SHM-Learn-Dist: An optimized GPU-Powered C++ Distributed Deep Learning Library

Souham Biswas
Illinois Institute of Technology
sbiswas7@hawk.iit.edu

Abstract

To handle training over large datasets, distributed training approaches are very popular. There are mainly two types of distributed training – data-parallel & model-parallel. While data-parallel approaches are easier to implement and include greater support, model-parallelism is usually used to train large neural nets too big to fit on one GPU. In this paper, the workings of the SHM-Learn-Dist library is described which yields training performance boosts in a distributed setting. Support currently exists for data-parallel training.

1. Introduction

Data parallel training is particularly useful in cases where there is huge diversity in data or the mapping to be learnt is complex. The main idea behind data parallel training involves the following two modes –

- Gradient Averaged Data Parallelism
- Weight Averaged Data Parallelism

We shall discuss about both forms of data parallelism and give some pointers about the use cases of each.

Model parallelism is especially useful when we are training very big neural networks of the order of billions of parameters. Here, the model parameters may be distributed across different nodes. It is illustrated in Fig. 1.0.

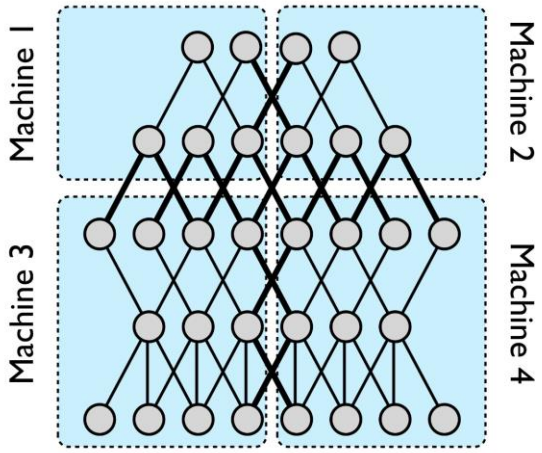

Fig. 1.0 – Model Parallelism illustration[1]

As is clearly evident, there is significant communication involved within every layer here.

The organization of the paper shall be in the following way –

1. Firstly, explore the 2 different types of data parallelisms mentioned previously.

2. Second, we delve in to understanding the fundamental effects of data parallelism on training by observing the parameters of a single neuron which is subject to distributed training.

3. Next, the results of some parameter searches for finding the optimal distributed setting to train on the CIFAR-10 dataset are presented.

4. The fourth section elicits the exact mathematical explanation of the memory-realignment algorithm which contributes greatly to the speedup and presents the GPU kernels and their invocation patterns.

5. Next, the plots involving distributed training on the CIFAR-10 dataset using a custom-augmented Tensorflow are presented to give a general idea of how the CIFAR-10 dataset behaves in a distributed setting.

6. In the sixth section, performance and scaling plots of distributed training experiments with SHM-Learn-Dist are presented and compared with other toolkits.

7. Finally, the future work is mentioned which is planned.

2. Types of Data Parallelism

Data parallel trainings are of 2 types namely -

1. Gradient Averaged
2. Weight Averaged

The main difference between these two methods is that in gradient averaging, every worker gets the same set of weights from the parameter server, computes gradients which are averaged element-wise by the parameter server and applied on the same set of weights which each worker started with. The update equation is presented in Eq. 2.1 –

$$w = w - \eta\left(\frac{1}{N_w}\sum_{i=1}^{N_w} g_i\right)$$

Eq. 2.1 Weight update equation in gradient averaged data parallelism.

Here, N_w is the number of workers and g_i is the gradient from the i^{th} worker; η is the learning rate.

