DSSTNE: Deep Learning At Scale For Large Sparse Datasets

https://github.com/amznlabs/amazon-dsstne

Scott Le Grand
Senior Scientist
Teza Technologies

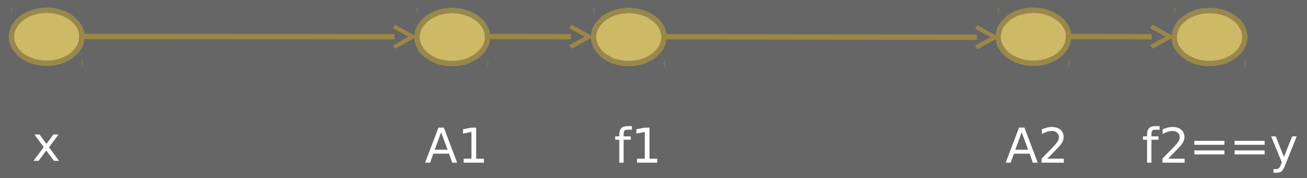
varelse2005@gmail.com

Outline

- What's Deep Learning?
- Why GPUs?
- Deep Learning for Recommendations at Amazon
- DSSTNE
- Benchmarks
- DSSTNE at scale
- Deep Learning for The 99%

What's Deep Learning (Neural Networks)?

- World’s most lucrative application of the chain rule from calculus (as applied to a graph)
- x is the input data
- A1 and A2 are linear transformations
- f1 and f2 are some sort of nonlinear function



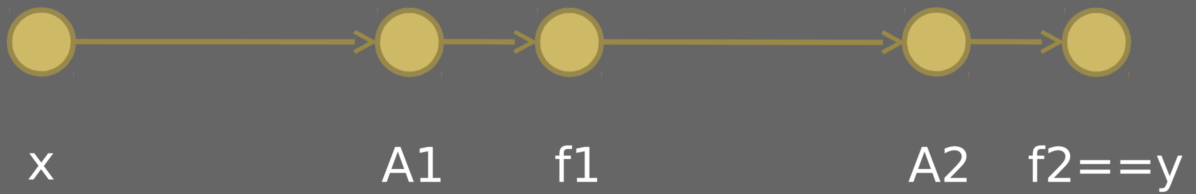
$$\mathrm{y}= \mathrm{f2}\left(A2\left(f1\left(A1(x)\right)\right)\right)$$

Nonlinear (Activation) Functions

- Linear: =x
- Sigmoid: $=\frac{1}{1+e^{-x}}$
- Tanh: $=\frac{e^{x}+e^{-x}}{e^{x}-e^{-x}}$
- Relu: =max(x, 0)
- SoftPlus: =log(1+e^{x})
- SoftSign: $=\frac{1}{1+|x|}$
- SoftMax: $=\frac{e^{x_i}}{\sum_j e^{x_{ij}}}$

Neural Network Training

Training: Minimize an Error Function E(y, t)



L1: E(y, t) = $|y - t|$

L2: E(y, t) = $(y - t)^2$

Cross Entropy: E(y, t) = -t*log(y) –(1-t)*log(1-y)

Neural Network Derivatives (BackPropagation）

$$\frac{@E}{@x} = \frac{@E}{@f2}\frac{@f2}{@A2}\frac{@A2}{@f1}\frac{@f1}{@A1}\frac{@A1}{@x}$$

$$\frac{@E}{@A2_{ij}} = \frac{@E}{@f2}\frac{@f2}{@A2}\frac{@A2}{@A2_{ij}}$$

$$\frac{@E}{@A1_{ij}} = \frac{@E}{@f2}\frac{@f2}{@A2}\frac{@A2}{@f1}\frac{@f1}{@A1}\frac{@A1}{@A1_{ij}}$$

Deep Learning/Neural Networks in One Slide*

$$X_{L+1} = X_L * W_{L \rightarrow L+1}$$

$$\delta_L = \delta_{L+1} * W^T_{L \rightarrow L+1}$$

$$\Delta W_{L \rightarrow L+1} = X^T_L * \delta_{L+1}$$

*The definitive answer to whether you should take Calculus, Statistics and Linear Algebra in college

Why GPUs?

“A Graphics Processing Unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display.”

No Man's Sky

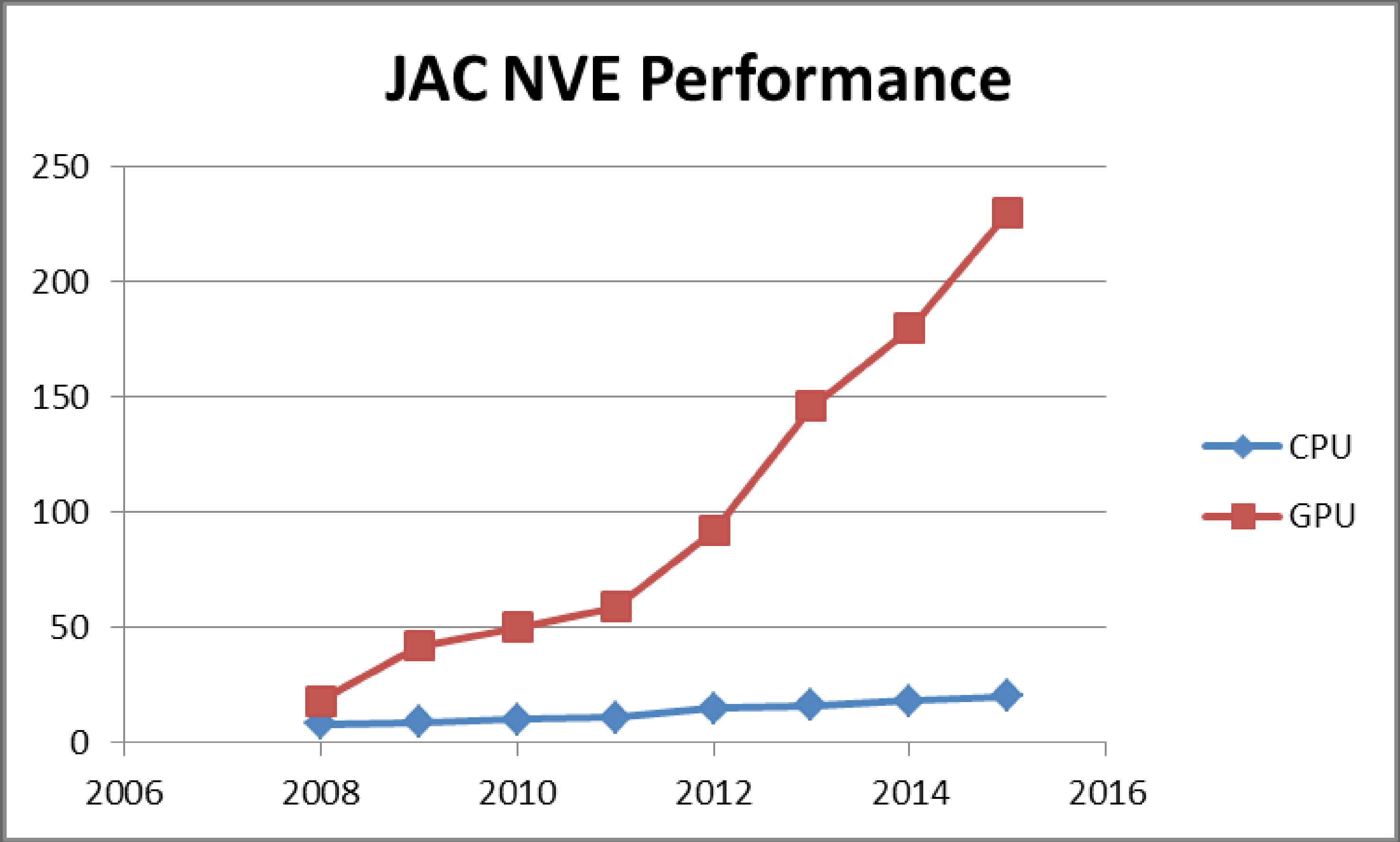
Horizon Zero Dawn

Pretty Pictures Require Lots of Math and Data

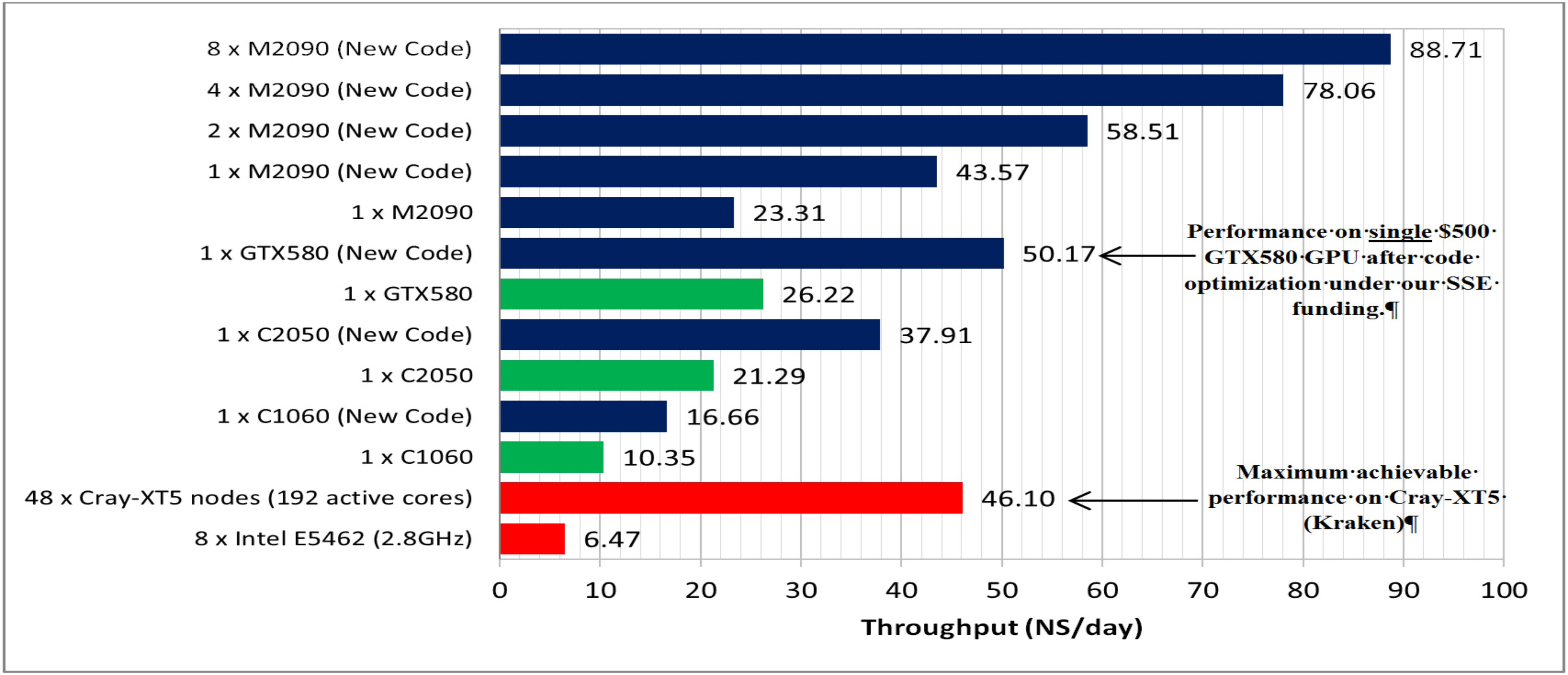
- Intel Core i7-6950x CPU: $1,723, 10 cores, 1.12 TFLOPS, 60 GB/s
- NVIDIA GTX Titan XP GPU: $1,200, 56 cores, 10.8 TFLOPS, 480 GB/s
- NVIDIA GTX 1080Ti GPU: $699, 56 cores, 11.2 TFLOPS, 484 GB/s*
- AMD R9 Fury X GPU: $500, 64 cores, 8.6 TFLOPS, 512 GB/s

*About 8-10x the performance for less than half the price

What can 11 TFLOPS do for you?



