using the following method. For each combination, we use the vendor compiler (Intel FPGA SDK for OpenCL offline compiler) to get F_{max}. Getting F_{max} for a combination of configurations with the compiler takes only 1-5 seconds. For an operation, there can be hundreds of combinations, and getting F_{max} for all of them only takes hundreds of seconds in total, much less than the time of compilation and execution of FPGA kernel with various configurations. The input sizes, A_{height}, A_{width} and B_{height} are obtained by offline analysis. For many DNN training workloads, once their hyperparameters (e.g., batch size) are determined, the input sizes for each operation can be known before the training takes place [24, 32, 33, 54].

We use `MatMul` with a specific problem input as an example to illustrate the usage of performance model. Suppose two input A (2048x1024) and B (1024x1024), and there are two combinations. We want to determine which one can lead to better performance. The first one has 2, 1, 128, and 64 as

the number of CU, SIMD width, work group size and degree of loop unrolling respectively; The second one has 4, 2, 64, and 64, respectively. We use OpenCL SDK to generate F_{max} for the two combinations, which are 210 MHz and 241 MHz. Based on Equation 1, LI of the two combinations are 81.81 and 142.57, respectively. Hence, we estimate that the first combination likely performs better because of a smaller LI.

Our performance models have two features, operation specific and input awareness. The two features distinguish the models from all the existing efforts that build generic models to predict performance of OpenCL programs for FPGAs [60, 72]. Our proposed models turn out to be extremely efficient and accurate in predicting the relative performance of different combinations, as shown later in Section 5.5.

Critical path analysis for FPGA kernel offloading. Besides using the characterization study to choose operations for FPGA offloading, we also use critical path analysis to choose those small operations that are not in the critical path to offload to FPGA. Running those small operations on FPGA might incur longer latency than on GPU. However, as long as the increased execution time is not exposed to the critical path, there will be no loss in training throughput.

We use the following method for the offline critical path analysis. We utilize the algorithm proposed in [39] and adapt their open-sourced simulator [38] to get the critical path. Based on that, we choose which small operations for offloading to FPGA. For any operation not in the critical path, we implement it with OpenCL and measure its performance on FPGA. The operation is offloaded only when the execution time of the operation on FPGA plus the data movement time between CPU and FPGA is shorter than the critical path, thus not prolonging the critical path.

Other FPGA kernel optimizations. We also apply a set of common optimization techniques to improve the performance of FPGA kernels, including local memory buffering on FPGA's on-chip scratchpad memory to reduce expensive off-chip RAM accesses, loop tiling to improve data locality, and double buffering to reduce data movement time.

We further optimize the operation performance via multi-kernels for input adaptiveness. Particularly, we generate three kernels for each FPGA operation. This is based on the fact that an operation with different input sizes needs different configurations for the best performance. Given an operation, its inputs used in the training are first collected and then grouped by size into three types, i.e., small, medium and large. For each type, we use the latency indicator-based approach to decide the best configuration and generate a kernel. Therefore, we have three kernels for an operation. They are combined into one single FPGA bitstream file. Before DNN training occurs, Hype-training loads the combined bitstream file into FPGA upon OpenCL initialization. During the training, Hype-training locates the kernel based on the operation input size to perform the operation.

After FPGA kernel optimization, we run the kernels on FPGA and compare with their counterpart on GPU. Only when an FPGA kernel outperforms the corresponding GPU one, can the kernel be used as a candidate for runtime scheduling on FPGA (next subsection). This makes sure that performance loss is avoided. It also eliminates the unpromising operations mistakenly selected by the threshold-based method during offline characterization.

3.4 Runtime Scheduling

The runtime system schedules operations based on the offline kernel characterization (Section 3.2) and offline critical path analysis (Section 3.3). The runtime system schedules operations for two use cases: *Case 1*: minimizing energy consumption without loss in training throughput; *Case 2*: achieving the highest possible performance (throughput) without violating a given power cap.

The runtime system assumes that the following information is available when scheduling operations. The peak and average power consumption of the operations running on GPU (P^{GPU}_{peak} and P^{GPU}_{ave}) and the power consumption of the operations running on FPGA (P^{FPGA}) is known. The above information can be collected through the offline characterization study. This is feasible because a DNN framework usually has a limited number of operation types and the operations are repeatedly used within a model and across models. The cost of offline characterization is acceptable and amortized. The same methodology is used in the existing work to direct runtime scheduling (e.g, [32, 33, 36, 54]). We discuss how the runtime schedules operations for the two cases as follows.

Handling Case 1. The runtime system examines the task queue in TensorFlow, whenever a GPU or FPGA becomes idle after finishing the previous operations. In the task queue, there are operations with dependency resolved and ready to launch. The runtime system will schedule operations to run on the idling GPU (or FPGA), if there are pending operations scheduled to run on that GPU (or FPGA) based on the offline analysis. If GPU is idle and there are only FPGA operations ready in the queue, the runtime system will run the operations on GPU without waiting (thus no worse than without Hype-training). If an operation has multiple FPGA kernels, each corresponding to one type of input size, the runtime system selects which kernel to run based on the operation input as discussed previously.

