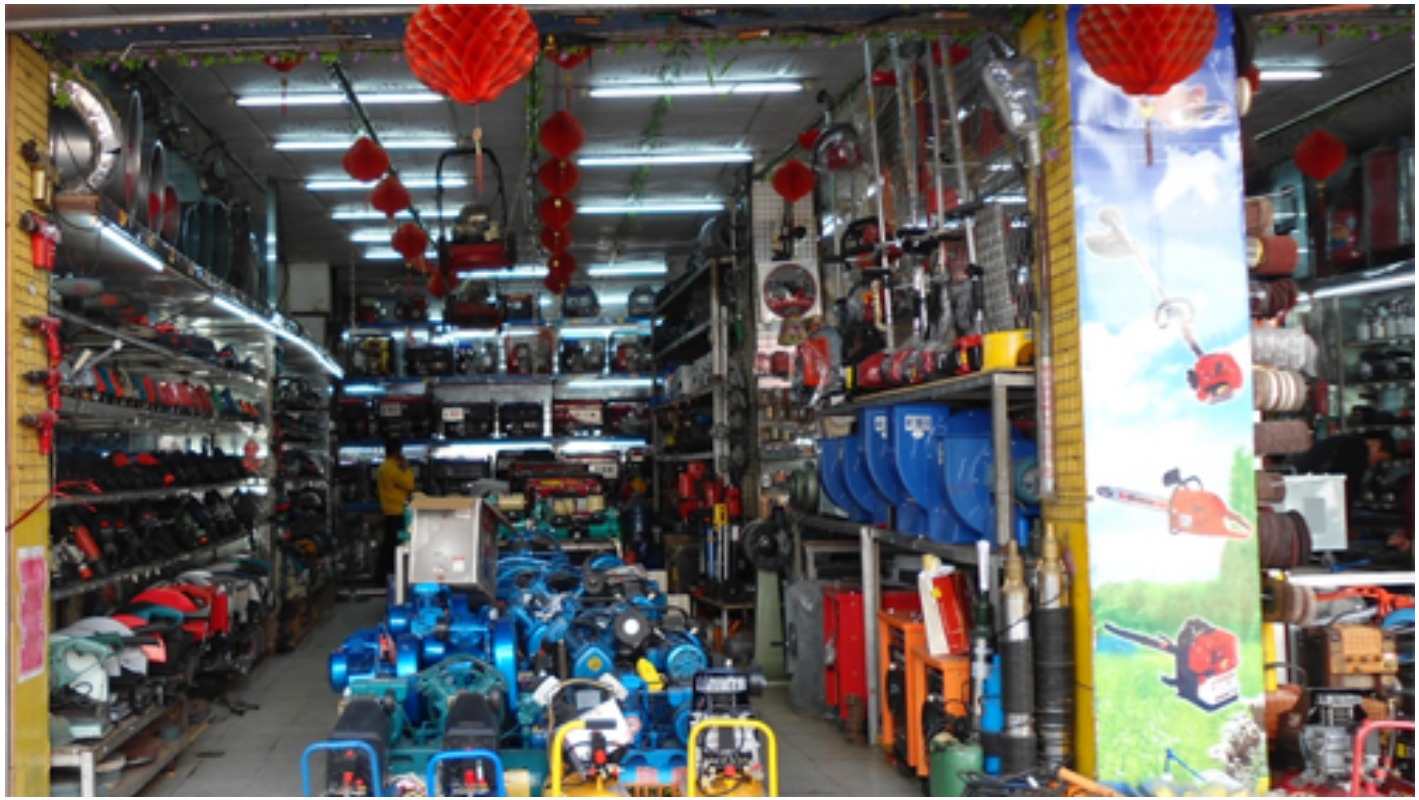
Deep Learning (hopefully faster)

Adam Coates

Silicon Valley AI Lab

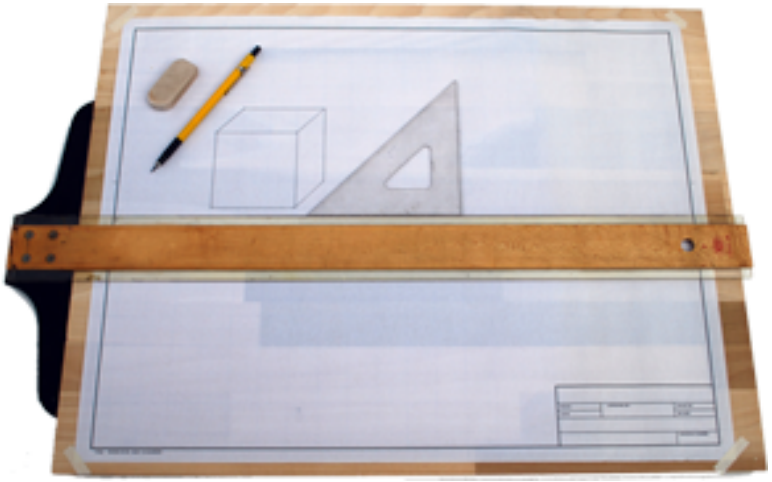
Scope

- AI and Deep Learning depend heavily on systems for training and deployment.
  - Many many tools to solve systems problems.



Scope

- Focus on making models train faster.
  - Huge topic! Best to see a ton of ideas over time.
- This talk: conceptual tools to help DL practitioner strategize and decide what to do next.

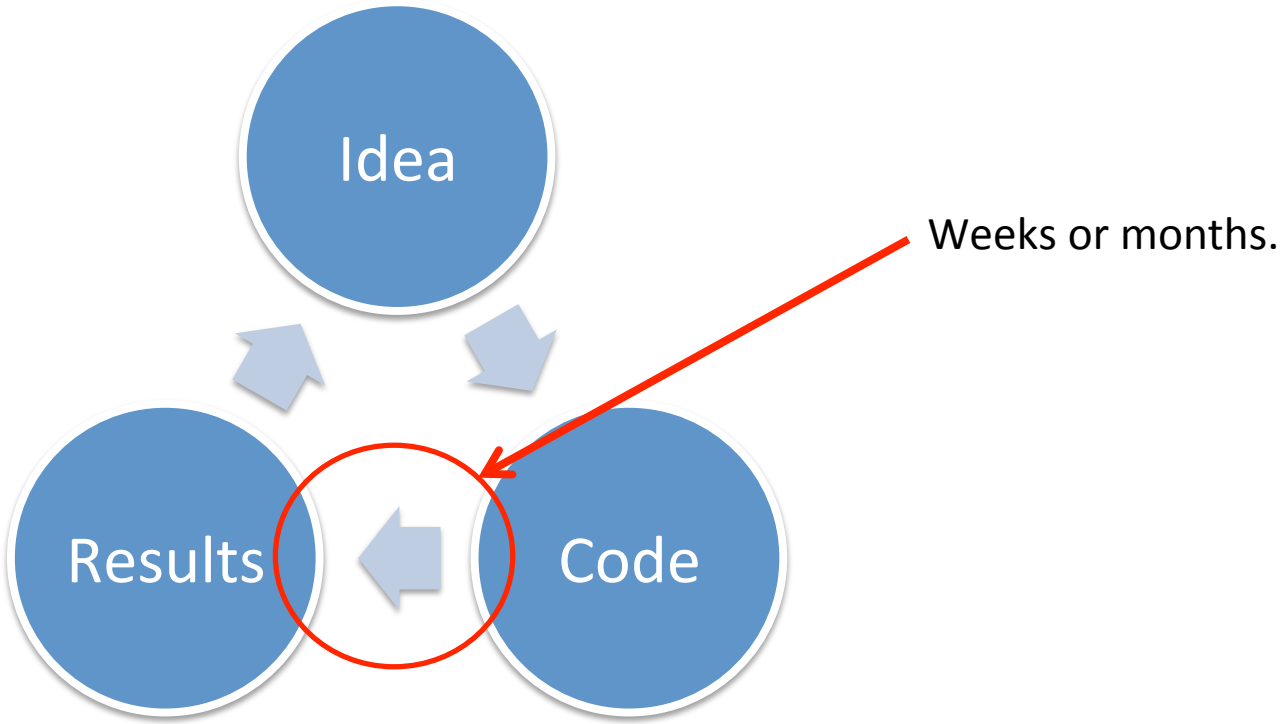


Overview

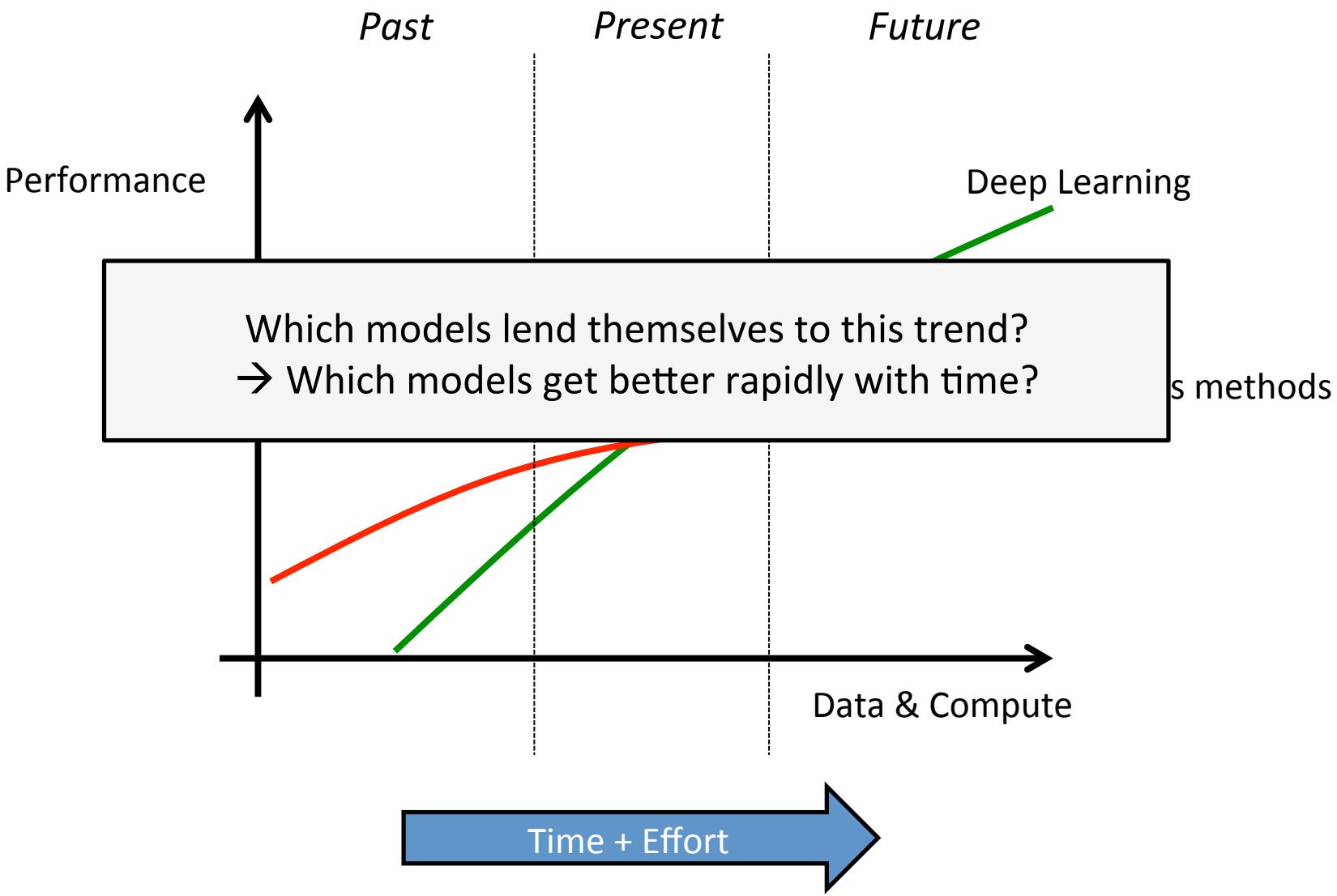
- Basic motivations & approach
- Single node: 1 GPU or 1 CPU.
- Multiple nodes.

Cycle time argument

- DL / ML research involves guided exploration.
  - We want shorter overall experiment time (wall time) so that we can make faster research progress!