JAC NVE Benchmark (2011)



Performance·on·single·$500· GTX580·GPU·after·code· optimization·under·our·SSE· funding.¶

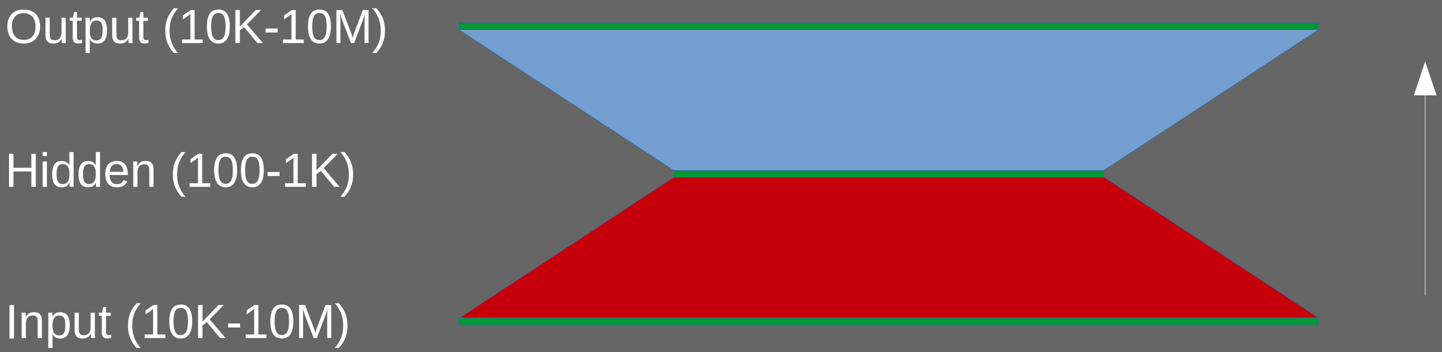
Maximum·achievable· performance·on·Cray-XT5· (Kraken)¶

Product Recommendations Also Require Lots of Arithmetic (2014)

What are people who bought items A, B, C...Z most likely to purchase next?

Traditionally addressed with variants of Matrix Factorization, Logistic Regression, Naive Bayes, Thompson Sampling, etc...

So why not Deep Learning?

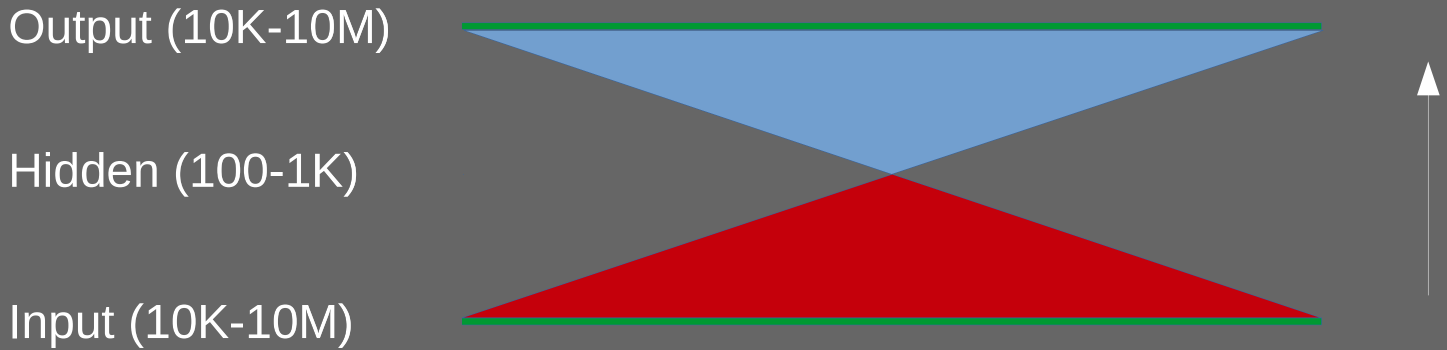


Output (10K-10M)

Hidden (100-1K)

Input (10K-10M)

Large Output Layers, Small Hidden Layers



Output (10K-10M)

Hidden (100-1K)

Input (10K-10M)

Existing frameworks were not designed to handle neural networks with input (purchase history) and output (recommendations) layers 10K to 10M units wide because…

This Is A Huge Sparse Data Problem

- Uncompressed sparse data either eats a lot of memory or it eats a lot of bandwidth uploading it to the GPU
- Naively running networks with uncompressed sparse data leads to lots of multiplications of zero and/or by zero. This wastes memory, power, and time
- Product Recommendation Networks can have billions of parameters that cannot fit in a single GPU so summarizing...

Framework Requirements (2014)

- Efficient support for large input and output layers
- Efficient handling of sparse data (i.e. don't store zero)
- Automagic multi-GPU support for large networks and scaling
- Avoids multiplying zero and/or by zero
- <24 hours training and recommendations cycle
- Human-readable descriptions of networks (API)

DSSTNE: Deep Sparse Scalable Tensor Network Engine*

- •A Neural Network framework released into OSS by Amazon in May of 2016
- •Optimized for large sparse data problems
- •Extremely efficient automagic model-parallel multi-GPU support
- •~6x faster than TensorFlow on such datasets (and that's just on one GTX Titan X (Maxwell), ~15x faster using 4 of them)
- •100% Deterministic Execution #reproducibilitymatters #noASGD
- •Full SM 3.x, 5.x, and 6.x support (Kepler or better GPUs)
- •Distributed training support OOTB (~20 lines of MPI Collectives)

*"Destiny"

Key Features

- Stores networks and data sets in NetCDF format with optional HDF5 support
- Multi-GPU handled with MPI and Interprocess CUDA P2P copies
- Initial emphasis on fully-connected networks, convolutional and pooling layer support was added late in 2016
- Dependencies are C++11, CUDA 7.x+, netcdf, a C++11-aware MPI library, libjsoncpp, and cuDNN*
- There are no computational shortcuts here, all we're doing is avoiding multiplying by zero and storing/copying zeroes

*Why isn't cuDNN just part of the CUDA Toolkit? Anyone? Bueller? Bueller?

Neural Networks As JSON Objects

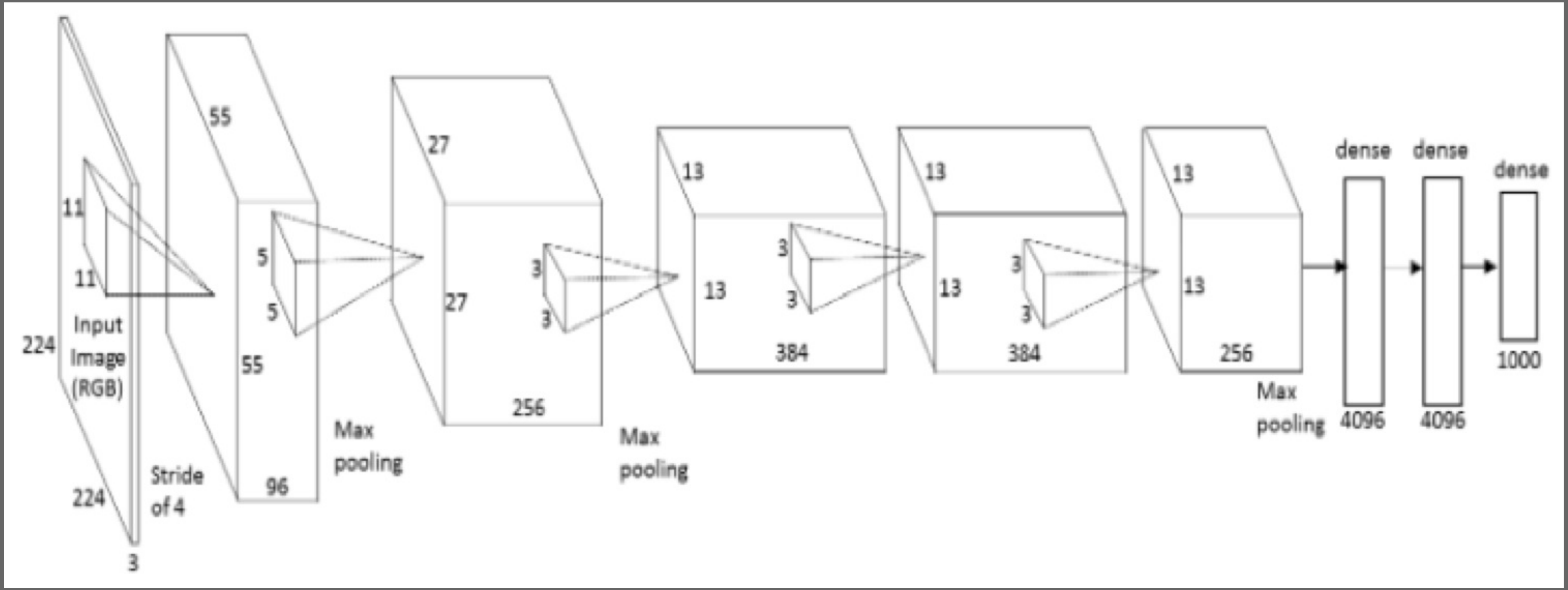
```
{
    "Version" : 0.7,
    "Name" : "AE",
    "Kind" : "FeedForward",
    "SparsenessPenalty" : {
        "p" : 0.5,
        "beta" : 2.0
    },

    "ShuffleIndices" : false,

    "Denoising" : {
        "p" : 0.2
    },

    "ScaledMarginalCrossEntropy" : {
        "oneTarget" : 1.0,
        "zeroTarget" : 0.0,
        "oneScale" : 1.0,
        "zeroScale" : 1.0
    },
    "Layers" : [
        { "Name" : "Input", "Kind" : "Input", "N" : "auto", "DataSet" : "input", "Sparse" : true },
        { "Name" : "Hidden", "Kind" : "Hidden", "Type" : "FullyConnected", "N" : 128, "Activation" : "Sigmoid", "Sparse" : true },
        { "Name" : "Output", "Kind" : "Output", "Type" : "FullyConnected", "DataSet" : "output", "N" : "auto", "Activation" : "Sigmoid", "Sparse" : true }
    ],

    "ErrorFunction" : "ScaledMarginalCrossEntropy"
}
```

AlexNet As A JSON Object*

```
{
    "Version" : 0.81,
    "Name" : "AlexNet",
    "Kind" : "FeedForward",
    "LocalResponseNormalization" :
    {
        "k" : 2,
        "n" : 5,
        "alpha" : 0.0001,
        "beta" : 0.75
    },
    "Layers" : [
        { "Kind" : "Input", "Type" : "Convolutional", "N" : "auto", "DataSet" : "input"},
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 96, "Kernel" : [11, 11], "KernelStride" : [4, 4],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Pooling", "Function" : "LRN" },
        { "Kind" : "Hidden", "Type" : "Pooling", "Function" : "Max", "Kernel" : [3, 3], "KernelStride" : [2, 2]},
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 256, "Kernel" : [5, 5], "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Pooling", "Function" : "LRN" },
        { "Kind" : "Hidden", "Type" : "Pooling", "Function" : "Max", "Kernel" : [3, 3], "KernelStride" : [2, 2] },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 384, "Kernel" : [3, 3], "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 384, "Kernel" : [3, 3], "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 256, "Kernel" : [3, 3], "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Pooling", "Function" : "Max", "Kernel" : [3, 3], "KernelStride" : [2, 2] },
        { "Kind" : "Hidden", "Type" : "FullyConnected", "N" : 4096, "Activation" : "Relu", "pDropout" : 0.5 },
        { "Kind" : "Hidden", "Type" : "FullyConnected", "N" : 4096, "Activation" : "Relu", "pDropout" : 0.5 },
        { "Kind" : "Output", "Type" : "FullyConnected", "N" : "auto", "DataSet" : "output", "Activation" : "SoftMax" }
    ],
    "ErrorFunction" : "CrossEntropy"
}
```

*Accidentally similar to Andrej Karpathy's ConvnetJS framework

AlexNet


224
224
3
11
11
Input
Image
(RGB)
Stride
of 4
55
55
96
5
5
Max
pooling
27
27
256
3
3
Max
pooling
13
13
384
3
3
13
13
384
3
3
13
13
256
Max
pooling
dense
4096
dense
4096
dense
1000

VGG16 As A JSON object

```
{
   "Version" : 0.81,
    "Name" : "VGG-16",
    "Kind" : "FeedForward",
    "LocalResponseNormalization" :
    {
        "k" : 2,
        "n" : 5,
        "alpha" : 0.0001,
        "beta" : 0.75
    },
    "Layers" : [
        { "Kind" : "Input", "N" : [224, 224, 3], "DataSet" : "input"},
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 64, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 64, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Pooling", "Function" : "Max", "Kernel" : [2, 2], "KernelStride" : [2, 2] },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 128, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 128, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Pooling", "Function" : "Max", "Kernel" : [2, 2], "KernelStride" : [2, 2] },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 256, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 256, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 256, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Pooling", "Function" : "Max", "Kernel" : [2, 2], "KernelStride" : [2, 2] },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 512, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 512, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 512, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Pooling", "Function" : "Max", "Kernel" : [2, 2], "KernelStride" : [2, 2] },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 512, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 512, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Hidden", "Type" : "Convolutional", "N" : 512, "Kernel" : [3, 3], "KernelStride" : [1, 1],  "Activation" : "Relu" },
        { "Kind" : "Pooling", "Function" : "Max", "Kernel" : [2, 2], "KernelStride" : [2, 2] },
        { "Kind" : "Hidden", "Type" : "FullyConnected", "N" : 4096, "Activation" : "Relu", "pDropout" : 0.5 },
        { "Kind" : "Hidden", "Type" : "FullyConnected", "N" : 4096, "Activation" : "Relu", "pDropout" : 0.5 },
        { "Kind" : "Output", "Type" : "FullyConnected", "N" : "auto", "DataSet" : "output", "Activation" : "SoftMax" }
    ],
    "ErrorFunction" : "CrossEntropy"
}
```

Human-Readable Doesn't Suck..

```
[illegible]
```

TLDR: 278 Lines of Code for AlexNet in Caffe...

