# Towards Fully Automatic Programming Systems for Very Large-Scale ML

(It's time to go back to the 70's)

Chris Jermaine

Rice University

# Assertion at the Heart of This Talk

- Existing ML systems (TensorFlow, PyTorch) have had huge impact
- But ML systems seem to have reached a dead end
- Current SOTA in ML systems:
  - Excellent for small-ish models, one GPU
  - OK/adequate for single machine, multi-GPU learning/inference
  - Quite poor for distributed learning/inference, big models, big data
  - Very difficult to train/use models approaching size of GPU RAM

# Example Application

- LLaMA 65B large language model inference
  - PyTorch
  - AWS `p4d.24xlarge` machines $8\times$ A100 GPUs
  - "prefill" (first token inference)
  - 4096-length prompt requires 2.9E14 multiply ops
  - 8192-length prompt requires 6.2E14 multiply ops
- Server has 50E14 FLOPS (BFLOAT16 tensor core)
  - 4096 inference should take roughly $\frac{50}{2.9\times 2} = \frac{1}{8}$ second
  - 8192 inference should take roughly $\frac{50}{6.2\times 2} = \frac{1}{4}$ second

# Example Application

- LLaMA 65B large language model inference
  - PyTorch
  - AWS `p4d.24xlarge` machines 8× A100 GPUs
  - "prefill" (first token inference)
  - 4096-length prompt requires 2.9E14 multiply ops
  - 8192-length prompt requires 6.2E14 multiply ops

- Server has 50E14 FLOPS (BFLOAT16 tensor core)
  - 4096 inference should take roughly $\frac{50}{2.9 \times 2} = \frac{1}{8}$ second
  - 8192 inference should take roughly $\frac{50}{6.2 \times 2} = \frac{1}{4}$ second

| | First token inference | | | |
|---|---|---|---|---|
| Seq len | Time (s) | Mults/sec | Mult/sec per GPU | % peak performance |
| 4096 | 6.11 | 4.7E13 | 5.9E12 | 2% |
| 8192 | 7.06 | 8.8E13 | 1.1E13 | 4% |
| 16348 | OOM | N/A | N/A | N/A |

# This Is a Software Problem

- But people buy hardware to get around this
  - And that hardware costs a lot

# This Is a Software Problem

- But people buy hardware to get around this
  - And that hardware costs a lot
- Consider the following simple experiment
  - $\mathbf{X}$ is a 20K by 20K matrix
  - Compute $\mathbf{Y} = \mathbf{X} \times \mathbf{X} \times ...\mathbf{X}$
  - 10 multiplications
  - FP32

# Run on Four Different Processors

- Intel Xeon "Ice Lake" (ca 2021) 32 cores
  - CPU is $1500 then $2 per GB RAM (DDR4)
  - Time for MM chain is **63 seconds**
- Nvidia P100 GPU
  - GPU is $500 and comes with 16GB RAM
  - Time for MM chain is **16.8 seconds**
- Nvidia V100 GPU
  - GPU is $1500 and comes with 16GB RAM
  - Time for MM chain is **10.4 seconds**
- Nvidia A100 GPU
  - GPU is $20000+ and comes with 80GB RAM
  - Time for MM chain is **8.5 seconds**

# Consider Price for Performance

- Say I want to do 100 of these MatMuls per second
  - Intel Xeon: $630,000
  - P100 GPU: $83,900
  - V100 GPU: $208,500
  - A100 GPU: $1,700,000
- P100 GPU is by far the best
  - A100 is by far the worst

# Consider Price for RAM

- Say I want to store 5TB in RAM
  - Intel Xeon: \$15,000 $= 5 \times$ (\$1000 + \$2000)
  - P100 GPU: \$160,000
  - V100 GPU: \$470,000
  - A100 GPU: \$1,200,000
- CPU is by far the best
  - A100 is by far the worst

# So Why Does Anyone Buy an A100?

- Not because compute is super fast
- Rather:
  - High RAM number compared to other GPUs mitigates need for partitioning
  - Fast interconnect limits performance hit when I do

# Could Software Make A100's Obsolete?

- What if the system automatically partitioned?
  - And autmatically hid communication behind computation?
- You might choose CPU for memory-intensive and low compute
  - Ex: LLM toxicity check with 100K tokens (more than 600GB RAM for LLaMA 65B)
- You might choose P100 for compute-intensive
  - Training with a more moderate horizon
- But you'd never choose an A100
  - And you'd save $$

# Could Software Make A100's Obsolete?

- What if the system automatically partitioned?
  - And autmatically hid communication behind computation?
- You might choose CPU for memory-intensive and low compute
  - Ex: LLM toxicity check with 100K tokens (more than 600GB RAM for LLaMA 65B)
- You might choose P100 for compute-intensive
  - Training with a more moderate horizon
- But you'd never choose an A100
  - And you'd save $$

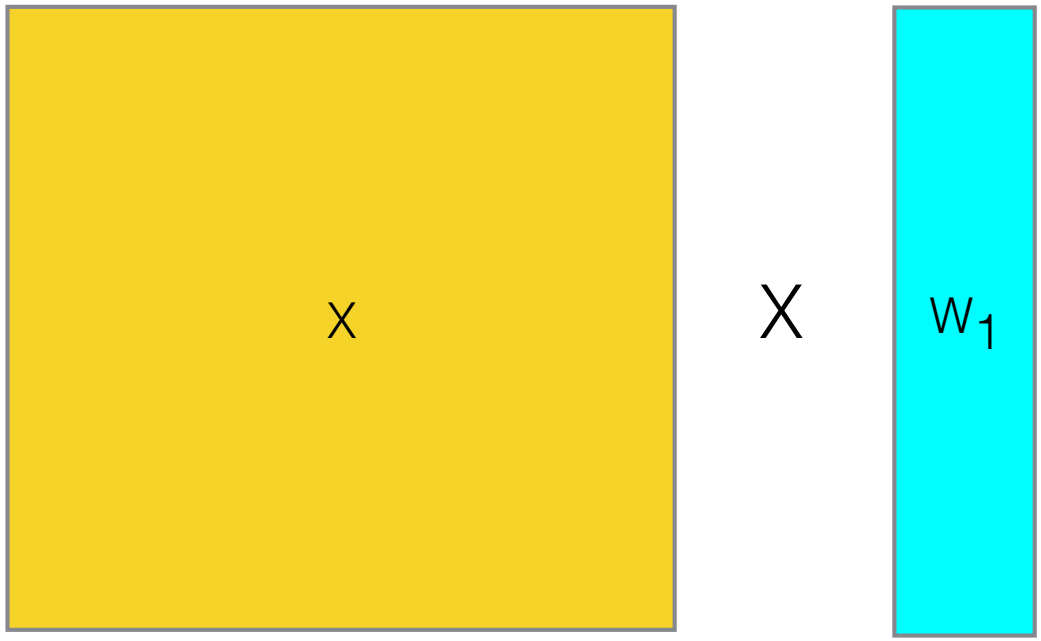
**It's our goal to build this software!**

# Why Can't Systems Effectively Use the Hardware?

- Fundamentally, ML systems lack abstraction
  - At the lowest level, these systems execute compute graphs
  - Vertices are operations, edges data flow
- But these are not graphs of logical operations!
  - They are physical operations that need to be run somewhere
  - If they can't run well, system can't "figure it out"