Scaling argument

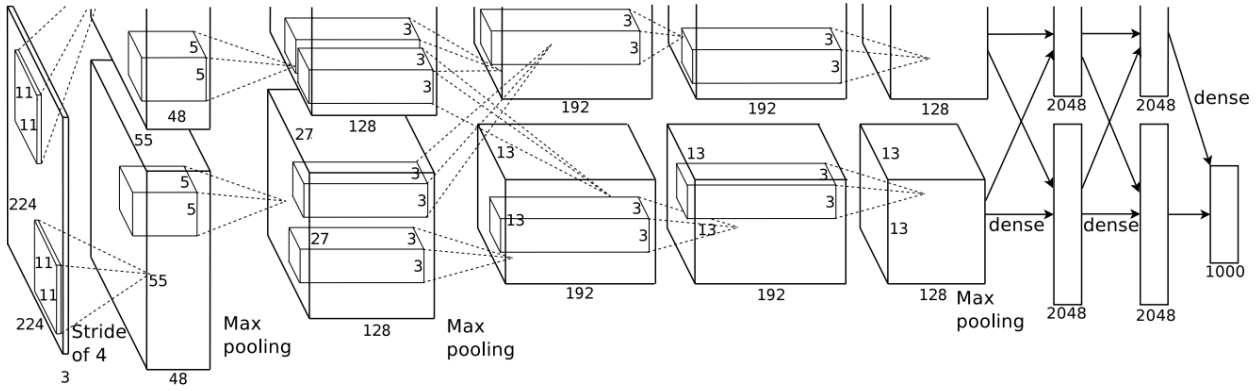


Approach

- Several ways to try to make system faster.
  - Change the software.
  - Change the hardware.
  - Change the model / algorithm.
    - Hard to chase systems+accuracy at once.
  - We'll talk about performance modeling: Basic idea applicable to all of these decision processes.
    - We'll work some examples.

Workload

- Most DL workloads built on common operations:



Convolution with small filters: r = conv(filters, data)
Point-wise nonlinearities: r = max(0, z)
Dense linear/affine operations: r = A*z + b
Reductions: Z = sum(p)
…

Workload

- Given *fixed* problem size, we will work on maximizing *throughput*.
  - Rate at which operations are completed.

Throughput = (#operations) / (running time)

If #operations is a constant → Same as minimizing running time.

Caveat

- Throughput doesn't consider convergence time.
  - Convergence depends on hyperparameters, etc., not systems.
- If you're trying to make changes to model or hyperparameters, beware:
  - Throughput is gameable.
  - E.g., Minibatching:
    - Bigger minibatch = higher throughput!
    - But not always best wall time for whole experiment.

SINGLE NODE PRINCIPLES

Setting goals

- While thinking through speed and systems issues, best question to keep asking:
  - How much could be gained? (Is it worth it?)

- To answer: need to be able to assess potential gain.
  - Go for biggest, cheapest gains.
  - Keep going until you hit diminishing returns.

The speed of light

- Your baseline is not how slow your current code runs.
  - 10x speedup over slow code would be great.
  - How do you know if you can get 10x?
  - How do you know if there's more to do?

The speed of light

- Baseline is the *fastest your code can ever run.*
  - I.e., maximum potential *throughput*.
  - This is "the speed of light" for your system.
    - 0.5c is pretty good. Potential ~2x speedup left.
    - 0.8c is very good. Only ~1.25x speedup left.
  - Usually costs more effort to go faster if already close to speed of light.
    - Also: could be time to buy more GPUs.

The speed of light

- Your baseline is not how slow your current code runs.
  - Your "baseline" is the *fastest it can ever run.*
    - This is "the speed of light" for your system.

Goal: for single node, quickly estimate speed of light for DL operations.

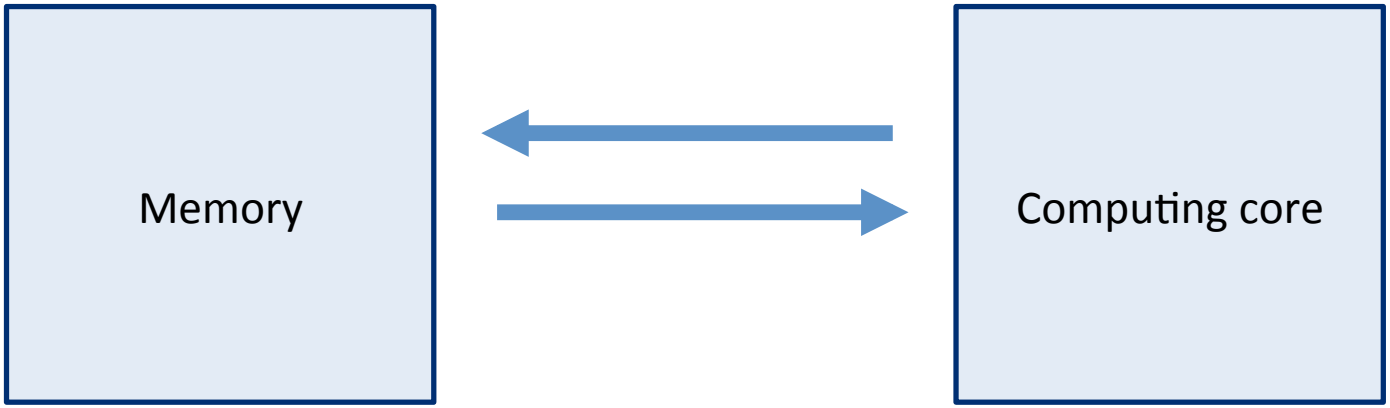
Performance modeling

- Given a fixed computation to perform, how do we estimate maximum potential throughput?
  - Hard to do in general. Modern processors are complicated!

- We'll use a simple scheme that is quick and will give you intuition.

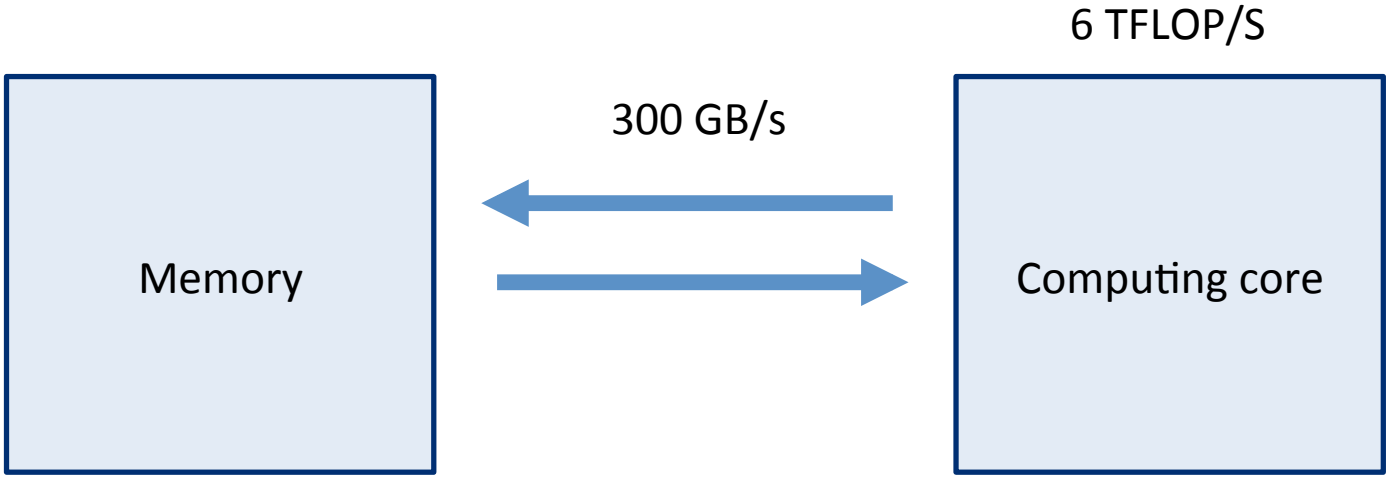
Model of a compute node

- Represent computation and memory only.
  - Only represents two key hardware limitations:
    - Total computation system can perform.
    - Total bandwidth available to memory.



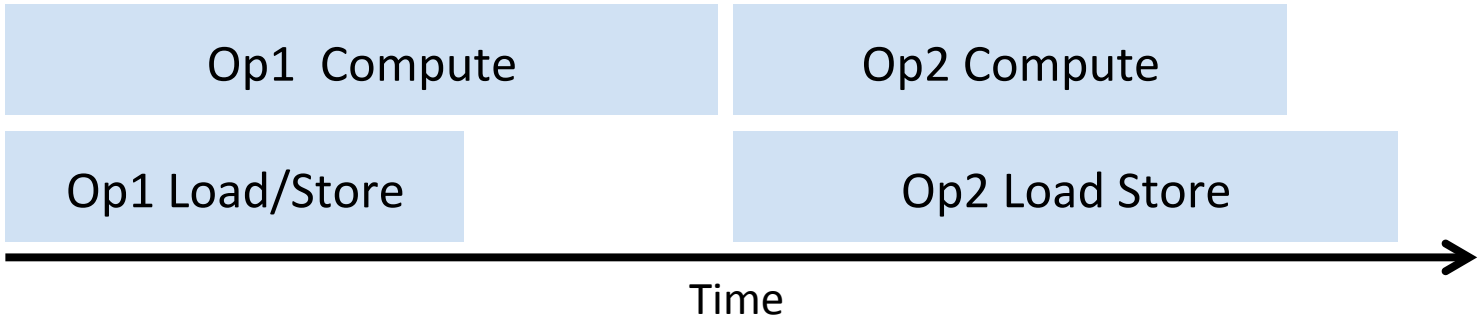
Model of a compute node

- Example: GPU circa 2015
  - Computing limit: ~6 TFLOP/S
  - Memory bandwidth: ~300 GB/s
  - **Key assumption:** we can always stream memory simultaneously with computation.



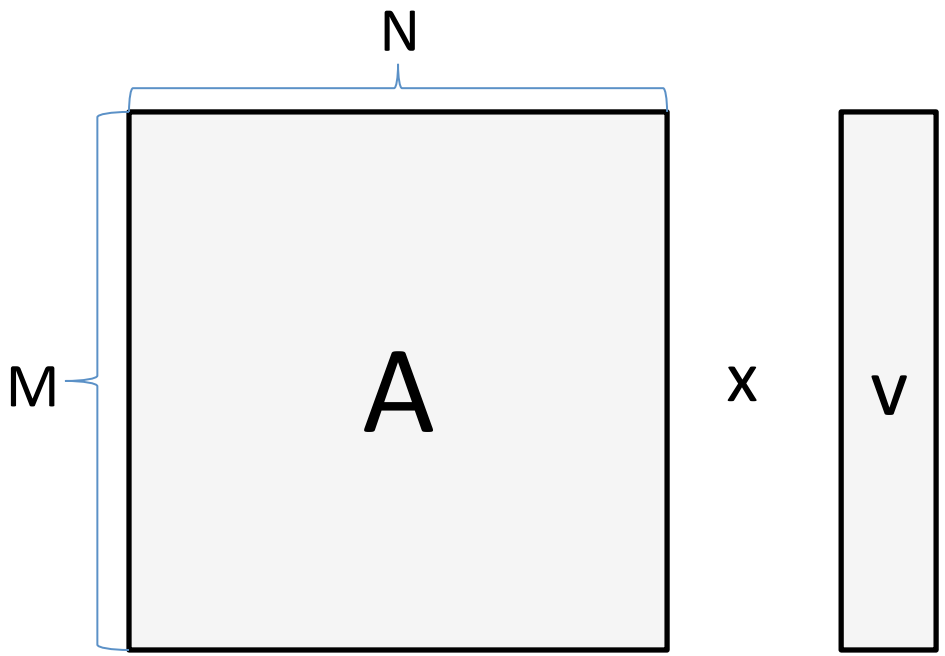
Model of a compute node

- If we run a sequence of operations, timeline might look like:



Example: Matrix-vector multiply

Compute: $A\,v$ for single-precision operands.



How much data do we need to load from memory?	4 bytes $\times$ (MN + N)
How much data do we need to store to memory?	4 bytes $\times$ M
How many FLOPs?	M (2N – 1) $\approx$ 2MN

Example: Matrix-vector multiply

For M=1024 and N=512, what is the best possible throughput (in operations per second)?

Memory: 4 bytes $\times$ (1024$\times$512 + 512 + 1024) = 2.1e6 bytes

FLOPs: 2$\times$1024$\times$512 = 1e6 FLOPs

Running time = max{ 2.1e6 bytes / (300e9 bytes/s),
1e6 FLOPs / (6e12 TFLOP/s) }

= max{ 7us, 0.16us }

Even substantial change in this number is irrelevant.