JSON API Is Just An Interface to DSSTNE

```
struct NNNetworkDescriptor
{
    string                        _name;                   // Optional name for neural network
    NNNetwork::Kind               _kind;                   // Either AutoEncoder or FeedForward (default)
    ErrorFunction                 _errorFunction;          // Error function for training
    vector<NNLayerDescriptor>     _vLayerDescriptor;       // Vector containing neural network layers
    vector<NNWeightDescriptor>    _vWeightDescriptor;      // Vector containing preloaded weight data
    bool                          _bShuffleIndices;        // Flag to signal whether to shuffle training data or not
    uint32_t                      _maxout_k;               // Size of Maxout (default 2)
    NNFloat                       _LRN_k;                  // Local Response Normalization offset (default 2)
    uint32_t                      _LRN_n;                  // Local Response Normalization spread (default 5)
    NNFloat                       _LRN_alpha;              // Local Response Normalization scaling (default 0.0001)
    NNFloat                       _LRN_beta;               // Local Response Normalization exponent (default 0.75)
    bool                          _bSparsenessPenalty;     // Specifies whether to use sparseness penalty on hidden layers or not
    NNFloat                       _sparsenessPenalty_p;    // Target sparseness probability for hidden layers
    NNFloat                       _sparsenessPenalty_beta; // Sparseness penalty weight
    bool                          _bDenoising;             // Specifies whether to use denoising on input layers
    NNFloat                       _denoising_p;            // Probability of denoising inputs (for sparse layers, only denoise on non-zero values)
    NNFloat                       _deltaBoost_one;         // Adjusts scaling of nonzero-valued outputs
    NNFloat                       _deltaBoost_zero;        // Adjusts scaling of zero-valued outputs
    NNFloat                       _SMCE_oneTarget;         // Relaxed target for non-zero target values (Default 0.9)
    NNFloat                       _SMCE_zeroTarget;        // Relaxed target for zero target values (Default 0.1)
    NNFloat                       _SMCE_oneScale;          // Scaling factor for non-zero target values (Default 1.0)
    NNFloat                       _SMCE_zeroScale;         // Scaling factor for zero target values (Default 1.0)
    string                        _checkpoint_name;        // Checkpoint file name
    int32_t                       _checkpoint_interval;    // Number of epochs between checkpoints
    int32_t                       _checkpoint_epochs;      // Number of epochs since last checkpoint
    NNNetworkDescriptor();
};
```

DSSTNE's Engine is API-Agnostic

Amazon's Definition of Sparsity

- 0.01% to 0.1% Density, far lower than the optimal sparsity for the cuSparse library (too cuSlow)
- Sparse data stored in CSR format (Index, value) or just indices
- Sparse SGEMM is 5-20x faster than a full SGEMM depending on density (ultimately memory-limited)
- Sparse input layers are nearly “free”

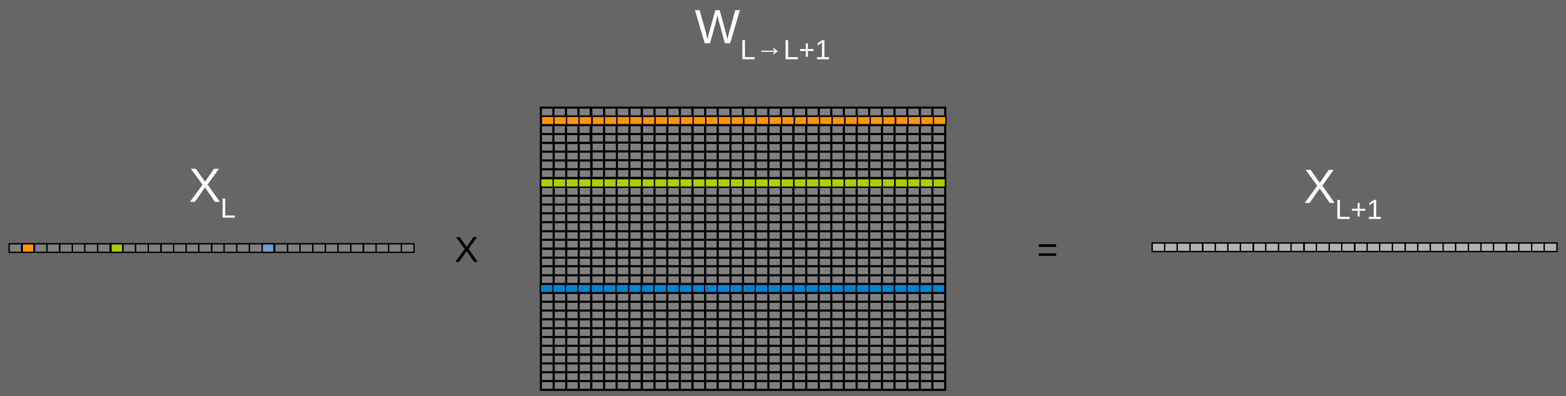
Sparse Neural Network Training*

$$X_{L+1} = X_L * W_{L \rightarrow L+1}$$

$$\Delta W = X^T_L * \delta_{L+1}$$

*Sparse output layers are easy (exercise for the listener)

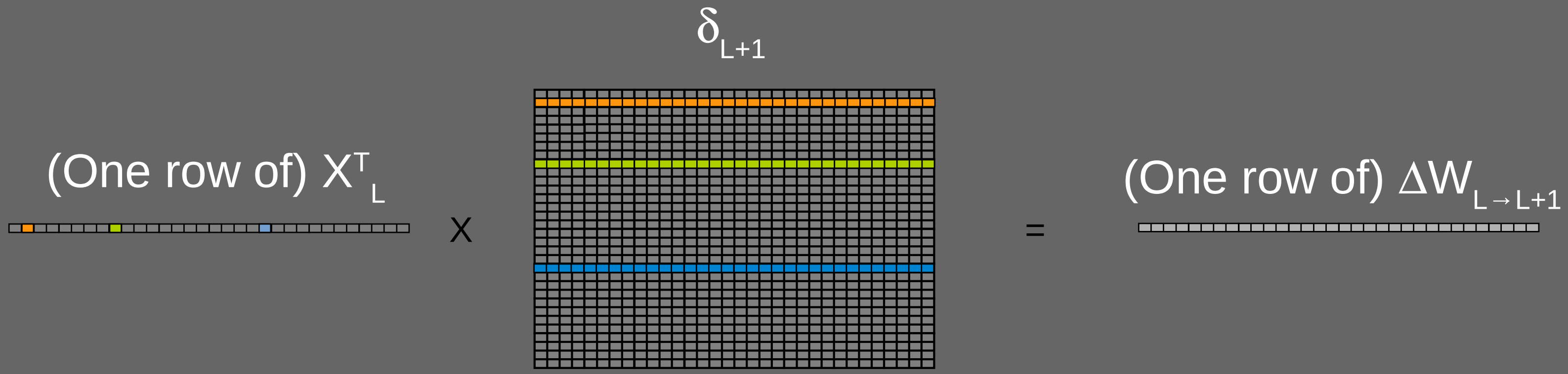
Sparse $X_{L+1} = X_L \quad * \quad W_{L \to L+1}$



Sparse $\Delta W = X^T_L * \delta_{L+1}$

- Need to transpose X_L matrix in parallel
- This is easy to do with atomic ops
- But the transpose ordering is not deterministic, floating point math is not associative (A + B + C) != (C + A + B)
- Solution: use 64-bit fixed point summation because fixed point accumulation is associative (A + B + C) == (C + A + B)
- 64-bit fixed point adds are also 32-bit instructions on NVIDIA GPUs (that means they aren't stupid slow on consumer GPUs)

Sparse $\Delta W_{L \to L+1} = X^T_L \ * \ \delta_{L+1}$



δ_{L+1}

(One row of) X^T_L

X

=

(One row of) $\Delta W_{L \to L+1}$

Determinism

- Consumer GPU failure rate is up to 20%
- Tesla GPU ECC only covers system memory, run twice
- 2 consumer GPUs cost $2400 versus $5000+ for a Tesla GPU
- Otherwise, race conditions become impossible to detect
- Otherwise, uninitialized variables become impossible to detect
- Data Scientists can provide simple bug repros
- Non-deterministic behavior is a bug
- invest engineering hours upfront, save lots more later

Large Networks

“My belief is that we’re not going to get human-level abilities until we have systems that have the same number of parameters in them as the brain.” - Geoffrey Hinton

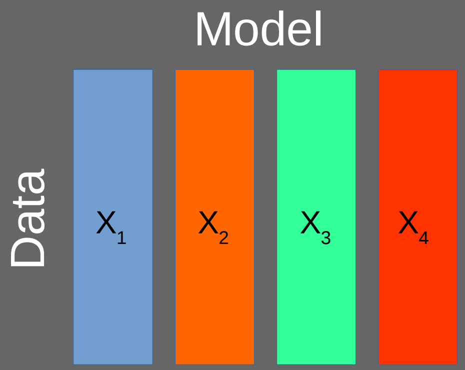
Two Definitions

Data Parallel: Shard dataset(s) by example across all processors, each of which has a full copy of the neural network model

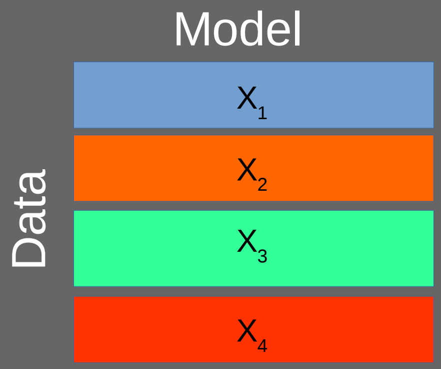
Model Parallel: Shard neurons and parameters across all processors

One O(D/N) Weird(er) Trick

Model Parallel



Data Parallel



Model Parallel vs Data Parallel

- Amazon Product Categories range from 10K to 10M items
- The Amazon Catalog is billions of items (Holy Grail)
- GPUs have up to 12 (2015) $ I mean 24 (2016) $$ oops I mean 32 GB (2016) $$$ of memory
- All the interesting problems need >12 GB of memory ruling out data-parallel
- Data Parallel Implementation unacceptably slow for networks with fully connected layers (GBs of weight gradients)

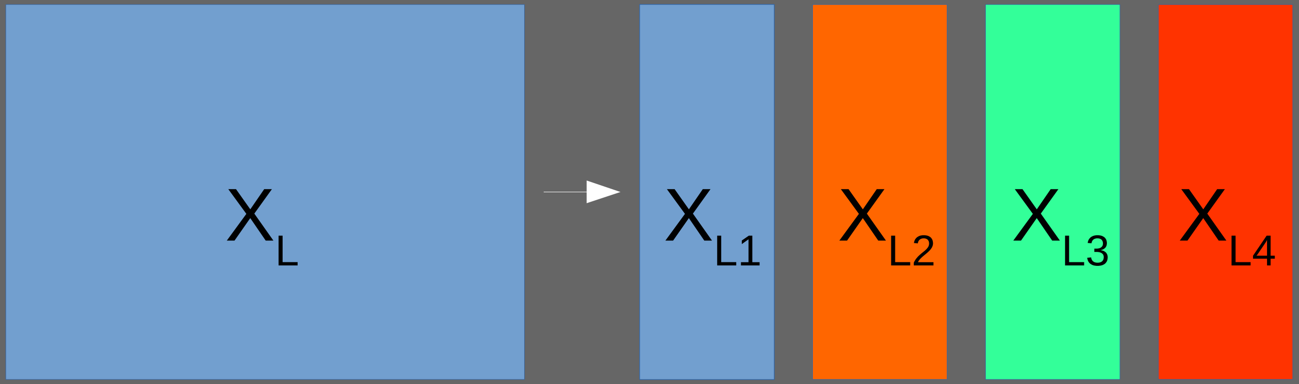
“Automagic” Model Parallel

- Uses the same JSON Object
- 1 GPU/process because reasons(tm) and simplicity
- DSSTNE Engine automatically distributes the neural network based on the number of processes running