# Even the Term "Data Parallel" Is Problematic

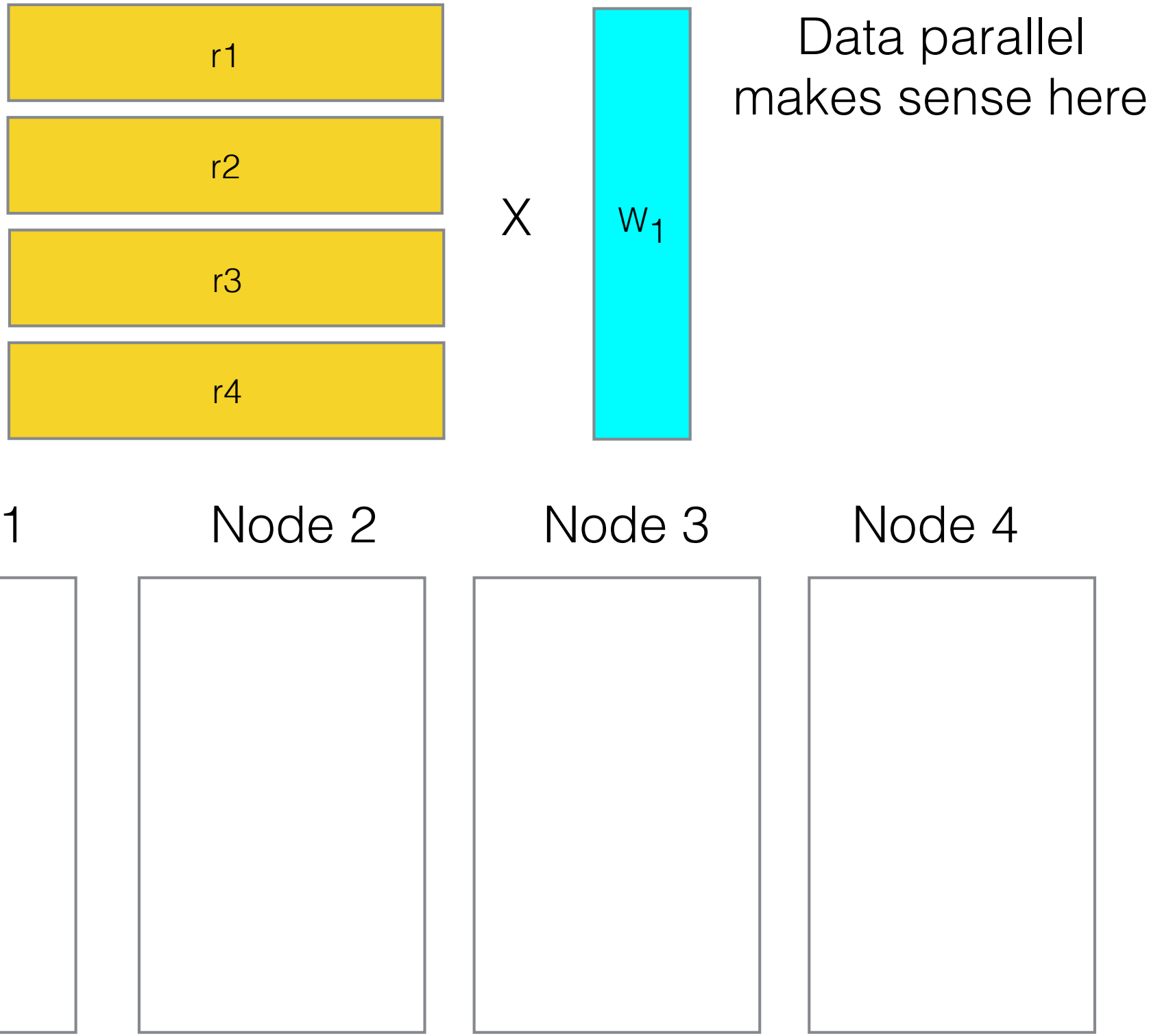
- It assumes there is something different/special about data
  - But that's not true!
  - In the end you are just doing some variant of MatMul
  - Who cares which intput matrix is data and which is model?
  - Just automatically run it the best way

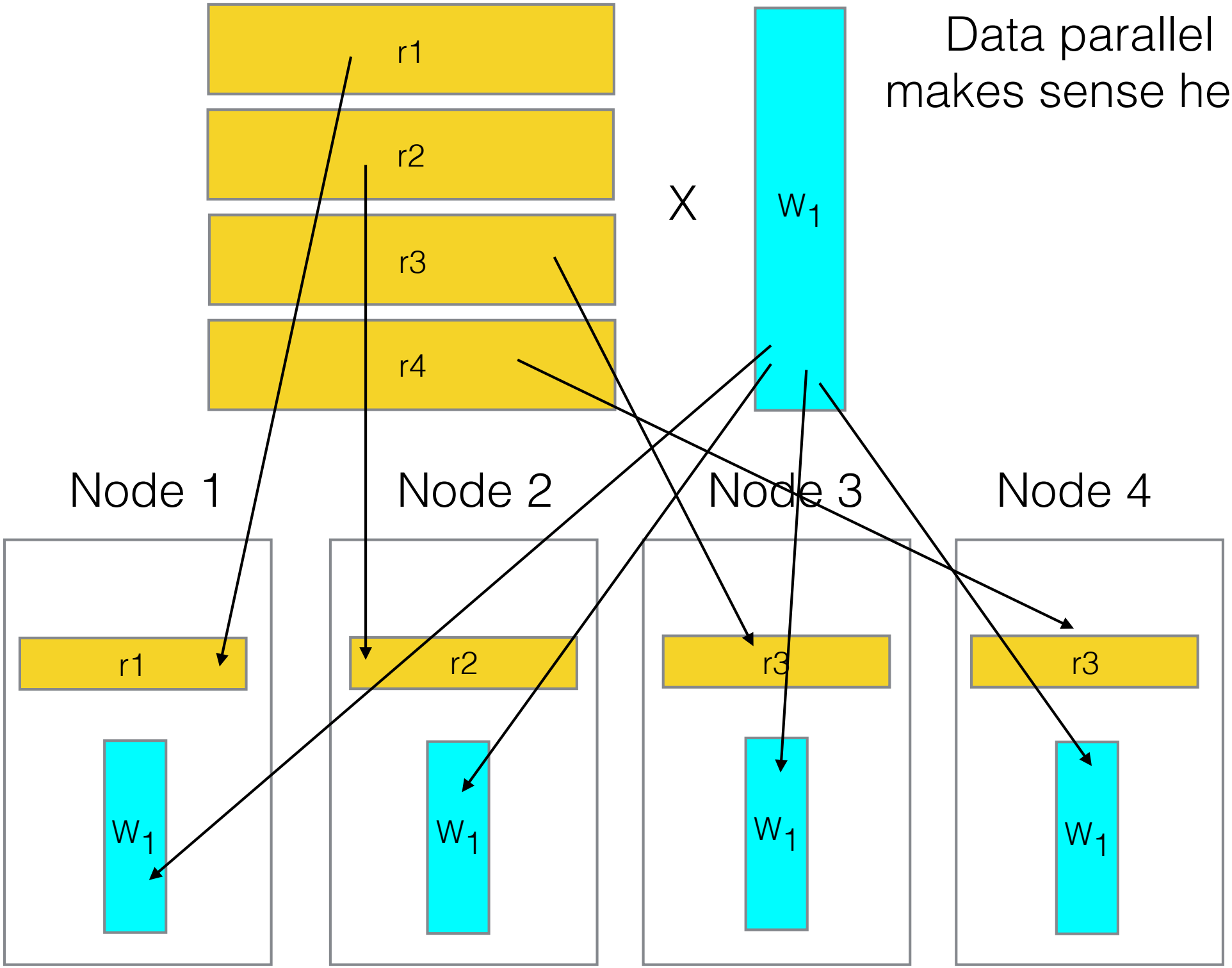


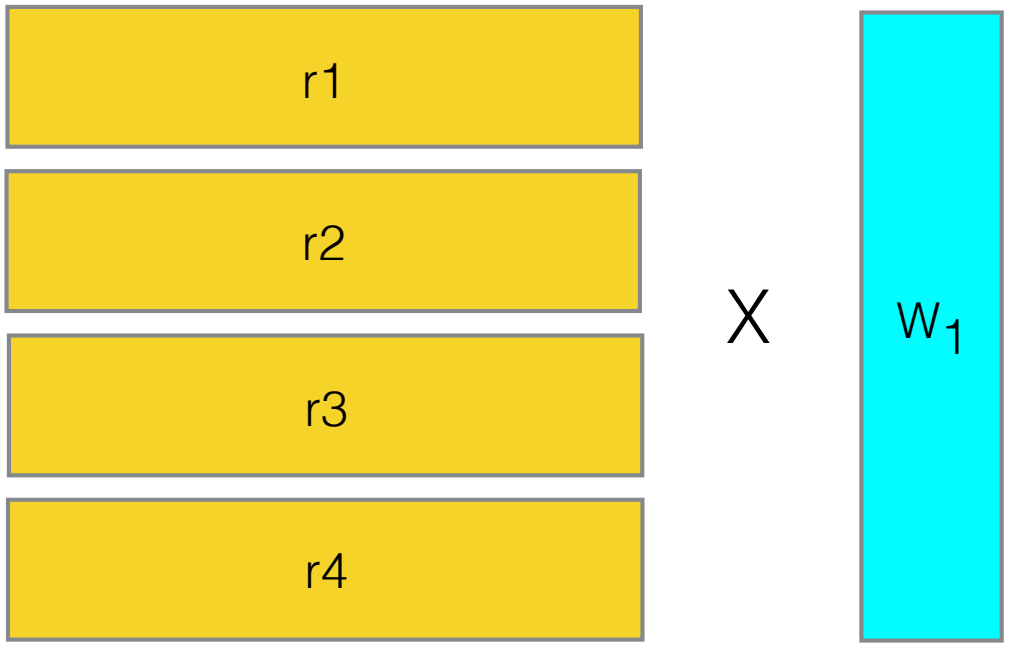

Data parallel makes sense here

Compute:

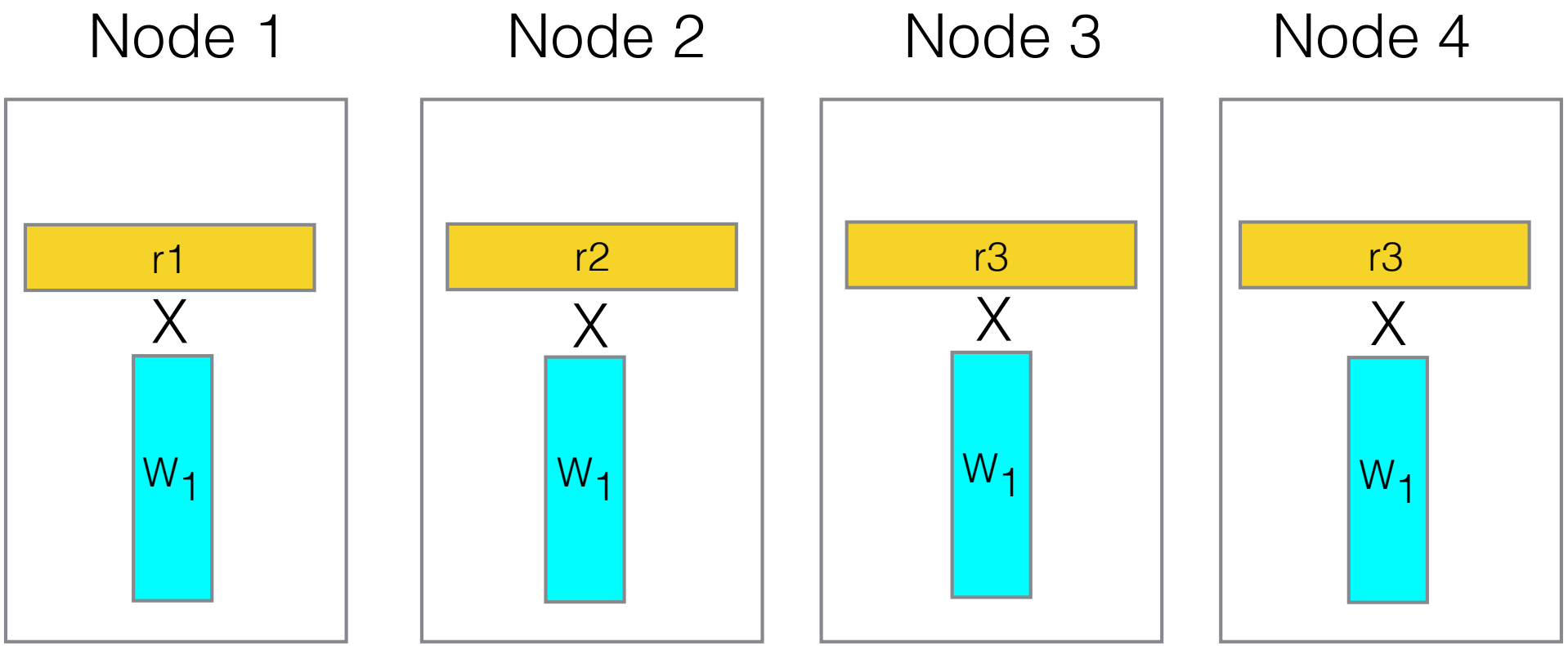
$f(XW_1)$

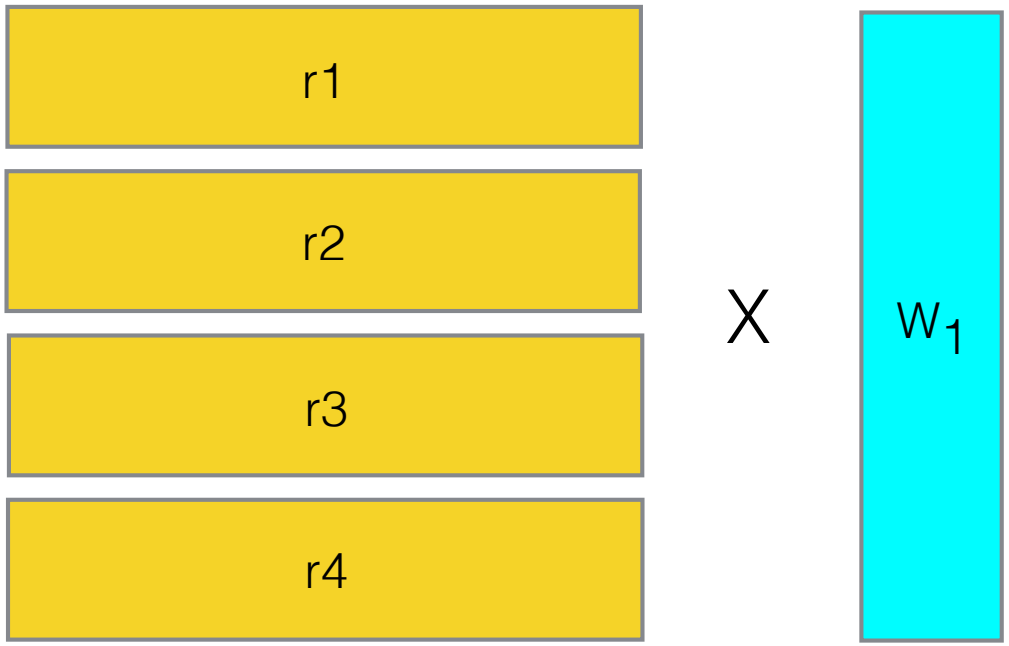

r1
r2
r3
r4
X
$W_1$
Data parallel
makes sense here
Node 1
Node 2
Node 3
Node 4


r1
r2
r3
r4
X
$W_1$
Data parallel
makes sense here
Node 1
Node 2
Node 3
Node 4
r1
r2
r3
r3
$W_1$
$W_1$
$W_1$
$W_1$



