A Complexity Analysis of the Greedy Allocation Algorithm

Flotilla creation first searches for the reference training rate (time complexity is $\mathcal{O}(N)$), then iteratively finds the best candidate DNN to add in the flotilla (time complexity is $\mathcal{O}(N_k \times N \times M)$, and finally assigns all the remaining GPUs available to the DNNs in the flotilla (time complexity is $\mathcal{O}(N_k \times M)$). So the time complexity of flotilla creation is $\mathcal{O}(N_k \times N \times M)$. Algorithm 1 shows the flotilla creation algorithm.

GPU assignment first prunes the factorial solution space by identifying and assigning GPUs to the DNNs whose training rate meets certain requirements in $\mathcal{O}(N_k)$ time complexity. It then searches for the optimal GPU assignment strategy for the remaining DNNs. The algorithm is shown in Algorithm 2. This algorithm assumes the number of GPUs per node is the same among nodes (*GPUsPerNode*), which holds in the major supercomputers. Let N'_k be the number of remaining DNNs. The solution space is $N'_k!$. Most of the time, N'_k is a small number less than five. However, enumerating all the possible solutions is still in factorial time complexity. We set the maximum number of solutions to explore as 1024, reducing the time complexity to $O(1)$. The time complexity of GPU assignment is thus $\mathcal{O}(N_k)$.

Algorithm 1 createFlotilla

Input: $cands, R, M$
Output: $\mathcal{F}_k, \mathbf{m}_k$

D_{fast}, r_{fast} = fastestDNN($cands, R$) // Find the DNN with the largest training rate with a single GPU
$\mathcal{F}_k, M_k, \mathbf{m}_k = [D_{fast}], 1, [1]$
while $|\mathcal{F}_k| < |cands|$ **do**
  $D_{best}, r_{best}, M_{best}$ = findNext($r_{fast}, R, cands, \mathcal{F}_k, M - M_k$) // Find the next DNN, its training rate and required GPU count
  if $D_{best} == -1$ **then**
    break
  end if
  $\mathcal{F}_k$.append(D_{best})
  $\mathbf{m_k}$.append(M_{best})
  $M_k += M_{best}$
end while
while $M_k < M$ **do**
  $D_{slow} = slowestDNN(\mathcal{F}_k, \mathbf{m}_k, R)$ // in terms of speed on the currently assigned GPUs
  $\mathbf{m_k}[slow] += 1$
  $M_k += 1$
end while
$\mathcal{F}_k, \mathbf{m}_k$

Algorithm 2 getGPUAssignment

Input: $\mathcal{F}_k, \mathbf{m_k}$
Output: A_k

$j, A_k, remaining, assigned = 1, 0_{N_k,M}, \{1, \cdots, N\}, \{\}$
for all $i \in remaining$ **do**
  if $m_i^{(k)} \% GPUsPerNode == 0$ **then**
    $assigned$.add(i)
    j = assignGPUs($A_k, i, j, m_i^{(k)}$)
  end if
end for
$remaining$ -= $assigned$
memo, assigned = {}, {}
for all $i \in remaining$ **do**
  if $-m_i^{(k)} \% GPUsPerNode$ not in $memo$ **then**
    $memo[-m_i^{(k)} \% GPUsPerNode] = i$
  else
    for $ii = i, memo[-m_i^{(k)} \% GPUsPerNode]$ **do**
      $assigned$.add(ii)
      j = assignGPUs($A_k, ii, j, m_{ii}^{(k)}$)
    end for
    del $memo[m_i^{(k)} \% GPUsPerNode]$
  end if
end for
$remaining$ -= $assigned$
if $|remaining| > 0$ **then**
  $\tilde{\mathbf{m}}_{(k)}, bestScore, bestA, j_{copy}, A_{copy} = [], \infty, j, clone(A_k)$
  for all $i \in remaining$ **do**
    $\tilde{\mathbf{m}}_{(k)}$.append($(i, m_i^{(k)})$)
  end for
  for $permutation$ in allPermutations($\tilde{\mathbf{m}}_{(k)}$) **do**
    $j, A_k = j_{copy}, clone(A_{copy})$
    for $i, m_i^{(k)}$ in $permutation$ **do**
      j = assignGPUs($A_k, i, j, m_i^{(k)}$)
    end for
    $score$ = calculateScore(A_k) // Score is calculated based on the loss function in Eq. 7
    if $score < bestScore$ **then**
      $bestScore, bestA = score, A_k$
    end if
  end for
  $A_k = bestA$
end ifA_k

