Computer Science Course Notes — 1st Semester

Dario Loi

Contents

Big Data Computing

First Lecture

Classroom: phdlyrl

Second Lecture

What's Big Data?

It is sometimes used as a buzzword, but it describes an actual phenomenon.

5 V's:

1. Value: Extracting knowledge from data.
2. Volume: The amount of data. (Measuring in terabytes, petabytes, exabytes)
3. Variety: Different formats, structured (SQL tables) and Unstructured (text, images, videos).
4. Velocity: The speed at which data is generated. (usually in real-time, you can't parse it all)
5. Veracity: The quality of the data. (Is it reliable?)

Scaling

When scaling, you have two options:

- Scaling *up*, you buy better hardware for your monolithic machine, however, moore's law is slowing down, and you will get diminishing returns, in general, you will not be able to keep up with increases in Volume/Velocity.
- Scaling *out*, you buy more machines, and distribute the workload, this is the way to go for big data, since this allows for exponential growth.

Scaling *out*, or *horizontal scaling*, also allows you for more flexibility, as once you deal with the orchestration of the workload, you can dynamically increase/reduce the number of allocated machines depending on a rolling cost/benefit analysis.

Network Bottlenecks

A bottleneck during a request is generated when:

- There is a skew in the ratio of communication/computation
- There is a non-overlappable part of communication/computation

Third Lecture

In this lecture, we will take a look at a use case of Big Data, *Distributed Deep Learning*.

Naturally, Deep Learning performs better with more data, this is actually an empirically derived law that has come up in recent OpenAI research: exponentially more data leads to a linear increase in model performance, irrespective of the model architecture (assuming a reasonable architecture).

How does a DNN work

To put it simply, we want to build a function estimator over a dataset D:

$$\hat{f} : D \rightarrow Y$$

Such that:

$$\hat{f}(x) \approx y$$

Where x is the input data, and y is the output data.

To do this, we use iterative optimization, where we start with a random function, and we iteratively update it to minimize the error between the output of the function and the actual output. We use the data from the dataset to *fit* our model, under an *inductive* bias, which is an assumption that the *test* data is similarly distributed (so that our fit will generalize).

Forward and Backward passes

Our iterative fitting process can be split in two passes:

1. Forward pass, where we compute the output of the model given the input.
2. Backward pass, where we compute the gradient of the error with respect to the model parameters.

We use the obtained gradient to refine the model parameters, and we repeat the process until we reach a stopping criterion (which indicates that the model has converged).

In general, most of these operations are matrix operations, which can be parallelized on the GPU in an efficient manner.

Parallelism Models

There are three orthogonal parallelism directions in which we can distribute the workload:

1. Data Parallelism
2. Pipeline Parallelism
3. Model Parallelism

The last two together are also known as *Model Parallelism.*

Data Parallelism

The simplest one is *Data Parallelism.* We split the data in n subsets, and we train n models in parallel, each on a different subset. We then perform *gradient aggregation* across these sets, and we update the model parameters.

In this way all the models are updated in parallel, and the only bottleneck is the gradient aggregation.

Gradient aggregation is *quite* costly, it contributes to a significant portion (20% to 50%) of total training time, with the portion increasing as the available bandwidth decreases.

Naturally, this means that this operation has been the subject of heavy optimization, algorithmically and infrastructurally.

Parameter Servers

An initial solution was to use a *Parameter Server*, which is a server that holds the model parameters, and is responsible for the aggregation of the gradients. This does *not* scale well, as it introduces a single point of failure, and a bottleneck in the network.

Splitting the Aggregation

We can split the aggregation into K servers, which will each aggregate a subset of the gradients, and then we will aggregate the results of these servers. This is a more scalable solution, but it still has a bottleneck in the aggregation.

Naturally, as long as K is small, this is a good solution, but as K grows, the aggregation time will grow linearly, and the bottleneck will be reached.

Cost Model

A cost model on a K-PS (Parameter Server) system is:

$$\max\left(\frac{n}{\beta}, \frac{pn}{k\beta}\right)$$

Where n is the number of bytes in the gradient vector, p is the number of GPUs/workers, k is the number of servers, and β is the bandwidth of the network.

Data Parallelism Recap

In data parallelism, each worker must keep a copy of the model, therefore you can also encounter out-of-memory errors, as the model grows in size.

Data parallelism is simple and easy to implement, but has limited potential to be scalable.

Replication of the model is also a waste of resources, as we are just keeping copies around, and we are not using them. It should only be used as a last resort if we have more resources than we can use after applying the other parallelism models, and we still want idle time to be minimized.