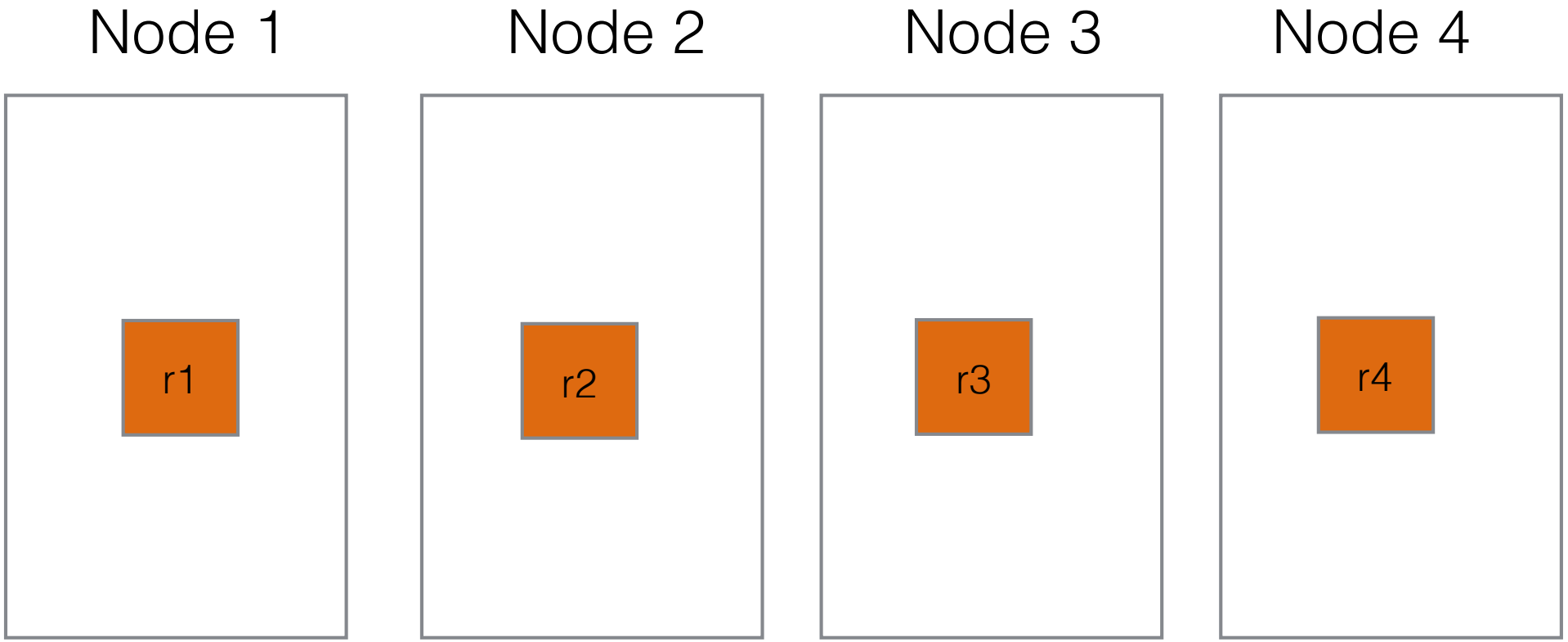

Data parallel makes sense here

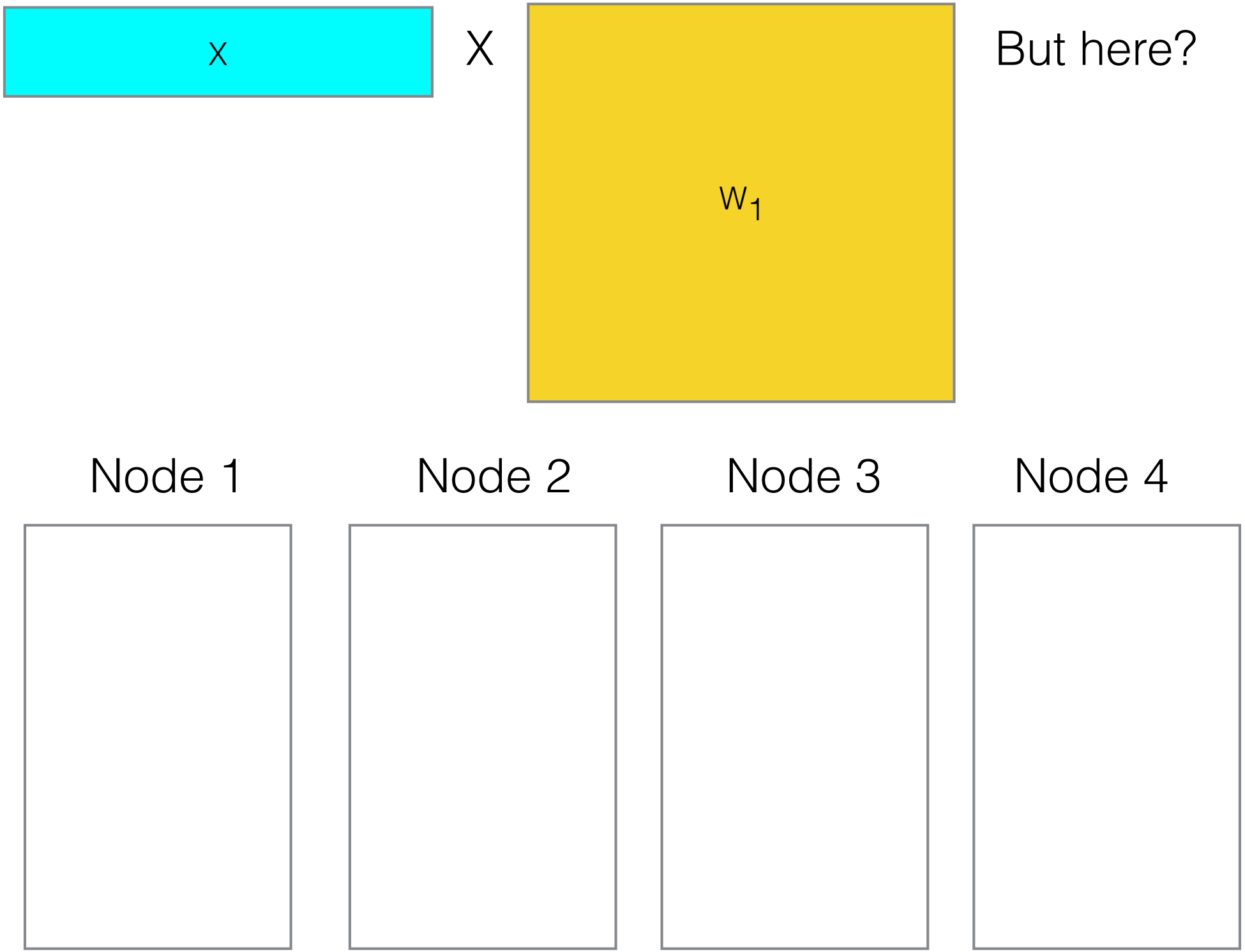


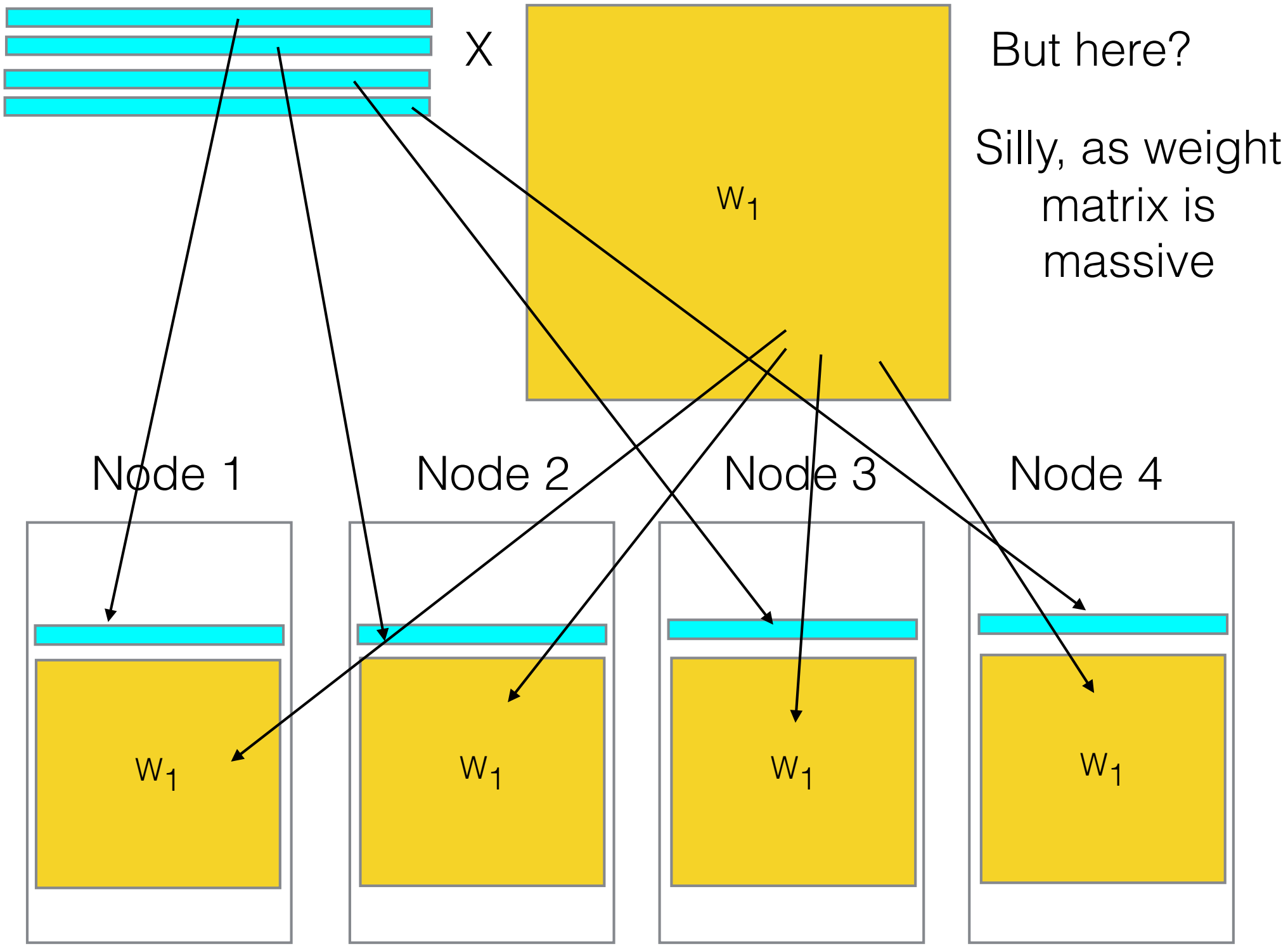


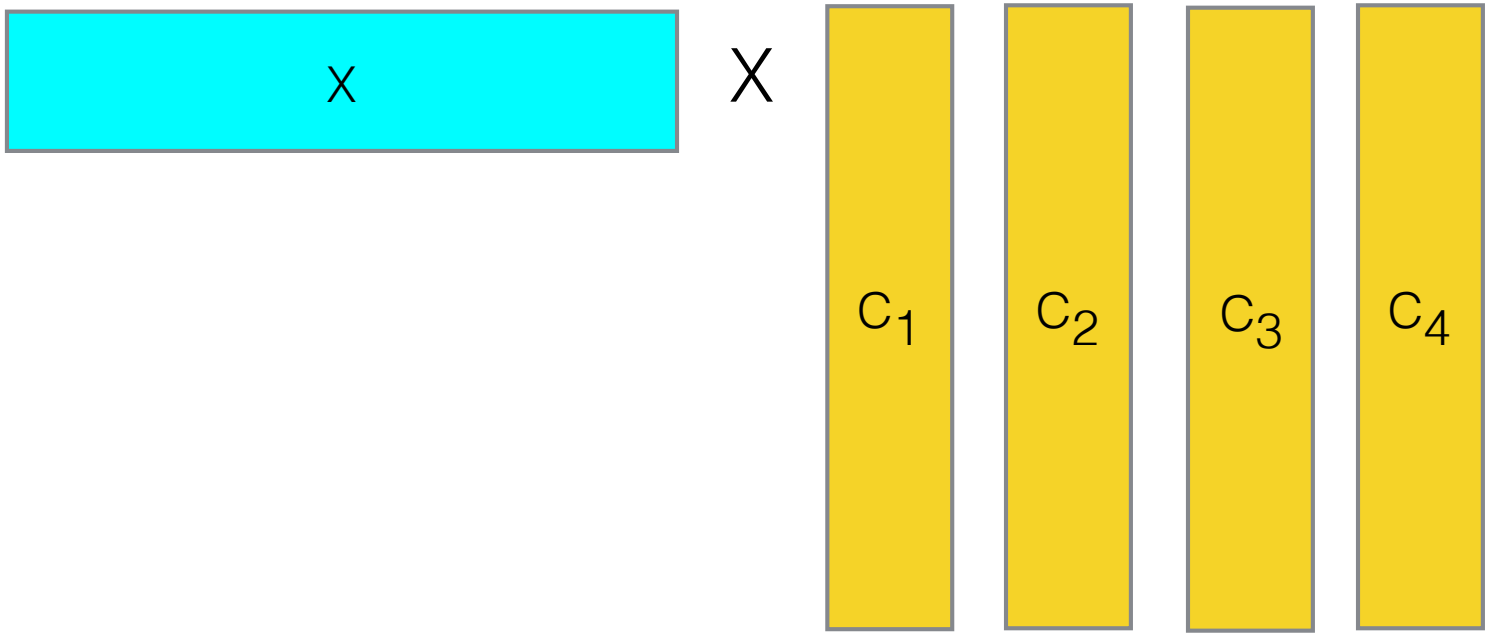

Total xfer:

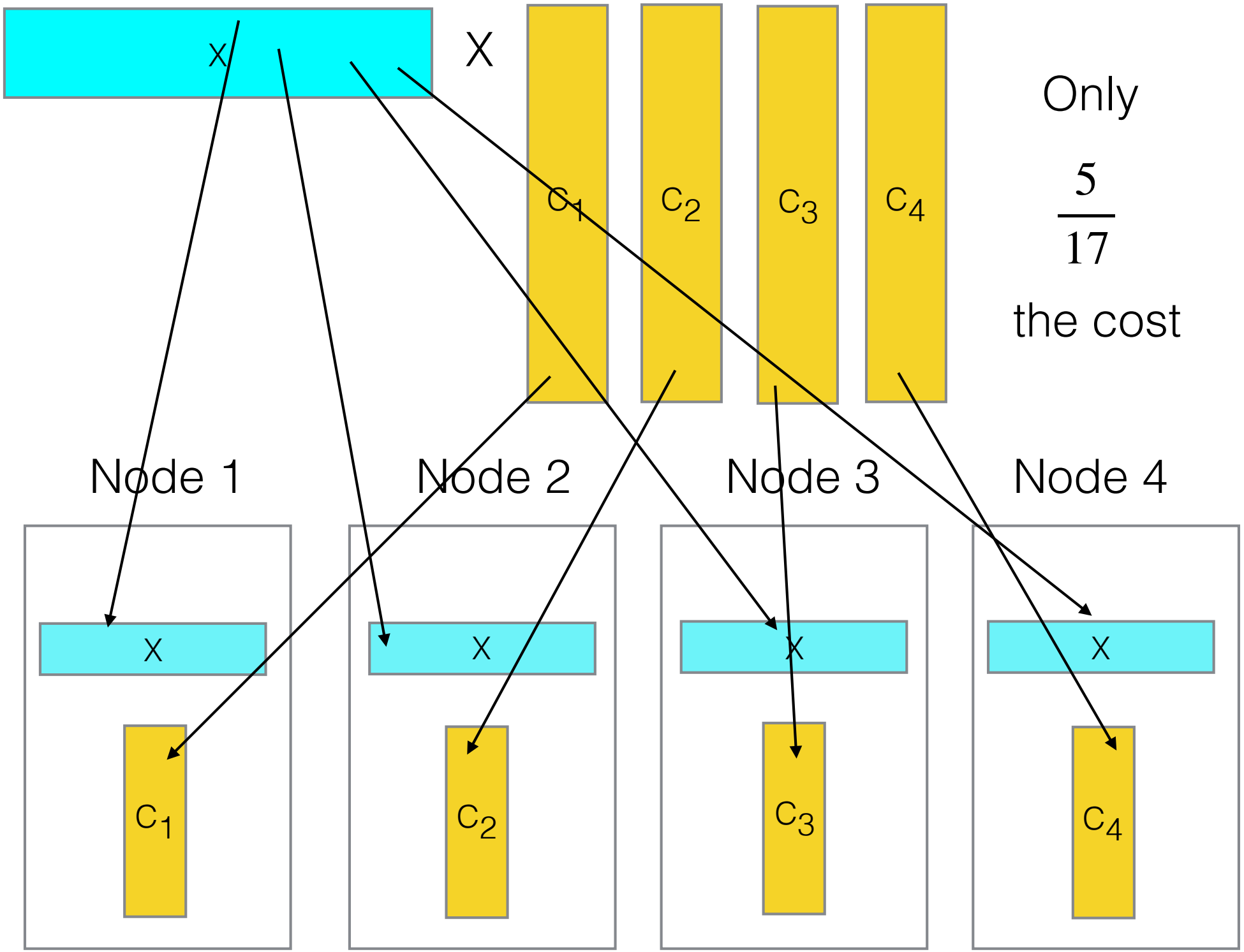
$$|X| + 4|W_1|$$




x
X
$w_1$
But here?
Node 1
Node 2
Node 3
Node 4


X
$W_1$
But here?
Silly, as weight matrix is massive
Node 1
Node 2
Node 3
Node 4
$W_1$
$W_1$
$W_1$
$W_1$


x
X
$c_1$
$c_2$
$c_3$
$c_4$
Instead:
Do this!
Node 1
Node 2
Node 3
Node 4


X
X
$C_1$
$C_2$
$C_3$
$C_4$
Only
$\frac{5}{17}$
the cost
Node 1
Node 2
Node 3
Node 4
X
X
X
X
$C_1$
$C_2$
$C_3$
$C_4$

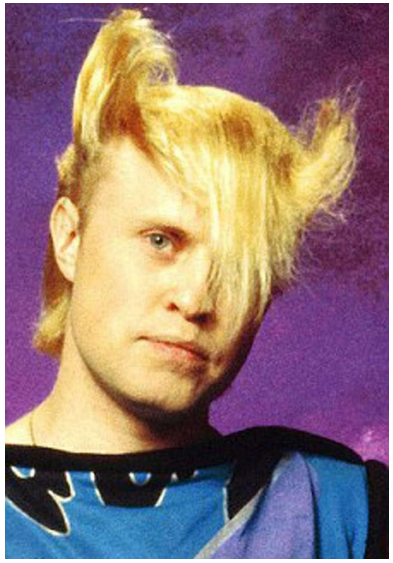
# ML Systems Can't Perform Even this Simple Opt

- Why? As distributed systems, they are poorly designed
  - Computations not abstracted
  - No real attempt at opt
  - Hence everything is on the programmer

# Our Goal

- All programmer does is specify model
  - Exposing the mathematics to the system
- But the system just works
  - System figures out the best implementation for given hardware
  - Automatic decomposition of computation as needed