Handling Case 2. The runtime system examines the power sensors and the task queue in TensorFlow, whenever a GPU or FPGA becomes idle after finishing the previous operations. From the power sensors, the runtime system gets the current system power consumption. The runtime will schedule an operation whose peak power on the idling device (GPU or FPGA) would not lead to a violation of the power cap if scheduled to that device. If there is no such operation, the runtime system will wait until a new operation is ready.

This ensures that the system does not violate the power cap. When a GPU and an FPGA becomes idle at the same time, without violating the power cap the runtime system uses GPU, in order to achieve the highest performance.

Reducing data movement latency. Prior work has reported that the data movement between CPU and FPGA memories is much faster than that between CPU and GPU memories [10]. This is consistent with the observation in our test system, where moving 512 KB data between an Intel 12-core CPU and S10 FPGA through PCIe 3.0 is 0.547 ms, but moving the same data through PCIe 3.0 between the same CPU and an NVIDIA Tesla V100 GPU is 1.198 ms. Therefore, using FPGAs does not introduce longer data movement time compared with GPU-only system.

However, there is a chance that two operations with dependency previously scheduled to run on GPU may now run on GPU and FPGA separately. In particular, when one operation is finished on GPU and the other one (referred to as the target operation in the rest of this discussion) is about to run on FPGA, there is an extra data movement from GPU to CPU and then from CPU to FPGA. To reduce this extra data movement time, the runtime system tries to overlap the data movement with computation on GPU as much as possible. To achieve that, the runtime system examines the next operation to run on GPU. The execution time of that operation has already been known from the offline profiling and kernel optimization. If the data movement time to offload the target operation to FPGA is shorter than the execution time of the operation right before the target operation on GPU, then the target operation is offloaded to FPGA. Otherwise, it does not run on FPGA to avoid performance loss. Note that the execution time of the target operation on FPGA does not need to be considered here, because after the offline kernel profiling and optimization, the target operation on FPGA is expected to perform better than the counterpart on GPU or off the critical path.

For example, in AlexNet, the operation `Conv2D` can be offloaded to FPGA without suffering from performance loss caused by data movement, when the immediately previous operation running on GPU is `BiasAddGrad`, whose computation time (5.146 ms) on the V100 GPU is bigger than the data movement time (4.931 ms).

4 Implementation

Hype-training is designed and implemented as a generic and extensible optimization framework based on TensorFlow. For some operations (e.g., `MatMul` and `Conv2D`), we use their OpenCL implementation from open-sourced DNN libraries (i.e., Intel clDNN and clBLAS) and apply the set of optimizations discussed in Section 3.3. For some operations (e.g., `Add` and `Mul`), we implement by ourselves with OpenCL. Those implementations are optimized with compiler-assisted optimization directives to expose more concurrency.

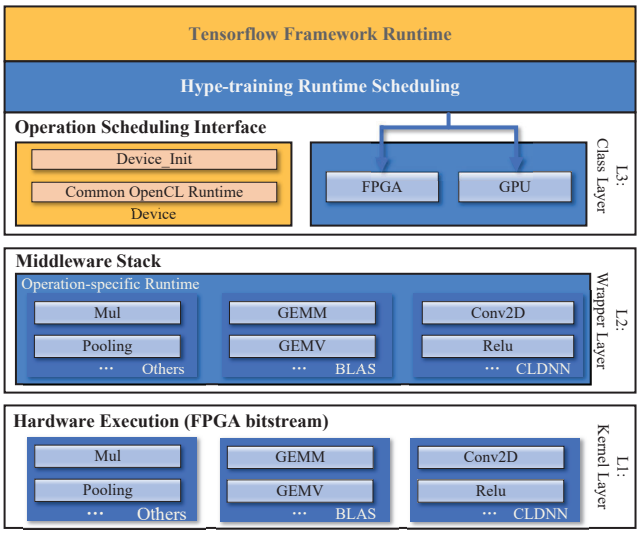

Figure 3. The hierarchical software design of Hype-training. The blue boxes are new contributions from Hype-training.

To support scheduling operations to run on FPGAs in DNN frameworks, we introduce a hierarchical software design to enable easy integration and flexible scheduling of FPGA-based operations. The essence of our design is to build a middleware to decouple the TensorFlow runtime system from FPGA-vendor specific runtime system. Fig. 3 delineates the proposed software design at the abstraction level, with our contribution highlighted in blue.

Our design has three layers. L3 (the top layer) encapsulates the OpenCL kernel designs and provides a uniform interface for each operation which may be implemented differently on different FPGAs. Given L3, the TensorFlow runtime system can manage those operations as usual. The runtime of Hype-training is an extension of the TensorFlow runtime.

