Two Essays on the Intersection between Deep Learning and Computer Architecture

- Scaling Deep Learning on GPU and Knights Landing Clusters
- ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware

Scaling Deep Learning on GPU and Knights Landing Clusters

Outline

Introduction

- Deep Learning Applications require large computing power
- Current solutions like GPGPUs or FPGA need to fetch data from either CPU memory or remote processors at runtime, we use multi-GPU platform and Intel Knights Landing cluster(self-host chip) as target platform.
- Algorithm side, we improved the Elastic Averaging SGD(EASGD) method by redesigning four efficient distributed algorithms:
- Async EASGD -> Async MEASGD -> Hogwild EASGD -> Sync EASGD

Background

Intel Knights Landing Architecture:

1. Self-hosted Platform: self-hosted by an operating system, no need for CPU
2. Better Memory: 384 GB
3. Configurable NUMA: Non-uniform memory access

Convolutional Neural Networks

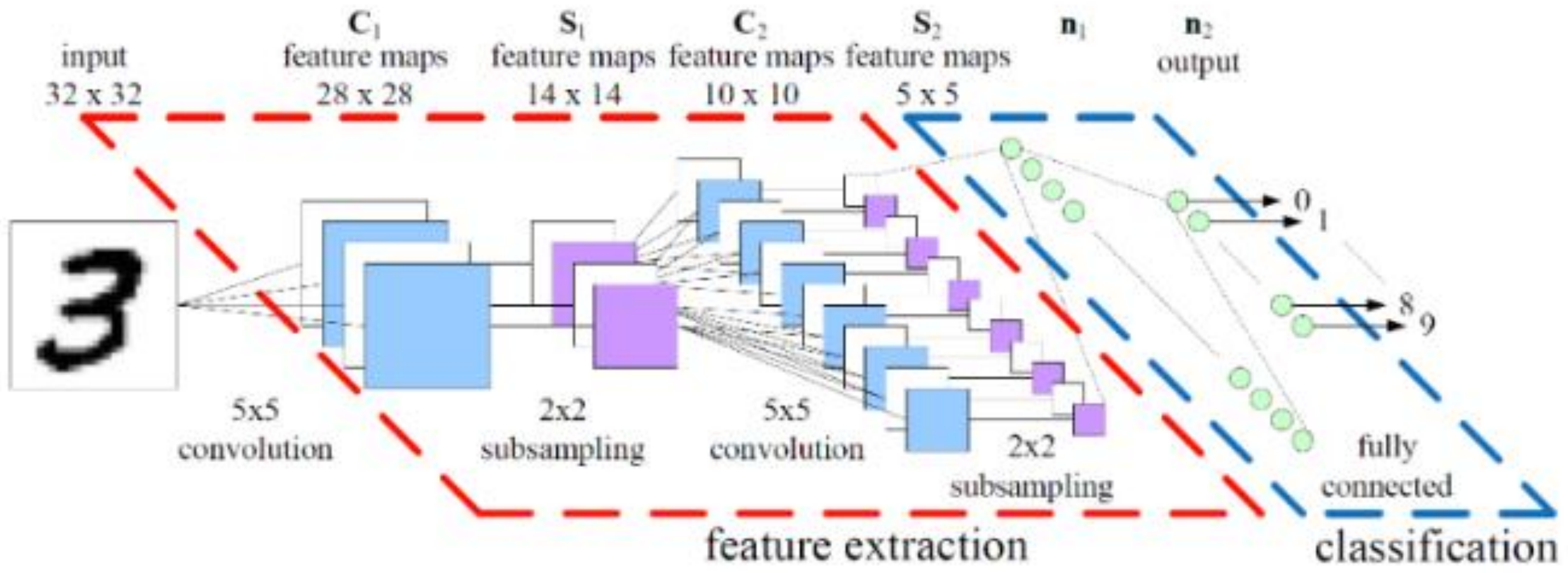


Figure 3: This figure [20] is an illustration of Convolutional Neural Network.