The *effective throughput* is (1e6 FLOPs / 7us) = **142 GFLOPs**

Arithmetic intensity

- A key quantity related to throughput is the arithmetic "intensity":

Intensity = (# arithmetic ops) / (# bytes to load or store)

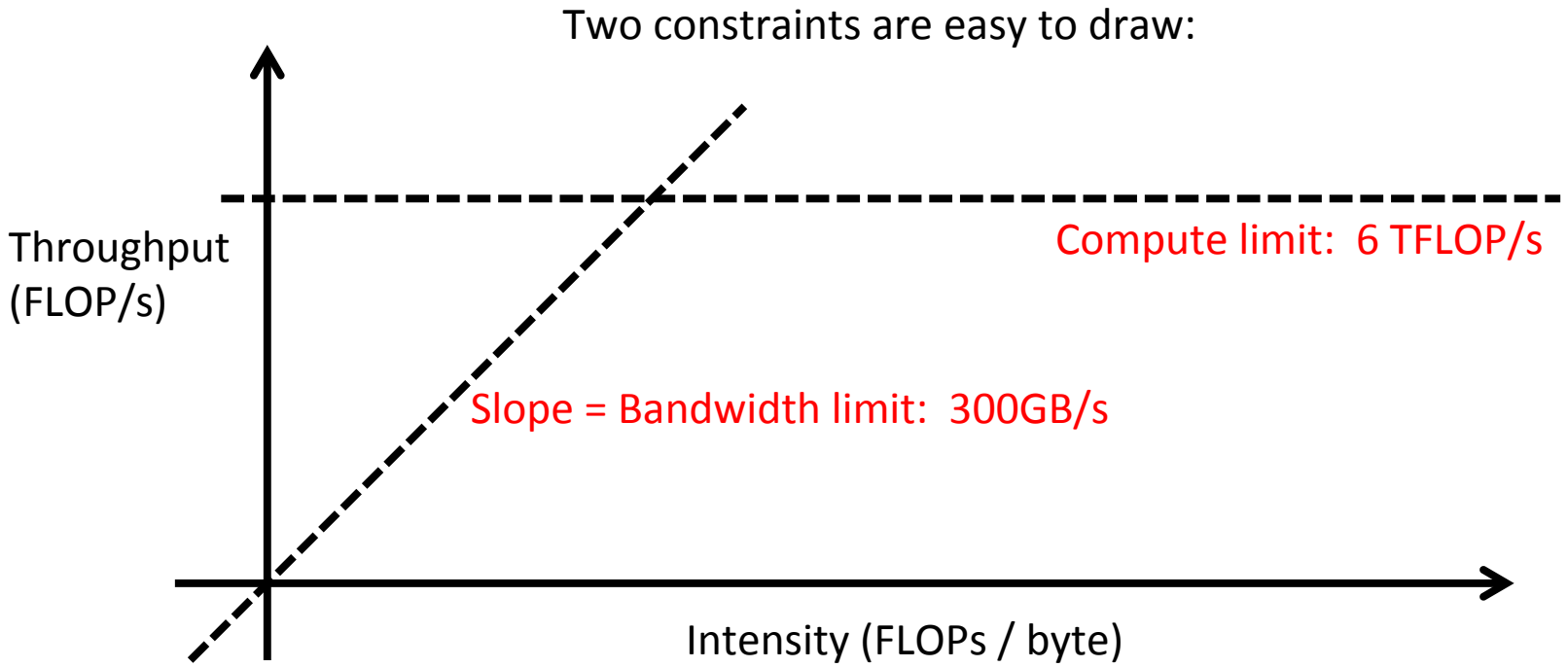
- E.g, for previous scenario, intensity is:

Intensity = (1e6 FLOPs) / (2.1e6 bytes) = 0.5 FLOPs/byte

➢ Low intensity = bottlenecked on memory.

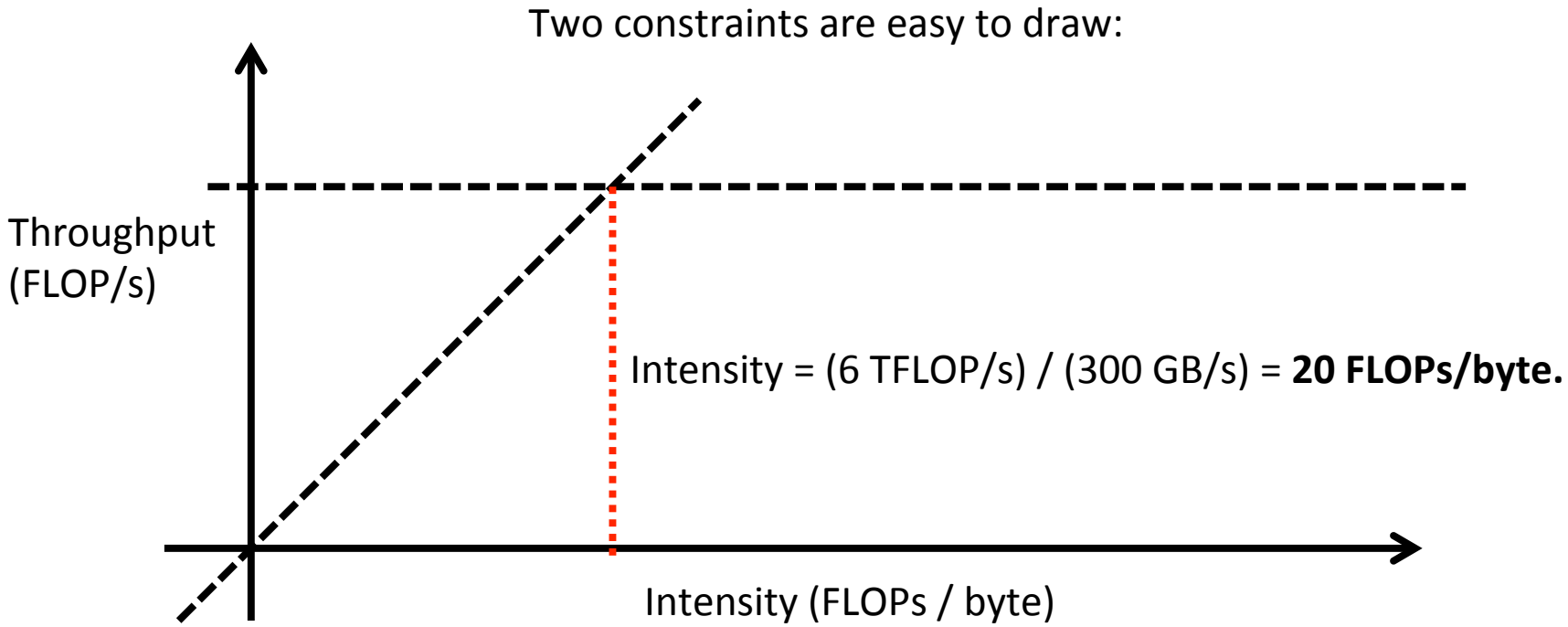
The "Roofline" model

- Williams, Waterson, Patterson 2009:
  - Visualize maximum throughput of our 2-part system as a function of intensity.



The "Roofline" model

- Williams, Waterson, Patterson 2009:
  - Visualize maximum throughput of system as a function of intensity.



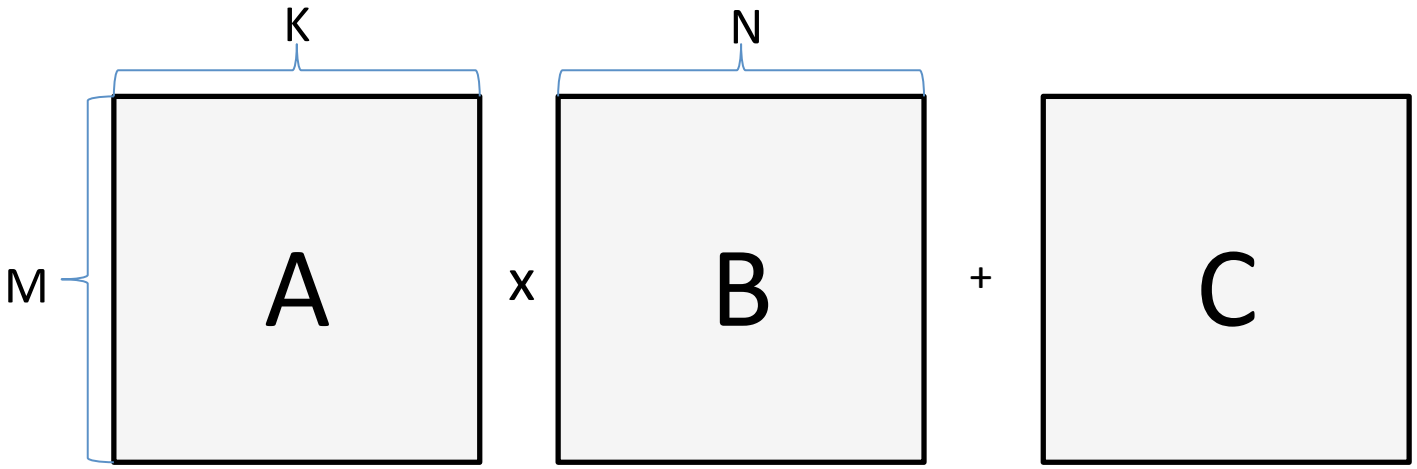
The “Roofline model”

- Easy to see relationship between memory-bound and compute-bound work.
  - Based on “theoretical” numbers: need intensity > 20 FLOPs/byte to be compute bound.

- Why is this useful to know?
  - Below 20 FLOPs/byte, compute is not constraining.

Example: matrix-matrix multiply

- Compute: C = C + A B for single precision matrices.

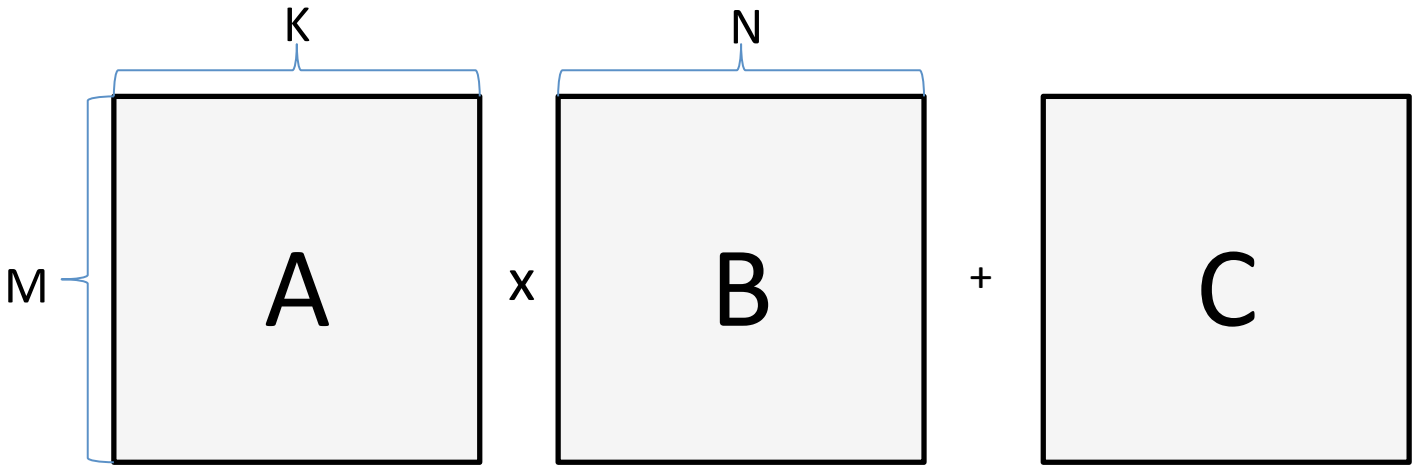


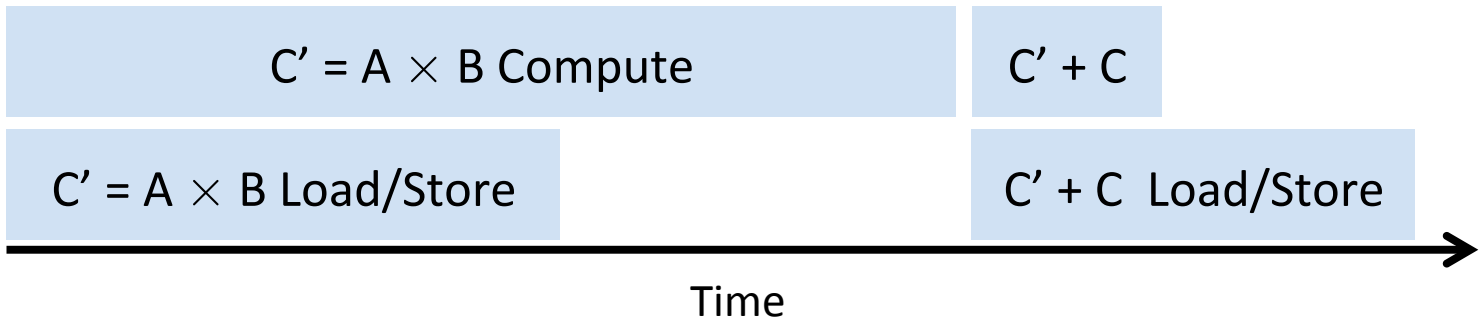
Memory to load + store: 4 bytes $\times$ (MK + KN + 2MN)
FLOPs to compute: $\approx$ 2 $\times$ MKN

