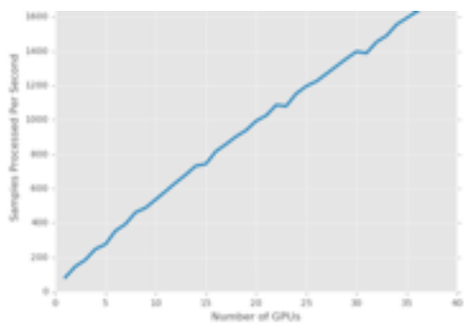
Effectively Scaling Deep Learning Frameworks

(To 40 GPUs and Beyond)



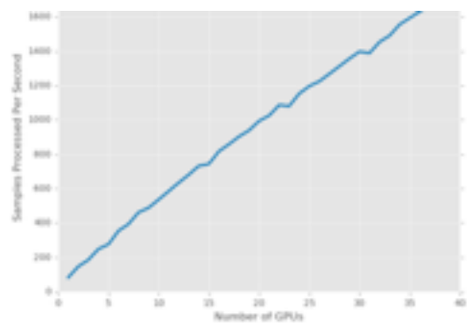
Welcome everyone!

I'm excited to be here today and get the opportunity to present some of the work that we've been doing at SVAIL, the Baidu Silicon Valley AI Lab.

This talk describes a change in the way we've been training most of our models over the past year or so, and some work we had to get there — namely, how we have managed to train our models on dozens of GPUs using common deep learning frameworks (TensorFlow).

Hitting the Limits of Data Parallelism

(Alternate Title)



I briefly considered an alternate title for this talk: Hitting the Limits of Data Parallelism.

The technique I'm presenting today lets you do that — let's you scale to as many GPUs as you'd like to, while retaining the same performance.

So why limit ourselves to 40? Well, it turns out that as your batch size gets large, you converge to a worse local minimum — so depending on your model, you'll have a different cap of your batch size, and in practice for our models, we can't go more than 40 GPUs worth of data while keeping the same final model performance.

Agenda

1. **Progression of a deep learning application**
2. Are deep learning frameworks “good enough”?
3. Writing fast deep learning frameworks
4. Scaling with TensorFlow
   (or any graph-based framework)
5. Questions

Before jumping into our technique ourself, I’d like to offer some wisdom. Whether or not you think this is wise is really something you’ll have to figure out for yourself, but it’s something that we talk about often at SVAIL, and so I figured I’d share it.

Progression of a Deep Learning Application

Step 1: Idea / Proof-of-Concept!

- AlexNet (Krizhevsky, 2012)
- CTC (Graves, 2006)
- GANs (Goodfellow, 2014)
- WaveNet (van den Oord, 2016)

In our mental model of a deep learning application lifecycle, everything starts with an IDEA.

Before going further, we need to prove the IDEA works — whether you're looking at object classification with AlexNet, the CTC loss function, recent work on GANs for image generation, or sample-by-sample speech synthesis with WaveNet, you can often start with fairly small datasets that can easily fit on one or two GPUs.

Progression of a Deep Learning Application

Step 2: Refinement

- VGG (Simonyan, 2014)
- CTC for Speech Recognition (Graves, 2013)
- W-GAN (Arjovsky, 2017)
- Deep Voice (Arik, 2017)

After you’ve nailed the idea, there’s a period of refinement that can take a few months or years. Object classification accuracy jumped by something like 10 to 20 percent through better architectures, and we’ve seen similar things happen for speech recognition, GANs, speech synthesis, and other applications.

Progression of a Deep Learning Application

Step 3: Scaling

- Deep Image (Wu et al, 2015)
- Deep Speech 2 (Amodei et al, 2016)
- *GANs – upcoming?*
- *Text-to-speech – upcoming?*

Finally, after the idea seems to work and work well, we can talk seriously about large-scale training.

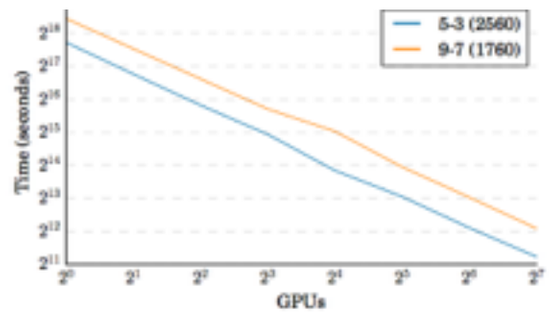
We've seen this for image classification and speech recognition — training on enormous datasets of tens of thousands of hours of audio, as we did with Deep Speech 2 at SVAIL.

We have yet to see this for speech synthesis or GANs.

Why Bother Scaling?

Fraction of Data	Hours	Regular Dev	Noisy Dev
1%	120	29.23	50.97
10%	1200	13.80	22.99
20%	2400	11.65	20.41
50%	6000	9.51	15.90
100%	12000	8.46	13.59

Table 10: Comparison of English WER for Regular and Noisy development sets on increasing training dataset size. The architecture is a 9-layer model with 2 layers of 2D-invariant convolution and 7 recurrent layers with 68M parameters.



But scaling *does* matter. Although it takes a lot of effort and engineering, the results speak for themselves. With Deep Speech 2, we went from an error rate of 30% to 8% by scaling by a factor of 100.

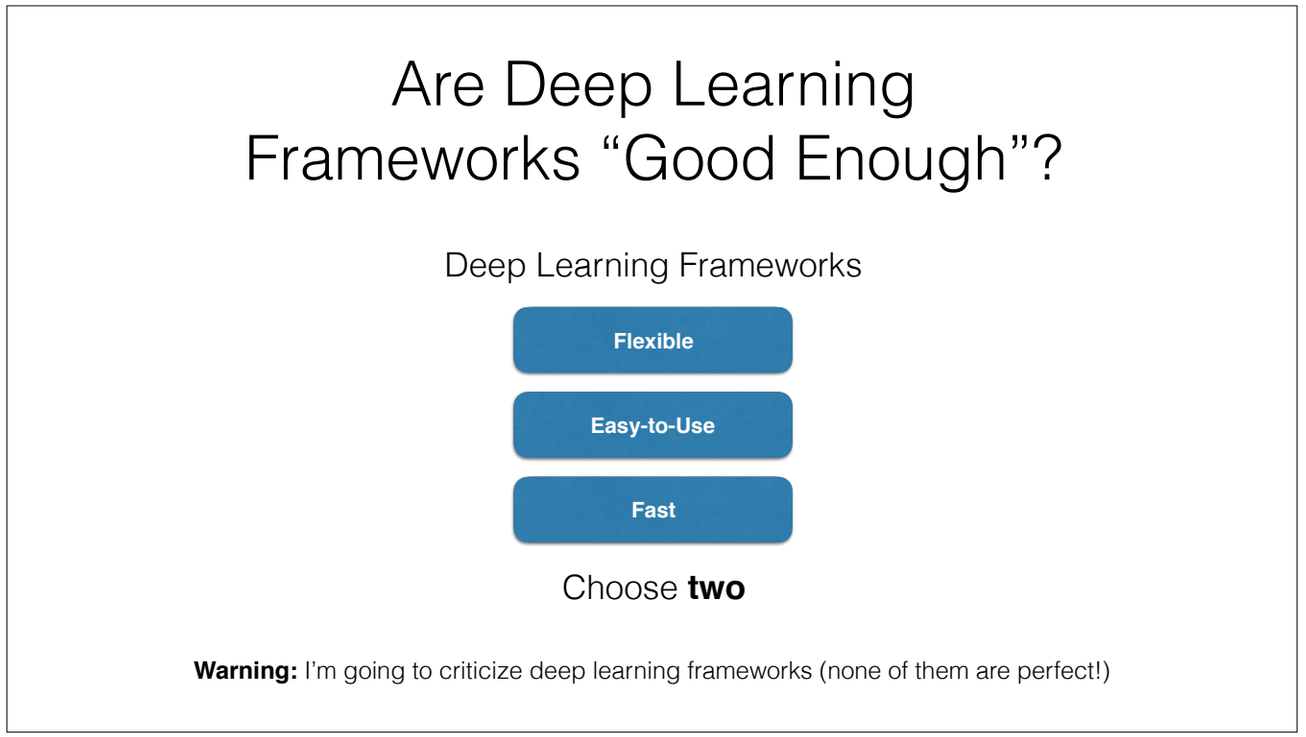
Don’t scale until you need to.