Stochastic Gradient Descent

$$w := w - \eta \nabla Q_i(w)$$

Data Parallelism and Model Parallelism

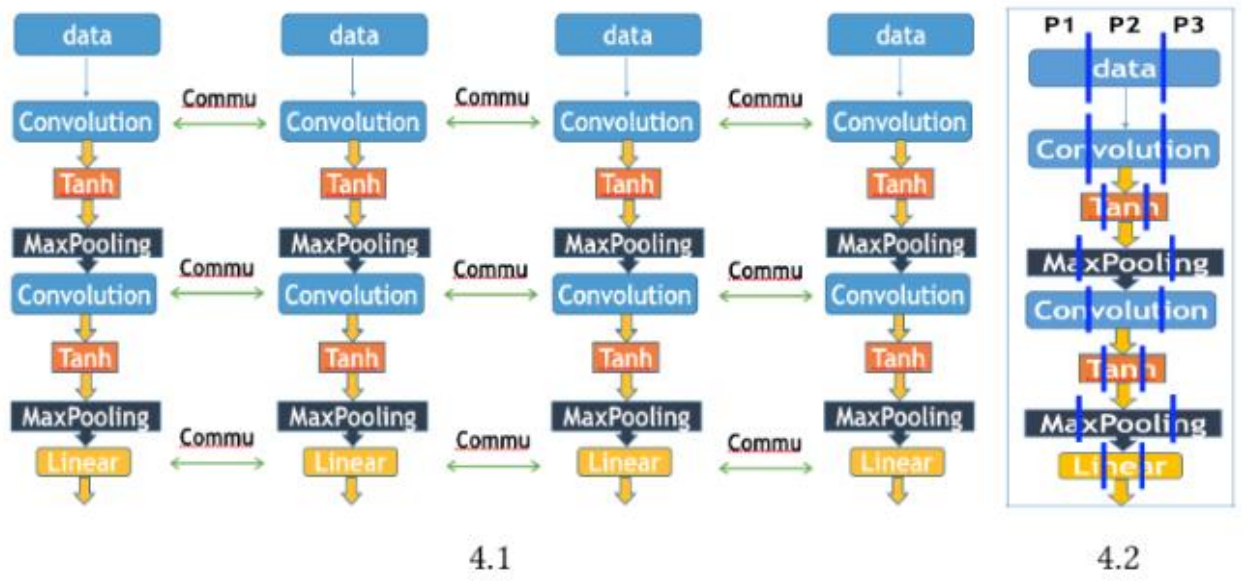


Figure 4: Methods of parallelism. 4.1 is data parallelism on 4 machines. 4.2 is model parallelism on 3 machines. Model parallelism partitions the neural network into P pieces whereas data parallelism replicates the neural network itself in each processor.

Related Work

- Parameter Server
- Hogwild
- EASGD

Parameter Server(Async SGD)

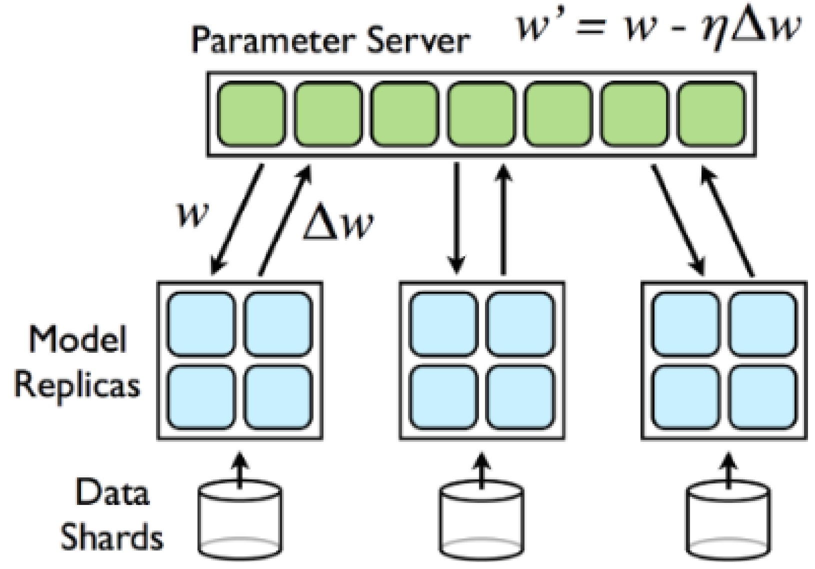


Figure 5: This figure [5] is an illustration of Parameter Server. Data is partitioned to workers. Each worker computes a gradient and sends it to server for updating the weight. The updated model is copied back to wokers

Hogwild (Lock Free)

- Async SGD has a lock to avoid weight update conflict, make sure the master only processes one sub-gradient at a time.
- The hogwild removes the lock and allows the master to process multiple sub-gradients at the same time.