For M=K=N=512:
Intensity = 64 FLOPs / byte (Should be compute-bound)

Example: matrix-matrix multiply

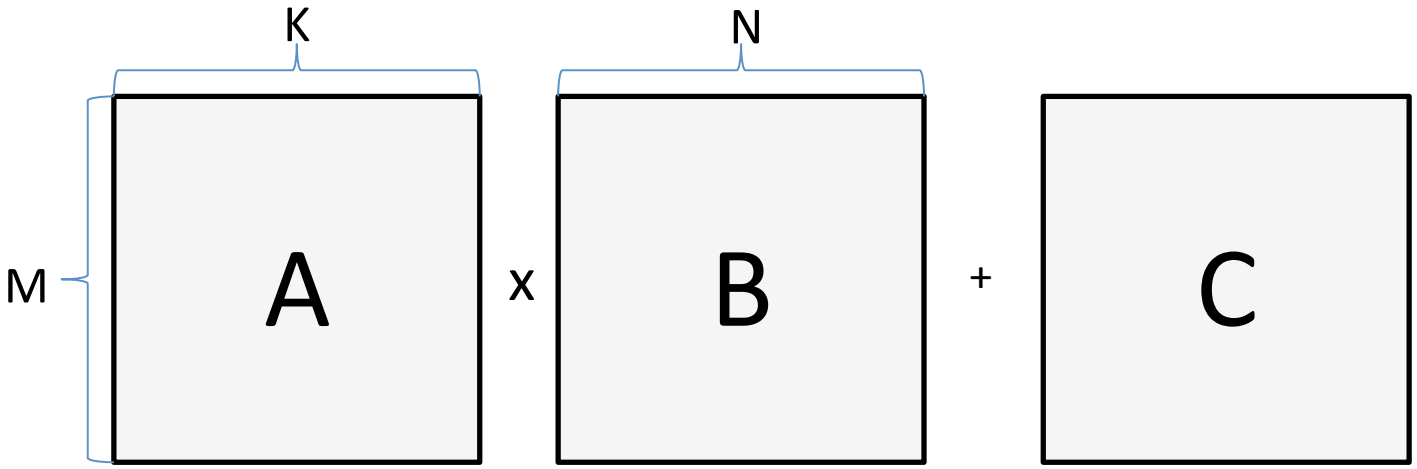
- Notice: we analyzed two operations as one.

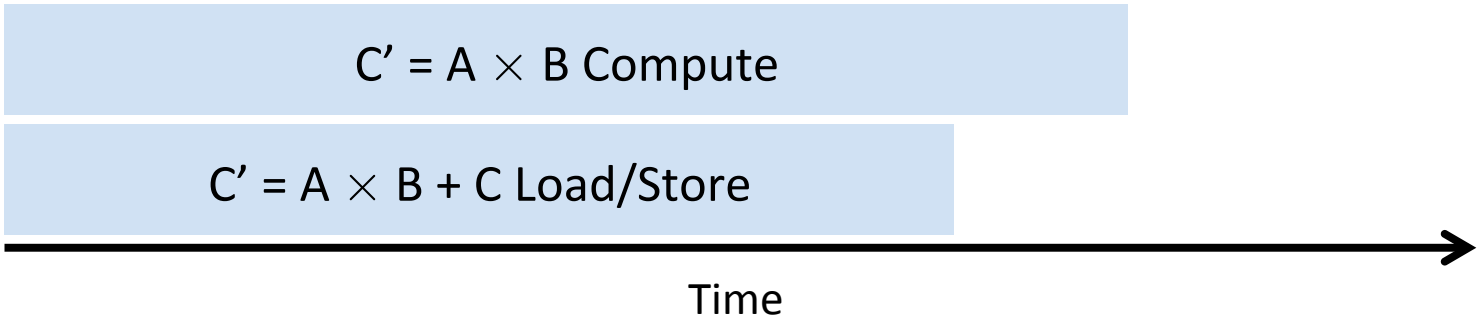




Example: matrix-matrix multiply

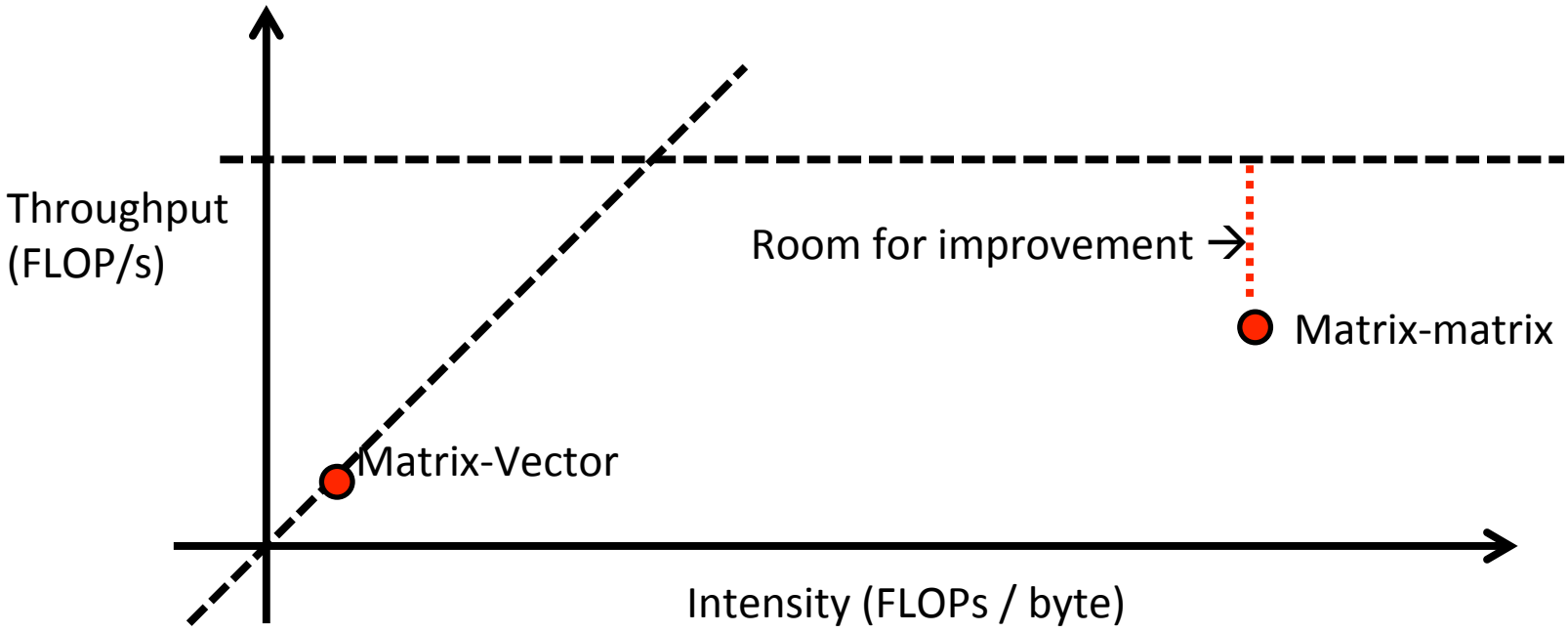
- Implicitly assuming we can overlap load/store of C to save time.





The "Roofline" model

- Roofline is the upper speed limit.
  - In practice, your code probably doesn't reach it.
  - Pick the piece of code that:
    (i) is responsible for most of running time.
    (ii) has some headroom for improvement.

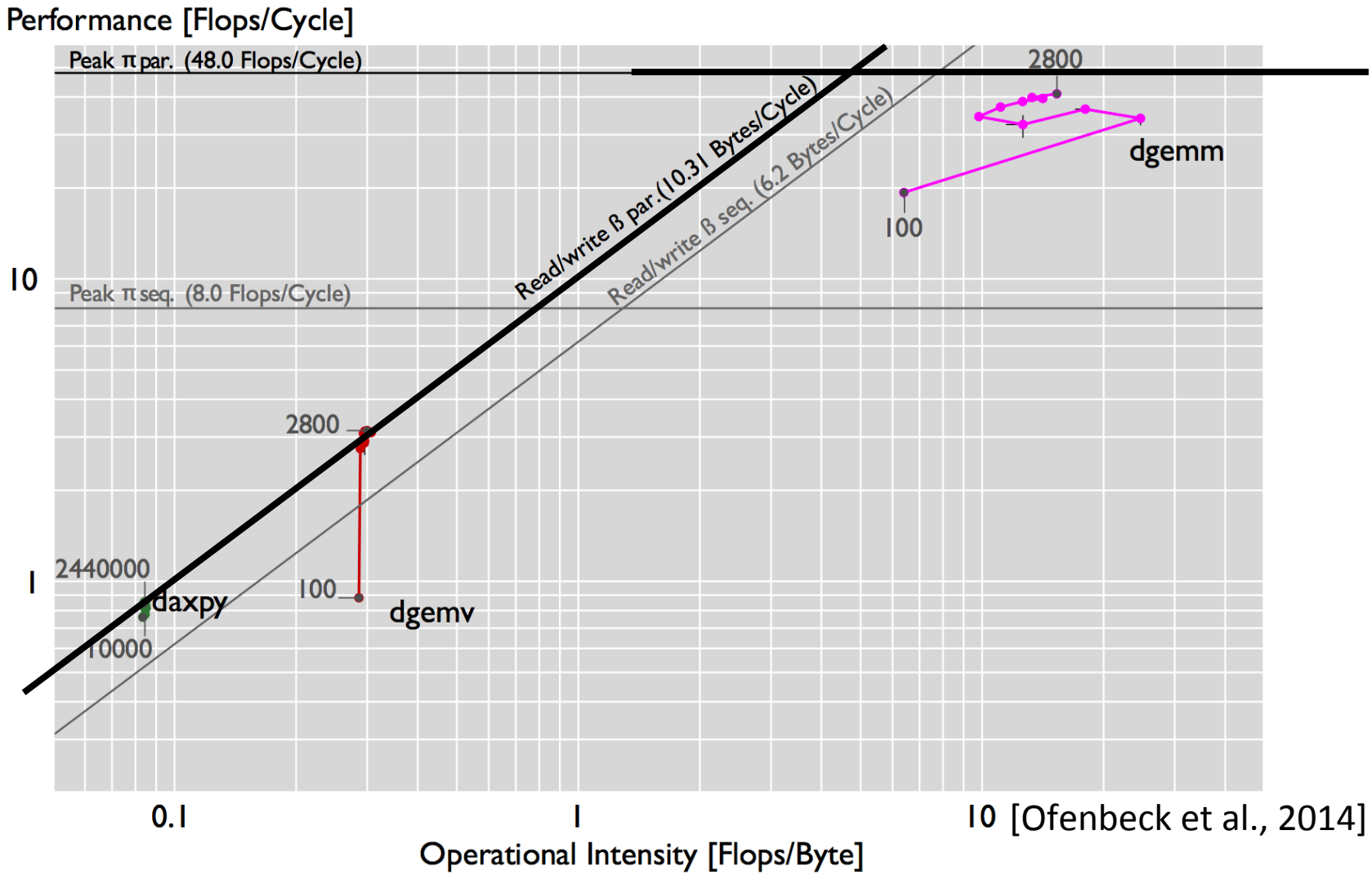


Roofline in practice

- Theoretical limit is hard to reach with fully generic code.
  - E.g., CuBLAS sgemm can achieve peak with large matrices, but tends to do badly for small matrices (bandwidth-bound).
  - Might need to sanity-check boundaries with small benchmarks.
    - E.g., Many Kepler GPUs could not achieve > 50% floating point peak using CUDA code.

Roofline in practice

- Often decent:



[Ofenbeck et al., 2014]

Summary

- Want to find maximum potential throughput (“speed of light”) to know best performance we can ever get.
  - Benchmark against this.
  - Factor speedup is nice; but not actionable.

- Use operational intensity and roofline model to quickly spec out what performance you might be able to achieve.