L2 (the middle layer) and L3 build a middleware to separate the TensorFlow runtime from the OpenCL runtime for FPGAs. The OpenCL runtime consists of two parts: one common runtime provided by FPGA vendors to create context, command queue and memory allocation at L3, and the other runtime specific to operations and provided by us to configure, launch and release kernels at L2. The common runtime provided by the vendors is shared by all the kernels. With L2 and L3, adding support for FPAGs does not require disruptive change to the TensorFlow runtime.

L1 (the lowest layer) includes FPGA design files (i.e., bitstreams to run on FPGA) for potential operations to be offloaded to FPGA. The design files are generated from OpenCL-based programs by FPGA-vendor specific compilers.

The bitstream files are preloaded into FPGA before training starts, such that it does not impact training performance. To provide consistent interfaces to invoke operations on GPUs and FPGAs, we define a set of unified interfaces for kernel invocation. Overall, our implementation for runtime

scheduling incurs negligible overhead with only 0.1% execution time overhead and no memory overhead on the host.

Supporting a new operation in Hype-training goes through the following workflow: (1) implementing the operation in OpenCL; (2) changing the TensorFlow Op kernel module to integrate the operation; and (3) performing offline operation characterization. In our experience, it typically takes 5-8 hours (excluding OpenCL programming efforts).

Discussion on deployment cost. In our architecture, FPGA works as an accelerator to reduce energy consumption of GPU. This design introduces extra costs, including the ownership cost of FPGA and static power consumption of FPGA when the system is idle. Given the considerable energy saving, we argue that the ownership cost can be paid off in the long term. For example, on our platform with three S10 FPGAs ($15000) per V100 GPU, using Hype-training saves 1.29 million Joules per hour (the average saving in our evaluation) leads to $17900 saving in one year. Also, FPGA can be turned off to save static power when the system is idle. Hence, the deployment cost of our system is acceptable.

5 Experimental Evaluation

5.1 Experimental Setup

Experimental platforms. Our experimental system contains two Xeon E5-2630 CPUs, two GPUs (V100) and six FPGAs (Intel S10, whose production model is GX 2500). Each GPU and FPGA can be individually disconnected or disabled to create various configurations for testing and comparison. GPUs and FPGAs are attached to the server by PCIe 3.0. Note that CPUs do not compute operations as CPUs are in charge of scheduling and their power consumption is usually greater than FPGA. The operating system is Ubuntu 16.04. We use *training samples per second* to measure training throughput. We use *training throughput per Watt* to measure energy efficiency (which can be calculated as: total_ops/total_energy = (ops/second×t)/(power×t) = throughput/power). Unless indicated otherwise, all tests use the default GPU setting without power capping.

Tools. We use Intel FPGA SDK for OpenCL 19.1.0 for programming FPGA and use TensorFlow 1.8 [7]. We use CUDA 10.1 [4] and cuDNN 7.5 [14] for GPU. The measured system power includes both the dynamic and static power consumption of CPUs (including memory), GPUs, FPGAs, and data movement between CPUs and GPUs/FPGAs. This is achieved by using a collection of industry-standard tools including NVIDIA System Management Interface [46], Intel Running Average Power Limit (RAPL) Interface [5], and Intel Powerplay Analyzer [12].

Workloads. We use four ImageNet winner DNNs including Alexnet [28], Inception3 [57], Vgg16 [53] and Resnet50 [22]. We also evaluate DCGAN [49] and BERT-Large [3]. For the first four models, Imagenet is used as the training dataset [16]. For DCGAN and BERT-Large, we use CelebA [34] and SQuAD [50] as the training dataset respectively. The training batch size for BERT-Large is 10, and for other models is 256, according to [6]. When running an operation on multiple FPGAs, we use the method described in Section 3.3.

5.2 Overall Results

We compare GPU-only (one GPU) and GPU-FPGA (one GPU + three FPGAs, enabled by Hype-training) in this subsection, and other combinations of GPUs and FPGAs are presented in the next subsection. Fig. 4 compares the performance, average power, and peak power for the six models. Table 2 gives the details on which operations are offloaded and their ratios to the total number of operations and execution time. In general, if a larger portion of operations is offloaded to FPGA, there tends to be larger improvement on the power and energy efficiency, as presented below.

Performance. Fig. 4 shows that, compared with GPU-only, there is no loss in the training throughout in Hype-training for all the models. The strict no-throughput-loss in Hype-training is attributed to our operation characterization study and the critical path analysis that only choose operations that do not cause performance loss to offload to FPGAs. In fact, there is even an average improvement of 10.1% in throughput across the six models for two main reasons.

First, after offloading some operations to FPGA, these operations no longer compete for GPU resources, such as memory bandwidth and GPU SM cores, with other operations on GPU. This allows operations that are not offloaded to FPGA to deliver higher performance on GPU. For example, as shown in Fig. 5(b), when some operations are offloaded to FPGA, the performance of other operations (e.g., `Conv2DBackpropFilter` and `Conv2DBackpropInput`, the top two most time-consuming operations for the first five models) on GPU is improved by 17.8% on average, and the performance of *MatMulGrad* and *Dropout* in BERT-Large on GPU is improved by 17.1%.