EASGD (Round-Robin)

- Round-robin VS FCFS strategy
- The master interact with one single worker at a time
- Baseline

$$W_{t+1}^{i} = W_{t}^{i} - \eta(\Delta W_{t}^{i} + \rho(W_{t}^{i} - \bar{W}_{t})) \qquad (1)$$

$$\bar{W}_{t+1} = \bar{W}_{t} + \eta \sum_{i=1}^{P} \rho(W_{t}^{i} - \bar{W}_{t}) \qquad (2)$$

Distributed Algorithm Design

Async EASGD

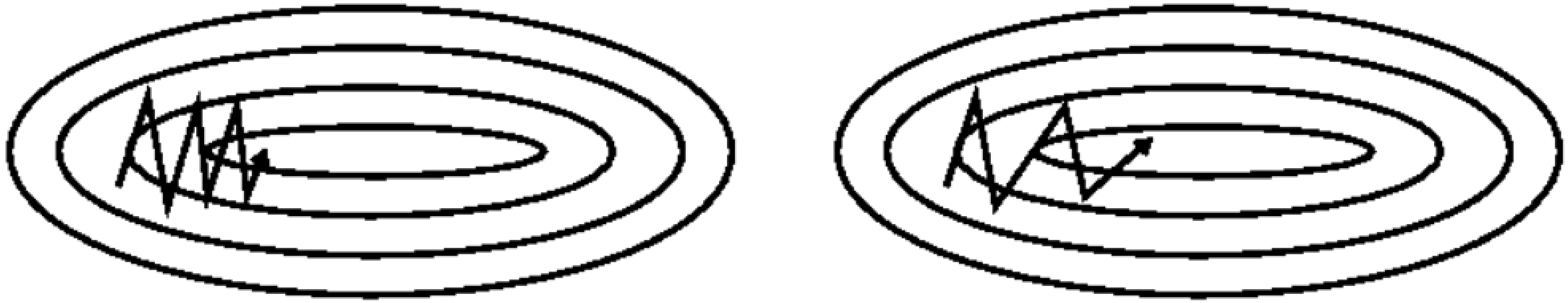
- Redesigning the parallel SGD methods
- The computation and update of different GPUs are ordered, good fault-tolerance but inefficient.
- To use parameter-server update to replace the round-robin update:
- Use FCFS strategy to process multiple workers

During the t-th iteration

- (1) i-th worker first sends its local weight W_t^i to master and master returns $\bar{W}_t$ to i-th worker ($i \in \{1, 2, \dots, P\}$).
- (2) i-th worker computes gradient ΔW_t^i and receives $\bar{W}_t$.
- (3) master does the update based on Equation (2) and worker does the update based on Equation (1).