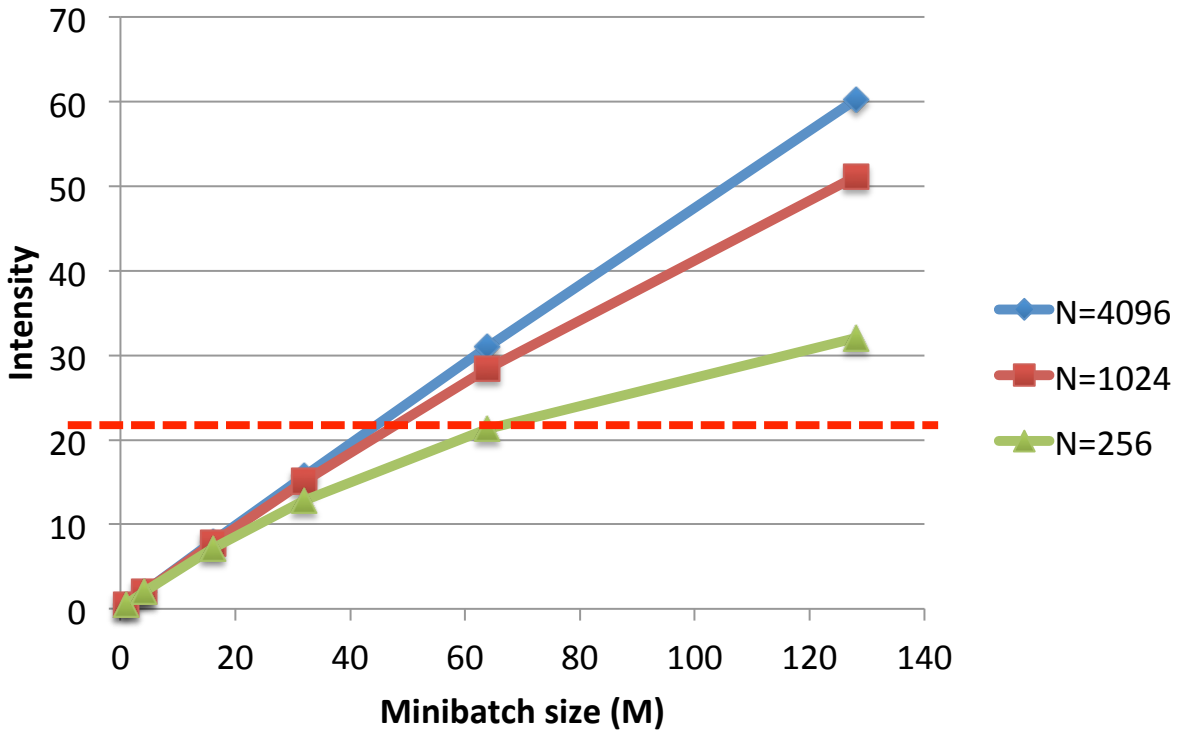
SINGLE NODE ISSUES

Minibatch size

- Common to process “minibatch” of examples.
  - Historically, minibatch size=1 has led to faster convergence. But this does not imply fastest experiment.
- What size should we use then?

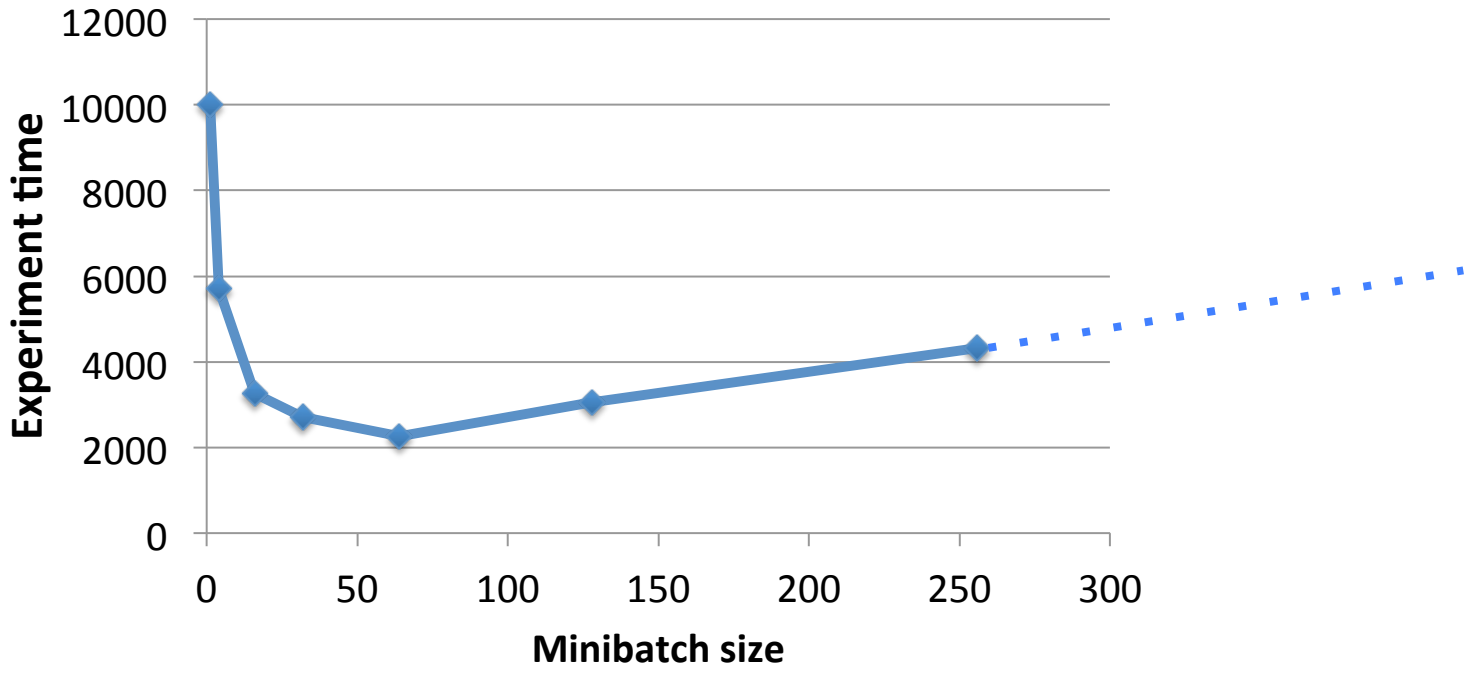
Minibatch size

- For DNN with N $\times$ N weights, minibatch size M:
  - Ops = $2N^2$ M, Memory = $4(N^2 + 2NM)$
  - Consider intensity for M=1…1024:



Minibatch size

- Below $\approx$ M=64, operations are memory-bound.
  - Increasing M leads to sub-linear increase in compute time.
- Beyond 64, DNN operations will be compute-bound.
  - Increasing M further leads to linear increase in time.
- Effect: experiment time falls, then rises again with M.



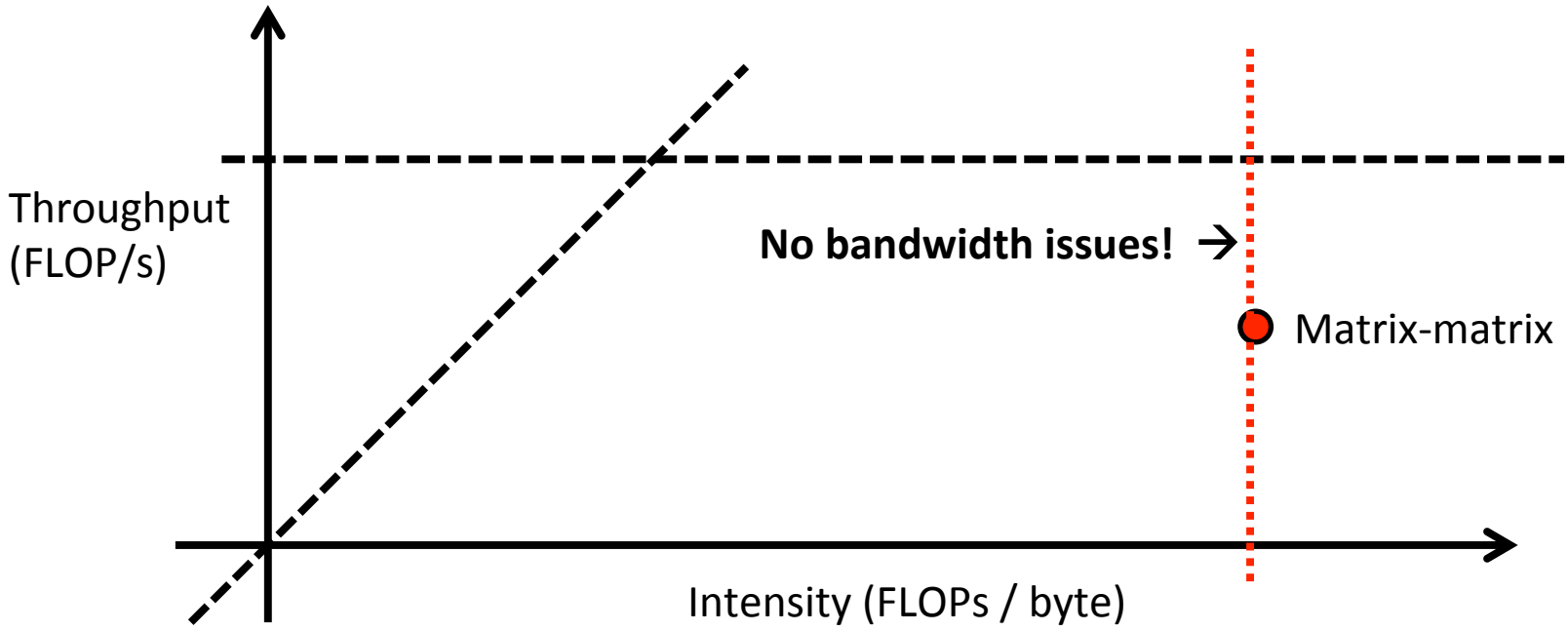
Moral of story

- For minibatch size:
  - Not much harm in raising until you are compute-limited; not much to gain beyond this point.

- In general, if you're *not* compute limited, there could be a free lunch in your future.
  - Bigger model = fit more data.
  - Bigger minibatch = faster convergence.

Optimizing software

- OK – your model is supposed to be compute limited now. But you're not achieving throughput you expect.
  - How do you make it fast?
  - Roofline model suggests some tactics over others.



Things to try

- Low intensity workloads:
  - Try to increase intensity by accessing memory less.

    (Try this first if you're in the "middle ground"!)
    - Look for data-reuse that will help you avoid redundant loading.

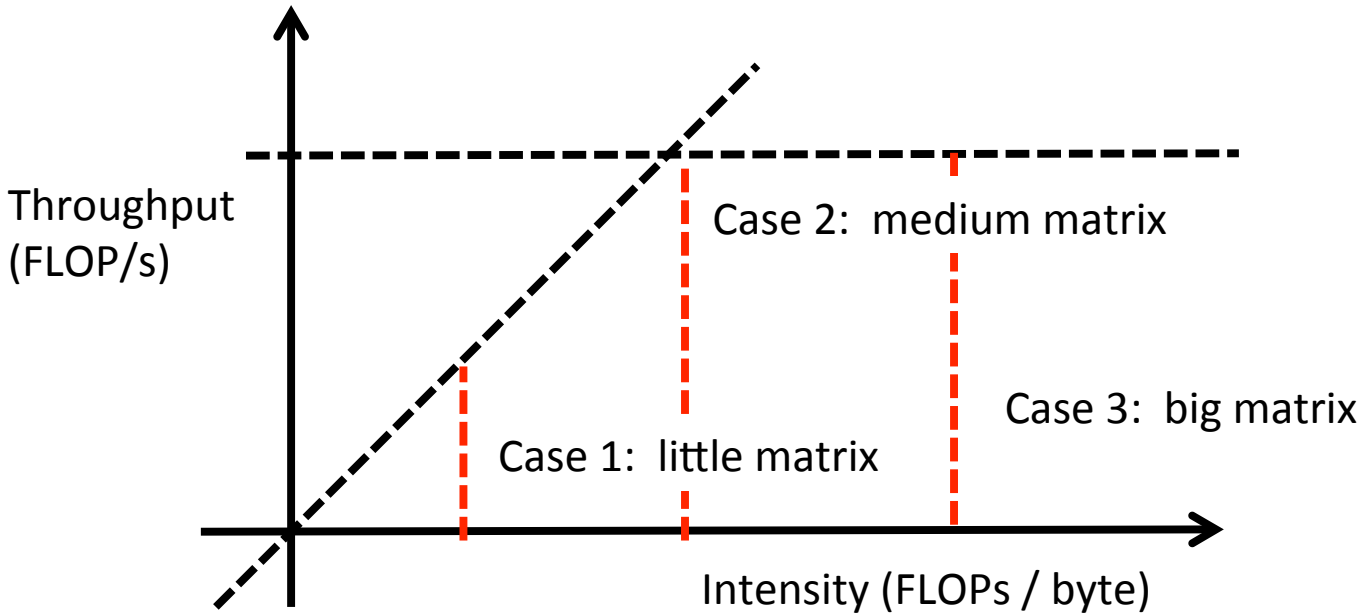
  - Focus on improving memory performance.
    - Sequential accesses on CPU / coalesced access on GPU.
    - Prefetch by hand.