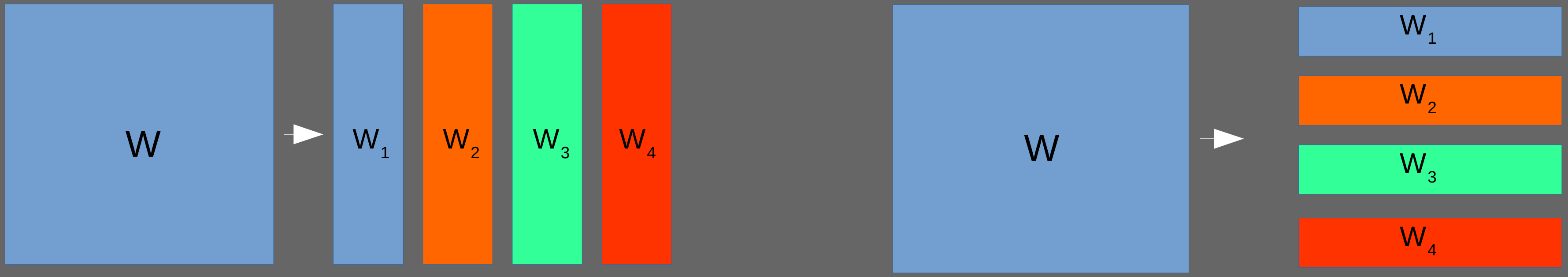
```
./trainer                     (serial job, 1 GPU)
mpirun -np 3 ./trainer        (model parallel, 3 GPUs)
mpirun -np n ./predictor      (model parallel, n GPUs)
```

One Weird Trick For Model Parallelism

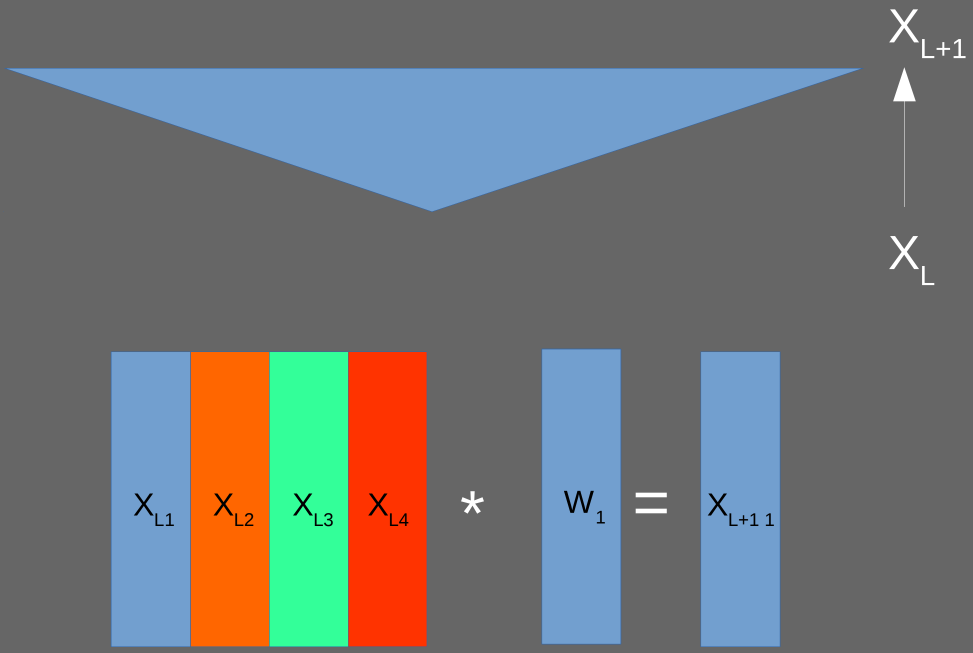
To parallelize an SGEMM operation, first one shards the input data across all N GPUs (N = 4 here)



Two Ways To Shard The Weights

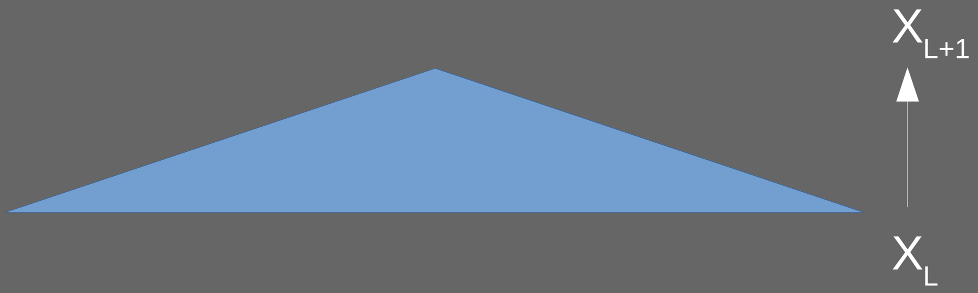


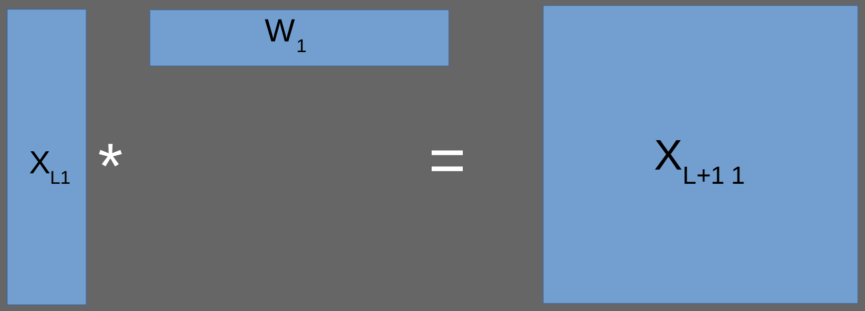
Output Layer Larger Than Input Layer?
allGather* Input Layer Data Then SGEMM



*Using custom 2d allGather code, not NCCL/MPI

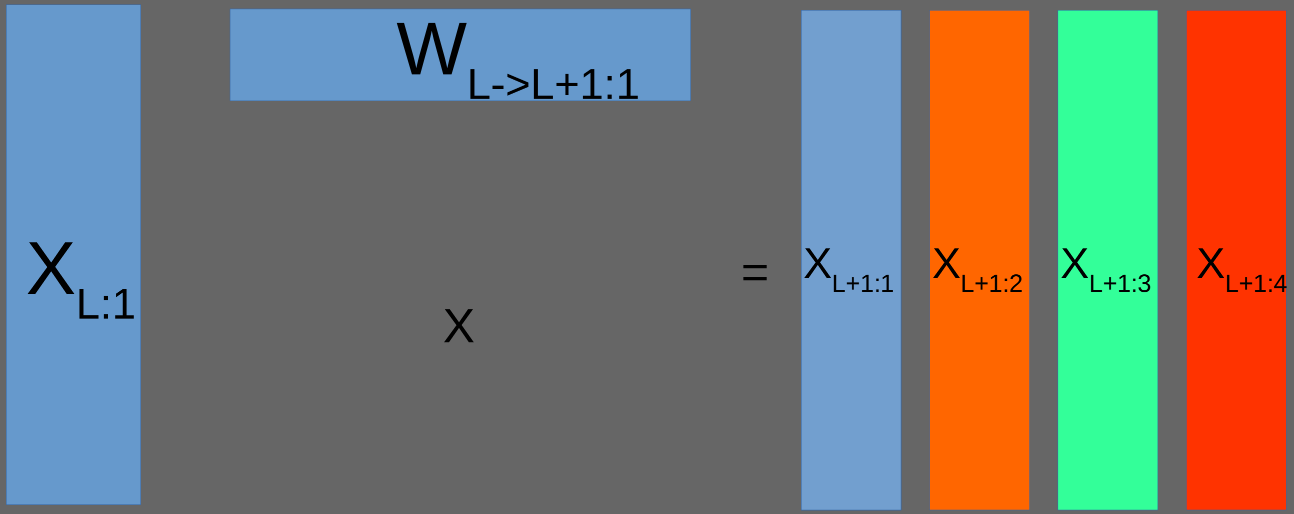
Input Layer Larger Than Output Layer? SGEMM Then Reduce Outputs*

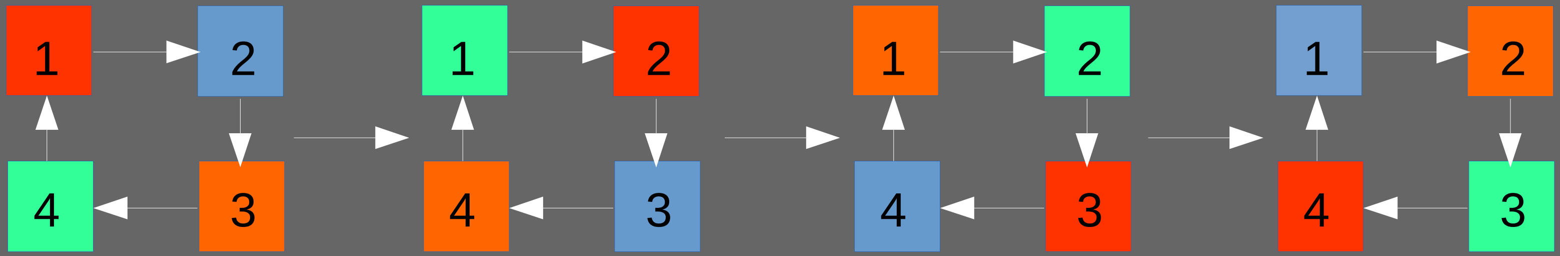




*Using custom 2D partial reduction code which is also O(n)

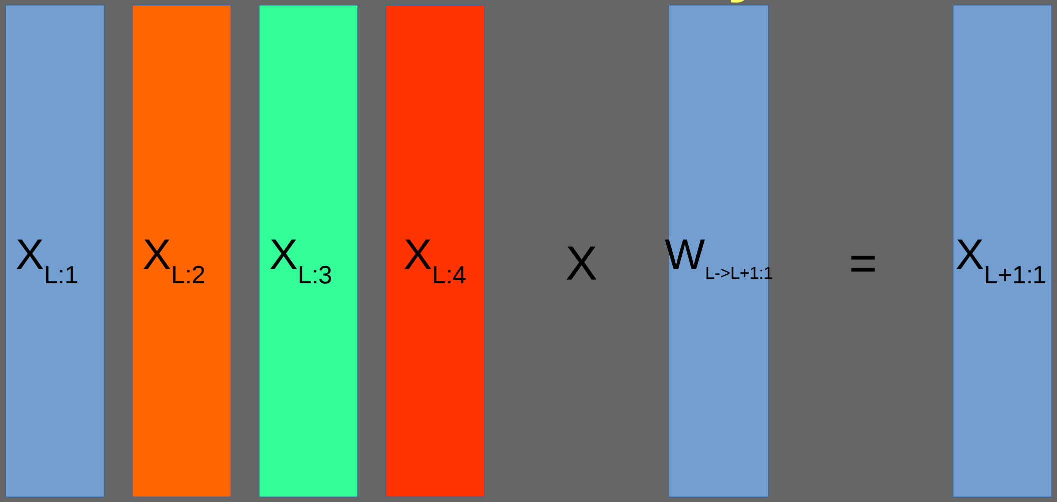
We can even mix math and communication

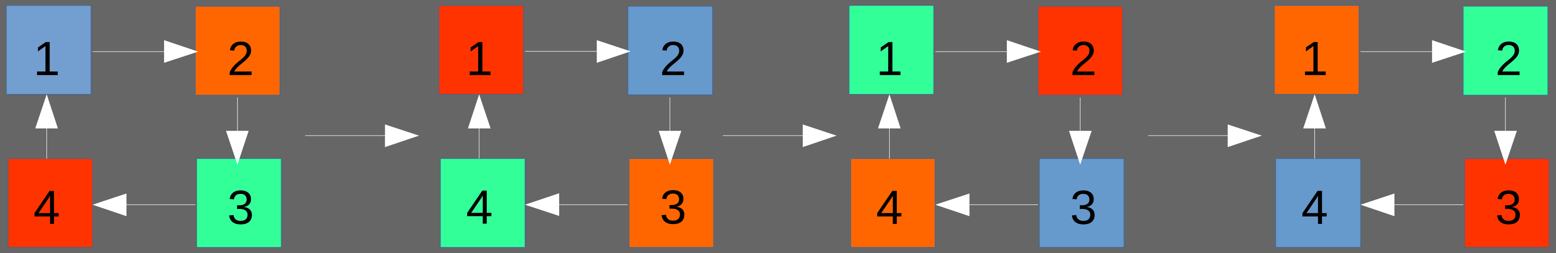




* Reduce the outputs over N-1 communication steps if the model outputs are smaller than the model inputs