Async MEASGD

- Add a momentum parameter V, help accelerate gradient in the right direction

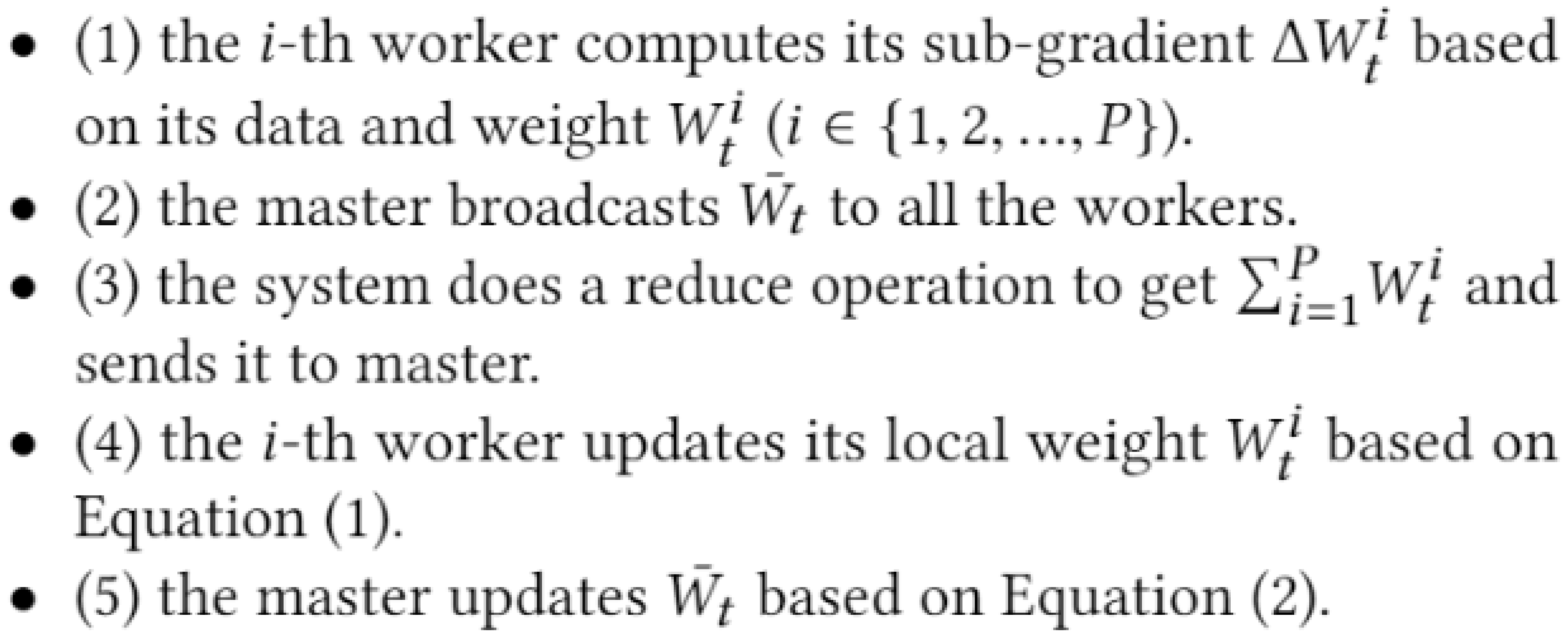


Left — SGD without momentum, right — SGD with momentum. (Source: Genevieve B. Orr)