- High intensity workloads:
  - Focus on improving compute performance.
    - Specialized instructions (SIMD, FMA = fused multiply add).
    - Adjust instruction mix.
    - Loop unrolling.

Note on code complexity…

- Very hard to write kernels that employ many optimizations at once.
  - And best optimization depends on problem parameters!

- Usually: dispatch problems into separate pieces of code optimized for different scenarios.



MULTINODE

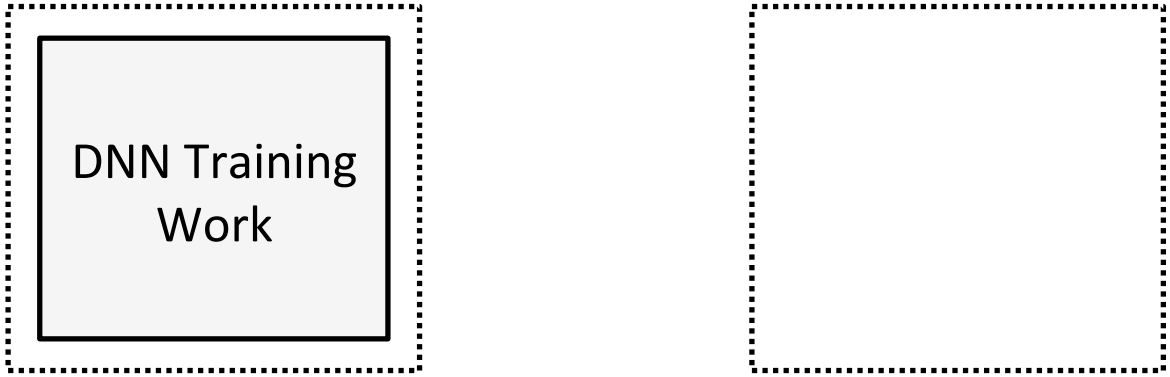
Training with clusters

- To go very fast, we want to use many CPUs, GPUs, or many machines at once.
  - Relatively fewer tools and libraries to help.
    - It's not that easy to automate.

- Re-use some of analysis tools to guide your decisions on how to parallelize work.

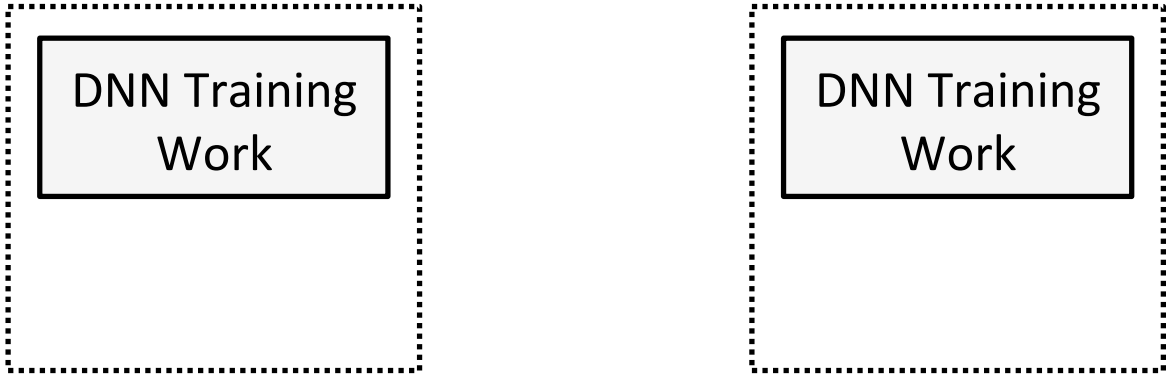
What can we hope to achieve?

- Ideal case: starting from single-node job, achieve higher throughput using more nodes for same job.



What can we hope to achieve?

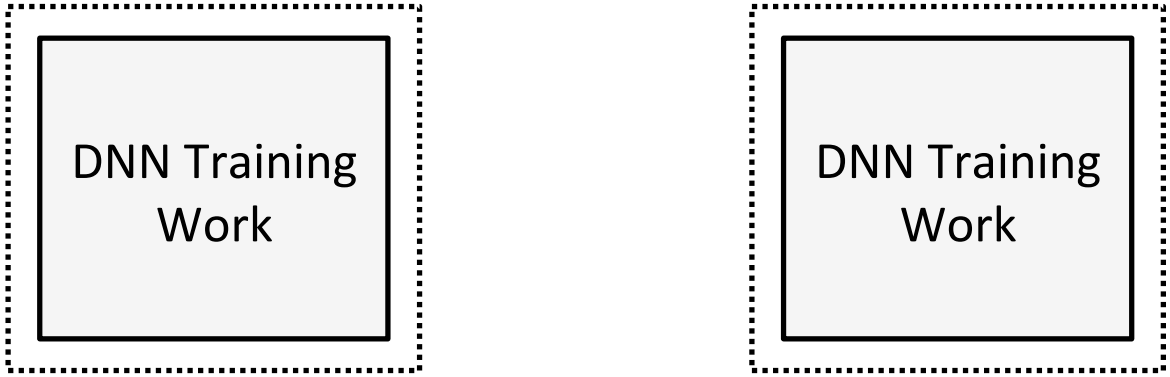
- Starting from single-node job, achieve higher throughput using more nodes for same job. (Ideally, 2x throughput.)



This is “strong scaling”: run same job in half the time.

What can we hope to achieve?

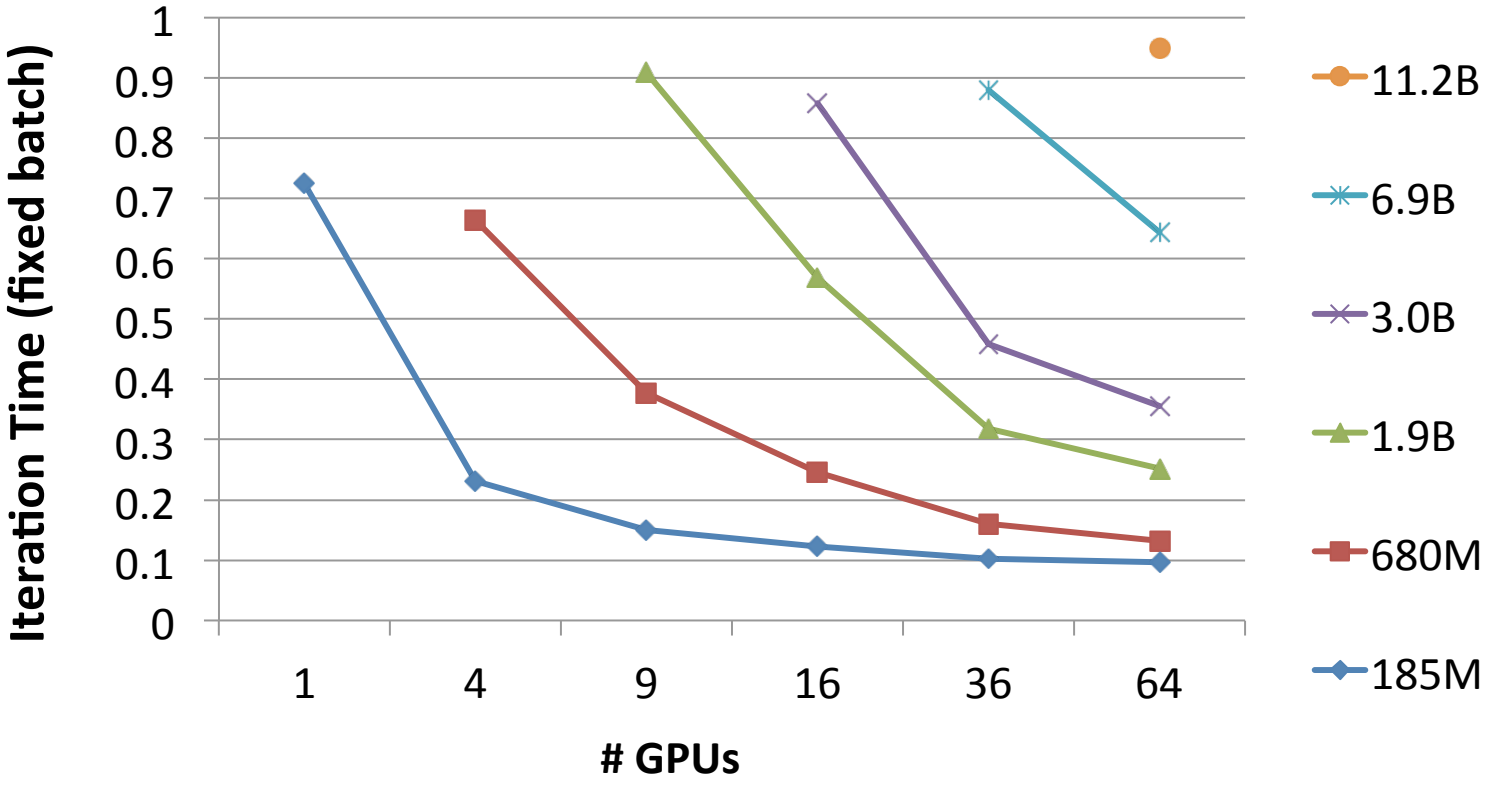
- Alternatively, we could parallelize *and* make workload larger (bigger model, bigger minibatch)



This is “weak scaling”: run larger job without slowing.

Example: weak scaling

Small network doesn't get faster with more GPUs.
But giant networks run about same speed.



[Coates et al., 2013]

Weak vs. strong

- If you can use a bigger model, or if a 2x increase in minibatch would help:
  - Job is a good candidate to scale up.
  - ***Recommend doing this first.***

- In practice:
  - Sometimes don't want a big net (e.g., data)
  - Minibatch size has already hit diminishing returns.
  - Want faster *cycle time* so we can learn quickly.

- What makes strong scaling difficult?

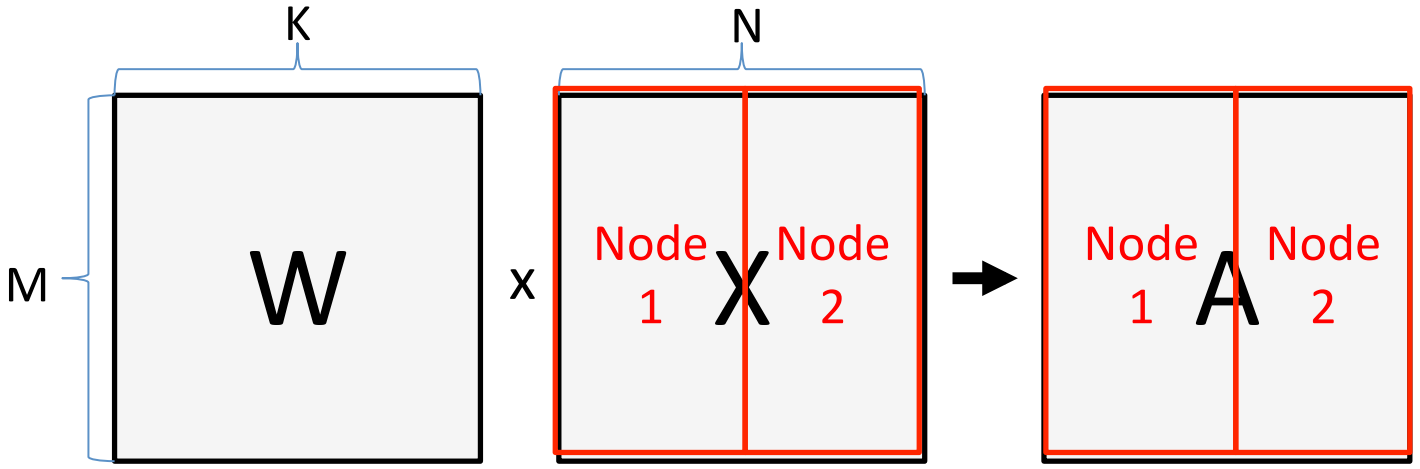
Performance modeling

- To understand this, need to analyze performance of multi-node system.

- First: let's partition work and just start by assuming infinite network bandwidth.