Second, the data movement time between CPU and FPGA is smaller than between CPU and GPU, as discussed in Section 3.4. Moreover, as operations are offloaded to FPGA, the data movement time may be partially overlapped by the runtime scheduler. Fig. 5(a) supports this analysis, showing that the data movement time exposed to the critical path (i.e., the non-overlapped part) is reduced by 11.1% on average, after some operations are offloaded to FPGA.

Power consumption. Fig. 4 also shows that GPU-FPGA with Hype-training consumes 23.3% less average power than GPU-only. In particular, Resnet50 has the largest power savings (28.3%), and Alexnet has the smallest (20%). This is expected as Resnet50 has the large portion of computation (41.8% of execution time in Table 2) offloaded to FPGA.

Energy efficiency. Fig. 5 (a) plots the energy efficiency improvement compared with GPU-only. Hype-training achieves a significant improvement with an average of 44.3% (up to 59.7%). This improvement in energy efficiency comes from

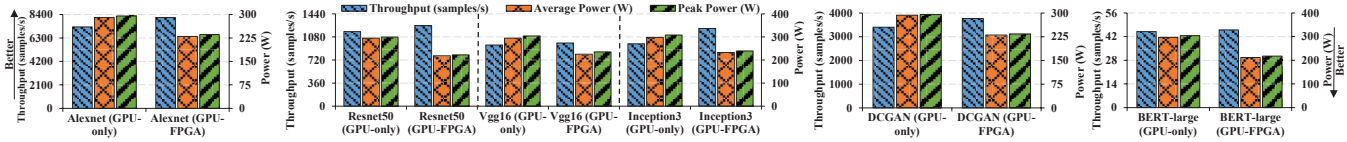

Figure 4. Training throughput and power consumption for GPU-only (one GPU) vs. GPU-FPGA (one GPU+three FPGAs).

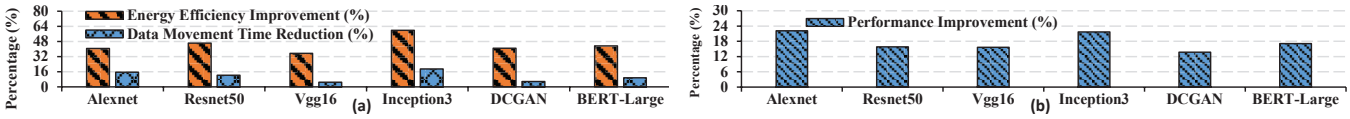

Figure 5. (a) Improvement in energy efficiency and reduction in data movement time with Hype-training. (b) Performance improvement for some operations on GPU, after offloading some operations to FPGA.

the large power savings in GPU-FPGA with no performance loss (thus higher throughput to watt ratio).

5.3 Results for Other GPU and FPGA Combinations

We evaluate Hype-training with different number of GPUs and FPGAs in the hybrid accelerator system.

Multiple GPUs. To use multiple GPUs for training, we use the data parallelism-based training [65], which builds on top of the Mirrored Strategy in TensorFlow. This strategy duplicates the model on each GPU device. Model parameters are updated synchronously among multiple GPUs. Fig. 6 compares the results of GPU-only (using 2 GPUs) and GPU-FPGA (2 GPUs and 6 FPGAs). Fig. 8(a) shows the energy efficiency improvement for GPU-FPGA over GPU-only. On average, Hype-training achieves 11.2% performance improvement while saving energy and power by 46.6% and 24.1%, respectively, compared with GPU-only.

Two GPUs vs. one GPU + one FPGA. Fig. 7 presents the training throughput and power consumption. We see that GPU-only (2 GPUs) performs 85.1% better than GPU-FPGA in terms of training throughput, because of GPU-only has an extra GPU. However, this does not mean that Hype-training has performance loss. Hype-training does not aim to achieve better performance than 2 GPUs with only one GPU and one FPGA (which is unlikely, if not impossible). Rather, Hype-training aims to utilize available FPGA(s) in addition to GPU(s) in the system to achieve the same training throughput while reducing the total energy or without violating the power cap. In this particular case, GPU-only (2 GPUs) actually consumes 124% more power than GPU-FPGA, which may not be acceptable in cases with power capping (next subsection presents results under power caps in more detail). It is worth noting that GPU-FPGA achieves an average of 20.9% better energy efficiency than GPU-only, as shown in Fig. 8(b).

Different numbers of FPGAs. We also evaluate scenarios where the number of FPGAs increases gradually from 1 to 4 to process training operations coupled with one GPU. Although detailed results are not presented here due to space limitation, the proposed Hype-training has no throughput loss in any of the cases, while achieving varying degree of power savings (e.g., 14.2% with 4 FPGAs) and energy savings (e.g., 40.4% with 4 FPGAs). This further illustrates the potential of hybrid GPU-FPGA accelerators and the effectiveness of Hype-training.