  - No more "model parallel" or "data parallel"
  - It's just a math computation, run optimally
- How to get there?
  - I'm a database researcher
  - Databases are awesome
  - Time to go back to the 1970's?

# The 70's and 80's Were an Amazing Time!



- At least for database research
- Advent of distributed RDBMS software
  - Arguably the only widely-used distributed/parallel programming systems
  - (OK, might argue for systems like Spark... but Spark is basically a DB)
- Why did they work so well?
  - Beautiful mix of theory and systems
  - 1970's and 80's, people asked a series of foundational questions...

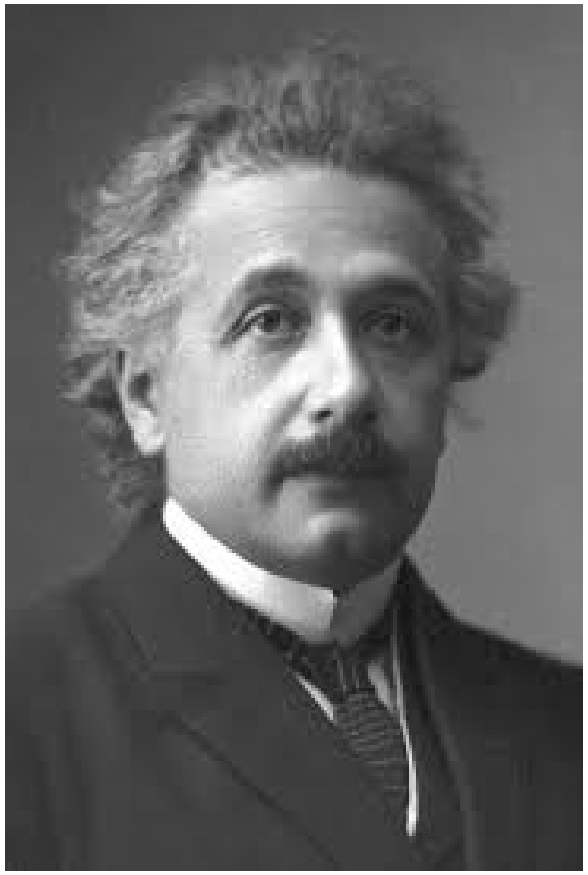
# Foundational Questions To Ask

- (1) What is the programming abstraction for data access?
  - RDBMS Answer: relational calculus and variants (SQL)
  - MLSys Answer: ???
- (2) What is the implementation abstraction for data access?
  - RDBMS Answer: relational algebra
  - MLSys Answer: ???
- (3) How to implement that abstraction?
  - RDBMS Answer: query optimization, indexing, distributed joins, etc.
  - MLSys Answer: ???