Example: Data Parallelism

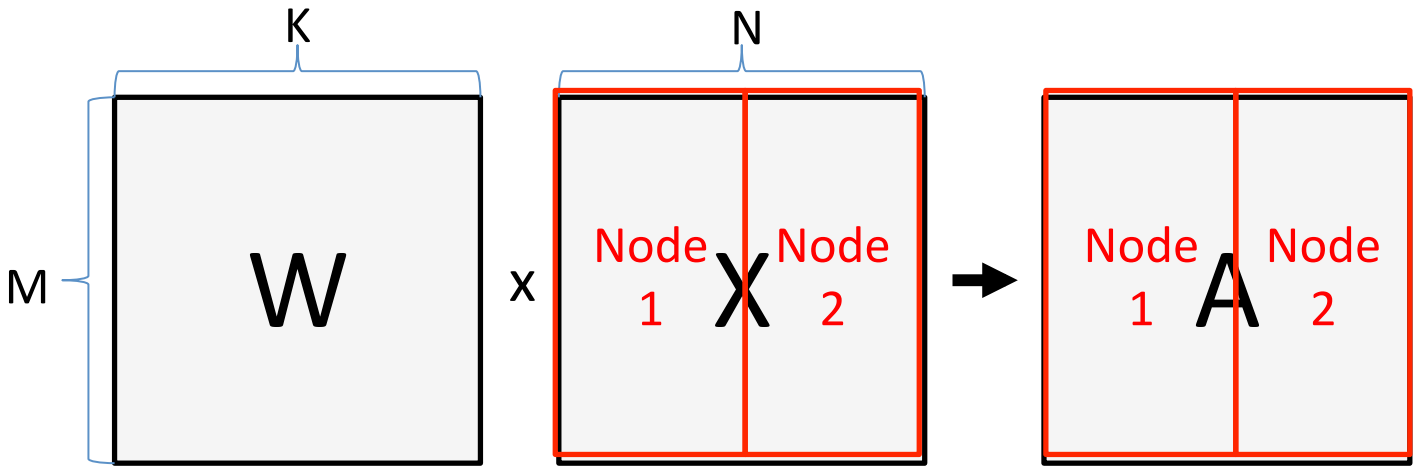
- Common practice: partition training job by splitting minibatch (X) in half.
  - Keep model (W) synchronized over network.



- What happens to workload on Node 1?

Example: Data Parallelism

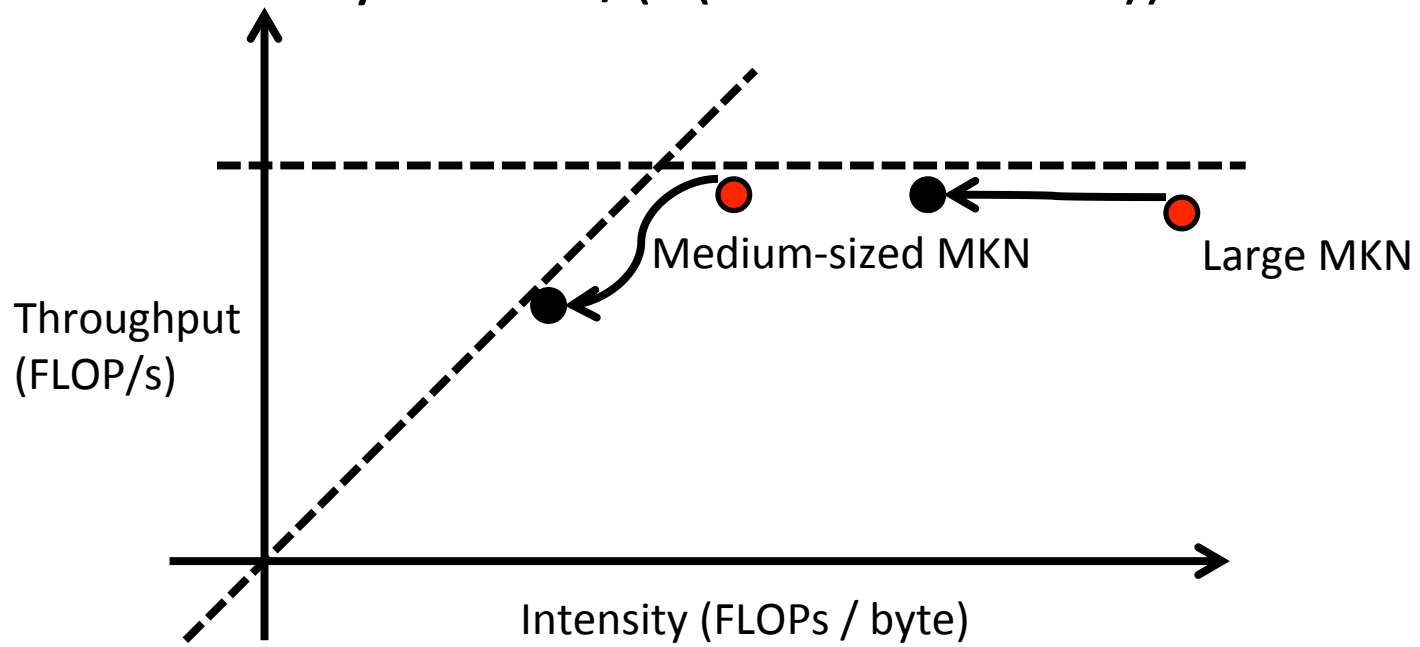
	FLOPs	Memory	Intensity
Before:	2 MKN	4(MK+KN+MN)	MKN/(2(MK+KN+MN))
After:	MKN	4MK + 2KN + 2MN	MKN/(2(2MK+KN+MN))



- Node 1 operational intensity falls!

Local throughput

- This may or may not cause a problem depending on size of model.
  - Intensity = MKN/(2(2MK+KN+MN))



We'll assume that Node 1 can still run at max throughput.
Otherwise, need to prorate Node 1's throughput limit for any other analysis.

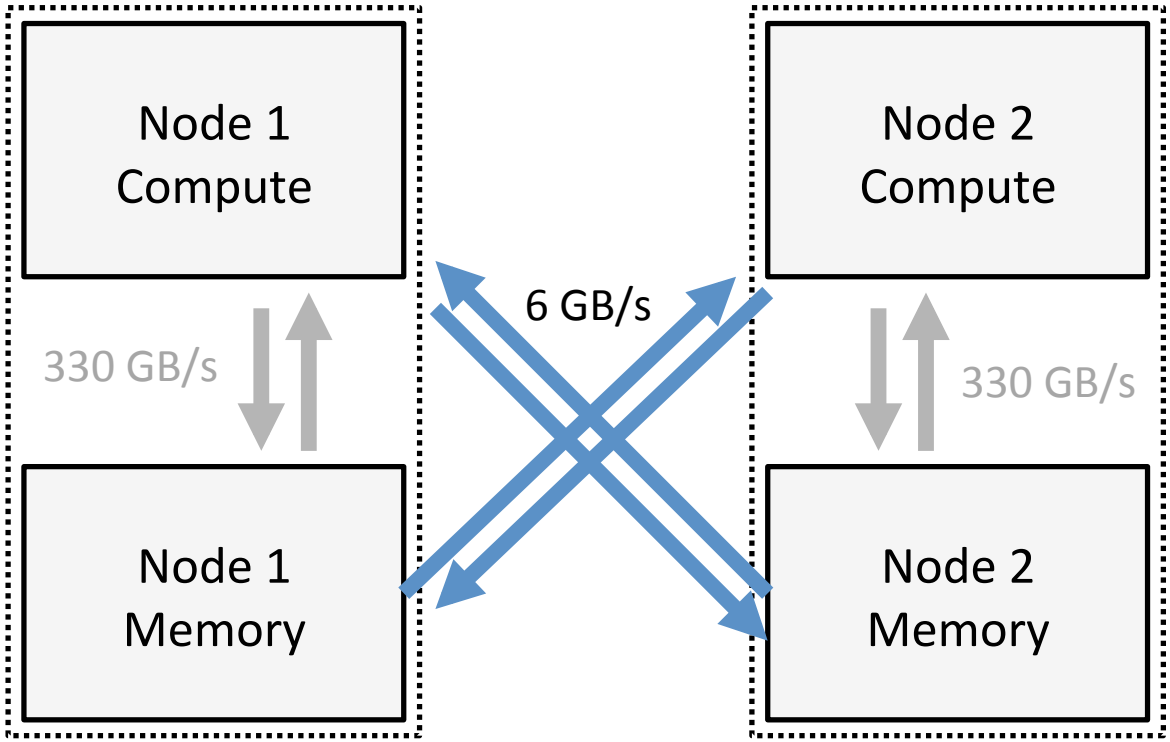
Performance Modeling

- Even with infinite network bandwidth, we might not be able to scale.
  - Have to be mindful of how distributing affects local node's efficiency.

- Next:
  - Assume local throughput is nice: 6 TFLOP/s
  - How do we analyze communication?

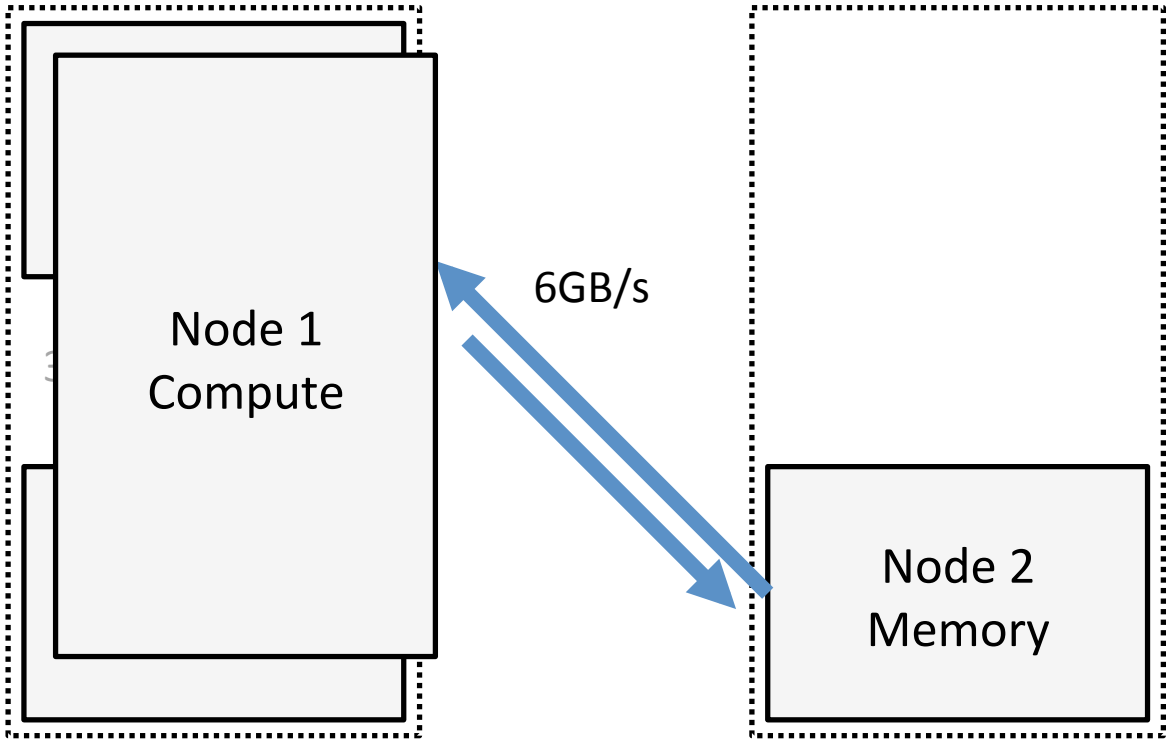
Performance modeling

- Approach we used to analyze operations for single node also useful for thinking about multiple nodes.
  - But make distinction between *local* and *remote* memory.