In weight averaging, every worker starts with its own set of weights which vary between workers. The workers compute *and apply the gradients* to get new weights which are sent to the parameter server. The parameter server averages these weights element-wise and sends them back to the workers who continue the process. The update equation for the weight averaged technique is presented in Eq. 2.2 –

$$w = \frac{1}{N_w}\sum_{i=1}^{N_w}(w_i - \eta g_i)$$

Eq 2.2 Weight update equation in weight averaged data parallelism

Upon simplifying, it becomes Eq. 2.3 –

$$\Rightarrow w = \left(\frac{1}{N_w}\sum_{i=1}^{N_w} w_i\right) - \eta\left(\frac{1}{N_w}\sum_{i=1}^{N_w} g_i\right)$$

Eq. 2.2 Simplified weight update equation in weight averaged data parallelism

Here, w_i is the set of network parameters of the i^{th} model; the remaining symbols have the same meanings as explained previously.

The main difference intuitively observable here is that the weight averaged technique promotes diversity in the weights along with the gradients; whereas in the gradient averaged technique, only the gradients are averaged.

3. Single Neuron Distributed Training

An experiment script was written which simulates distributed training of one single neuron with a sigmoid activation function which is trying to learn the following simple function as stated in Eq. 3.1.

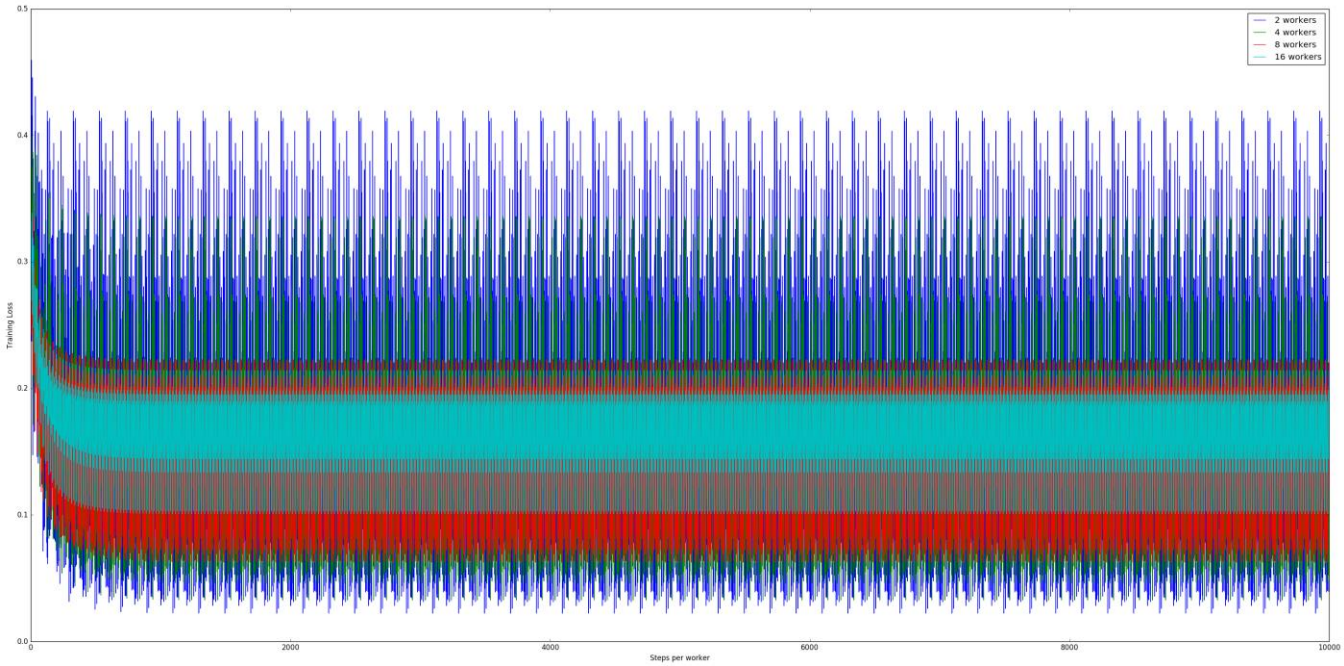

Fig. 2.1 Single Sigmoid Neuron distributed training loss plots over 2, 4, 8 & 16 nodes.

$$f(x) = \begin{cases} 0 \; if \; x < 0.5 \\ 1 \; if \; z \geq 0.5 \end{cases}$$

Eq. 3.1 Function subject to distributed learning by a single sigmoid Neuron

The distributed training was done over 10,000 steps per worker for different numbers of workers for each data parallel training strategy. The learning rate was kept constant at 0.5. Stochastic Gradient Descent was used as the optimization algorithm with no regularization for simplicity coupled with a sigmoid activation function.