# Let's Start Here

- (1) **What is the programming abstraction for data access?**
  - RDBMS Answer: relational calculus and variants (SQL)
  - MLSys Answer: some sort of tensor calculus/summation notation
- (2) What is the implementation abstraction for data access?
  - RDBMS Answer: relational algebra
  - MLSys Answer: ???
- (3) How to implement that abstraction?
  - RDBMS Answer: query optimization, indexing, distributed joins, etc.
  - MLSys Answer: ???

# Example starting point: Einstein notation



- Already some adoption in ML
  - Will refer to as "EinSum"

# Einstein Notation

- To multiply matrices $\mathbf{A}$ and $\mathbf{B}$:

$$\forall_{ik}\mathbf{C}_{ik} \leftarrow \sum_{j} \mathbf{A}_{ij} \times \mathbf{B}_{jk}$$

- Note how this differs from just calling a MatMul
  - ▷ Innards of operation are now visible to the system

# EinSum: Can Implement (Almost) Any ML Comp

- Example: MLP $\mathbf{SoftMax}(\mathbf{W}^{(2)} \times \mathbf{ReLU}(\mathbf{W}^{(1)} \times \mathbf{X}))$

$$\mathbf{A}_i \leftarrow \sum_j \mathbf{W}^{(1)}_{i,j} \times \mathbf{X}_j$$

$$\mathbf{B}_i \leftarrow \sum_{\emptyset} \mathbf{ReLU}(\mathbf{A}_i)$$

$$\mathbf{C}_i \leftarrow \sum_j \mathbf{W}^{(2)}_{i,j} \times \mathbf{B}_j$$

$$\mathbf{D}_\emptyset \leftarrow \sum_i \exp(\mathbf{C}_i)$$

$$\mathbf{E}_i \leftarrow \sum_{\emptyset} \frac{\exp(\mathbf{C}_i)}{\mathbf{D}_\emptyset}$$

# What About the Implementation Abstraction?

- (1) What is the programming abstraction for data access?
  - RDBMS Answer: relational calculus and variants (SQL)
  - MLSys Answer: some sort of tensor calculus/summation notation
- (2) **What is the implementation abstraction for data access?**
  - RDBMS Answer: relational algebra
  - MLSys Answer: tensor relational algebra
- (3) How to implement that abstraction?
  - RDBMS Answer: query optimization, indexing, distributed joins, etc.
  - MLSys Answer: ???

# Tensor Reltional Algebra (TRA)

- Algebra over ***tensor relations***
- What is a tensor relation?
- Informally...
  - One takes a rank $r$ tensor (array)
  - And represents it as a set of rank $m \leq r$ sub-tensors

# Tensor Relations

- Example tensor relation:
  - Consider the matrix $\mathbf{A}$

$$\begin{bmatrix} 1.4 & 2.2 & 1.2 & 2.1 \\ 2.3 & 2.6 & 1.1 & 2.2 \\ 1.4 & 1.0 & 1.1 & 1.4 \\ 1.1 & 1.4 & 2.5 & 2.3 \end{bmatrix}$$

  - Decompose $\mathbf{A}$ into a set of $(\texttt{rowID}, \texttt{colID}, \texttt{chunk})$ triples:

$$\bar{\mathbb{R}} = \left\{ \left( \langle 1,1 \rangle, \begin{bmatrix} 1.4 & 1.2 \\ 2.3 & 2.6 \end{bmatrix} \right), \left( \langle 1,2 \rangle, \begin{bmatrix} 1.2 & 2.1 \\ 1.1 & 2.2 \end{bmatrix} \right), \right.$$
$$\left. \left( \langle 2,1 \rangle, \begin{bmatrix} 1.4 & 1.0 \\ 1.1 & 1.4 \end{bmatrix} \right), \left( \langle 2,2 \rangle, \begin{bmatrix} 1.1 & 1.4 \\ 2.5 & 2.3 \end{bmatrix} \right) \right\}$$

# The Algebra

- Won't formally define it
  - But is a lot like algebra executed by modern RDBMSs
  - Easiest to think of this is SQL over tensor relations
  - (though not totally accurate)

- Example MatMul over tensor relations in SQL:

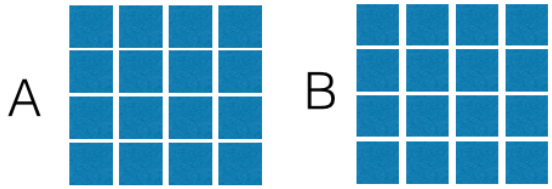
```
SELECT lhs.rowID, rhs.colID
  mat_sum (mat_mul (lhs.chunk, rhs.chunk))
FROM A AS lhs, B AS rhs
WHERE lhs.colID = rhs.rowID
GROUP BY lhs.rowID, rhs.colID
```

# Schema Design Has Always Been Important In DBs

- No different in tensor relational systems
  - There are many ways to implement the same MatMul relationally

# What Are the Options?

- We might fully decompose inputs into scalars:

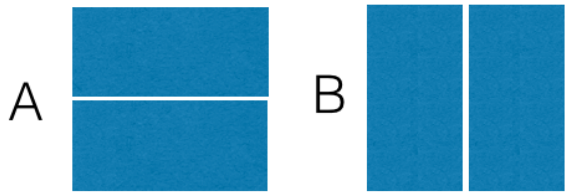



- Then TRA is:

```
SELECT lhs.rowID, rhs.colID
  SUM (lhs.val * rhs.vl)
FROM A AS lhs, B AS rhs
WHERE lhs.colID = rhs.rowID
GROUP BY lhs.rowID, rhs.colID
```

# Another Option

- Or decompose $\mathbf{A}$ into row strips and $\mathbf{B}$ into col strips:

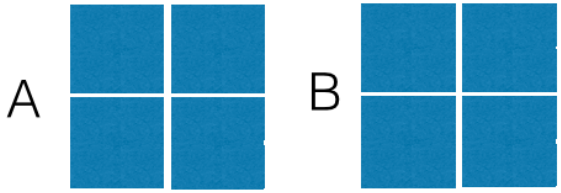



- Then TRA is:

```
SELECT lhs.rowID, rhs.colID
  mat_mul (lhs.chunk, rhs.chunk)
FROM A AS lhs, B AS rhs
WHERE lhs.colID = rhs.rowID
```

# Yet Another

- Or decompose **A** and **B** into chunks:




- Then TRA is:

```
SELECT lhs.rowID, rhs.colID
  mat_sum (mat_mul (lhs.chunk, rhs.chunk))
FROM A AS lhs, B AS rhs
WHERE lhs.colID = rhs.rowID
GROUP BY lhs.rowID, rhs.colID
```

# Which Implementation Is Best?

- Each implementation has its own performance characteristics
- Fully relational (relations store scalars) is sometimes good, why?
  - The most cross-device parallelism
  - Take full advantage of sparsity
- Fully tensor (relations have one tuple with a full tensor) is sometimes good, why?
  - Push fewer tuples through the system
  - Less communication
  - Take full advantage of accelerators
- We need to be able to systematically explore these options
  - How?
  - Let's do it now...