Hogwild EASGD

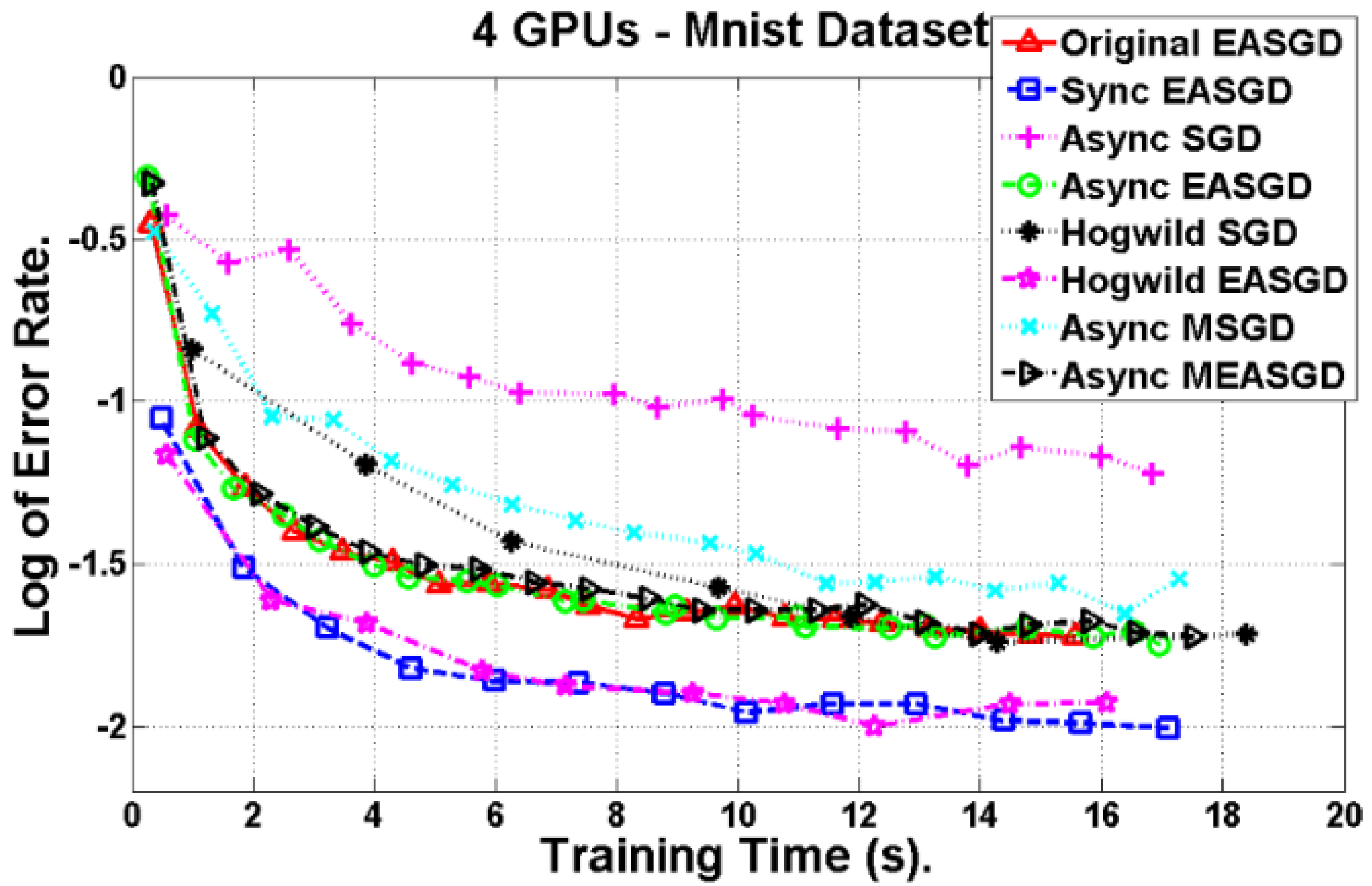
- Remove the lock for faster convergence

Sync EASGD



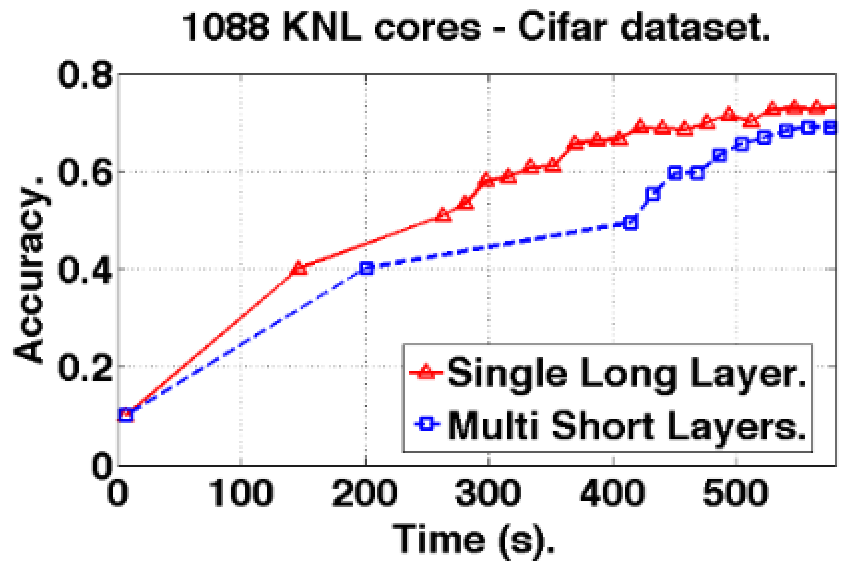
- (1) the i-th worker computes its sub-gradient ΔW_t^i based on its data and weight W_t^i ($i \in \{1, 2, \ldots, P\}$).
- (2) the master broadcasts $\bar{W_t}$ to all the workers.
- (3) the system does a reduce operation to get $\sum_{i=1}^{P} W_t^i$ and sends it to master.
- (4) the i-th worker updates its local weight W_t^i based on Equation (1).
- (5) the master updates $\bar{W_t}$ based on Equation (2).