The loss plots so obtained from subjecting the single sigmoid neuron to gradient averaged distributed training are presented in Fig. 2.1.

As is clearly evident, the convergence is slower with more workers; however, the loss is less noisy with more workers which corresponds with our observations from distributed CIFAR-10 experiments presented in section 6.

The noisy loss plot can be attributed to the fact that the function being learnt is too simple to be scaled across multiple compute nodes. Hence, visually illustrating that data-parallelism is most useful on complex mappings and/or on large datasets.

It is not unusual to notice that some optimization algorithms perform better with certain distribution schemes. For example, asynchronous Stochastic Gradient Descent scales better with lesser nodes whereas synchronous Stochastic Gradient Descent scales better with more nodes[2].

4. Mathematical explanation of Memory Re-alignment algorithm

An algorithm to minimize the number of matrix multiply operations by combining the weights and biases efficiently in to a single matrix was presented in the previous paper[3]. In this section, the exact mathematical explanation of its working and the GPU kernels along with their invocation patterns shall be presented.

Two types of re-alignment procedures are implemented namely `ShiftRight` and `ShiftLeft` which are explained in sections 4.1 & 4.2 respectively.

4.2 ShiftRight Memory Realignment Matrix transform-

In this scheme, the following transform is done, as shown in Fig. 4.2.1 –

$$\begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & a_0 & a_1 \\ 1 & a_2 & a_3 \end{bmatrix}$$

Fig. 4.2.1 ShiftRight matrix transform illustration

Recall[3] that this when represented in the row-major memory layout as in the case of NVIDIA GPUs, actually looks like what is shown in Fig. 4.2.2 below –

$$[a_0, a_1, a_2, a_3] \rightarrow [1, a_0, a_1, 1, a_2, a_3]$$

Fig. 4.2.2 ShiftRight transformation actual memory layout illustration

The way this is done has been explained at a high level in [3]; here, the actual implementation details are presented. Let us make the following assumptions about the variable names used to explain the approach –

- `rows` – The original number of rows in the input matrix.
- `cols` – The original number of columns in the input matrix.
- `d_mat` – GPU pointer to the memory location holding the original matrix in a row-major format as depicted on the LHS in Fig. 4.2.2.
- `d_helper` – GPU pointer to the memory location to hold the values which may be damaged due to Write-After-Read race conditions[3].

Firstly, we initialize the `d_helper` pointer to a memory location with a size of $\frac{rows(rows-1)}{2}$. Next, we populate the `d_helper` array with values at the affected indices[3]. Ideally, we'd require $\frac{rows(rows-1)}{2}$ threads to do this job as each thread simply performs a copy operation. However, to satisfy the constraint of launching threadblocks with number of threads being a multiple of the GPU warp size[4] (32), the number threadblocks of size $4 \times warp\_size$ to be launched will be as shown in Eq. 4.2.1.

$$num\_threadblocks = \left\lceil \frac{rows(rows-1)}{2 \times (4 \times warp\_size)} \right\rceil$$

Eq. 4.2.1 Number of threadblocks needed to populate `d_helper`.