5.4 Evaluation with Power Capping

We evaluate if Hype-training can meet power caps while maximizing performance. For comparison, we use the power management mechanism (PMM) on GPU [27, 30], which changes core frequency and voltage to adjust power. Both Hype-training and PMM try to keep GPU power consumption under a specified power target as much as possible due to the concerns on system reliability and cooling [27, 71]. With a system-wise power cap of 230 Watts, Fig. 9 plots the runtime power consumption for 40 iterations during training, and Table 3 compares the training throughput.

As can be seen from Fig. 9, Hype-training consistently keeps the system power below the power cap. In fact, there is about a 50-70 Watts power margin between the power cap and runtime power; whereas for PMM, the power cap is sometimes violated, indicating the inefficiency of PMM on GPU. Meanwhile, Hype-training has 11.3% better throughput than PMM on average, according to Table 3. The improvement in throughput comes from smaller data movement time and less resource contention, as discussed previously.

Besides the above 230 Watts power cap, we also test two other power caps, 250 Watts and 210 Watts. For PMM, in the case of 250 Watts, among the six models, Vgg16 has the largest number of power cap violation (127459 times); In the case of 210 Watts, Resnet50 has the largest number of power cap violation (306811 times). In contrast, Hype-training has zero violation in both cases for all the models.

5.5 Effectiveness of Performance Models

We evaluate the effectiveness of performance models. The four configuration parameters can form a large design space.

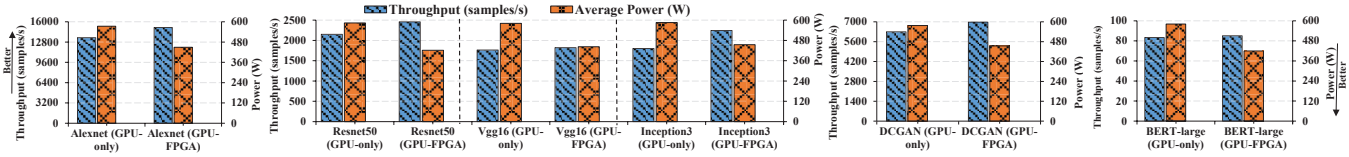

Figure 6. Training throughput and power for GPU-only (two GPUs) vs. GPU-FPGA (two GPUs+six FPGAs).

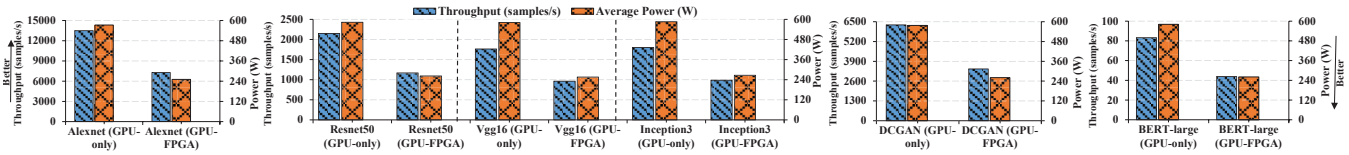

Figure 7. Training throughput and power for GPU-only (two GPUs) vs. GPU-FPGA (one GPU+one FPGA).

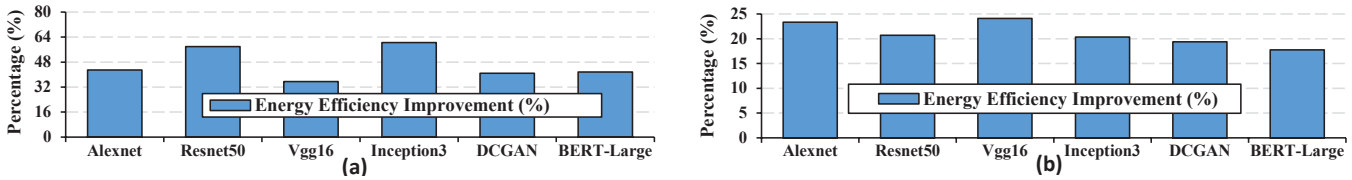

Figure 8. (a) Energy efficiency improvement for GPU-FPGA (two GPUs+six FPGAs) over GPU-only (two GPUs). (b) Energy efficiency improvement for GPU-FPGA (one GPU+one FPGA) over GPU-only (two GPUs).

Table 2. Percentage of offloaded operations to FPGA in all operations (in terms of number of operations and execution time).

DNN models	Offloaded Op. (%)	Exe. Time (%)	Offloaded Op.
Alexnet	22.22	36.05	MatMul Conv2D, Relu, Mul, AddN, MaxPool
Resnet50	27.78	41.84	MatMul, BatchNorm, L2Loss, Conv2D, Relu, Mul
Vgg16	27.78	23.33	MatMul, BiasAdd, Conv2D, Relu, Mul
Inception3	23.81	38.75	MatMul, Transpose, Conv2D, Relu, ReluGrad, Mul
DCGAN	17.54	24.31	MatMul, Transpose, Conv2D, Relu, Mul
BERT-Large	19.32	25.16	MatMul, Add, Norm, Mul, Transpose, Softmax

Table 3. Throughput for six models under 230W power cap for GPU-only (one GPU) vs. GPU-FPGA (one GPU+three FPGAs).