The moral of the story is — don’t scale until you need to. Know *why* you’re working on scaling, and what you hope to get out of it. It will take a lot of time to do well, so don’t jump into it just because it’s the thing to do.

Agenda

1. **Progression of a deep learning application**
2. **Are deep learning frameworks “good enough”?**
3. Writing fast deep learning frameworks
4. Scaling with TensorFlow
   (or any graph-based framework)
5. Questions

Next up, you’ll recall that the talk was titled “scaling deep learning FRAMEWORKS”. So, why the emphasis on frameworks? Does it matter?



I’m going to introduce a stupid analogy here.

Deep learning frameworks are either flexible, easy to use, or fast. But so far, none that I’ve seen can really hit all three criteria. And that’s where this talk comes in!

Beware — in the next few slides I’m going to criticize pretty much every deep learning framework I’ve used. I hope no one takes offense here — no framework is perfect, and we all have our favorites that we use more or less often.

Flexible and **Fast**

Example: Internal SVAIL Framework

- Flexible: Write any operation with NumPy-like framework
- *Very* fast:
  - Highly optimized for our cluster and for RNNs
  - Lots of performance debugging tools
- **Not** easy-to-use: Must writing fwd/bwd prop in C++

We can definitely do flexible and fast.

At SVAIL, much of our work was done with a custom-build internal framework for training RNNs. It was written in C++, and was *incredibly* fast, and had great debugging tools for performance, and we could hit something like 50% of the peak possible throughput of our cluster on average over training — which is very high, most of the TensorFlow models I've worked with have a hard time hitting 20%.

And it was flexible — you could write any operation you wanted.

But it wasn't easy to use — you had to write both the forward prop and the backward prop in C++, and that was error prone and tricky. And in C++, too.

Flexible and **Easy-toUse**

Example: TensorFlow, Theano

- Flexible: Easy to build complex models
- Easy-to-use: Autograd, Python interface
- **Not** fast:
  - Overhead from graph representation
  - Do not scale well to dozens of GPUs

You can definitely be flexible and easy to use, as I think we've seen with graph-based frameworks like TensorFlow, Theano, Caffe2. Auto diff and Python interfaces make developing new models relatively painless, and you can build pretty much any model (not even limited to deep learning).

But these don't tend to be fast — they can introduce overhead from the graph representation, and in my experience don't scale well to dozens of GPUs by default, as they don't do a great job of optimizing communication in a synchronous gradient descent.

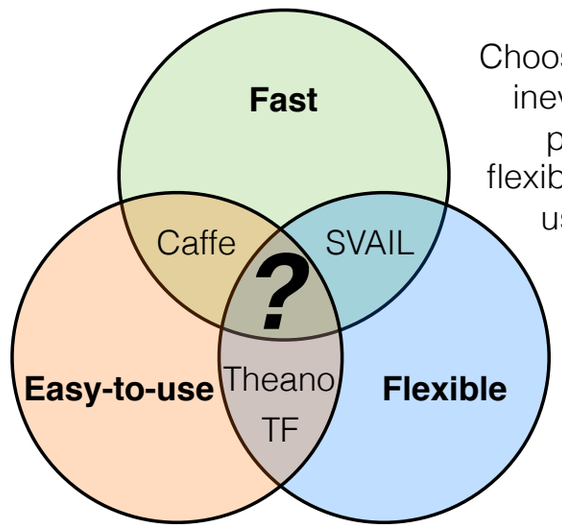
Fast and **Easy-toUse**

Example: Caffe (not Caffe2)

- Easy-to-use: Pre-packaged layers, etc.
- Fast: Can be well-optimized, have scaled.
- **Not** flexible: Limited to built-in layers, optimizers, algorithms

And you can be fast and easy to use — like Caffe, which gives you a predefined set of layers types and optimizers, and is very fast as a result. And we've seen Caffe work with MPI on many dozens of GPUs, so it succeeds there too.

Are Deep Learning Frameworks “Good Enough”?



Choosing a framework inevitably involves performance, flexibility, and ease of use trade-offs.

…close, but not quite yet!

But I have yet to see any framework which does all three of these successfully.

No framework is fast, flexible, *and*
easy to use.

Framework choice involves trade-offs.

So, the next moral of the story is this: Choosing a framework involves trade-offs between these three… and we're close to getting past that, but not quite there.

Agenda

1. **Progression of a deep learning application**
2. **Are deep learning frameworks “good enough”?**
3. **Writing fast deep learning frameworks**
4. Scaling with TensorFlow
   (or any graph-based framework)
5. Questions

So, how did we manage to make our internal SVAIL framework so fast?

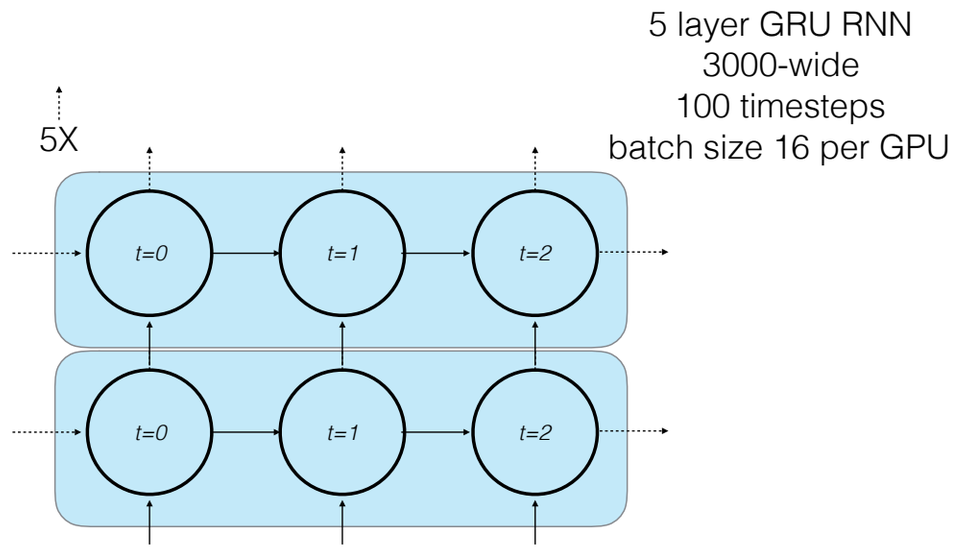
Writing fast deep learning frameworks

- You can't optimize what you can't measure.
- Extract the signal from the noise.
- Choose the right hardware for the job.
- ***Focus Today:*** *Minimize communication overhead.*

Well, there are a few bits here.

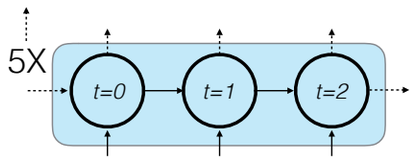
But the focus today is one in particular – minimize the overhead from communication.