Overall Comparison



Single Layer Communication

- Allocate the neural networks in a contiguous way and pack all the layers together and conduct one communication each time.



Multi-GPU Optimization

- Sync EASGD1, there are 8 potentially time-consuming parts:

1. Data I/O	ignore
2. Data and weight initialization	ignore
3. GPU-GPU parameter communication	communication
4. CPU-GPU data communication	communication
5. CPU-GPU parameter communication	communication
6. Forward and Backward propagation	computation
7. GPU weight update	computation
8. CPU weight update	computation

CPU-GPU data communication is the key

- Sync EASGD1: P blocking send/receive operations can be efficiently processed by a tree-reduction operation(standard MPI reduction)
- Sync EASGD2: Put all training and test data on the CPU, put all weights on GPU to reduce communication overhead.
- Sync EASGD3: overlap critical path 7-10 and 11-12

```
Algorithm 3: Sync EASGD2 & Sync EASGD3
master: GPU$_1$, workers: GPU$_1$, GPU$_2$, ..., GPU$_P$
  Input: samples and labels: $\{X_i, y_i\}$ $i \in 1, ..., n$
         #iterations: $T$, batch size: $b$, # GPUs: $G$
  Output: model weight $W$
1 Normalize $X$ on CPU by standard deviation: $E(X) = 0$ (mean)
   and $\sigma(X) = 1$ (variance)
2 Initialize $W$ on CPU: random and Xavier weight filling
3 for $j = 1; j <= G; j{+}{+}$ do
4     create local weight $W_1^j$ on GPU$_j$, copy $W$ to $W_1^j$
5 create global weight $\bar{W}_1$ on GPU$_1$, copy $W$ to $\bar{W}_1$
6 for $t = 1; t <= T; t{+}{+}$ do
7     for $j = 1; j <= G; j{+}{+}$ do
8         CPU randomly pick $b$ samples
9         CPU asynchronously copy $b$ samples to $j$-th GPU$_j$
10    Forward and Backward Propagation on all the GPUs
11    GPU$_1$ broadcasts $\bar{W}_t$ to all the GPUs
12    GPU$_1$ gets $\sum_{j=1}^{G} W_t^j$ from all the GPUs
13    All the GPUs update $W_t^j$ by Equation (1)
14    GPU$_1$ updates $\bar{W}_t$ by Equation (2)
```

Knights Landing Optimization

- 68 cores or 72 cores, data locality, divide-and-conquer method:
- Divide: replicating data and

Copying weights

- Conquer: sum up the gradients

From all partitions

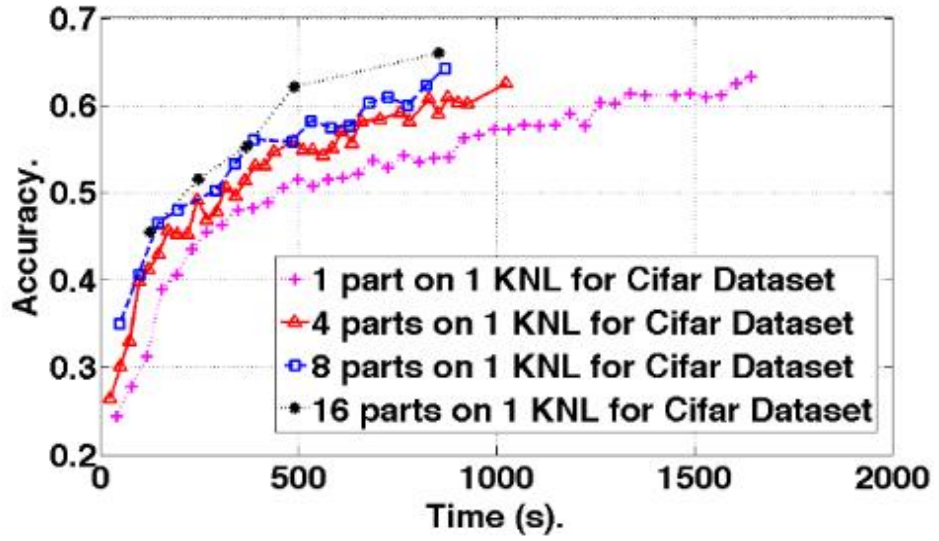


Figure 12: Partitioning a KNL chip into group and making each group process one local weight can improve the performance.