DNN models	GPU-only (samples/s)	GPU-FPGA (samples/s)
Alexnet	5666	6313
Resnet50	904	1077
Vgg16	741	775
Inception3	757	882
DCGAN	2641	2928
BERT-large	35	37

For example, the operation `MatMul` has 20480 combinations with 4 different settings for the number of CUs, 5 for the SIMD width, 4 for the work group size, and 256 for the degree of loop unrolling. To explore the large design space, we first employ a binary search algorithm to eliminate those unpromising combinations. Those combinations are either infeasible due to hardware limitation or inefficiency, compared to the combinations that are already evaluated. Second, we eliminate those combinations whose reported F_{max} is within 10% of the F_{max} of others, as we observe that less than 10% difference in F_{max} between two combinations induces little actual performance difference.

The above methods significantly reduce the number of combinations. Among the remaining combinations, we report the results of the top five best-performing combinations for each operation. Table 4 lists the configurations of the top fives for the three most common operations offloaded to FPGA (i.e., `MatMul`, `Conv2D`, and `MaxPool`. The table also shows the corresponding `LI` values, and the actual execution time when we run them on FPGA for verification. It can be seen that the proposed proxy `LI` has exactly the same relative ranking as the actual execution time, indicating that `LI` can serve as a reliable indicator for fast comparisons.

6 Related Work

GPU for DNN training. As a prevalent high-performance architecture, GPU has been widely used for accelerating computation-intensive workloads [23, 62–64], particularly DNN training [8, 13, 25, 26, 42, 43, 55, 56, 56, 58, 66]. Mirhoseini et al. [41, 42] use a reinforcement learning model (RL)

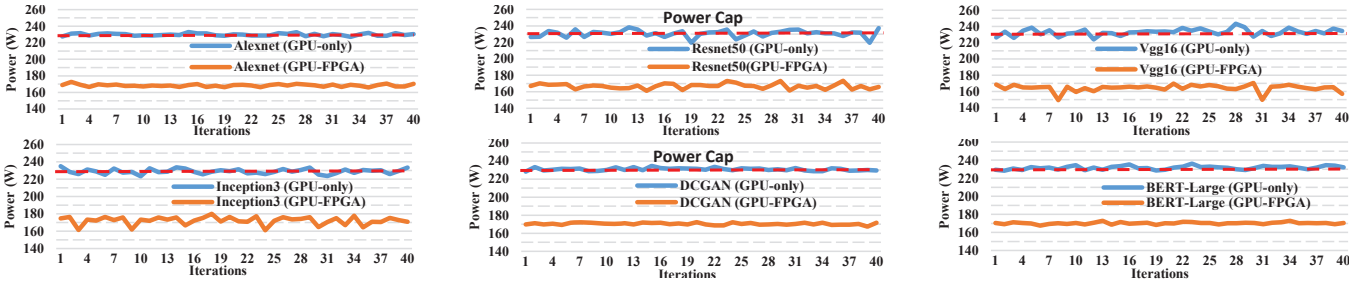

Figure 9. Runtime system power for six models under 230W power cap for GPU-only (one GPU) vs. GPU-FPGA (one GPU+three FPGAs).

Table 4. Latency Indicator (LI) prediction vs. execution time.

Operations	#CU	SIMD width	work_group_size	Degree of loop unrolling	Perf. (exe. time)	LI
MatMul	1	8	256	256	2.595 ms	17.9
	2	4	256	128	3.480 ms	20.95
	1	16	128	128	4.188 ms	49.09
	2	1	128	64	9.594 ms	81.81
	4	2	64	64	16.515 ms	142.57
Conv2D	2	4	64	64	2.532 ms	19.34
	1	8	64	32	5.154 ms	26.47
	2	16	32	32	8.134 ms	55.88
	4	4	32	16	11.173 ms	61.48
	4	8	16	8	18.576 ms	128.94
MaxPool	3	8	256	256	0.284 ms	30.62
	2	4	128	64	0.793 ms	74.7
	2	16	128	32	1.396 ms	88.33
	1	8	64	32	2.275 ms	130.89
	4	1	32	1	2.768 ms	148.42

or a feed forward (FF) model plus a LSTM model to decide which operations in a DNN model should run on which devices in a distributed environment with a mixture of devices such as CPUs and GPUs. However, using RL or FF+LSTM is too time-consuming (hundreds of GPU hours to make the decision or train [41]), which is especially problematic for large DNN models. Hype-training uses offline characterization plus runtime scheduling to make the decision, which avoids the lengthy model training; the offline characterization results, once collected, can be used repeatedly for any DNN model.