Our Benchmark Model



In order to explain why that’s so important, let’s consider this RNN-based benchmark — a 5 layer, 3000 wide, 100 timestep GRU, with a batch size of 16 per GPU.

Our Benchmark Model



5 layer GRU RNN
3000-wide
100 timesteps
batch size 16 per GPU

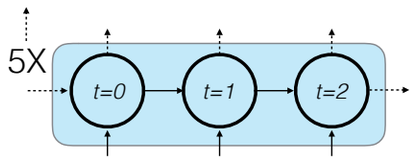
Parameters:

$$5 \text{ layers} \times \frac{6000 \times 9000 \text{ float matrix}}{\text{layer}} \times \frac{4 \text{ bytes}}{\text{float}} = 1 \text{ GB}$$

If you count the number of parameters this model has, it ends up being about a gigabyte of data.

The GRU matrix ends up being twice as wide and three times as tall as the width of the hidden state, due to the gating and forget gates in the GRU; that's where the 6000 and 9000 come from.

Our Benchmark Model



5 layer GRU RNN
3000-wide
100 timesteps
batch size 16 per GPU

Compute:

$$5 \text{ layers} \times \frac{6000 \times 9000 \text{ matrix}}{\text{layer-timestep}} \times$$

$$\times \text{batch size} \times 100 \text{ timesteps} \times 2 \text{ FLOPs} = \sim 1 \text{ TFLOPs}$$

If you count the number of FLOPs, floating point operations, you get about a teraflop, ten to the twelfth operations.

This is just accounting for the matrix multiplies, so it's a somewhat simplified model. There's a factor of two that comes from the fact that the dot product in a matrix multiply has both additions and multiplications.

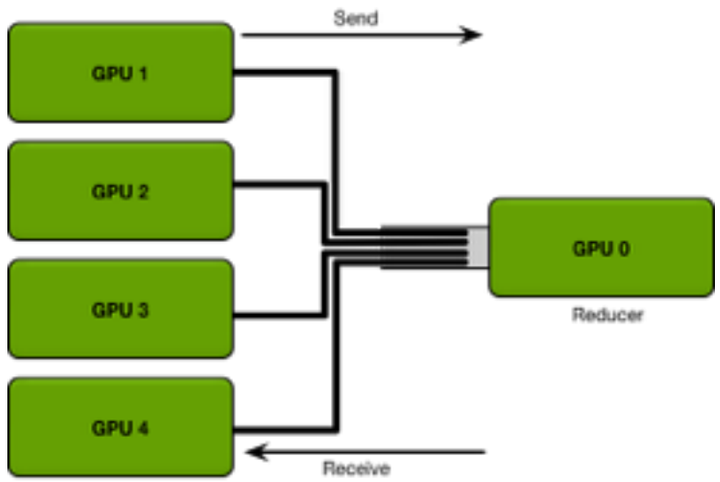
So, remember those numbers: 1 teraflop and 1 gigabyte of data.

Gradient Descent (SGD)

1. Each GPU: Forward propagation (~ 1 TFLOP)
2. Each GPU: Backward propagation (~ 1 TFLOP)
3. **All GPUs: Average gradients (communication)**
4. All GPUs: Apply gradients to weights
5. Loop from Step 1.

We’re going to use a fairly standard synchronous stochastic gradient descent, SGD. On each GPU we run forward and backward prop (both a teraflop), average the gradients from all GPUs, and then apply the gradients, and then repeat.

Multi-Tower Scaling



We’ll start with the simplest way of doing this:

Just send all the gradients to one GPU (or to a CPU) to do the averaging.

Multi-Tower Scaling

- **Step 3: Average gradients** – by sending all gradients to single GPU, or to CPU.
- Performance: For a system of *N* GPUs, main GPU sends / receives *N* copies of parameters (2(*N - 1)* GB transfer).
- **Compute:** 2 TFLOP / 6 TFLOP/s peak = ~300 ms
- **Communication**: 2(*N* - 1) GB / 6 GB/s bus = (*N* - 1) / 3 s

The issue with this is that the reducer GPU has to receive a ton of data!

For N GPUs with our benchmark model, it'll have to receive N - 1 gigabytes of data. And that will be a bottleneck in our training.

If we're using Titan X Maxwell GPUs, with a peak throughput of 6 teraflops, then we will spend at least 300 milliseconds on compute every iteration. But if we have a 6 GB / s bus, which is not crazy for a PCI-Express bus or for Infiniband, then we'll spend N - 1 / 3 seconds on communication.

Multi-Tower Scaling

- **Training on 4 GPUs, *max* 25% of time is spent in compute. 8 GPUs, 12% of time spent in compute.**
- ***Option 1:*** Give up on SGD, use variant of Async SGD.
- ***Option 2***: Use a better communication algorithm!

For 4 GPUs, 75% of time is communication overhead; for 8 GPUs, it's more like 90%.

This just doesn't work.

So we have two options: give up on synchronous SGD, or just do better.

Some frameworks (like TensorFlow) are build around the idea that you *have* to take option one.

But at SVAIL, we reject that. Synchronous SGD is convenient — its deterministic, it can converge to better models, it's simpler. We'd like to stick with it.

So we go with option 2: do your communication better.

Ring Allreduce

- **Algorithm:** Given equal-size linear array of floats on each GPU, do an in-place element-wise sum of the arrays.
- **Performance:** For a system of *N* GPUs, each GPU sends / receives *one copy* of parameters (2 GB transfer)
- **Result:** Scale to (almost) *any* number of GPUs, with minimal communication overhead.

We use an algorithm called the ring all reduce.

The ring all reduce can take a linear array of floats on each GPU, and then do an element-wise in-place sum of the arrays, which is effectively what you want to do for averaging gradients across GPUs.

Most importantly, for N GPUs, every GPU does just 1 GB of data transfer, and so you can use *any* number of GPUs. Your overhead doesn’t grow with the number of GPUs.

Ring Allreduce

- **Algorithm:**

  1. Set up communication ring.
  2. Divide array into N equal-size chunks.
  3. Scatter-reduce (N - 1 iterations).
  4. Allgather (N - 1 iterations).

This is a well-known algorithm from high performance computing, but let’s go over how it works nonetheless.

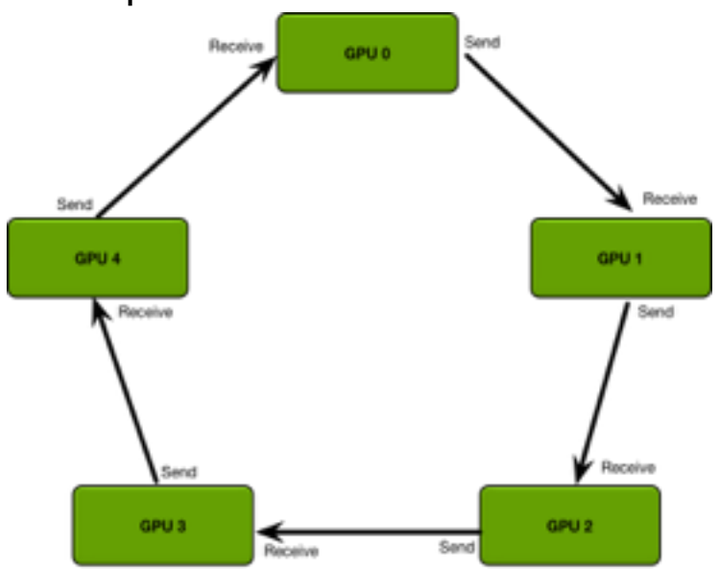
Ring Allreduce

- **Algorithm:**

  1. **Set up communication ring.**
  2. Divide array into N equal-size chunks.
  3. Scatter-reduce (N - 1 iterations).
  4. Allgather (N - 1 iterations).