# Inner Indices

- Example tensor $\mathbf{U}$
  - Three indices that range from 0 to 7, from 0 to 3, and from 0 to 3
  - The bound $\mathbf{b}$ of the tensor (aka the shape) is a vector $\langle 8, 4, 4 \rangle$.
- To access the item at $\mathbf{i} = \langle i, j, k \rangle$ we use the standard notation:

$$\mathbf{U}_{ijk}$$

Or:

$$\mathbf{U}_{\mathbf{i}}$$

- Call $\mathbf{i}$ a vector of "inner" indices

# Outer Indices

- Introduce the notion of "outer" indices
  - $\mathbf{U}$ could also be represented as $\bar{\mathbf{U}}$ with outer indices and inner indices
  - Let:

$$\bar{\mathbf{U}} \equiv \mathbf{U}_{/\mathbf{d}}$$

  - Then if $\times$, $/$ and $+$ are applied operator-wise, then:

$$\bar{\mathbf{U}}^{\mathbf{o}}_{\mathbf{i}} = \mathbf{U}_{\mathbf{i}+\mathbf{o}\times\frac{\mathbf{b}}{\mathbf{d}}}$$

# Tensor Relations Revisited

- Why do we do this?
- $\bar{\mathbf{U}} \equiv \mathbf{U}_{/\mathbf{d}}$ is a tensor-relational representation of $\mathbf{U}$
  - The tensor $\mathbf{U}$ is broken into $\prod_i \mathbf{d}_i$ "tiles"
  - Each tile is a tuple in the relation
  - The outer indices of $\bar{\mathbf{U}}$ serve as keys for the relation
  - The inner indices of $\bar{\mathbf{U}}$ serve index into tensors in a relation
- Example:

$$\left(\mathbf{U}_{/\langle 4,4,4\rangle}\right)^{3,2,1}_{1,0,1}$$

  - Refers to the value at position $\langle 1, 0, 1\rangle$ in the tensor
  - In the tuple with key $\langle 3, 2, 1\rangle$ (64 tuples if $\mathbf{b} = \langle 8, 4, 8\rangle$)

# Trivial Indices

- Convention: drop trivial (useless) indices
  - The $i$th outer index is trivial when $\mathbf{d}_i = 1$ (no slicing)
  - The $i$th inner index is trivial when $\mathbf{d}_i = \mathbf{b}_i$ (full slicing)
  - So we write $\left(\mathbf{U}_{/\langle 4,4,4\rangle}\right)^{3,2,1}_{1,0,1}$ as $\left(\mathbf{U}_{/\langle 4,4,4\rangle}\right)^{3,2,1}_{1,1}$

# Why Represent Tenor Relations Like This?

- Can now define tensor-relational re-writes over Einstein notation
  - That allow us to move between tensor-relational expressions
  - Consider matrix multiply:

$$\mathbf{C}_{i,j} \leftarrow \sum_{k} \mathbf{A}_{i,k} \times \mathbf{B}_{k,j}$$

  - This is the same as:

$$\left(\mathbf{C}_{/\langle 1,1 \rangle}\right)_{i,j} \leftarrow \sum_{k} \left(\mathbf{A}_{/\langle 1,1 \rangle}\right)_{i,k} \times \left(\mathbf{B}_{/\langle 1,1 \rangle}\right)_{k,j}$$

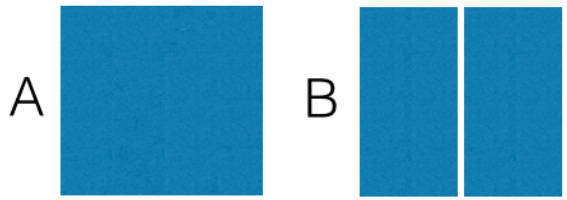
# This Re-Write Gives Us a New TRA Expression

- Now split 2nd dim of $\mathbf{B}$:

$$\left(\mathbf{C}_{/\langle 1,2\rangle}\right)^{a}_{i,j} \leftarrow \sum_{k} \left(\mathbf{A}_{/\langle 1,1\rangle}\right)_{i,k} \times \left(\mathbf{B}_{/\langle 1,2\rangle}\right)^{a}_{k,j}$$

  - Let the kernel function $K(\mathbf{A}, \mathbf{B})$ compute $\mathbf{C}_{i,j} \leftarrow \sum_k \mathbf{A}_{i,k} \times \mathbf{B}_{k,j}$
  - Then the above Einstein notation can be re-written to:
  - `SELECT` $\mathbf{B}.a$, `SUM` $(K(\mathbf{A}.\texttt{array}, \mathbf{B}.\texttt{array}))$ `FROM` $\mathbf{A}$, $\mathbf{B}$ `GROUP BY` $\mathbf{B}.a$.

- Is a mult of the following two tensor relations:



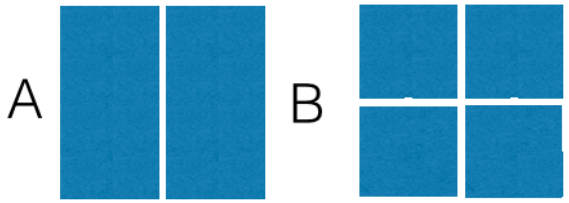
# We Can Keep Going!

- We can perform one more re-write to get:

$$\left(\mathbf{C}_{/\langle 1,2\rangle}\right)^{a}_{i,j} \leftarrow \sum_{b}\sum_{k}\left(\mathbf{A}_{/\langle 1,2\rangle}\right)^{b}_{i,k} \times \left(\mathbf{B}_{/\langle 2,2\rangle}\right)^{b,a}_{k,j}$$

  - Use the same kernel function (as the inner sum does not change)
  - Equivalent to the SQL: `SELECT` $\mathbf{B}.a$, `SUM` $(K(\mathbf{A}.\texttt{array}, \mathbf{B}.\texttt{array}))$ `FROM` $\mathbf{A}$, $\mathbf{B}$ `WHERE` $\mathbf{A}.b = \mathbf{B}.b$ `GROUP BY` $\mathbf{B}.a$.

- Is a mult of the following two tensor relations:



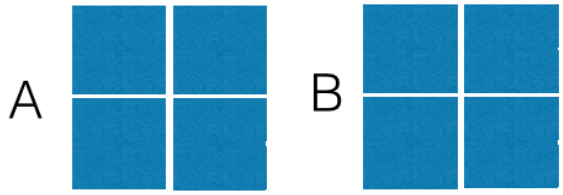
# How About One More?

- Another re-write gives:

$$\left(\mathbf{C}_{/\langle 2,2\rangle}\right)_{i,j}^{c,a} \leftarrow \sum_{b}\sum_{k}\left(\mathbf{A}_{/\langle 2,2\rangle}\right)_{i,k}^{c,b} \times \left(\mathbf{B}_{/\langle 2,2\rangle}\right)_{k,j}^{b,a}$$

  - Again uses the same kernel function
  - Equivalent to the SQL:
  - SELECT $\mathbf{A}.c$, $\mathbf{B}.a$, SUM $(K(\mathbf{A}.\texttt{array}, \mathbf{B}.\texttt{array}))$ FROM $\mathbf{A}$, $\mathbf{B}$ WHERE $\mathbf{A}.b = \mathbf{B}.b$ GROUP BY $\mathbf{A}.c$, $\mathbf{B}.a$.

- Is a mult of the following two tensor relations:



# Given a Tree of Binary EinSum Ops...

## Optimizing schema design is (somewhat) tractible

- Number of ways to choose $\mathbf{d}$ so $|\mathrm{U}_{/\mathrm{d}}| = 2^n$ is small
  - If we assume entries in $\mathbf{d}$ are powers of 2
  - Powers of 2 not a big resriction
  - Choose $2^n = c\times$ number of GPUs/machines
  - For $d$ dim tensor only $\frac{(n+d-1)!}{n!(d-1)!}$ relational designs
  - Ex: $n = 10$ (1024 processors) and $d = 6$ only 3003 designs

- Suggests a DP algorithm
  - Given a cost model for joins/aggs
  - And a cost model for repartitioning
  - For each EinSum op compute best cost for left and right inputs
  - Over all possible schema designs for left and right inputs
  - Brute force compute best for each possible output design

# Some Challenges

- The kernel specs generated during this process may be arbitrary
  - Need the ability to generate efficient accelerator implementations
  - With no human involvement
- Don't always have a tree of binary expressions
  - DAGs more often than not (backprop)
  - And not always binary

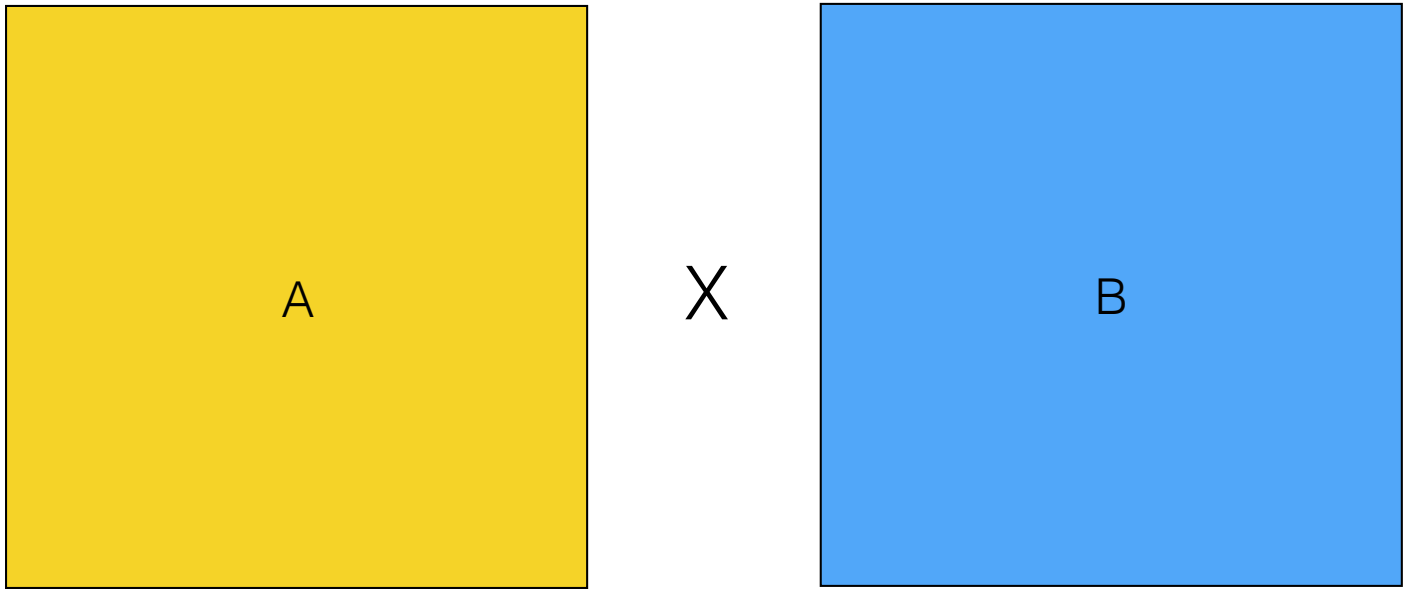
# Finally: What About Implementation?