FPGA for DNN training. A few efforts have investigated using FPGA for DNN training. Fox et al. [19] use Zynq FPGA for low-precision DNN training using 8-bit integer numbers. Zhao et al. [74] introduce a pipelined structure implementing convolution and pooling layers, and train a specific DNN model LeNet on two FPGAs. Geng et al. [21] train CNN models on a FPGA cluster with 5-83 FPGAs. The above two researches show that memory bandwidth is a limiting factor for DNN training on FPGAs. Besides, the number of multiply-accumulate (MAC) units of FPGA is also limiting factors. To address this problem, Pinjare et al. [47] implement the back propagation algorithm in a gradient descent form to reduce the number of multipliers. Our work is different from those existing efforts, as we explore fine-grained hybrid use of both GPUs and FPGAs for DNN training.

7 Conclusions

Current and future DNN training needs to be highly energy-efficient, particularly with the rapidly growing model complexity and dataset size. In this paper, we propose a framework, Hype-training, that automatically schedules individual training operations to a hybrid of GPU and FPGA accelerators. We characterize operations in terms of power and performance on GPUs and FPGAs, apply a set of techniques to optimize FPGA kernel performance, identify optimal configurations with fast performance models, and develop a runtime system for dynamic scheduling. Evaluation results show significant energy savings in DNN training for Hype-training, without loss in training throughput.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on this work. We thank Dr. Lizhong Chen for providing valuable feedback to the research and paper writing. This work was supported by the National Key Research and Development Program of China (No.2018YFB1003502), National Science Foundation of China under Grants 61772183 and 61972137 and China Scholarship Council (CSC).

References

[1] 2018. NVProf - NVIDIA Developer Documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[2] 2018. Optimize TensorFlow performance using the Profiler. https://www.tensorflow.org/guide/profiler.

[3] 2019. BERT, RoBERTa, DistilBERT, XLNet — which one to use? https://towardsdatascience.com/bert-roberta-distilbert-xlnet-which-one-to-use-3d5ab82ba5f8.

[4] 2019. CUDA Toolkit Documentation v10.1. https://developer.nvidia.com/cuda-toolkit-archive.

[5] 2019. Intel's RunningAverage Power Limit (RAPL) interface. https://01.org/rapl-power-meter.

[6] 2020. NVIDIA Data Center Deep Learning Product Performance. https://developer.nvidia.com/deep-learning-performance-training-inference.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[8] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. 2017. An in-depth performance characterization of CPU-and GPU-based DNN training on modern architectures. In *Proceedings of the Machine Learning on HPC Environments*. 1–8.

[9] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. 2017. An opencl™ deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 55–64.

[10] Ray Bittner, Erik Ruf, and Alessandro Forin. 2014. Direct GPU/FPGA communication via PCI express. *Cluster Computing* 17, 2 (2014), 339–348.

[11] Martin Burtscher, Ivan Zecena, and Ziliang Zong. 2014. Measuring GPU power with the K20 built-in sensor. In *Proceedings of Workshop on General Purpose Processing Using GPUs*. 28–36.

[12] Deming Chen, Jason Cong, Yiping Fan, and Zhiru Zhang. 2007. High-level power estimation and low-power design space exploration for FPGAs. In *2007 Asia and South Pacific Design Automation Conference*. IEEE, 529–534.

[13] Xiaoming Chen, Danny Z Chen, and Xiaobo Sharon Hu. 2018. moDNN: Memory optimal DNN training on GPUs. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 13–18.

[14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[15] Jeff Dean. 2019. Google AI chief Jeff Dean interview: Machine learning trends in 2020. https://venturebeat.com/2019/12/13/google-ai-chief-jeff-dean-interview-machine-learning-trends-in-2020/.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[18] Mariza Ferro, André Yokoyama, Vinicius Klôh, Gabrieli Silva, Rodrigo Gandra, Ricardo Bragança, Andre Bulcao, and Bruno Schulze. 2017. Analysis of GPU power consumption using internal sensors. In *Anais do XVI Workshop em Desempenho de Sistemas Computacionais e de Comunicação*. SBC.

[19] Sean Fox, Julian Faraone, David Boland, Kees Vissers, and Philip H. W. Leong. 2019. Training Deep Neural Networks in Low-Precision with High Accuracy Using FPGAs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*.

[20] Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. 2013. Effects of dynamic voltage and frequency scaling on a k20 gpu. In *2013 42nd International Conference on Parallel Processing*. IEEE, 826–833.

[21] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rui Xu, Rushi Patel, and Martin Herbordt. 2018. FPDeep: Acceleration and load balancing of CNN training on FPGA clusters. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 81–84.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[23] Xin He, Yapeng Yao, Zhiwen Chen, Jianhua Sun, and Hao Chen. 2021. Efficient parallel A* search on multi-GPU system. *Future Generation Computer Systems* (2021).

[24] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 875–890.

[25] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of large-scale multi-tenant {GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference*. 947–960.

[26] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. 2018. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research* (2018).