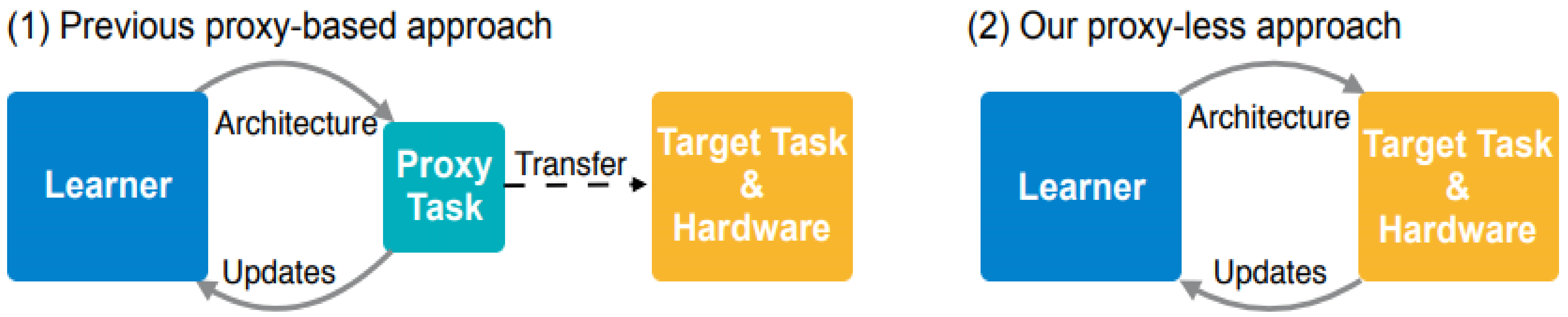
ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware

Neural Architecture Search (NAS)

- Automatically designing effective neural network architectures
- Problems: Large computational demand of conventional NAS algorithms, 48000 hours (Neural architecture search with reinforcement learning. In ICLR, 2017) to generate a CNN
- Use proxy tasks(smaller dataset, learning with a few blocks, training for a few epochs), but not guaranteed to be optimal for target task

ProxylessNAS

- Directly learn the architectures on the target task and hardware



Method

- Construct over-parameterized network:

$$\mathcal{N}(e = m_{\mathcal{O}}^{1}, \cdots, e_n = m_{\mathcal{O}}^{n})$$

e as certain edge, o as primitive operations in neural network like (convolution, pooling, identity, zero, etc), α as architecture parameters.

Path level binarization and pruning

$$g = \text{binarize}(p_1, \cdots, p_N) = \begin{cases} [1, 0, \cdots, 0] & \text{with probability } p_1, \\ \qquad \cdots & \\ [0, 0, \cdots, 1] & \text{with probability } p_N. \end{cases} \tag{2}$$

Based on the binary gates g, the output of the mixed operation is given as:

$$m_{\mathcal{O}}^{\text{Binary}}(x) = \sum_{i=1}^{N} g_i o_i(x) = \begin{cases} o_1(x) & \text{with probability } p_1 \\ \quad \cdots & \\ o_N(x) & \text{with probability } p_N. \end{cases}. \tag{3}$$

Path level binarization and pruning

- Store only one sampled path at a time so save an order of magnitude in memory consumption
- In training, all paths are initially given the same probability for selection
- Training weight parameters and architecture parameters in an alternative manner

- The algorithm then traces the paths — storing only one at a time — to note the accuracy and loss (a numerical penalty assigned for incorrect predictions) of their outputs. It then adjusts the probabilities of the paths to optimize both accuracy and efficiency. In the end, the algorithm prunes away all the low-probability paths and keeps only the path with the highest probability — which is the final CNN architecture.