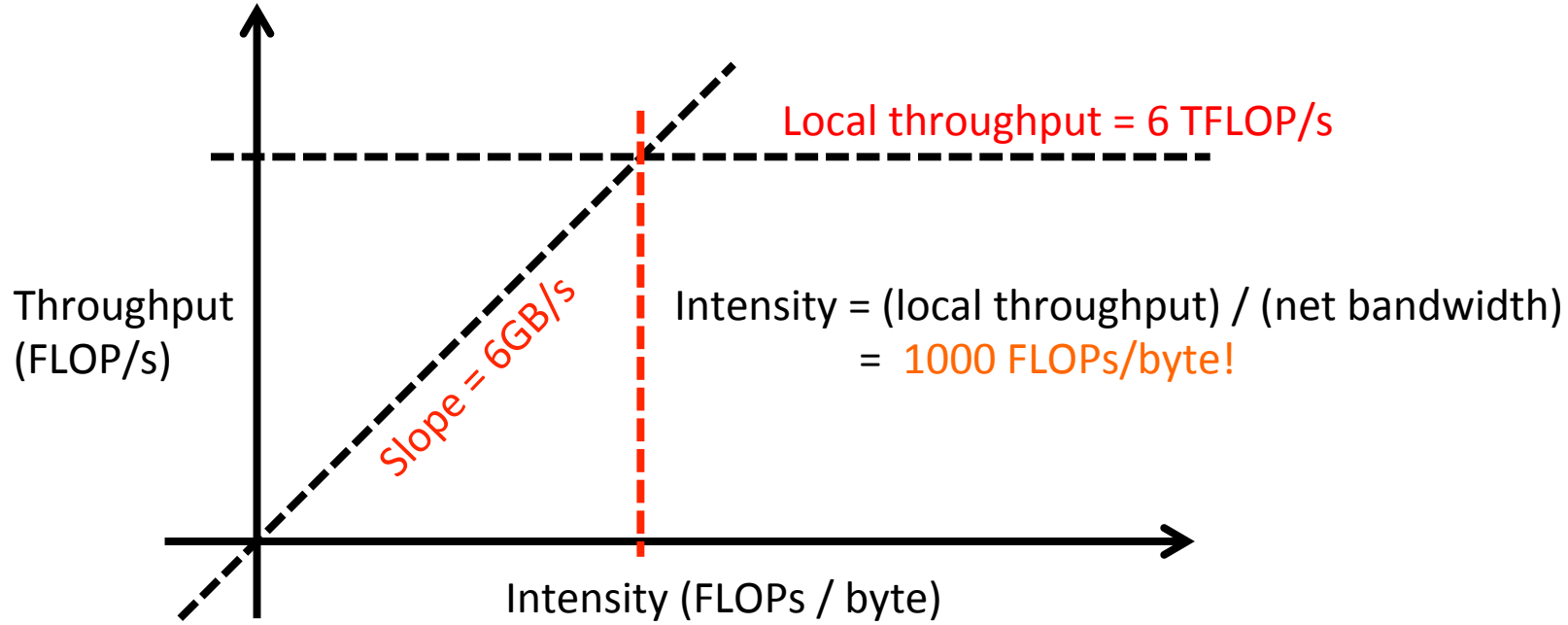
“Roofline” model

- Analyze performance of nodes in terms of their *local throughput* + bandwidth to *remote* data.



“Roofline” model

- Analyze performance of nodes in terms of their *local throughput* + bandwidth to *remote* data.



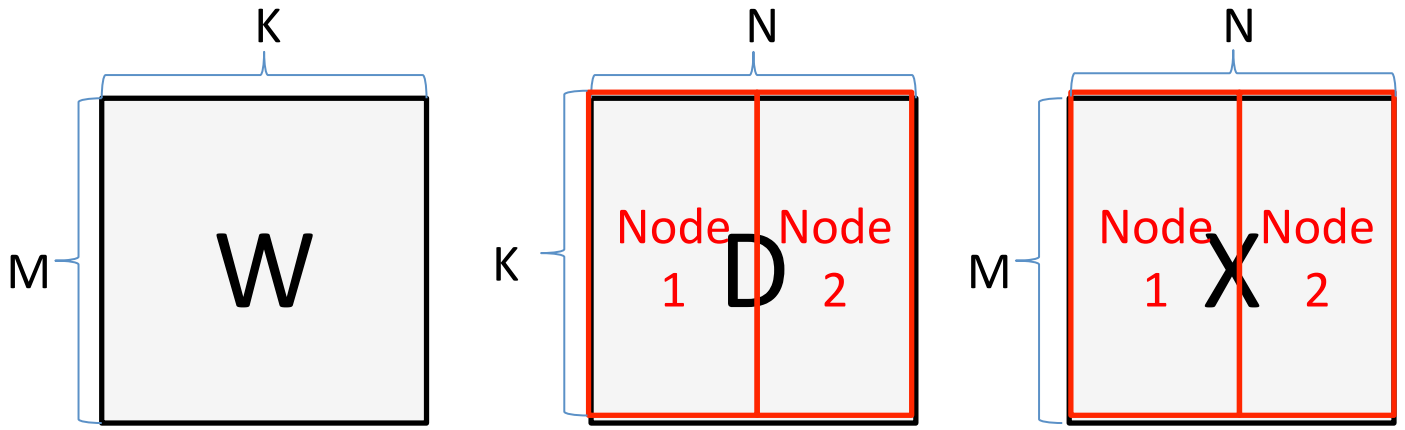
Note much higher intensity: need to do 1000 FLOPs locally *(at 6 TFLOP/s throughput*) for every 1 byte of network traffic.

Example: Data Parallelism

- What about gradient updates / communication?
- Analyze distributed operation for Node 1.

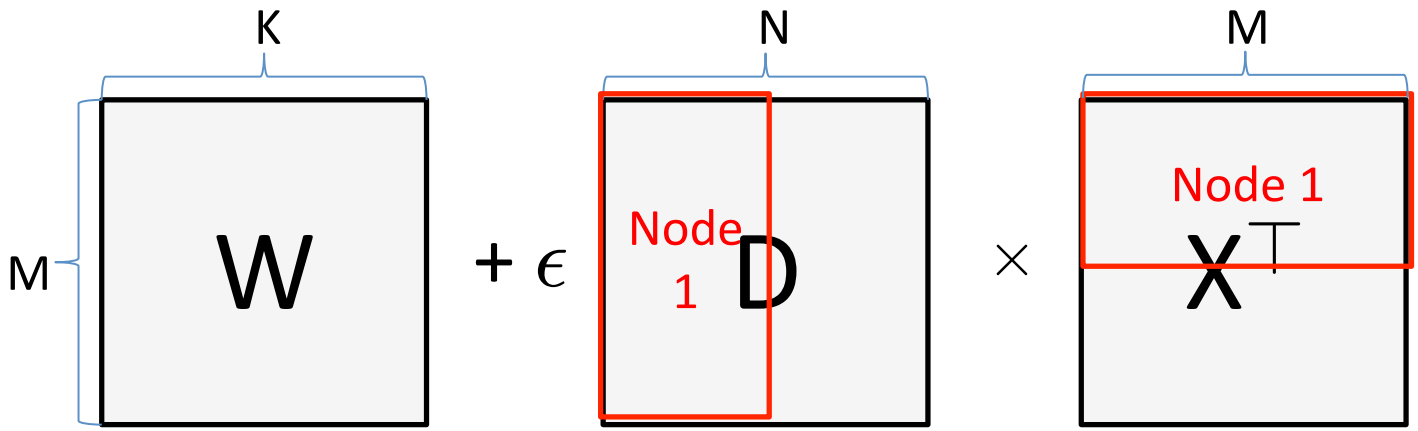
$W = W + \epsilon\, D\, X^{\top}$

Send(W, Node 2)



Example: Data Parallelism

- Node 1 needs to perform computation on local portion of D, X and local copy of W.
- Send updated W to Node 2.

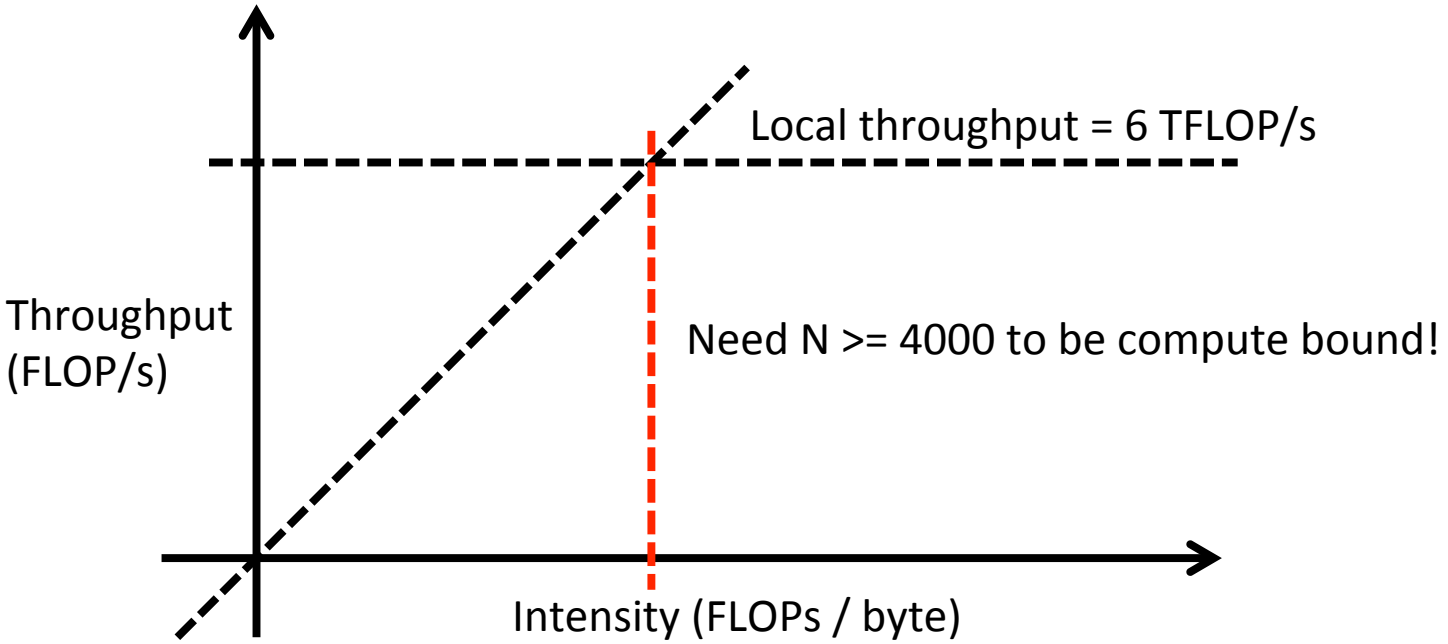


FLOPs	MNK + 2MK
Remote Memory	4MK
Overall Intensity	(MNK + 2MK)/ 4MK = N/4+1/2

Number of FLOPs we can carry out on Node 1 per byte of network traffic.

Overall throughput

- How big does N need to be to achieve high overall throughput?
  - Overall intensity $\approx$ N/4



Assumptions

- …wait. We violated a modeling assumption:

> **Key assumption:** we can always stream memory simultaneously with computation.

- But we introduced a dependency:

  $W = W + \epsilon\, D\, X^{\top}$

  Send(W, Node 2)

- We can deal with this a few ways:
  - More analysis to overlap Send() with other ops.
  - Actually stream W while it's being computed.

> Don't forget overlap assumption.
> Optimize code to make it true.

Putting everything together

- Seen how partitioning affects our ability to scale.
  - Changes size/shape and intensity of local work.
  - Distribution introduces network bandwidth limit.
  - Use roofline to get a sense for both issues!

Putting everything together

- Suggested design process:
  1. Scale up weakly if you can. (Strong scaling is hard.)
     - I.e., Make your model + minibatch as large as practical before parallelizing.
  2. Choose a partition of the work and data over nodes.
  3. Estimate local node max throughput (via roofline or benchmarking)
  4. Use local throughput and cluster network bandwidth to create multi-node roofline model.
  5. Estimate overall max throughput of work on each node.
  6. Are you happy?
     - No: Go to next slide, or try new partition.
     - Yes: Go back to deep learning.

Optimization strategy

- Like single-node: find operations that use bulk of time.
- Hunt for partitioning scheme that has a lot of potential (i.e., high "speed of light")
- Search for opportunities to increase communication+compute overlap.
- Judiciously apply hardware.
  - Compute limited: more GPUs / CPUs.
  - Bandwidth limited: faster network.
    - E.g., dual-rail connection, or 100G networks.

CONCLUSION

Key ideas

- Measure against the “speed of light”: the fastest your code could ever run.
- Use simple performance models to understand tradeoffs; identify approaches with high potential.
- Challenging part of multinode training is partitioning and communication.
  - Build intuition for good/bad schemes by trying out different choices and calculating max throughput.

Thank you!

Thanks: Greg Diamos & Bryan Catanzaro

References:

Coates, Huval, Wang, Wu, Ng, Catanzaro. "Deep Learning with COTS HPC." ICML 2013.

Samuel Williams, Andrew Waterman, David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures."
http://www.eecs.berkeley.edu/~waterman/papers/roofline.pdf

Ofenbeck, Steinmann, Caparros, Spampinato, Püschel. "Applying the Roofline Model" to appear in Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014.