Model Parallelism

In order to train models that do *not* fit into a single GPU, we can use *Model Parallelism*. in essence, model parallelism is the process of splitting the model into k parts, and training each part on a different GPU.

Now, the bottleneck is the GPU-to-GPU communication, since we have a sequential dependency between the parts of the model (each part depends on the output of the previous part). This happens in both the forward and backward passes.

Pipelining

Naturally, you can reduce the *bubble overhead* by pipelining the computation, so that the forward pass of the next part can start before the backward pass of the previous part has finished. The same concept can be applied on subsections of the forward/backward passes that use layers that are distributed across a network.

Operator Parallelism

In the case of *Operator Parallelism*, we split the model into operators, and we distribute the operators across the network. This is a more fine-grained approach, and it is usually used in conjunction with the other two parallelism models.

Here, the bottleneck is the communication between the operators, and the synchronization of the operators, however, we reduce the idle time of GPUs drastically.

Hierarchical Parallelism

Naturally, *Operator Parallelism* is applied on GPUs that are on the same node, so that the available bandwidth is maximized. Whereas *Model Parallelism* is applied on GPUs that are on different nodes, where communication is more costly.

This allows us to maximize the utilization of the available resources, and minimize the communication overhead.

Hybrid Parallelism

Hybrid parallelism (also called 3D-Parallelism), combines all three techniques, and offers maximum scalability and potential for resource utilization.

In general, one must be working with a sufficient quantity of data to *justify* the use of these techniques, as the overhead of the parallelism models can be significant, and can outweigh the benefits of the parallelism.

Challenges

Compute-Communication Overlap

When employing hybrid-parallelism, we perform computations and communications in a very fine-grained manner. The reason is that we want to perform gradient aggregation after each layer, so that we minimize the ***bubbling*** in our pipeline (we don't want to keep anyone waiting).

We want to maximize the overlap in windows of computation and communication, so that we can keep the GPUs and network as busy as possible.

Gradient Compression

Since DL is a stochastic process, one can afford to have some noise in the gradients, and still converge to a good solution. This allows us to compress the gradients, and reduce the communication overhead.

We have various techniques:

1. Quantization: Reducing the precision of the gradients.
2. Rank Decomposition: Decomposing the gradients into a low-rank matrix.
3. Sparsification: Sample the top k gradients, and send only those.

In sparsification, we can use various techniques to select the top k gradients, such as:

1. Top-k: Select the top k gradients by magnitude.
2. Thresholding: Select the gradients that are above a certain threshold λ.

There are theoretical proofs that show that asymptotically, model convergence is not affected by gradient sparsification, as long as the sparsification is done in a *smart* way.

Compression Tradeoff By compressing gradients, we lower the cost of a single iteration, but we increase the number of iterations needed to converge. This is a tradeoff that must be taken into account when designing a distributed deep learning system.

We have a real speed up if:

$$T_{comp} \cdot I_{comp} < T_{orig} \cdot I_{orig}$$

Aggregation Consistency

We observe that gradient accumulation acts as a *barrier*, which means that if a single GPU is, for some reason, considerably slower than the others, it will slow down the whole process. The probability of this happening is significant in large-scale systems, and it is a problem that must be addressed.

Possible solutions include: * Gradient Accumulation: Accumulate the gradients over multiple iterations, and then send them. * Ignoring Slow Workers: Ignore the gradients from slow workers, and only aggregate the gradients that you have currently available after a certain time.

Reduce Operations

We can use smart reduction techniques common in IPC (Inter-Process Communication) to reduce the number of operations needed to aggregate the gradients.

We keep a table of operations and total volume of communication:

Operation	Volume	Steps
Ideal	n	1
Parameter Server	$max(n, \frac{pn}{k})$	1
Näive AllReduce	$(p-1) \cdot n$	1
Ring AllReduce	$2 \cdot N$	$2(p-1)$
Bandwidth-optimal recursive doubling	$2 \cdot N$	$2\log_2(p)$

We explain each:

- Ideal: The ideal case, where we have a perfect system that can aggregate the gradients in a single operation.
- Parameter Server: The case where we have a parameter server, and we have to send the gradients to the server, and then the server sends the aggregated gradients back.
- Näive AllReduce: The case where we have a simple all-reduce operation, where each worker sends the gradients to all the other workers, and then each worker aggregates the gradients.
- AllReduce (ReduceScatter-AllGather): The case where we have a more sophisticated all-reduce operation, we scatter p subsets of the gradients to p workers, each worker aggregates the subset, and then we gather the results across all workers.