First, we set up a ring of our GPUs.

Set Up Communication Ring



Each GPU sends to its right neighbor and receives from its left neighbor. This never changes — this is fixed for the entire execution of the training program.

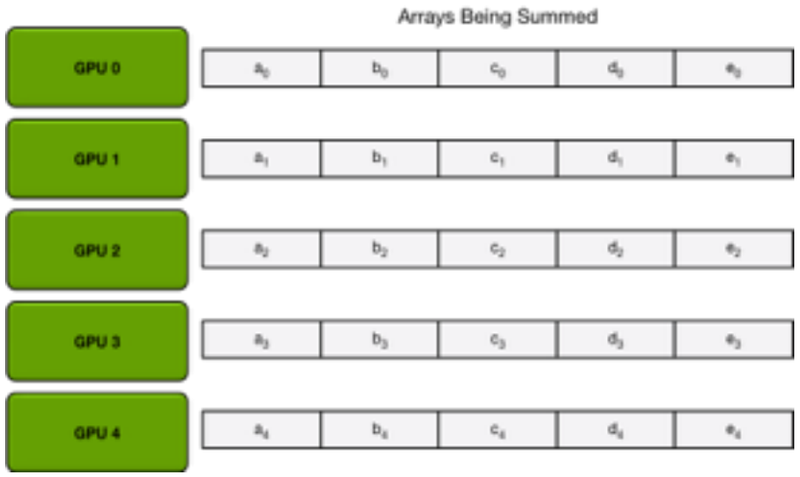
Ring Allreduce

- **Algorithm:**

  1. **Set up communication ring.**
  2. **Divide array into equal-size chunks.**
  3. Scatter-reduce (N - 1 iterations).
  4. Allgather (N - 1 iterations).

Next, we take the array we're currently reducing, and we divide it into equal chunks.

Divide Array



In this example, we have 5 GPUs. For simplicity let's just say that the array has 5 elements, although you can imagine that each of these elements is actually a block of numbers, and addition is elementwise.

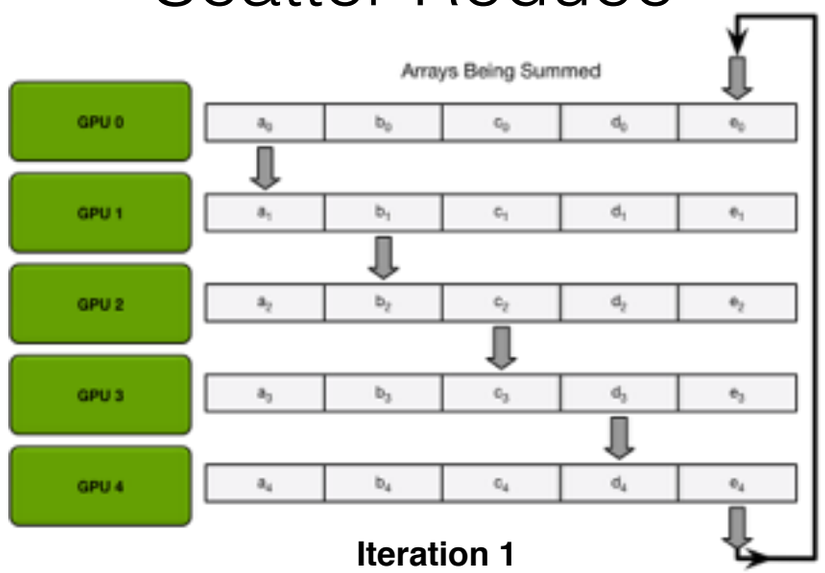
Initially, each array on each GPU is totally independent of the others, with its own values.

Ring Allreduce

- **Algorithm:**
  1. **Set up communication ring.**
  2. **Divide array into equal-size chunks.**
  3. **Scatter-reduce (N - 1 iterations).**
  4. Allgather (N - 1 iterations).

Next, we have N - 1 iterations of a scatter reduce. The purpose of a scatter reduce is to add up these chunks across the GPUs, so that at the end, every GPU has one “final” block.

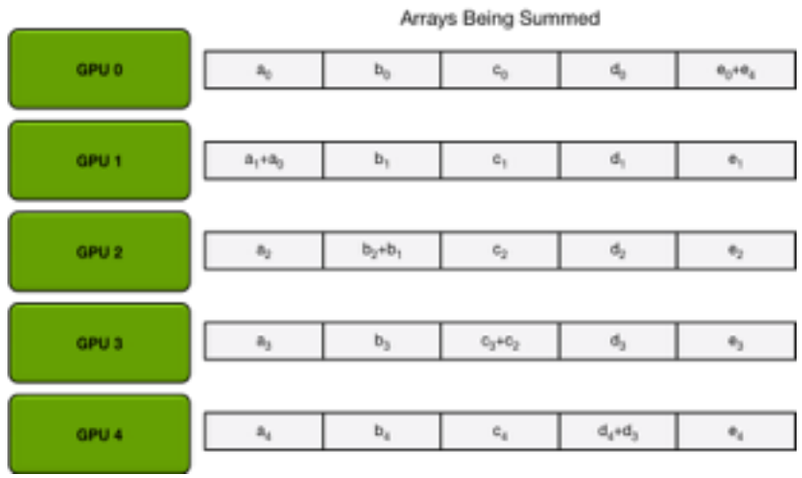
Scatter-Reduce



Iteration 1

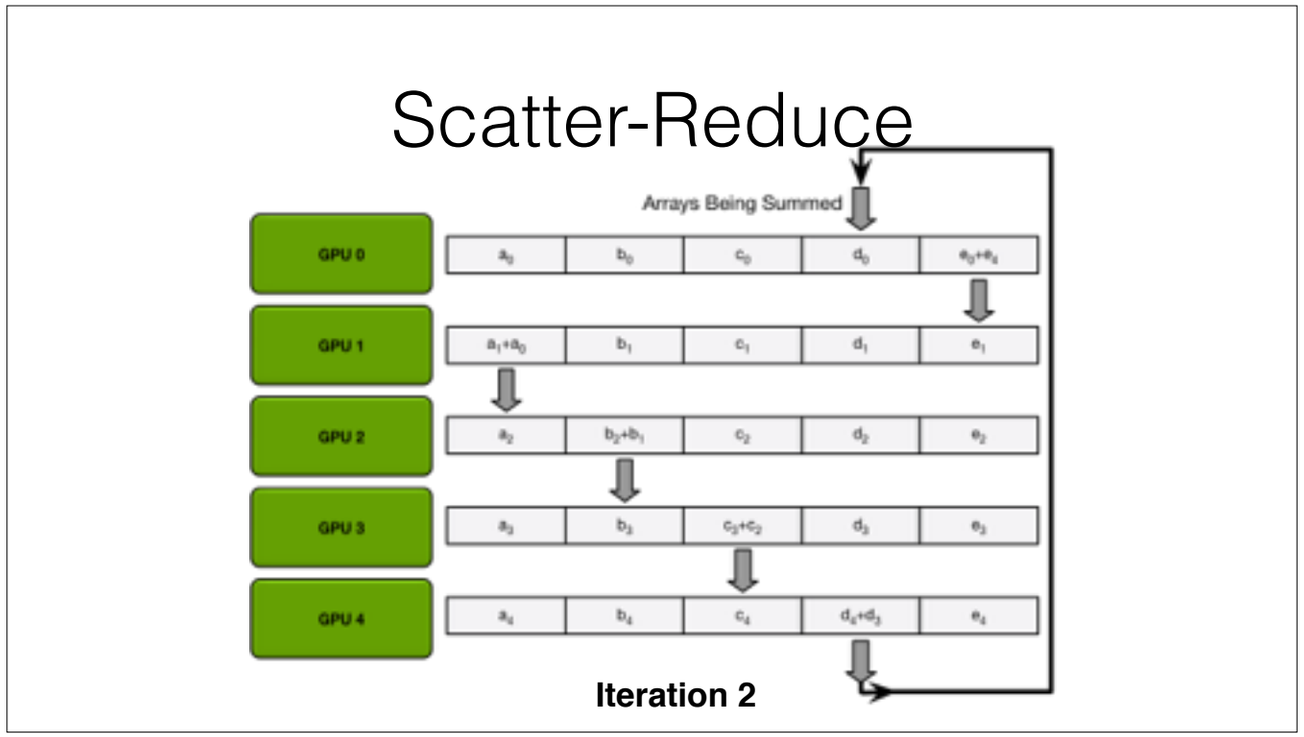
We start by having GPU 0 send its 0th block to GPU 1. GPU 1 its 1st block to GPU2, and so on, going down the ring and around.