And we have Eq. 4.2.2 -

$$threadblock\_size = 4 \times warp\_size$$

Eq. 4.2.2 Size of each threadblock to populate `d_helper`

This operation in done by the following GPU kernel–

```
__global__ void
ShiftRight_PopulateHelper_GPUKernel(float
*d_mat, float *d_helper, int damaged_elems,
int rows, int cols) {
  int idx = (blockDim.x * blockIdx.x +
threadIdx.x) % damaged_elems;
  int i = floor(0.5f * (sqrt((float)1 + 8 *
idx) - 1.0f)) + 1;
  int j = idx - i * (i - 1) / 2;
  int read_idx = j + i * cols;
  d_helper[idx] = d_mat[read_idx];
}
```

Mapping each thread to its respective element in the original matrix which its copying to the `d_helper` array, if `idx` is the global threadIdx of a particular thread and assuming the 2D index (in row, column form) from where it shall read in the original matrix `d_mat` as (i, j), we have the following relations as shown in Eq. 4.2.3 and Eq. 4.2.4 –

$$i = \left\lfloor \left(\frac{1}{2} \sqrt{1 + 8 \times idx} \right) + 1 \right\rfloor$$

Eq. 4.2.3 Index row of value in `d_mat` from which thread with global index `idx` reads.

$$j = idx - \frac{i(i-1)}{2}$$

Eq. 4.2.4 Index column of value in `d_mat` from which thread with global index `idx` reads.

Notice that the calculation of j utilizes the calculated value of i in the previous step. Finally, mapping them to unrolled 1D indices respective to `d_mat`, the final read index `read_idx` is given by Eq. 4.2.5 –

$$read\_idx = j + i \times cols$$

Eq. 4.2.5 1D unrolled read index in `d_mat` from where `d_helper` is populated by thread at global index `idx`

Invocation of the GPU kernel is done in the following way –

```
ShiftRight_PopulateHelper_GPUKernel << <
num_threadblocks,
    threadblock_size >> >
    (d_mat, d_helper, reqd_threads,
     rows, cols);
```

Following this, another GPU kernel is launched which first, naively does a ShiftRight transform as mentioned in [3], destroying some values in `d_mat` in the process. Secondly, the damaged values are repopulated with the values in `d_helper` which was populated in the previous step. The GPU Kernel concerning this operation is presented below.

```
__global__ void
ReAlignMemory_ShiftRight_GPUKernel(float
*d_mat, float *d_helper, int total_size, int
cols, int thread_chunk_size) {
  extern __shared__ float read_vals[];
  int shared_mem_idx, read_idx,
read_idx_row, write_idx;
  int row_linear_idx = blockIdx.x * cols;
  int read_idx_base = row_linear_idx
    + (threadIdx.x * thread_chunk_size) %
cols;
  int row_last_linear_idx = row_linear_idx +
cols;
  for (read_idx = read_idx_base; read_idx <
row_last_linear_idx;
       read_idx++) {
    read_idx_row = read_idx / cols;
    shared_mem_idx = read_idx -
row_linear_idx;
    if (read_idx >= read_idx_row * (1 +
cols)) {
      read_vals[shared_mem_idx] =
d_mat[read_idx];
    }
    else {
      read_vals[shared_mem_idx] =
d_helper[read_idx - cols * read_idx_row
        + (read_idx_row - 1)
        * read_idx_row / 2];
    }
  }
  __syncthreads();
  for (read_idx = read_idx_base; read_idx <
row_last_linear_idx;
       read_idx++) {
```

```
    write_idx = (read_idx +
ceil((float)read_idx / cols)) + !(read_idx %
cols);
    d_mat[write_idx] = read_vals[read_idx -
row_linear_idx];
    if ((write_idx - 1) % (cols + 1) == 0) {
      d_mat[write_idx - 1] = 1.0f;
    }
  }
}
```

Here, each threadblock populates it's shared memory space selectively with values from the original `d_mat` matrix and the `d_helper` array based on whether the final write index of the value in the transform could potentially be damaged by the Write-After-Read race condition. Once this is done, the values from the shared memory array are simply written to their new locations in `d_mat` and the 'empty' spaces are padded with 1s.