Algorithm 3 assignGPUs

Input: $A_k, i, j, m_i^{(k)}$
Output: j

while $m_i^{(k)} > 0$ **do**
  $a_{i,j}^k = 1; j += 1; m_i^{(k)} -= 1$
end while$j$

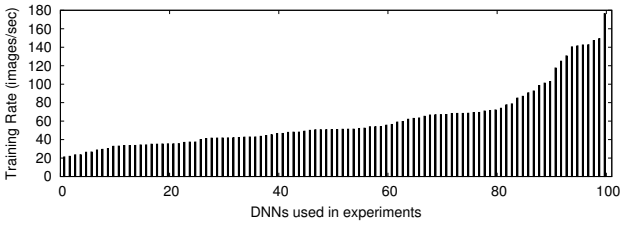

Figure 1: Training rate of each DNN on single GPU.

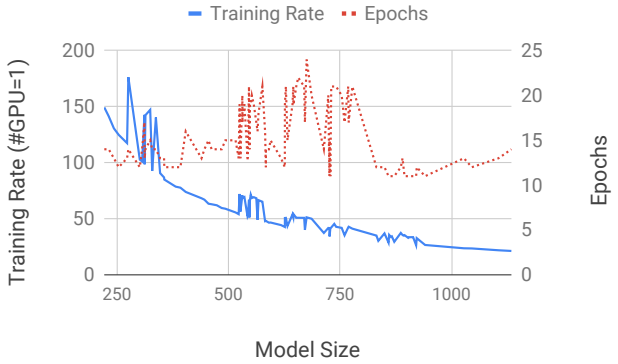

Figure 2: Correlations between model size of a DNN and the training rate and the number of epochs until convergence.

B Experiment Details

B.1 Characteristics of Experimental DNNs

The DNNs used in this experiment are derived from six popular DNNs, DenseNet-121, DenseNet-169, DenseNet-201, ResNet-50, ResNet-101 and ResNet-152. The first three are variations of DenseNet (Huang et al., 2017). The three variations share the same structure, but differ in the number of DNN layers, indicated by their suffixes. The latter three are variations of ResNet (He et al., 2016).

The 100 DNNs used in our experiments have a range of model sizes, from 232 MB to 1.19GB. Different DNNs have different GPU memory requirements and thus requires different batch sizes to maximize GPU utilization. For each, we use the maximum batch size that can fit into GPU's memory. Figure 1 shows the distribution of their training rates on a single GPU which vary from 21 to 176 images/sec.

Figure 2 outlines the relations between the training rates and model sizes of the DNNs, as well as the relations between convergence rates (i.e., the number of epochs needed for the DNNs to converge) and their model sizes. As model size increases, the training rate tends to drop as more computations are involved in the DNN, but there are no clear correlations with the convergence rate. It is the reason that the resource allocation algorithm in FLEET primarily considers training rate explicitly, while relies on the periodical (re)scheduling to indirectly adapt to the variations of DNNs in the converging rates.