Scatter-Reduce

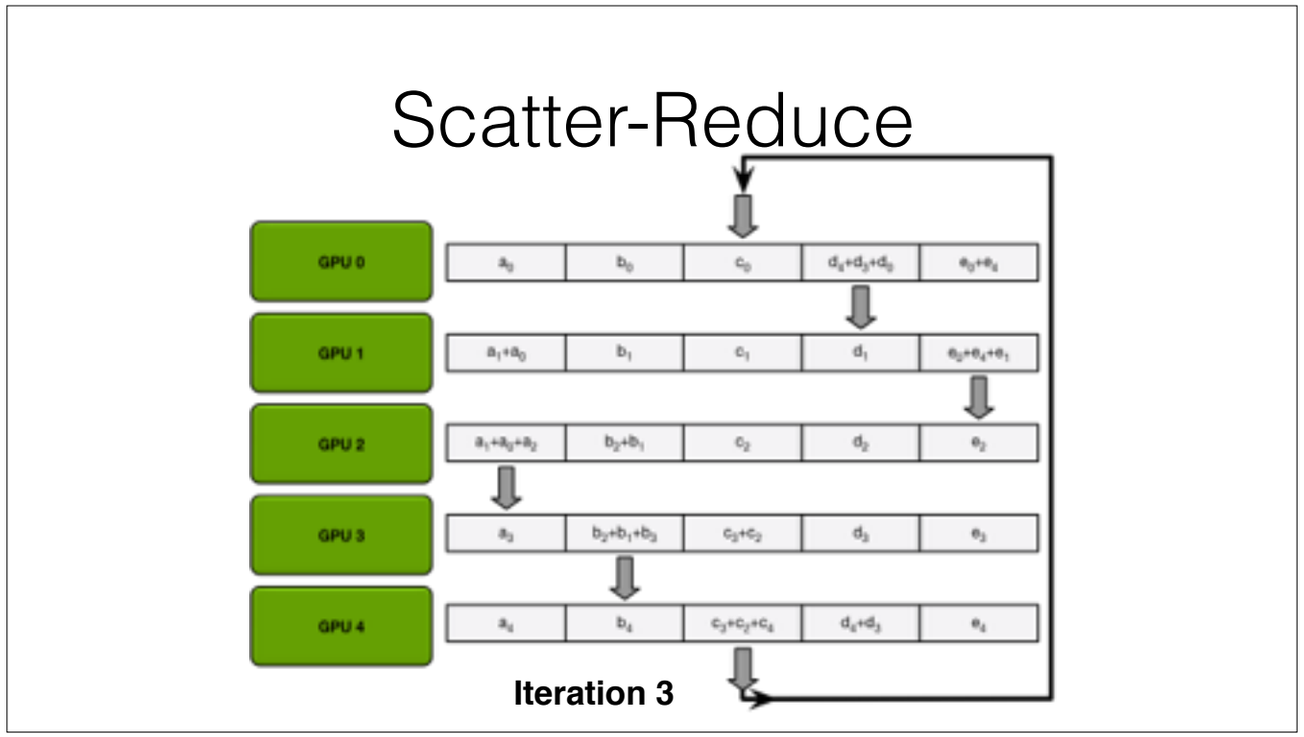


***After* Iteration 1**

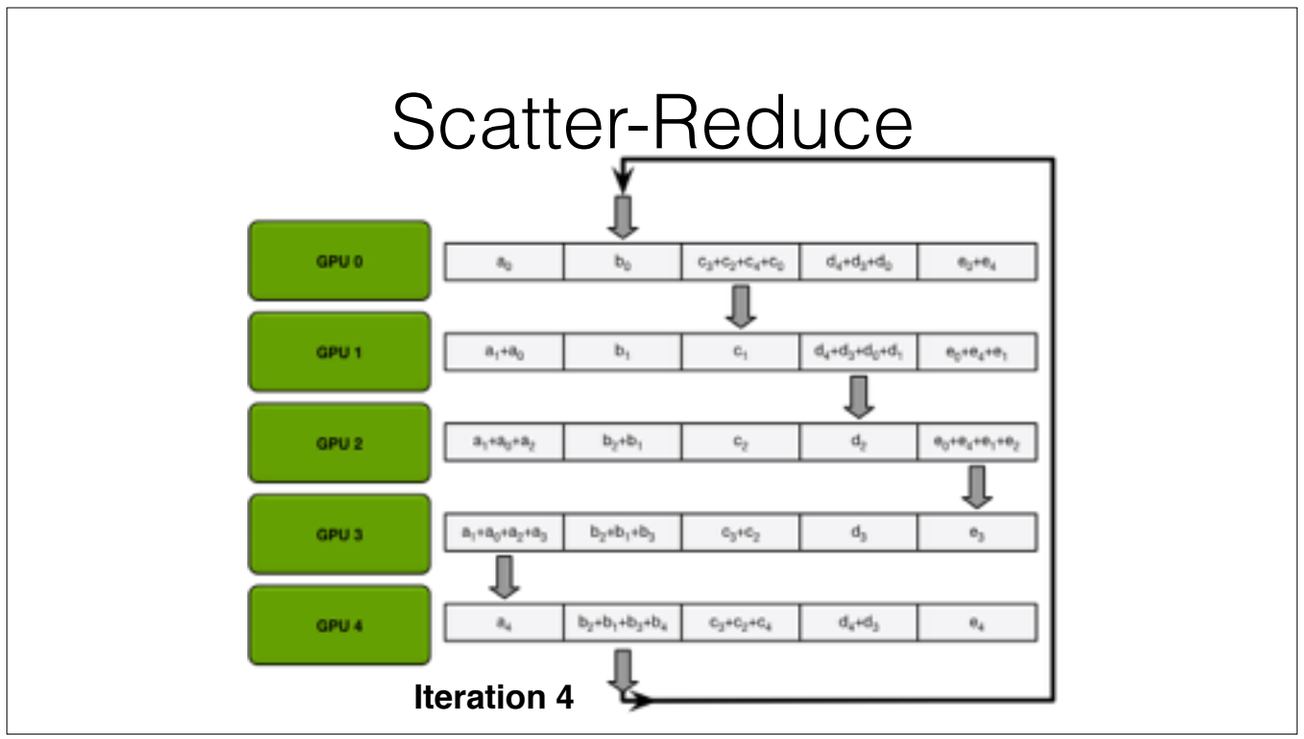
When a GPU receives data, it accumulates it into the right slot. You'll see here that GPU 1 has added a0 to a1, and so its 0th slot now has that sum in it.