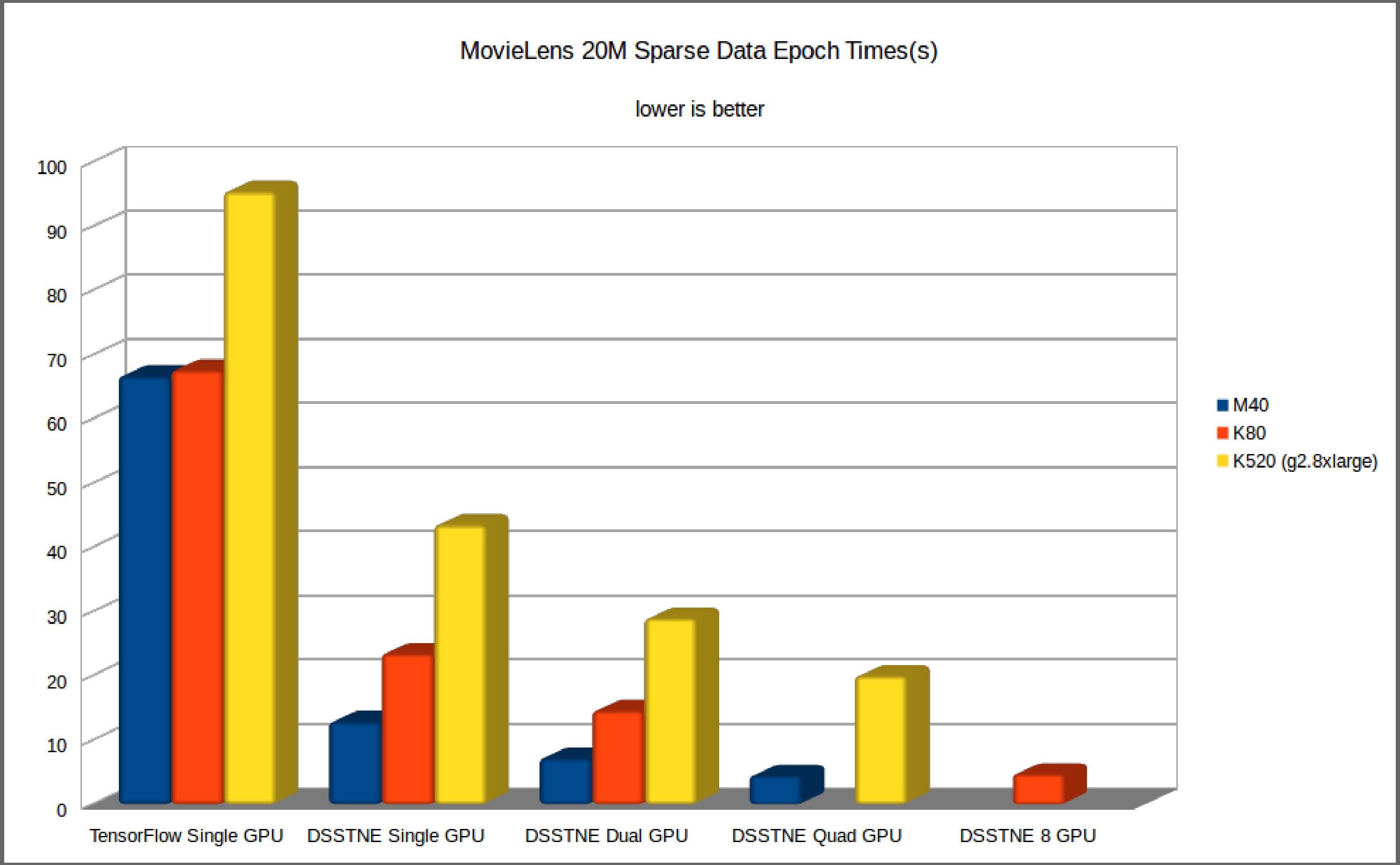
Both ways





*Scatter the inputs over N-1 communication steps if the model inputs
are smaller than the model outputs

How Well Does it Work?



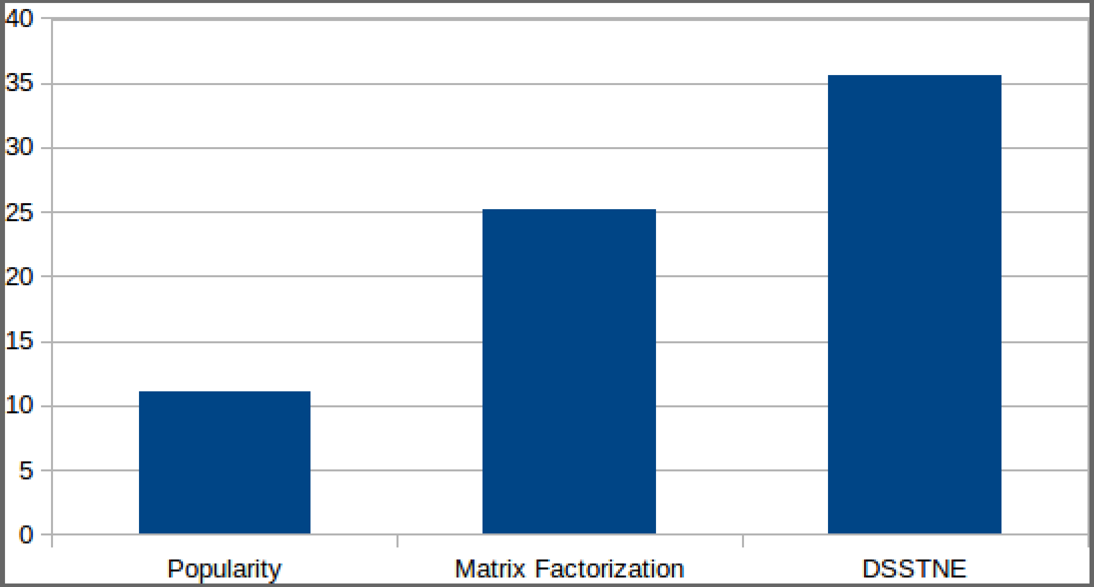
Yes Yes But How Good Are The Recommendations?

- This is a strange question IMO
- DSSTNE runs the same mathematics as everyone else
- Amazon OSSed the framework, not the actual networks, and definitely not how they prepare customer purchase histories
- So for a surrogate, let's use the binary prediction of a random 80/20 split of the MovieLens 20M dataset
- Competing numbers provided by Saul Vargas

MovieLens 20M DSSTNE

```
{
    "Version" : 0.8,
    "Name" : "AIV NNC",
    "Kind" : "FeedForward",

    "ShuffleIndices" : false,


    "ScaledMarginalCrossEntropy" : {
        "oneTarget" : 1.0,
        "zeroTarget" : 0.0,
        "oneScale" : 1.0,
        "zeroScale" : 1.0
    },
    "Layers" : [
        { "Name" : "Input", "Kind" : "Input", "N" : "auto", "DataSet" : "input", "Sparse" : true },
        { "Name" : "Hidden1", "Kind" : "Hidden", "Type" : "FullyConnected", "N" : 1536, "Activation" : "Relu", "Sparse" : false, "pDropout" : 0.37, "WeightInit" : { "Scheme" : "Gaussian", "Scale" : 0.01 } },
        { "Name" : "Hidden2", "Kind" : "Hidden", "Type" : "FullyConnected", "N" : 1536, "Activation" : "Relu", "Sparse" : false, "pDropout" : 0.37, "WeightInit" : { "Scheme" : "Gaussian", "Scale" : 0.01 } },
        { "Name" : "Hidden3", "Kind" : "Hidden", "Type" : "FullyConnected", "N" : 1536, "Activation" : "Relu", "Sparse" : false, "pDropout" : 0.37, "WeightInit" : { "Scheme" : "Gaussian", "Scale" : 0.01 } },
        { "Name" : "Output", "Kind" : "Output", "Type" : "FullyConnected",  "DataSet" : "output", "N" : "auto", "Activation" : "Sigmoid", "Sparse" : true , "WeightInit" : { "Scheme" : "Gaussian", "Scale" : 0.01, "Bias" : -10.2 }}
    ],

    "ErrorFunction" : "ScaledMarginalCrossEntropy"
}
```

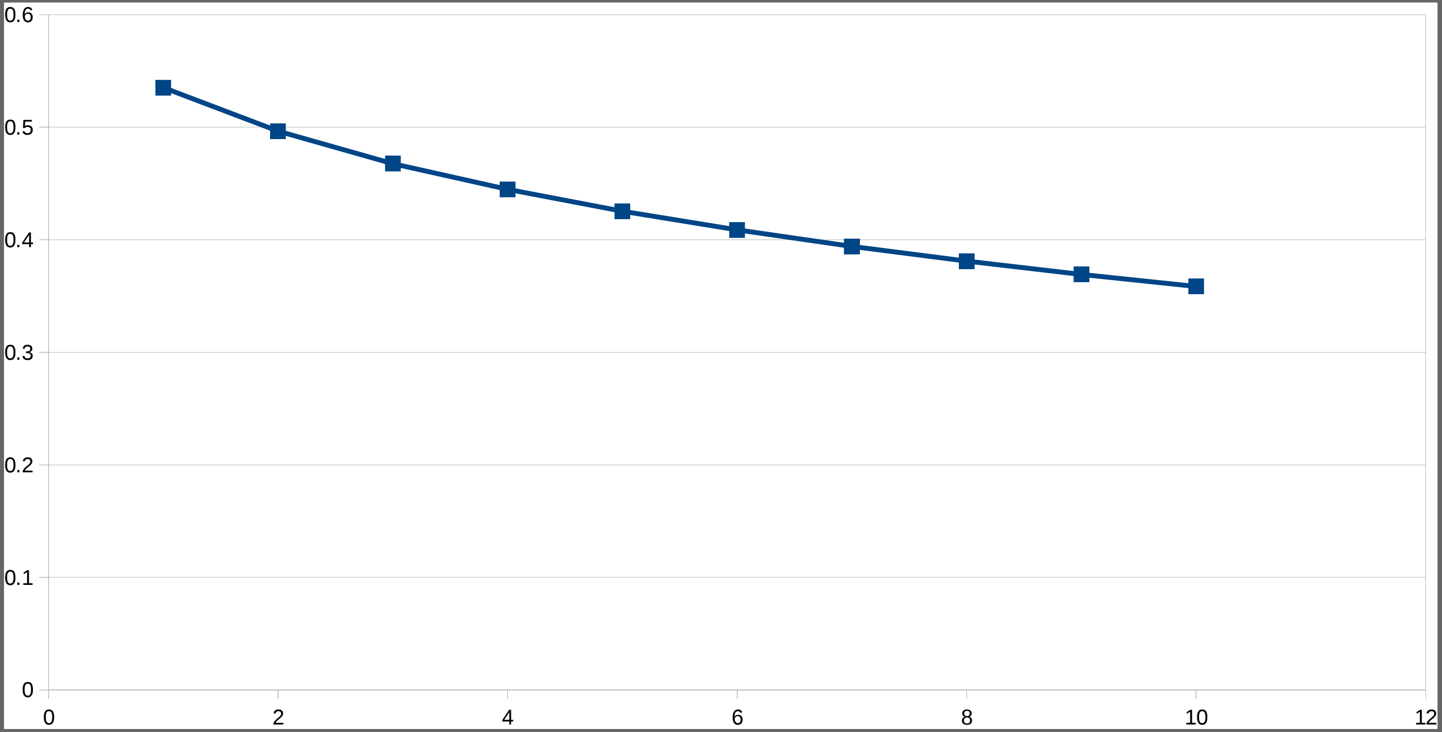
MovieLens 20M P@10



https://github.com/RankSys/RankSys

Raw Performance

P@K

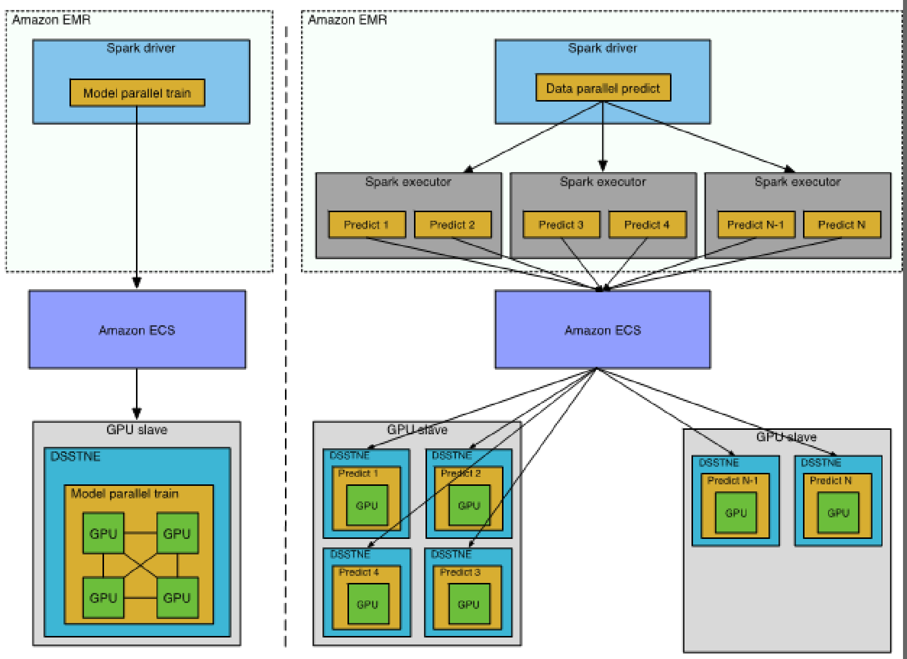


K

AWS Recommendations at Scale

- AWS released a blog by Kiuk Chung on how to perform product recommendations with DSSTNE (and other frameworks BTW) and Spark to experiment with deep learning for product recommendations at scale
- This is the Amazon Deep Learning Recommendations System minus the secret sauce networks, hyperparameters, and private customer data

Recommendations At Scale: How Do They Work?



http://blogs.aws.amazon.com/bigdata/post/TxGEL8IJ0CAXTK/Generating-Recommendations-at-Amazon-Scale-with-Apache-Spark-and-Amazon-DSSTNE

DSSTNE At Scale Summary

- Use SPARK to set up and launch training and inference tasks
- DSSTNE automagically scales each task based on process count
- And that's all folks...