Hardware-aware: handling Non-Differentiable Hardware Metrics

- To make latency differentiable, model the latency as a continuous function of neural network dimensions
- For each chosen layer of network, the algorithm samples the architecture on a latency-prediction model
- Use latency on each hardware platform as a feedback signal to optimize the architecture

Hardware-aware

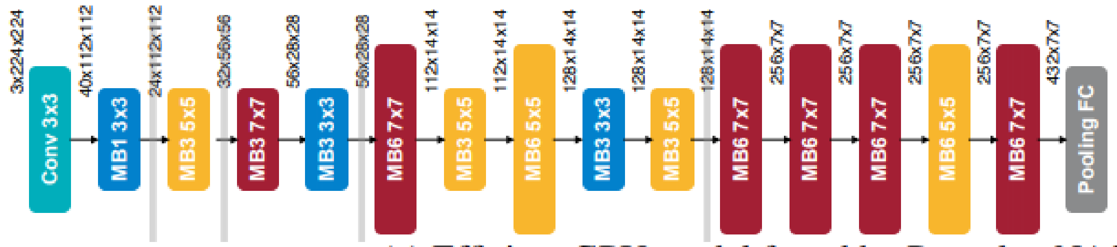
- The final loss function considers the trade-off between accuracy and latency:

$$Loss = Loss_{CE} + \lambda_1 ||w||_2^2 + \lambda_2 \mathbb{E}[\text{latency}]$$

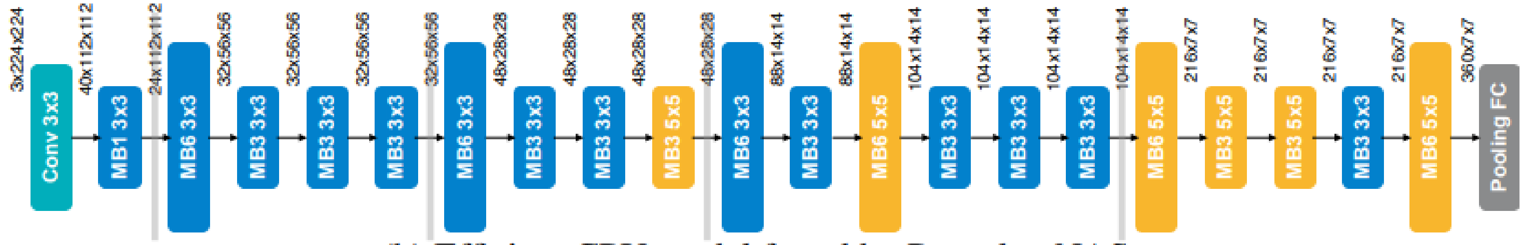
Experiments and Results

- On CIFAR-10, the model achieves 2.08% test error with only 5.7M parameters, better than STOA, while using 6* fewer parameters
- On ImageNet, the model achieves 3.1% better top-1 accuracy and being 1.2* faster with measured GPU Latency

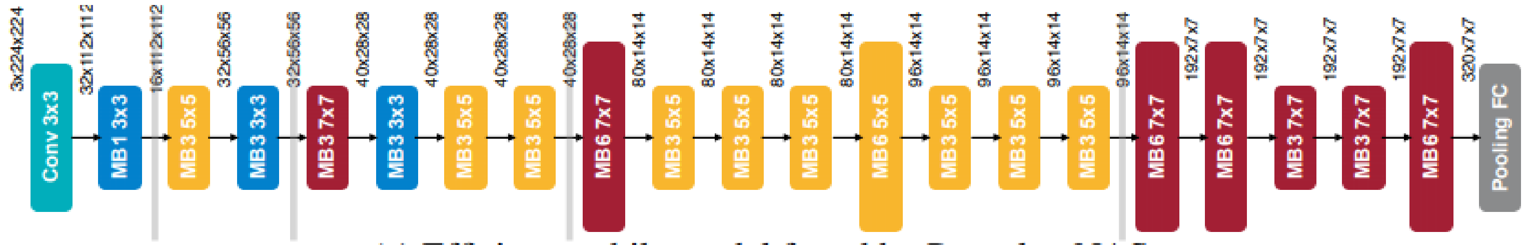
Efficient models found by ProxylessNAS



(a) Efficient GPU model found by ProxylessNAS.



(b) Efficient CPU model found by ProxylessNAS.



(c) Efficient mobile model found by ProxylessNAS.

Thanks!