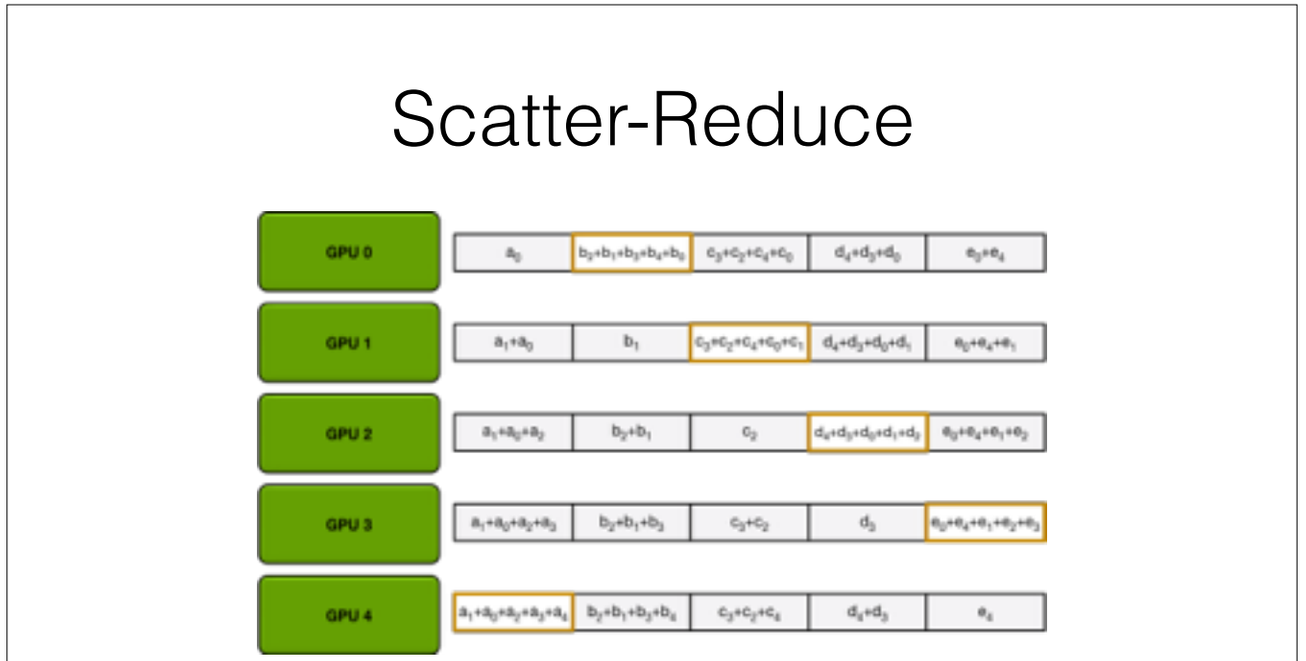
In the next iteration, we do the same thing, except this time the block that gets sent is the one that each GPU just received. For example, GPU 1 just received data from GPU 0, so it now sends its sum of those to GPU 2.



GPU 2 adds up what its receives and gets a0 plus a1 plus a2, and then in the next iteration sends THAT.



It happens again.



And finally, after N - 1 iterations, *some* GPU has the right value of each block. The first block is stored in its final form on GPU 4.

Ring Allreduce

- **Algorithm:**
  1. **Set up communication ring.**
  2. **Divide array into equal-size chunks.**
  3. **Scatter-reduce (N - 1 iterations).**
  4. **Allgather (N - 1 iterations).**

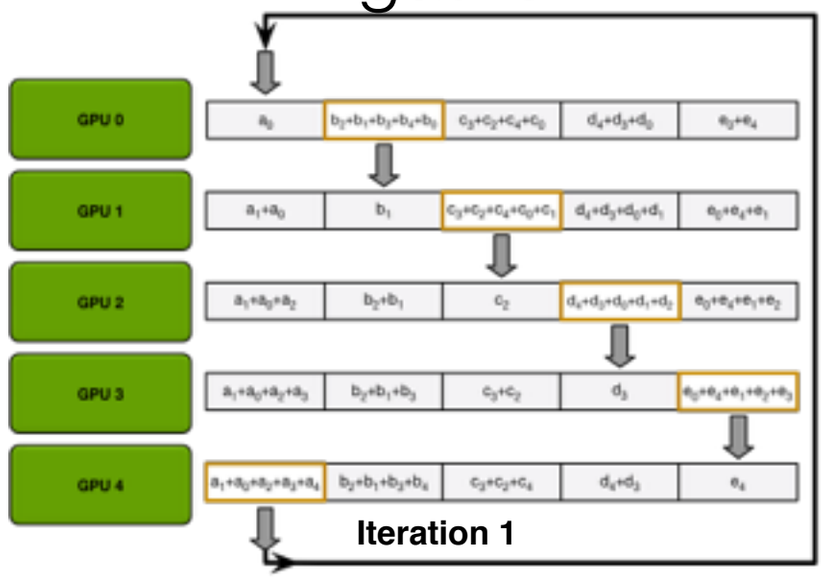
The last step is an allgather, which looks a lot like the scatter reduce. However, now, we transmit the finished aggregated chunks, and the goal is that at the end, *every* GPU has the right version of *every* block.