B.2 System Settings

All experiments are conducted on SummitDev (Sum, 2019), a development machine for Summit supercomputer at Oak Ridge National Lab. Each node is equipped with two IBM POWER8 CPUs and 256GB DRAM, and four NVIDIA Tesla P100 GPUs. Each POWER8 CPU has 10 cores with 8 HW threads each. The default SMT level is set to one unless noted otherwise. The number of cores allocated per GPU is five in all the experiments. NVLink 1.0 is the connection among all GPUs and between CPUs and GPUs within a node. EDR InfiniBand connects different nodes in a full fat-tree. The file system is an IBM Spectrum Scale file system, which provides 2.5 TB/s for sequential I/O and 2.2 TB/s for random I/O. our experiments show that thanks to the large I/O throughput of the file system, I/O is not the bottleneck of DNN training. The used CUDA version is 9.2.

FLEET is built on Tensorflow 1.12 (as the core training engine) , Horovod v0.15.2 (Sergeev & Del Balso, 2018) (as the basis for distributed DNN training), and mpi4py v3.0.0 (for the pipeline construction). We set `inter_op_parallelism_threads` and `intra_op_parallelism_threads` to # logical cores for parallel TensorFlow operaitons on CPU. The used CUDA version is 9.2.

B.3 Profiling Details

To minimize the overhead of profiling, we only profile the training rates of each DNN in the ensemble with the number of GPUs varying from one to $M_t (M_t < M)$. For $m = 1, \cdots, M_t$, we train a DNN for a maximum of 48 batches and use the training time of the last 20 batches to calculate the exact training rate: $r_i(m), i = 1, \cdots, N$. Based on the profiled training rates, we estimate the training rates of each DNN when $m > M_t$. Specifically, the profiling has three steps:

1. Collect the training rates of each DNN on a single GPU, $R(1) = \{r_i(1)\}, i = 1, \cdots, N$.
2. Estimate the number of GPUs required to make the DNN that has the smallest training rate on a single GPU achieve the largest single-GPU training rate, $M_a = [\frac{\max(R(1))}{\min(R(1))}]$.
3. Collect the training rates of each DNN with the number of GPUs varying from two to $M_t = \max(M_a, M_b)$, where $M_b = 2 \times GPUsPerNode$.

Note that steps 1 and 3 can be done in parallel because the trainings of different DNNs with different number of GPUs are independent. The training rate of the i-th DNN with

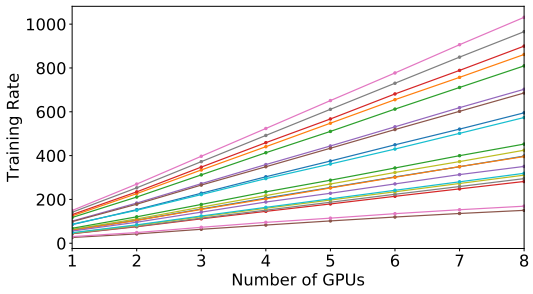

Figure 3: The profiled training rates (images/sec) of 100 DNNs in an ensemble with Imagenet.

the number of GPUs higher than M_t is estimated via the following equation:

$$r_i(m) = m \times \frac{r_i(M_b)}{M_b} \times \left(\frac{r_i(M_b)}{r_i(M_b - 1)} \times \frac{M_b - 1}{M_b} \right)^{m - M_b}. \quad (1)$$

The formula for M_b and Equation 1 are the result of performance modeling on our observations on the DNN performance trend as illustrated in Figure 3. It achieves a good tradeoff between the profiling cost and the performance prediction accuracy.

The profiling process also measures the throughput of a range of preprocessors (# cores=1, 2, 4, 8, 16, 32) in the pipeline. This step is quick since preprocessing does not exhibit large variations. Based on the profiled information, FLEET calculates the minimum number of preprocessors that can meet the demands of an arbitrary M DNNs (with one running on one GPU), and uses it to set the number of preprocessors.

REFERENCES

Summit user guide oak ridge leadership computing facility. https://www.olcf.ornl.gov/for-users/system-user-guides/summitdev-quickstart-guide/, 2019. Accessed 3/3/2019.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Sergeev, A. and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.