Deep Learning For The 99%

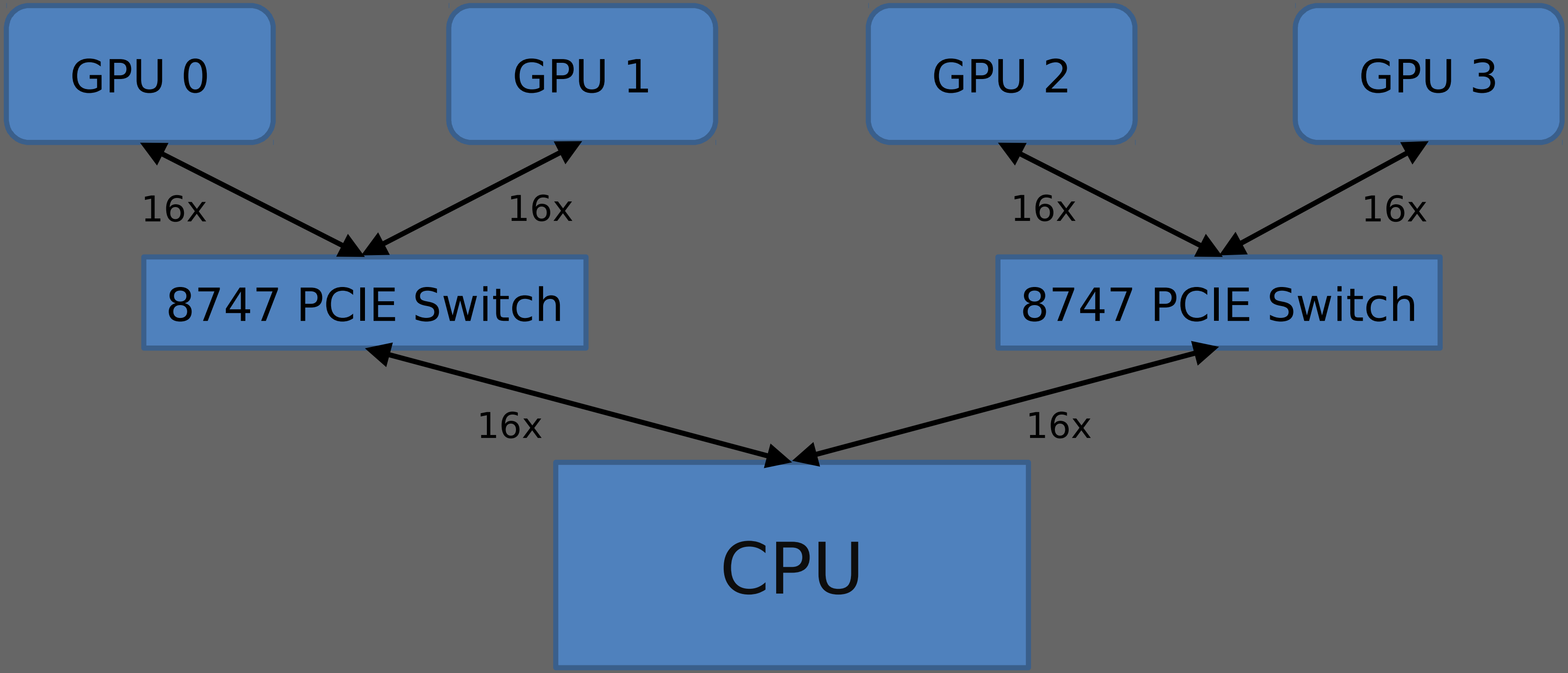
“You have to spend at least $24,000(US) to be able to run experiments on Atari or ImageNet competitively.” - Nando de Freitas

P13N at Amazon

Developed DSSTNE and changed the way Amazon does recommendations with:

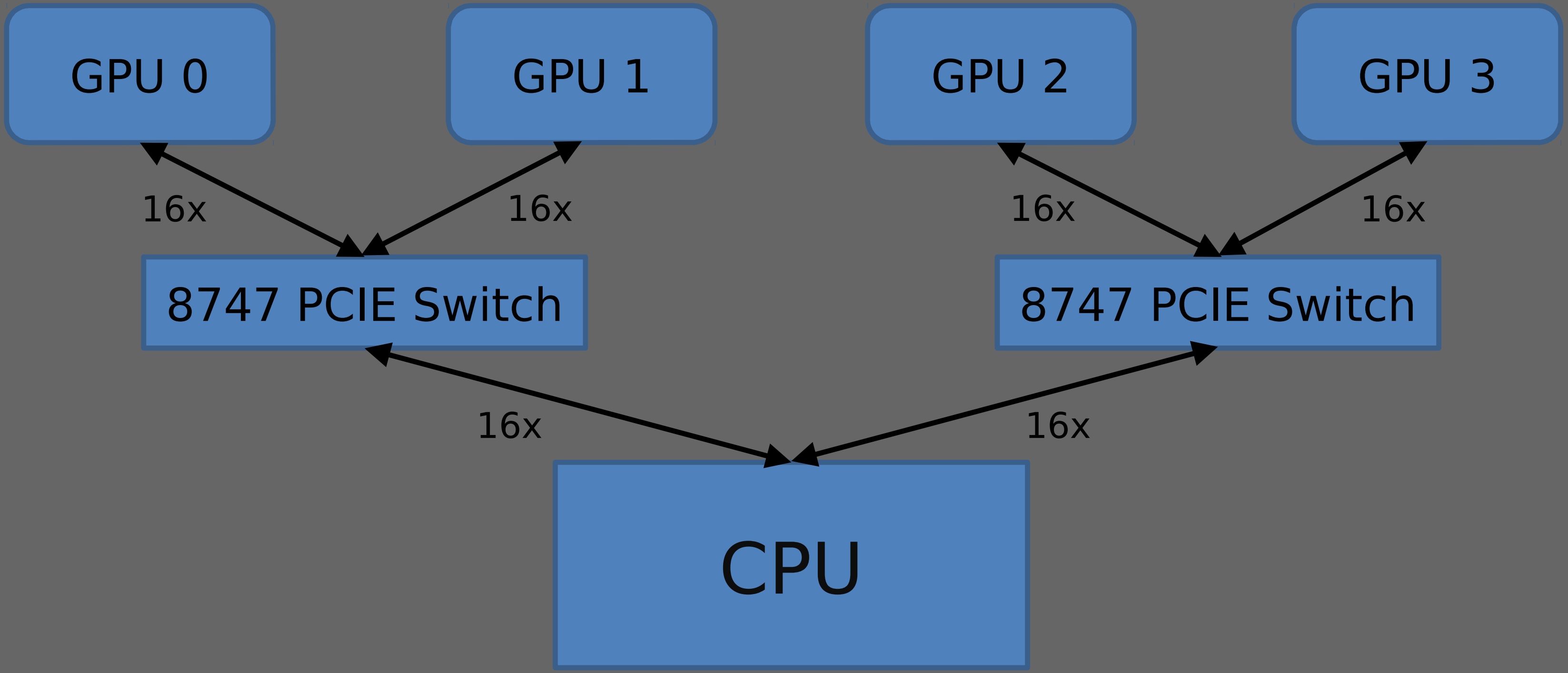
- A $3000 GTX 880M Laptop
- A $3000 GTX 980M Laptop
- A $2000 Desktop with two $1000 GTX TitanX GPUs
- A bunch of AWS time on GK104 GPUs from 2012

The AMBERnator (2013)*



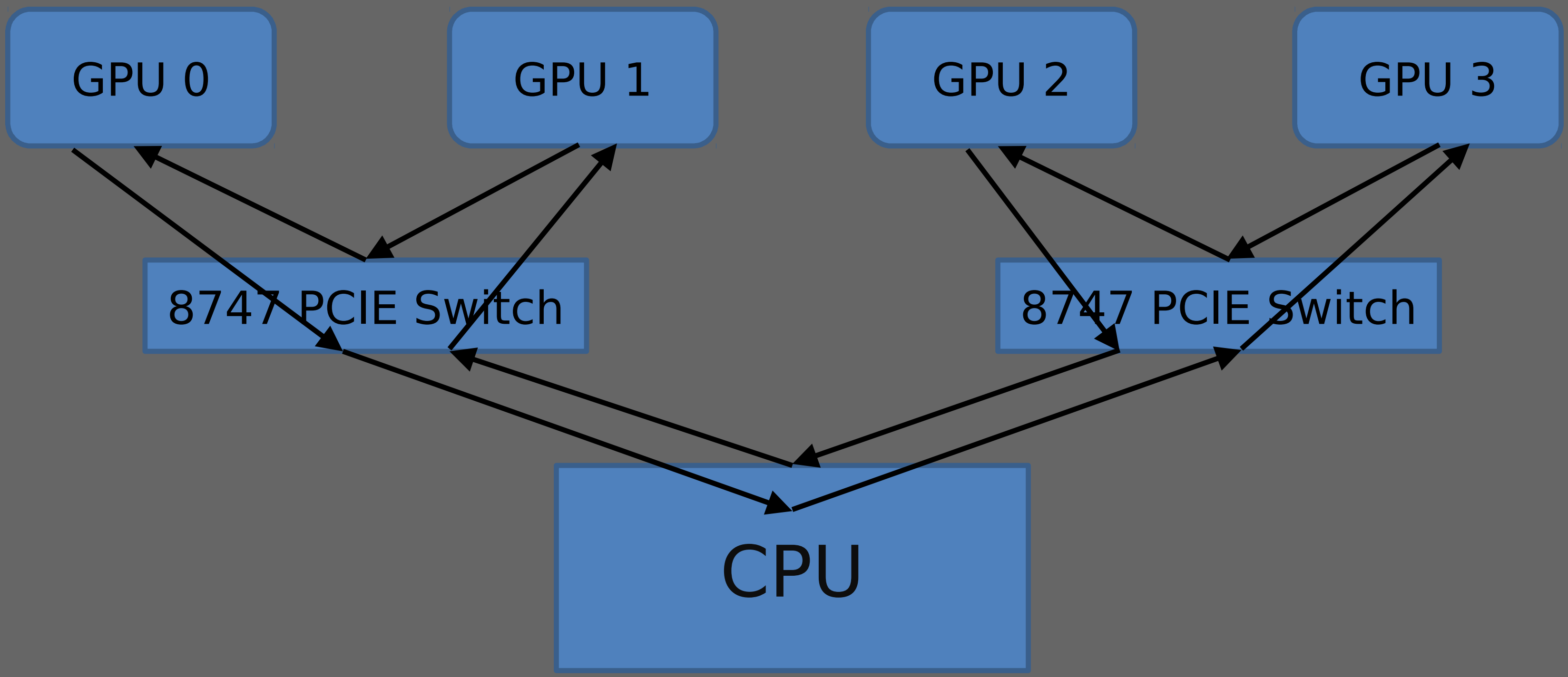
*You'll need about $7000 and a couple hours to build this

Digits Dev Box (2015)*

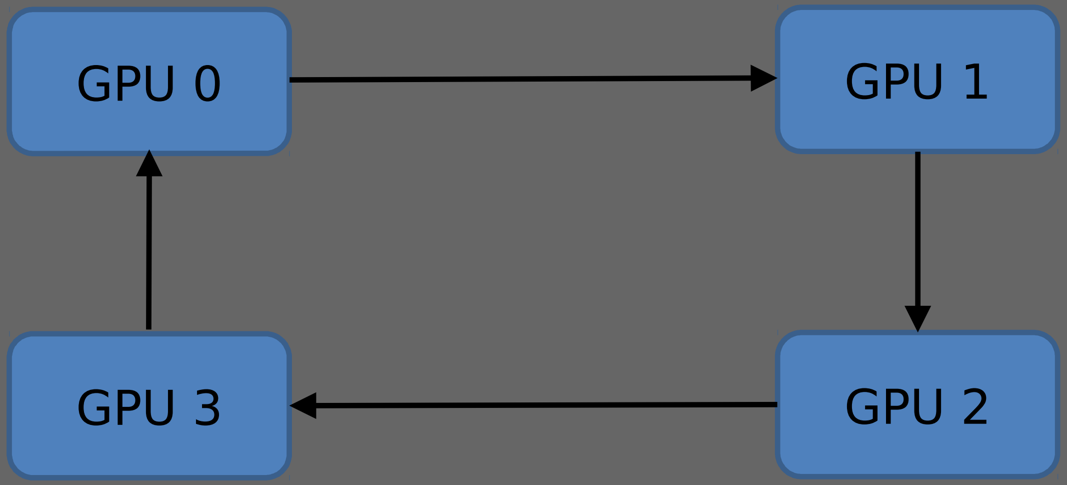


*But you'll need $15,000 for this. Maybe you can tell me what justifies the extra $8000?