All gather

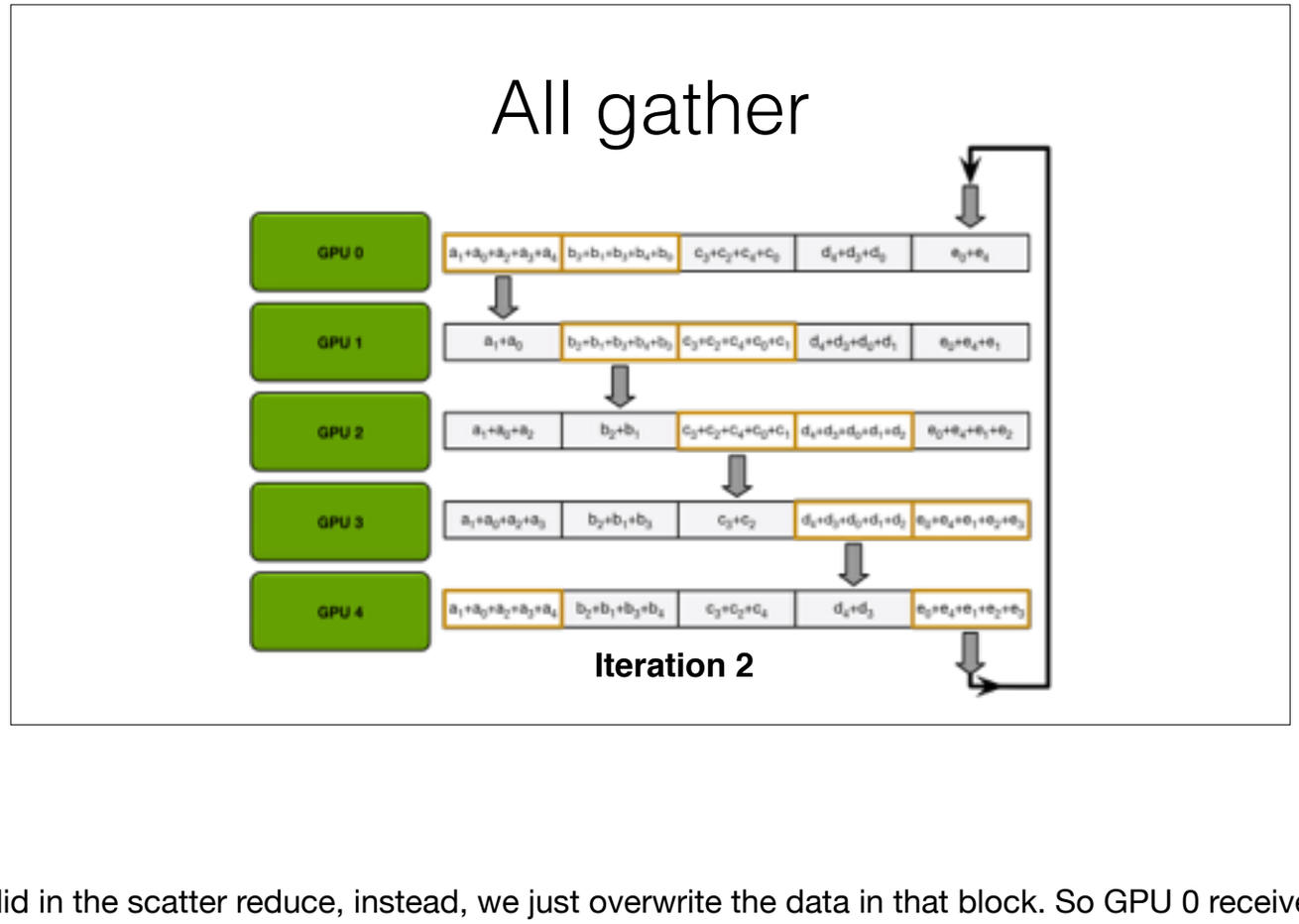
Initial State

We start from where we left off in the scatter reduce.

All gather



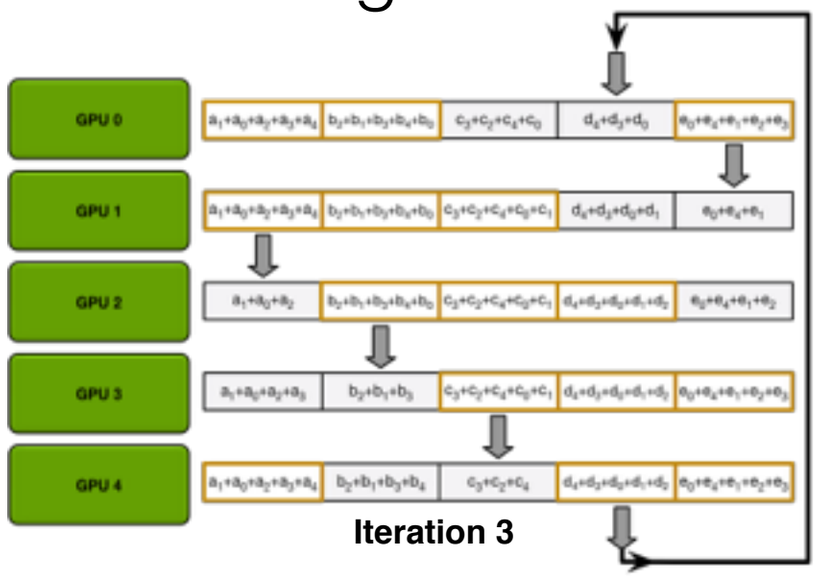
Each GPU starts by sending the chunk it has that is final to the next GPU. So GPU 4 will send chunk 0, since its fully accumulated there.



Instead of adding to the chunk like we did in the scatter reduce, instead, we just overwrite the data in that block. So GPU 0 receives some data from GPU 4 for block 0, and just writes it into the right place. So the yellow-outlined chunks are final values, and you see that GPU 0 has the final values in chunk zero.

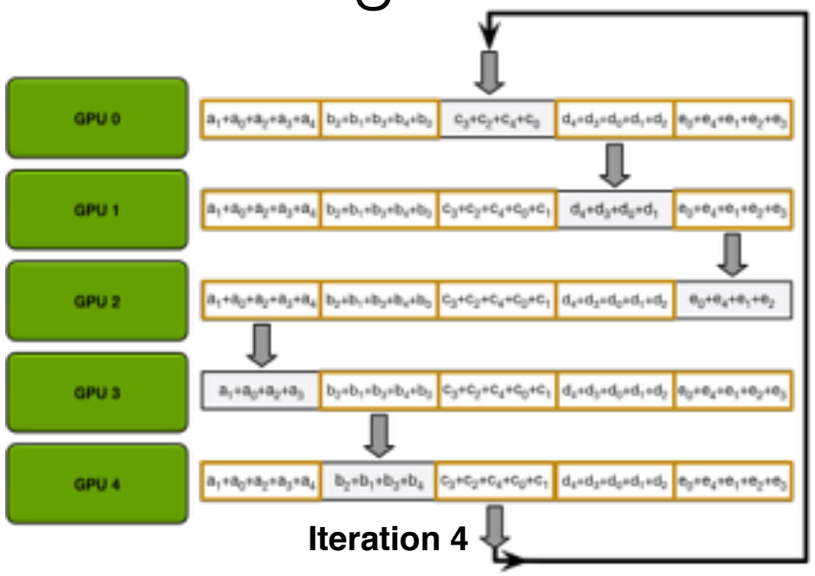
In the next iteration, each GPU sends along the values it just received, so GPU 0 sends chunk zero.

All gather

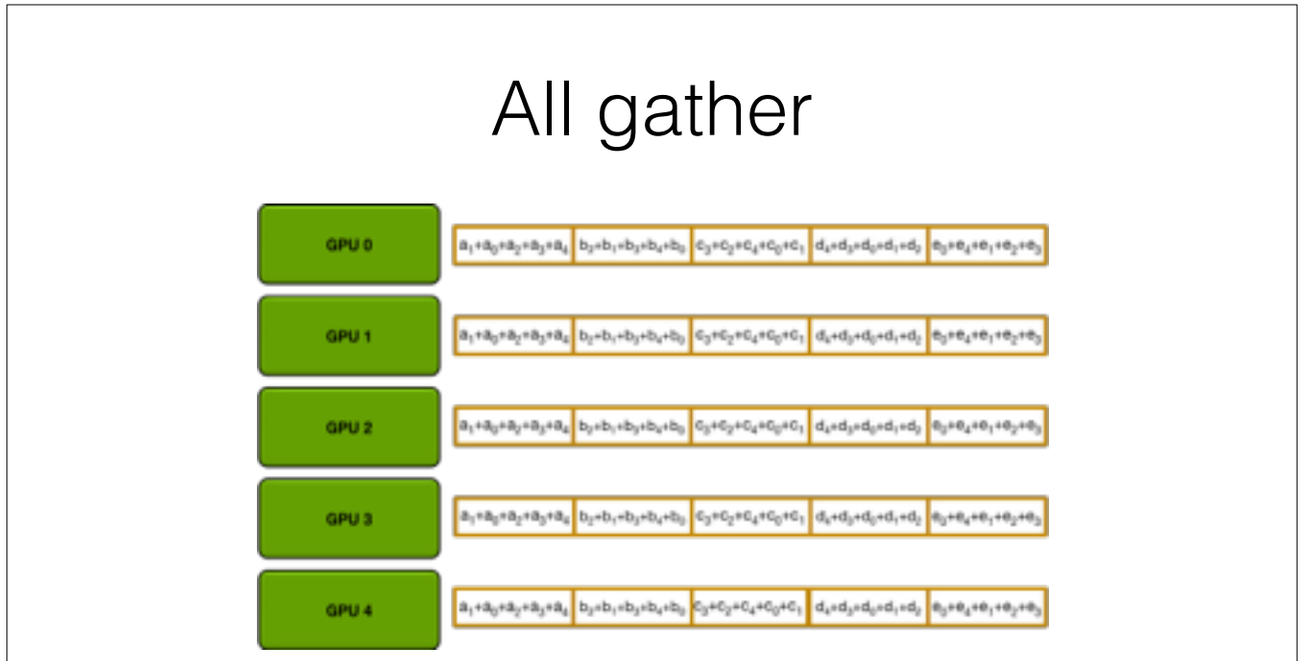


Again, we store the result, so every GPU has one more “final’ chunk stored on it.