Ring AllReduce

We go in depth into the AllReduce operation. First, we build a logical ring topology of N workers, such that each one is connected to the next one, and the last one is connected to the first one.

We take our vector g, we divide it into N parts, and we send each part to the next worker. Each worker then aggregates the received part with its own part, and then sends the result to the next worker.

This results in a full aggregation of the gradients in $(p-1)$ steps, another full rotation is then performed to gather the results, and we have our aggregated gradients.

The communication volume is:

$$2 \cdot \frac{pn}{k} \approx 2n$$

Which is a 2-approximation of the ideal case.

We use a Ring AllReduce instead of a Tree AllReduce, since the Ring AllReduce is capable of only sending parts of the gradients, and not the full gradients, which can be problematic in the case of large models.

Bandwidth-optimal recursive doubling

A variation of the Ring AllReduce is the Bandwidth-optimal recursive doubling, this allows us to perform the aggregation in a logarithmic number of steps, and it is optimal in terms of bandwidth.

Latency-optimal recursive doubling

Using the same binary-tree like topology as the Bandwidth-optimal recursive doubling, we can build a latency-optimal recursive doubling, which is optimal in terms of latency, by sending the entire gradient vector in a single step.

Fourth Lecture

This lecture went into a general overview of the CUDA GPGPU programming model.

An NVidia GPU is essentially a collection of *streaming multi-processors* (SMs), each of which is a collection of individual *threads*, these threads are grouped into *warps*, which are the smallest unit of execution on the GPU.

Warps are made up of 32 threads, and they are executed in *SIMD* (Single Instruction, Multiple Data) fashion, which means that all threads in a warp execute the same instruction at the same time.

Fifth Lecture

Today we will finish up the contents of the fourth lecture.

We started discussing on how GPUs are better for big data workloads. The main reason is that their specialization allows for better efficiency in throughput-oriented workloads.

The concept of *warps* was introduced, as well as the notion that control flow is incredibly disruptive to kernel throughput, as it can cause *divergence* in the warps.

GPU Interconnection

As mentioned before in the Deep Learning case study lecture, these GPUs are connected in a messy, non-uniform way, which can cause bottlenecks in the communication between the GPUs.

Tensor Processing Units

GPU hardware is still *too* general for workloads which only need linear algebra operations, such as fully connected layers in a neural network. They suffer from the Von Neumann Bottleneck, meaning that most of the cases, the processing units are waiting for data to be fetched from memory.

Accessing memory is impactful not only in terms of latency, but also in terms of power consumption, as a memory access can cost as much as 650x the cost of a FLop.

This is why Google developed the TPU, a specialized hardware accelerator, largely based on the concept of a systolic array, which is a type of parallel computing architecture.

A TPU is a collection of Tensor Cores, each posses a set of Matrix Multiplication Units(MXUs), Scalar Units, and high-bandwidth local memory.

MXUs are *not* designed according to the Von Neumann architecture, they are often termed as a spatial/dataflow architecture.

Dataflow Architecture

A dataflow architecture is composed by arranging simple circuits that perform elementary arithmetic in a 2D-grid, the grid possesses no memory, and the data *flows* through the grid, hence the name.

Matrix Multiplication

We give a recap of matrix multiplication, basically:

$$\underbrace{\begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix}}_{M} \times \underbrace{\begin{bmatrix} G \\ H \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} AG+BH \\ CG+DH \\ EG+FH \end{bmatrix}}_{V}$$

Where each letter can be seen as an individual *tile* of the matrix.

We observe how, for example, the G tile is multiplied by the first column of M to result in the first column of U, same for the H tile. in fact, each row of the V vector is the result of the multiplication of the respective M row by the column of U.

Therefore, we can *flow* the data through the systolic array, and perform a final aggregation of the results to obtain the final result.

This can be done *without* the need for registers or memory, as the computation is done on the fly at each clock cycle.

TPU Interconnections

Interconnecting TPUs is usually done through a 3D torus network, which is a simple and efficient way to connect the TPUs.

Performance Claims

Google claims that TPUs are in general faster than the comparable NVidia GPU solutions, however, other studies have shown that this is not always the case, and that the performance of the TPU is highly dependent on the workload.

In general it is still ambiguous whether TPUs are a better solution than GPUs, and it is still an open research question. The point of the lecture is more to convey the concept of the dataflow architecture, rather than its performance (which may change in the future).

Neuromorphic Computing

Another recent attempt at supplanting the Von Neumann architecture is the concept of Neuromorphic Computing, which is a type of computing that is inspired by the human brain.