P2P Ring Implementation



P2P Ring Simplified



O(D) Collectives on Closed Ring

•AllReduce:	$2D * (N – 1) / N$
•Reduce:	$D * (N – 1) / N$
•Gather:	$D * (N – 1) / N$
•AllGather:	$D * (N – 1) / N$

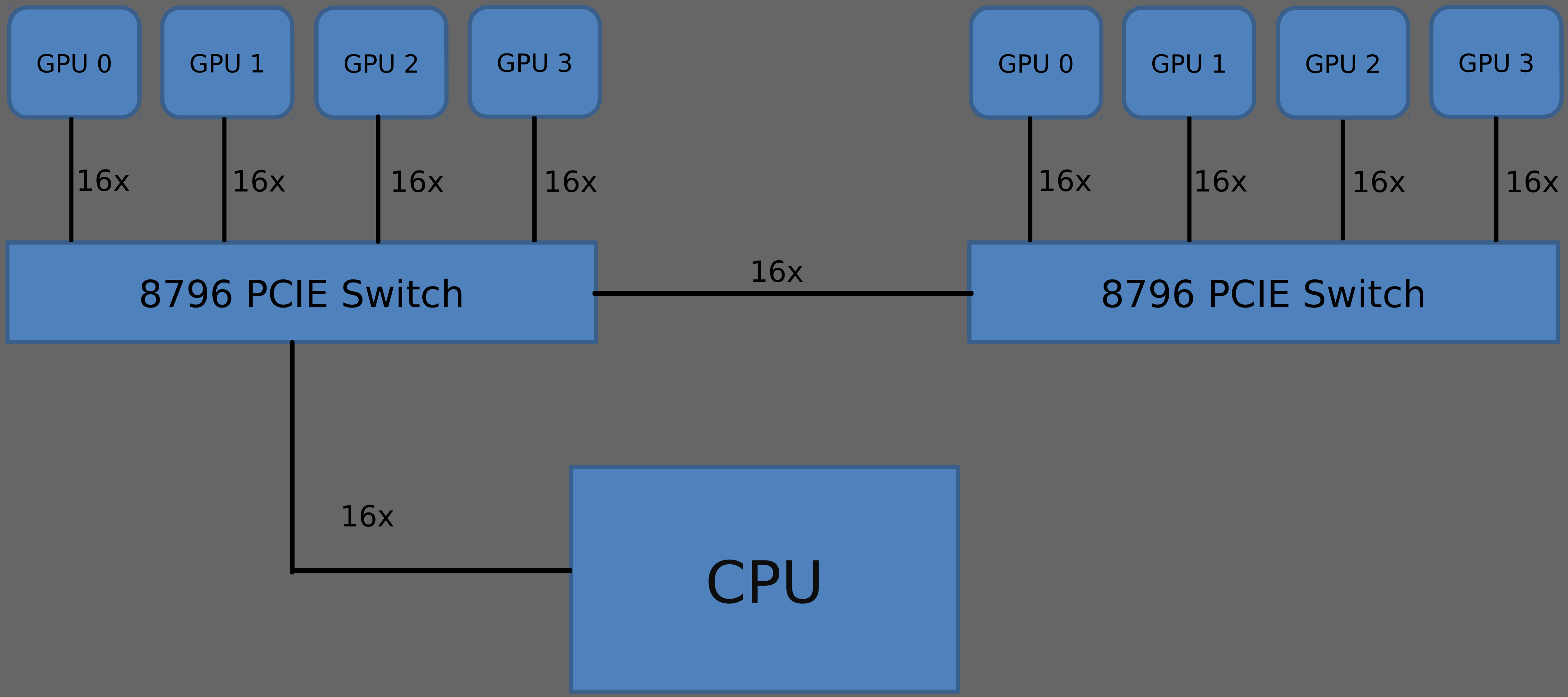
D: Bytes of data to collect/reduce

N: Number of processors in use

Total Cost: $15^H^H7,000 or less

- Asus P9X79-E WS MB ($500) plus Intel Core-i7 4820 (Ivybridge) CPU ($320)
- Asus X99-E WS MB ($520) plus Intel Core-i7 5930K (Haswell) CPU ($560)
- 4 Titan XP GPUs ($4,800)
- 44 TFLOPs for $7,000! (<<$24,000)
- NVIDIA prefers for you to pay $15,000

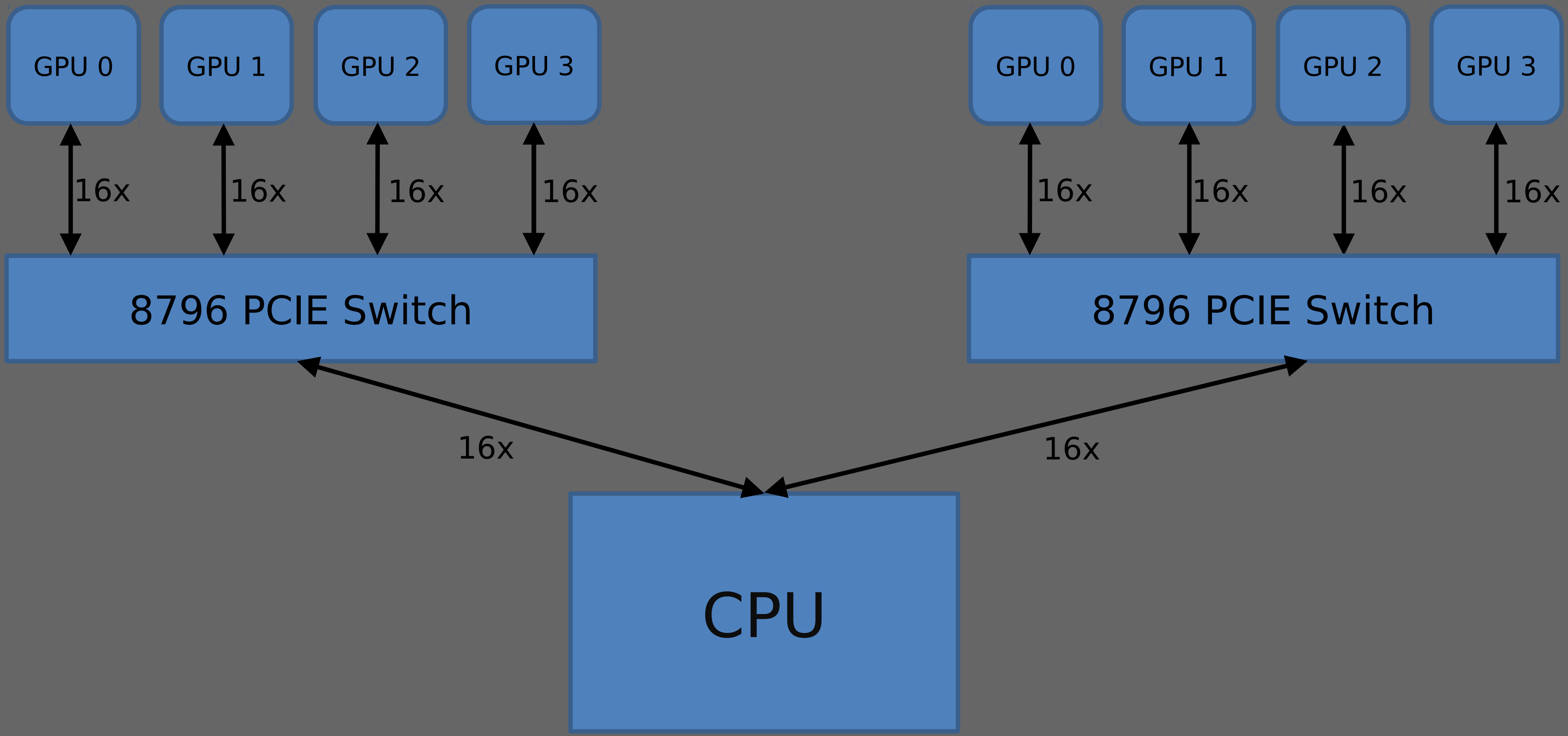
What if $24K is Burning A Hole in Your Pocket?



25 GB/s of Bidirectional P2P Bandwidth

	0	1	2	3	4	5	6	7
0	NA	24.97	24.96	24.95	24.95	24.95	24.96	24.95
1	24.97	NA	24.97	24.96	24.96	24.95	24.95	24.96
2	24.97	24.95	NA	24.95	24.96	24.96	24.95	24.95
3	24.95	24.95	24.95	NA	24.94	24.96	24.96	24.96
4	24.95	24.95	24.95	24.95	NA	24.94	24.95	24.94
5	24.95	24.95	24.94	24.94	24.95	NA	24.94	24.95
6	24.95	24.95	24.95	24.94	24.94	24.94	NA	24.95
7	24.94	24.94	24.95	24.94	24.95	24.95	24.96	NA

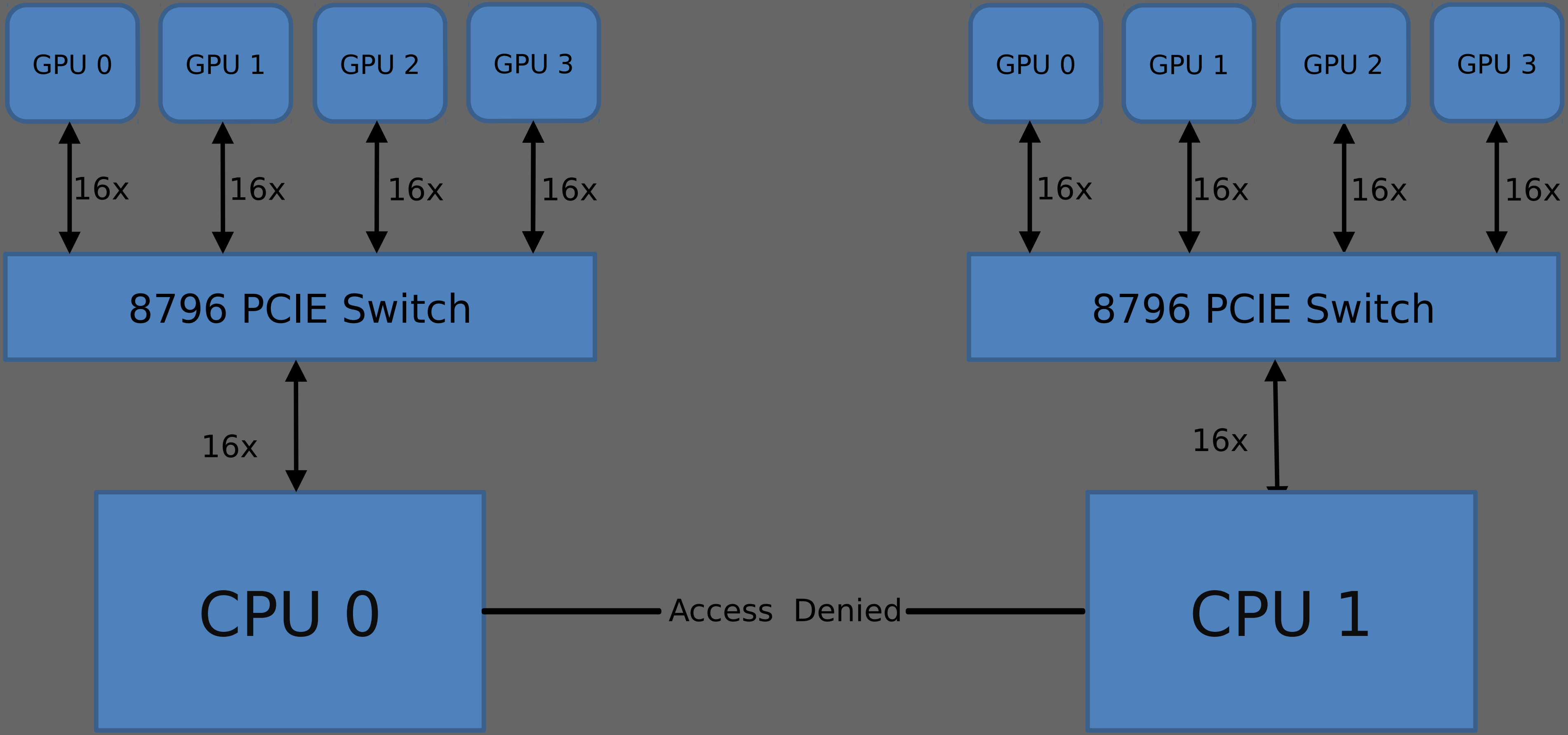
Sweat The Details



Craptastic Bandwidth

	0	1	2	3	4	5	6	7
0	NA	25.03	25.02	25.01	15.97	15.97	14.73	15.97
1	25.03	NA	25.04	25.02	15.96	15.97	14.73	15.97
2	25.02	25.04	NA	25.02	15.97	15.96	14.73	15.96
3	25.02	25.03	25.02	NA	14.69	14.69	14.7	14.69
4	15.98	15.98	15.99	14.73	NA	25.02	25.04	25.03
5	15.98	15.98	15.98	14.73	25.03	NA	25.02	25.03
6	14.69	14.7	14.69	14.7	25.03	25.02	NA	25.03
7	15.98	15.97	15.98	14.73	25.04	25.04	25.03	NA