[27] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, Shinobu Miwa, and Hiroshi Nakamura. 2013. Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping. In *2013 IEEE 31st International Conference on computer design (ICCD)*. IEEE, 349–356.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[29] Teng Li, Vikram K Narayana, and Tarek El-Ghazawi. 2015. A power-aware symbiotic scheduling algorithm for concurrent GPU kernels. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 562–569.

[30] Yang Li, Charles R Lefurgy, Karthick Rajamani, Malcolm S Allen-Ware, Guillermo J Silva, Daniel D Heimsoth, Saugata Ghose, and Onur Mutlu. 2019. A scalable priority-aware approach to managing data center server power. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 701–714.

[31] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. 2018. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* 275 (2018), 1072–1086.

[32] Jiawen Liu, Dong Li, Gokcen Kestor, and Jeffrey Vetter. 2019. Runtime concurrency control and operation scheduling for high performance neural network training. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 188–199.

[33] Jiawen Liu, Hengyu Zhao, Matheus A Ogleari, Dong Li, and Jishen Zhao. 2018. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 655–668.

[34] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015. Deep Learning Face Attributes in the Wild. In *Proceedings of International Conference on Computer Vision (ICCV)*.

[35] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 101–108.

[36] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 401–416.

[37] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. 2012. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *2012 41st International Conference on Parallel Processing*. IEEE, 48–57.

[38] Ruben Mayer. 2018. Simulation of TensorFlow partitioning and scheduling strategies. https://github.com/mayerrn/tensorflowPartitioningAndScheduling.

[39] Ruben Mayer, Christian Mayer, and Larissa Laich. 2017. The tensorflow partitioning and scheduling problem: it's the critical path!. In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*. 1–6.

[40] Xinxin Mei, Ling Sing Yung, Kaiyong Zhao, and Xiaowen Chu. 2013. A measurement study of GPU DVFS on energy conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*. 1–5.

[41] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *International Conference on Learning Representations*.

[42] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *The 34th International Conference on Machine Learning (ICML)*.

[43] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[44] Ripal Nathuji, Karsten Schwan, Ankit Somani, and Yogendra Joshi. 2009. VPM tokens: virtual machine-aware power budgeting in datacenters. *Cluster computing* 12, 2 (2009), 189–203.

[45] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. 2017. Can FPGAs beat GPUs in accelerating next-generation deep neural networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 5–14.

[46] Nvidia. 2018. NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface.

[47] SL Pinjare and Arun Kumar. 2012. Implementation of neural network back propagation training algorithm on FPGA. *International journal of computer applications* 52, 6 (2012).

[48] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 26–35.

[49] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[50] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 2383–2392. https://doi.org/10.18653/v1/D16-1264

[51] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. 2018. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600* (2018).

[52] Jiayi Sheng, Chen Yang, Ahmed Sanaullah, Michael Papamichael, Adrian Caulfield, and Martin C Herbordt. 2017. HPC on FPGA clouds: 3D FFTs and implications for molecular dynamics. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–4.

[53] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[54] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. 2019. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 909–923.

[55] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.

[56] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. 2019. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855* (2019).

[57] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[58] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 41–53.

[59] Qiang Wang and Xiaowen Chu. 2018. GPGPU performance estimation with core and memory frequency scaling. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 417–424.

[60] Shuo Wang, Yun Liang, and Wei Zhang. 2017. Flexcl: An analytical performance model for opencl workloads on flexible fpgas. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[61] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. 2016. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 114–125.

[62] Zhen Xie, Zheng Cao, Zhan Wang, Dawei Zang, En Shao, and Ninghui Sun. 2016. Modeling traffic of big data platform for large scale datacenter networks. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 224–231.

[63] Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. 2021. Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on GPU. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 426–440.

[64] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*. 94–105.

[65] Omry Yadan, Keith Adams, Yaniv Taigman, and Marc'Aurelio Ranzato. 2013. Multi-gpu training of convnets. *arXiv preprint arXiv:1312.5853* (2013).

[66] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. 2019. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650* (2019).

[67] Pengyuan Yu and Patrick Schaumont. 2007. Secure FPGA circuits using controlled placement and routing. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. 45–50.

[68] Reda Zhan, Xin and Sherief. 2013. Techniques for energy-efficient power budgeting in data centers. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[69] Xin Zhan and Sherief Reda. 2014. Power budgeting techniques for data centers. *IEEE Trans. Comput.* 64, 8 (2014), 2267–2278.

[70] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[71] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. *ACM SIGPLAN Notices* 51, 4 (2016), 545–559.

[72] Jialiang Zhang and Jing Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 25–34.

[73] Xufan Zhang, Ziyue Yin, Yang Feng, Qingkai Shi, Jia Liu, and Zhenyu Chen. 2019. NeuralVis: Visualizing and Interpreting Deep Learning Models. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1106–1109.

[74] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. 2016. F-CNN: An FPGA-based framework for training Convolutional Neural Networks. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 107–114.