Spike trains are introduced as an input, and the synthetic neural network prodcues a spike train as an output, this is done by using a network of *spiking neurons*. In theory, since this replicates the structure of the brain, it should also be capable of possessing memory capabilities.

Network Topology Design

We now move on to talk about ways to design interconnection between nodes of a big data computing system.

GPUs are connected by *switches*, which act as crossroads for the data. These can connect at most r numbers of nodes, this is called the switch's *radix*.

Topologies are *regular* if they are a regular graph (rings), *irregular* otherwise.

Hop count is the number of switches that a message must pass through to reach its destination.

The maximum hop count attainable in a network is the *diameter* of the network.

A network is *blocking* if two nodes cannot be connected by two disjoint paths. If there is this possibility, the network is *non-blocking*.

A network is *direct* if for all the switches, there is at least a connected *node*, if they are *indirect*, there may be some switches that are only connected to other switches.

Bisection Cut

In order to obtain a measure of a network's efficiency, we can perform a *bisection cut* analysis. This is done by cutting the network across the smalles possible cut that divides the nodes into *nearly* equal parts.

The *bisection bandwidth* is the bandwidth of the smallest cut that divides the network into two equal parts.

This is calulated as:

$$\text{Bisection Bandwidth} := \text{Number of Cuts} \cdot \text{Cut Links Bandwidth}$$

It is simply the multiplication between the number of cut links, and the bandwidth across the cut.

We produce a table for these metrics on different topologies:

Topology	Diameter	Bisection Cut
Chain	N	1
Ring	$N/2$	2
Mesh	$2(\sqrt{N}-1)$	$\sqrt{N}$
Torus	$\sqrt{N}$	$2\sqrt{N}$
Trees	1	$2 \cdot \log_{r-1}(N)$

Trees

Trees are a classical computer science structure that usually gives some form of logarithmic improvement over a metric when implemented as a data structure. Here we can lower the bisection cut cardinality to a logarithmic funciton of the number of nodes, which is a significant improvement over the other topologies.

However, we are clearly bottlenecking the system when we have to pass through the root of the tree, as the root is a single point of failure. The same reasoning applies inductively where lower nodes are responsible for the inter-communication of all the children.

Fat Trees

We can solve this through a *fat tree*, a data structure where at each higher level we have a double number of links than the previous level, this allows the higher nodes to account for the increased responsibility of the lower nodes.

Fat trees metrics:

- r bottom level switch radix
- n number of servers/nodes
- Diameter: $2log_{r/2}(n)$
- Bisection cut: $n/2$

A fat tree is a *non-blocking* network by construction, since we explicitly add a number of links that is double the number of the previous level.

In practice, we want to use switches that have the same radix level at each layer. This means that we need to aggregate multiple switches in a mesh network to form a single *virtual* switch node.

This is usually called a "folded CLOS network".

Blocking Fat Trees

We can save some money by using a *blocking* fat tree, where we use a radix r at the bottom level, and a radix $r/2$ at the upper level. This is a blocking network, but it is still a good solution for most cases.

The ratio between the lower and upper radix is known as the *blocking ratio*, for our example it's a 2:1 ratio.

Dragonfly

Another popular state-of-the-art topology is the *Dragonfly* network, which is a network that is composed of a number of *groups*, each group is a *clique* (a fully connected graph), and each group is connected to a number of other groups.

This allows for extremely low hop counts, and high bisection bandwidth.

For a fixed radix r, dragonfly networks can connect much more nodes than a fat tree. Diameters are also kept smaller. Naturally, this means that with similar requirements, the network will be cheaper to build.

Disadvantages include the lack of a guarantee of full bandwidth (blocking), expansion is more difficult, as is load balancing. A dragonfly topology also produces more loops, and therefore deadlocks.

Dragonfly+

Dragonfly+ is a slight variation where you have a fully connected *core* that connects roots of various trees that live at the edge of the network. The trees on the edge can

be fat trees, or blocking fat trees with various out-degrees.

Hamming Meshes

Hamming meshes are a type of architecture specific to deep learning workloads (where we employ three-dimensional parallelism). Here we wish to have:

1. A cheap architecture, such as a toroidal network
2. To still preserve some measure of speed for communications across the network

We solve this by essentially first producing a toroidal mesh, and then by interconnecting the edges (unwrapping the cylinder into a rectangle over an arbitrary cut) with fat trees.

In this way we create some sort of *information highway* that allows us to perform eventual edge-to-edge communications in a logarithmic number of hops.