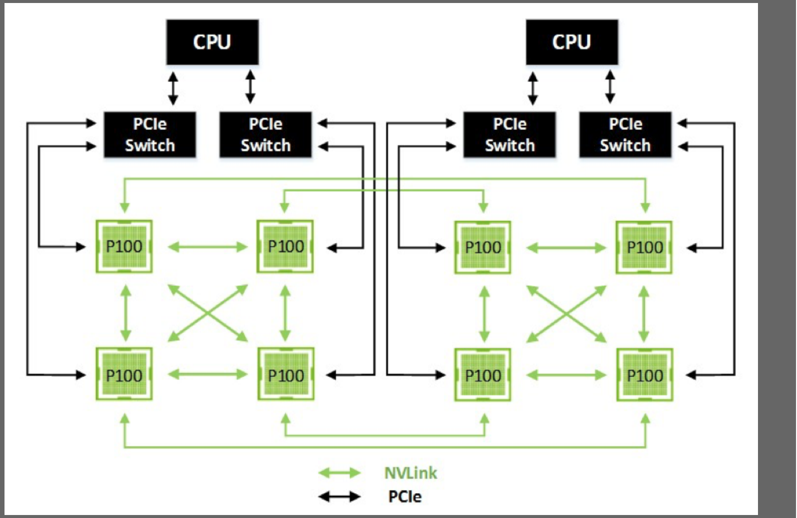
Watch Out For This



Or Do You Need A $149K DGX-1? TLDR: No

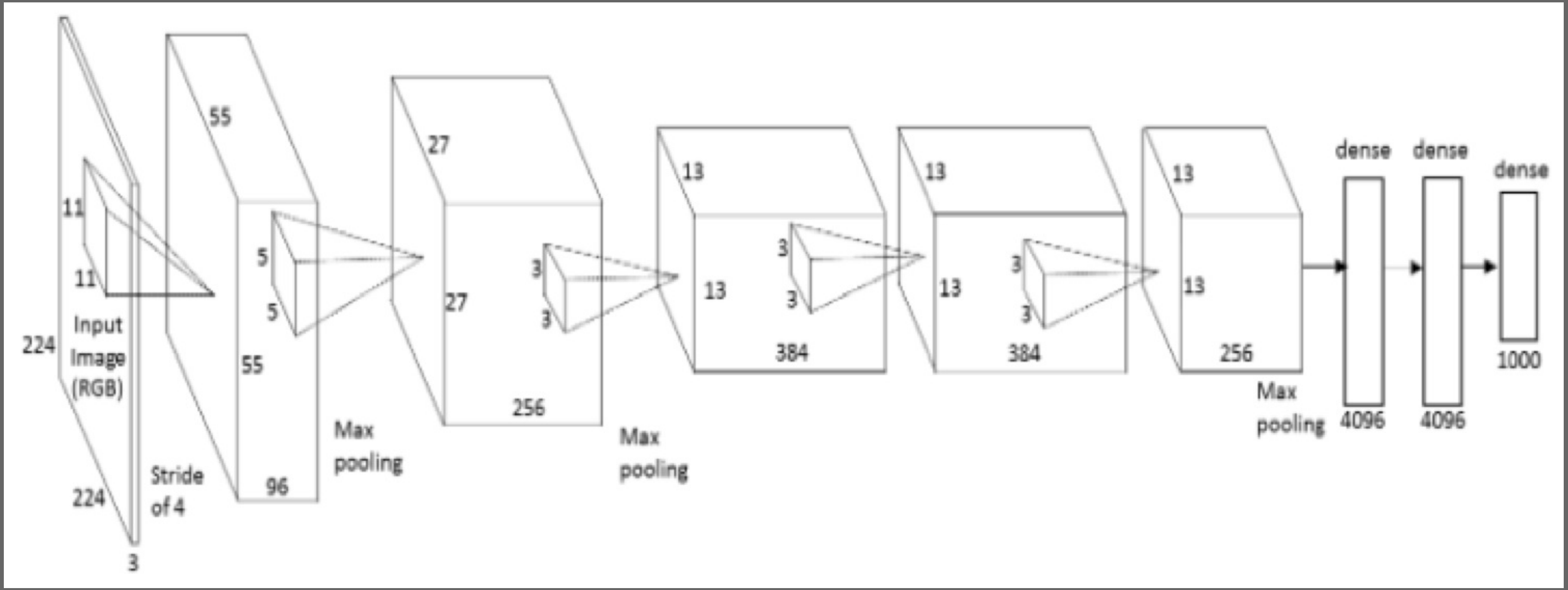
- 85 TFLOPS FP32 (~10.6 TFLOPS per GPU) no FP16 for now
- ~64 GB/s connected in a cube (N == 8)*

Reduction:	D * (N – 1) / N
Gather:	D * (N – 1) / N
AllReduce:	D * 2 * (N – 1) / N



But is your deep neural network really communication-limited?

AlexNet



224
224
3
11
11
Input
Image
(RGB)
Stride
of 4
55
55
96
5
5
Max
pooling
27
27
256
3
3
Max
pooling
13
13
384
3
3
13
13
384
3
3
13
13
256
Max
pooling
dense
4096
dense
4096
dense
1000

Are you running data-parallel?

- AlexNet has ~61M parameters
- We'll assume a total batch size of 128 (16 images per GPU)
- 8 * 16 images/GPU trains in 14.56 ms* on GTX Titan XP
- On DGX-1, AllReducing 61M (244 MB) parameters at ~64 GB/s is ~6.7 ms (buried 5.5 ms of backprop for overlapping copy and compute) for a final result of 1.2 ms.
- Using ~12.5 GB/s P2P, this would take ~34 ms

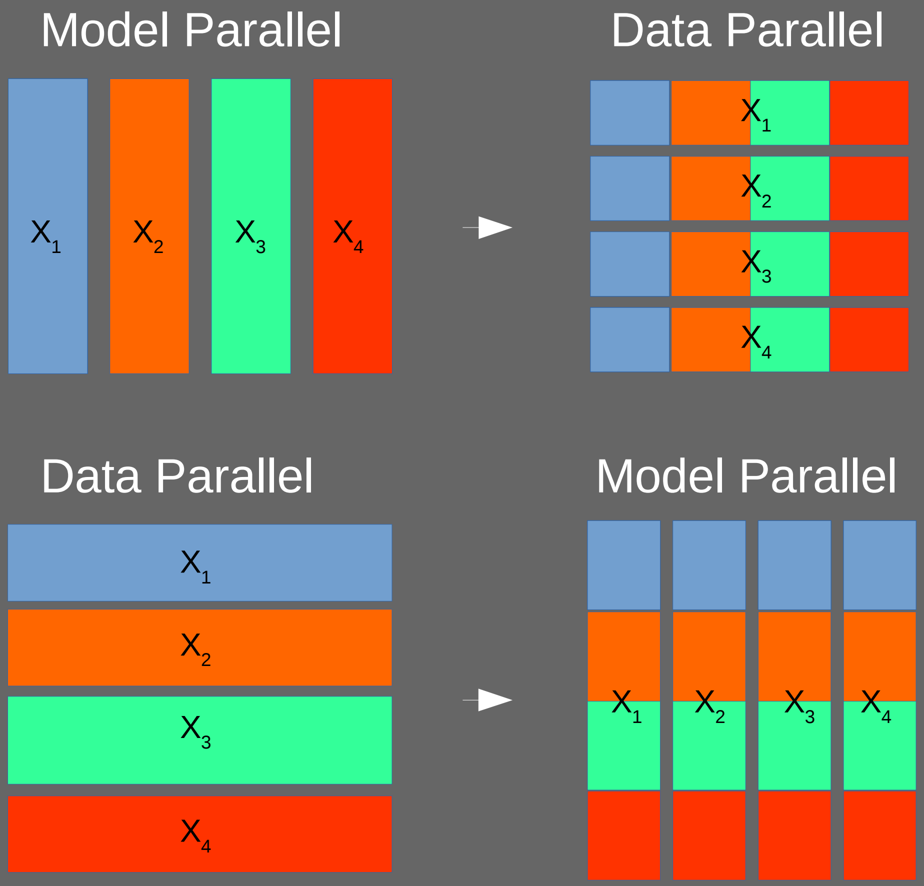
*https://github.com/jcjohnson/cnn-benchmarks

Alex Krizhevsky* to the Rescue! (or should you also run model-parallel?)

- Of AlexNet's ~61M parameters. ~4.3M are convolutional (data-parallel) and ~56.7M of which are fully-connected (model-parallel)
- Fully connected layers at a batch size of 128 is ~1.7M neurons
- P2P allReduce of 4.3M parameters takes ~2.4 ms
- P2P gather/reduction of 1.7M neurons is ~0.5 ms
- 2.9 ms is << 14.56 ms so once again it's effectively free(tm)
- It's also faster than NVLINK data-parallel…
- NVLINK model-parallel would of course win here but it doesn't exist...
- Similar arguments apply to INT16/INT8/FP16 and all the other greasy kids stuff

*https://arxiv.org/abs/1404.5997

One O(D/N) Weird(er) Trick



Model Parallel

X_1 X_2 X_3 X_4

Data Parallel

X_1

X_2

X_3

X_4

Data Parallel

X_1

X_2

X_3

X_4

Model Parallel

X_1 X_2 X_3 X_4

Implementation Matters (unless you're rich)

- Naive implementations of data-collectives are O(D log N) and/or reduce weight gradients when there's a better way(tm)
- Smart implementations are both O(D) and minimize communication costs
- GPUs connected to different CPUs cannot talk directly to each other because Intel says so
- Just say no to Xeon Phi (but that's another talk)
- Keep an eye on AMD's Vega GPUs
- You don't need **at least** $24K, you need **at most** $24K (and the right code)

Should You Embrace DSSTNE?

- Do you have a sparse data set? Yes
- Do you want to experiment with large models? Yes
- Do you want to run across multiple machines/GPUs? Yes
- Do you want to scale up conv nets? Soon
- Do you want to run Keras faster? Not yet
- Do you want to run TensorFlow faster? Not yet

DSSTNE RoadMap

- •Working on One Weird(er) Trick and General RNN (FoldFlow) support
- •Compile DSSTNE under Radeon Open Compute (ROC) and truly fulfill the promise of “CUDA Everywhere(tm)”*
- •Rather than port DSSTNE to OpenCL, AMD is porting CUDA to their GPUs to get instant access to all existing OSS CUDA applications
- •Provide a Python API through Python extensions (section 2.7.12)
- •Keras/TensorFlow (XLA) import
- •Automagic data streaming of models and data on SM 6.x and up

*https://github.com/RadeonOpenCompute

Summary

- DSSTNE's automagic model-parallel training is a big win
- DSSTNE's efficient sparse data processing is a big win
- DSSTNE required bespoke GPU code rather than CUDA libraries
- AWS has released all the tools for using DSSTNE for recommendations at scale
- Torch and Caffe have recently improved their sparse data support
- A lot of work-in-progress

Acknowledgments (DSSTNE/Amazon/AWS)

Rejith Joseph George
Kiuk Chung
Tristan Penman
Oleg Rybakov
Avishkar Misra
Jane You
Shruti Kamath
Mitchell Goodman
Cesar Romero
Jeff Bezos
Leo Dirac
Sebastian Gunningham
Srikanth Thirumulai
Henri Yandell
Matias Benitez

Acknowledgments (NVIDIA)

Jonathan Bentz

Mark Berger

Jerry Chen

Kate Clark

Simon Layton

Duncan Poole

Sarah Tariq

Acknowledgments (AMD/ROC)

Greg Stoner

Ben Sander

Michael Mantor