All gather



We do it again.



And finally, after N - 1 iterations, the all reduce is done — every GPU has all the right values.

Ring Allreduce

- **Algorithm:**
  1. **Set up communication ring.**
  2. **Divide array into equal-size chunks.**
  3. **Scatter-reduce (N - 1 iterations).**
  4. **Allgather (N - 1 iterations).**

To recap, after the setup we did 2(N - 1) iterations total. N-1 for scatter reduce, N-1 for allgather.

Ring Allreduce

- All nodes end up with reduced (summed result)
- Each node sends 2X * (N - 1) / N bytes, where X is total number of bytes in the array
- Constrained primarily by *slowest* communication mechanism in your network (usually inter-node communication, e.g. Infiniband)

If the total amount of data is X, and there are N GPUs, each GPU does 2 (N - 1) iterations and sends X/N data in each iteration. So in this case, each GPU does 8 iterations, and each iteration sends a fifth of the data, so in total sends 8/5 of the total data magnitude. And you'll note that as N grows larger, 2 X * (N - 1) / N is effectively constant, because (N - 1) / N just goes to one.

So your speed is contained just by your *slowest* communication link, which is usually Infiniband, not by the number of GPUs.

Agenda

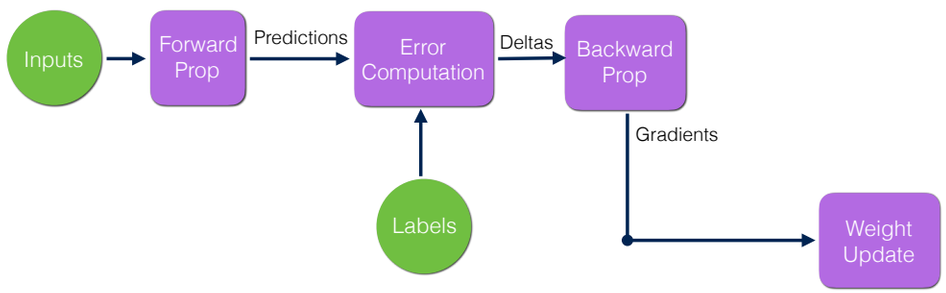
1. Progression of a deep learning application
2. Are deep learning frameworks “good enough”?
3. Writing fast deep learning frameworks
4. **Scaling with TensorFlow**
   (or any graph-based framework)
5. Questions

We spend two years doing this in our internal framework, and it works very well. We tested it up to something like 128 GPUs, and the performance hit is pretty much constant once you start using Infiniband.

But only recently in the past year or so have we figured out how to do the same thing with TensorFlow

Scaling with TensorFlow

- TensorFlow (graph-based frameworks) are flexible!
- View entire SGD step as a graph:

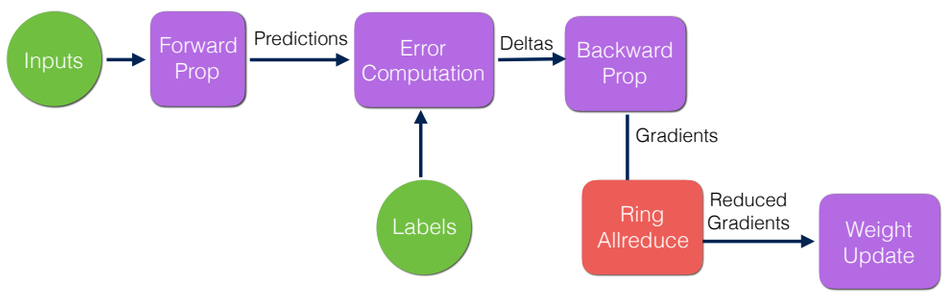


The key to doing this is to view the entire training step as a single graph (which is what TensorFlow encourages you do to anyways)

We take our inputs, do forward prop, do error computation, do backward prop, create gradients, and apply them.

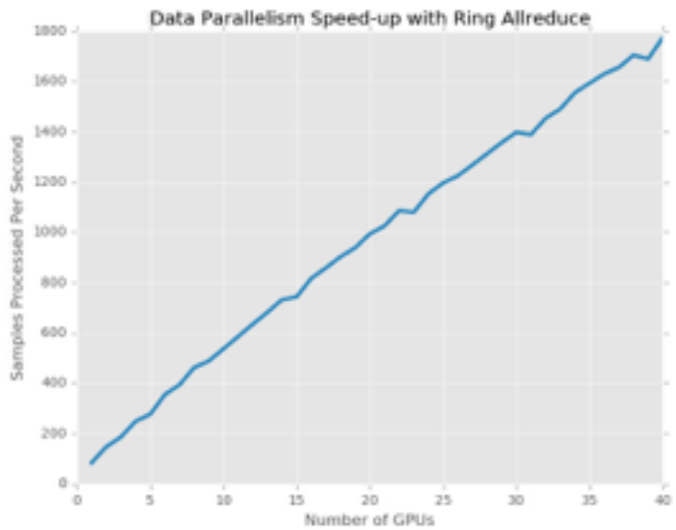
Scaling with TensorFlow

- Run many independent TensorFlow processes
- Insert allreduce as a node in the graph:



The only new element is that now, we can run *many* independent TensorFlow processes. And we add a new node into the graph — between backward prop and applying the gradients, we add an allreduce. And this can use something like nccl if you are only on one node, or use MPI like we do in order to scale to many nodes. MPI also lets you use CUDA interprocess communication if you use the right MPI implementation, and so you can do fast transfers between GPUs with it too.

Linear Scaling (40+ GPUs)



The result is what you expect: as we scale to 40 GPUs (and more), we have a completely linear performance curve. More GPUs, more data processed per second.

Scaling with TensorFlow

- In practice, framework-specific issues pop up – sparse gradients, inter-GPU communication on multiple GPU streams with CUDA IPC, etc.
- Hard to solve in a cross-framework manner, but same approach should work for many frameworks – nothing TensorFlow specific.
- Using system for past 9 months without major issues; slight performance hit vs. internal framework

In practice, this is a large amount of engineering work. You're dealing with several different libraries — CUDA, MPI, TensorFlow — none of which were designed with each other in mind. So it ends up being pretty tricky, but you can make it work

Because of this, it's hard to do it in a cross-framework manner, but the same ideas should work for any framework.

We've been using this system for a while internally now, and it works very well.

Scaling with TensorFlow

- Hard to overlay communication and compute
- We've released baidu-allreduce (demo implementation) and a TensorFlow patch with this system
- PaddlePaddle (Baidu's framework) does something very similar by default
- If you have a use-case, don't hesitate to talk to us!

We released baidu-allreduce as a demo implementation of this ring allreduce, and PaddlePaddle uses something very similar by default — but we'd like to see this become more common in other deep learning frameworks

We can't implement this for all frameworks, but we're happy to help out — so if you have any use cases of this, don't hesitate to reach out and talk to us.

Agenda

1. Progression of a deep learning application
2. Are deep learning frameworks “good enough”?
3. Writing fast deep learning frameworks
4. Scaling with TensorFlow
   (or any graph-based framework)
5. **Questions**

Thank you for listening!

We have a few minutes left for questions, and if there are any that aren’t answered by the end, feel free to catch me afterwards or reach out by email.