- (1) What is the programming abstraction for data access?
  - RDBMS Answer: relational calculus and variants (SQL)
  - MLSys Answer: some sort of tensor calculus/summation notation
- (2) What is the implementation abstraction for data access?
  - RDBMS Answer: relational algebra
  - MLSys Answer: tensor relational algebra
- (3) **How to implement that abstraction?**
  - RDBMS Answer: query optimization, indexing, distributed joins, etc.
  - MLSys Answer: new reltional system: the TOS???

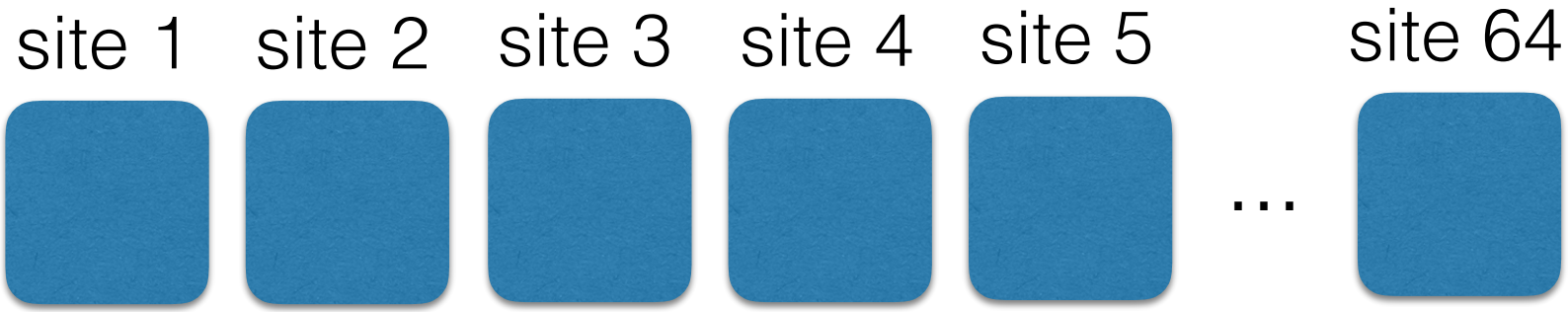
# Can We Just Re-Use Relational RDBMS

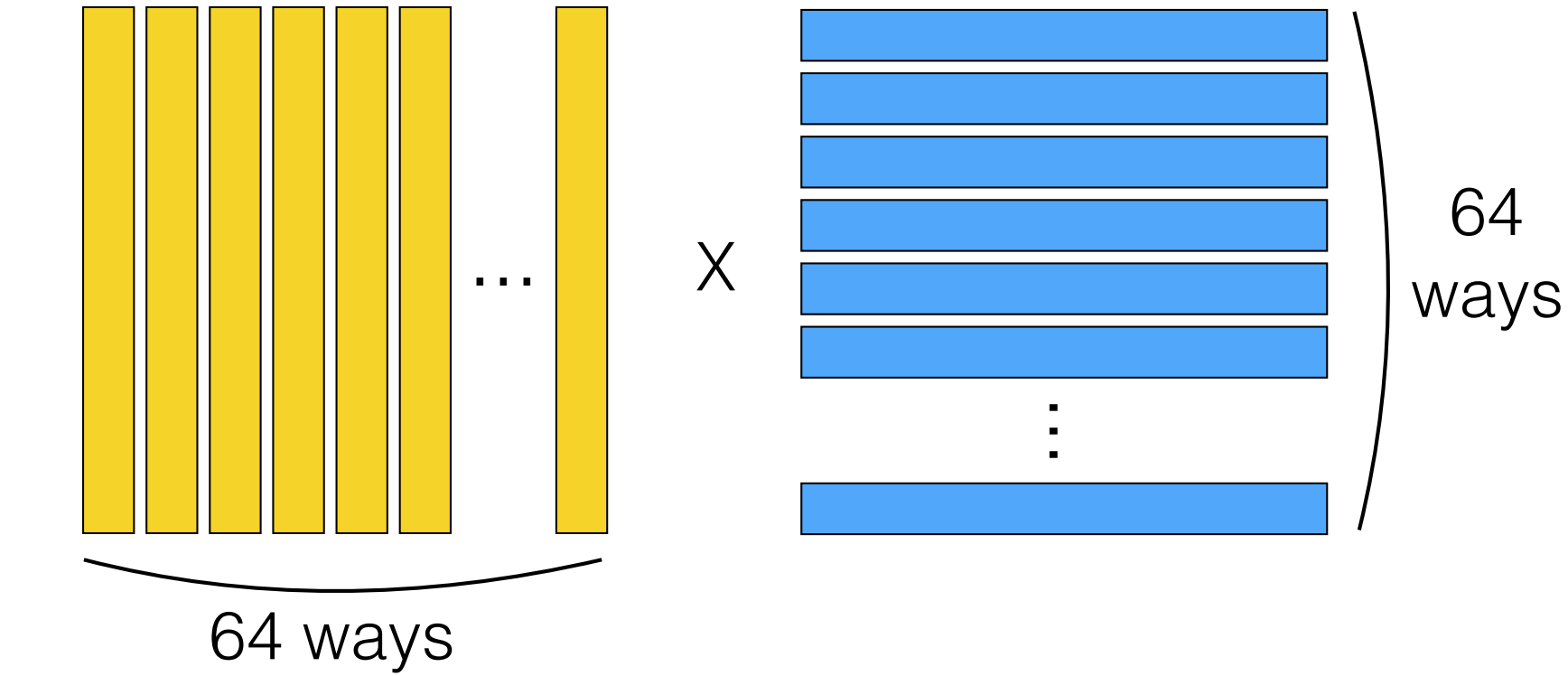
- Nope! RDBMS is operator-centric
- Not data-centric: problem for ML
- Consider matrix mutiplication
  - One join followed by aggregation
  - Say we have $n$ processors
  - How would a hash join/hash-based agg work?

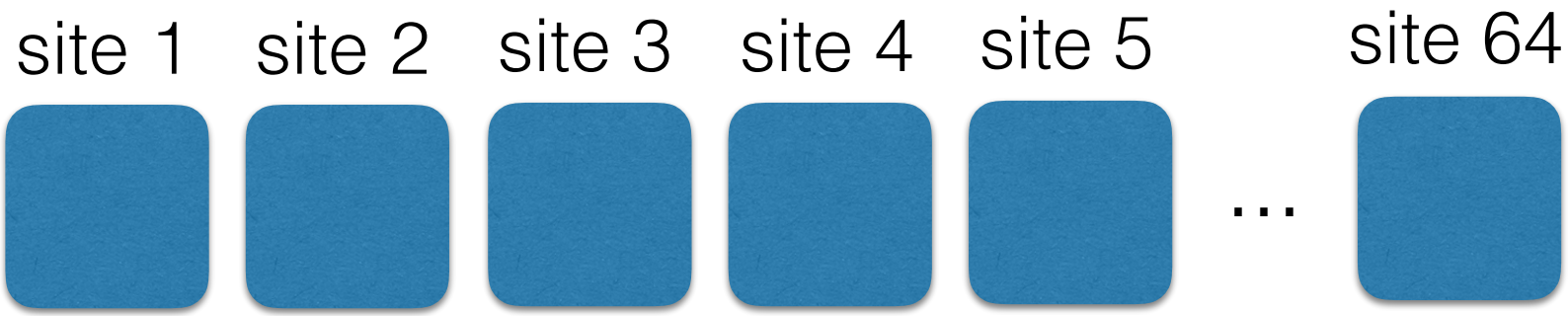
# We want to run this multiply on 64 processors

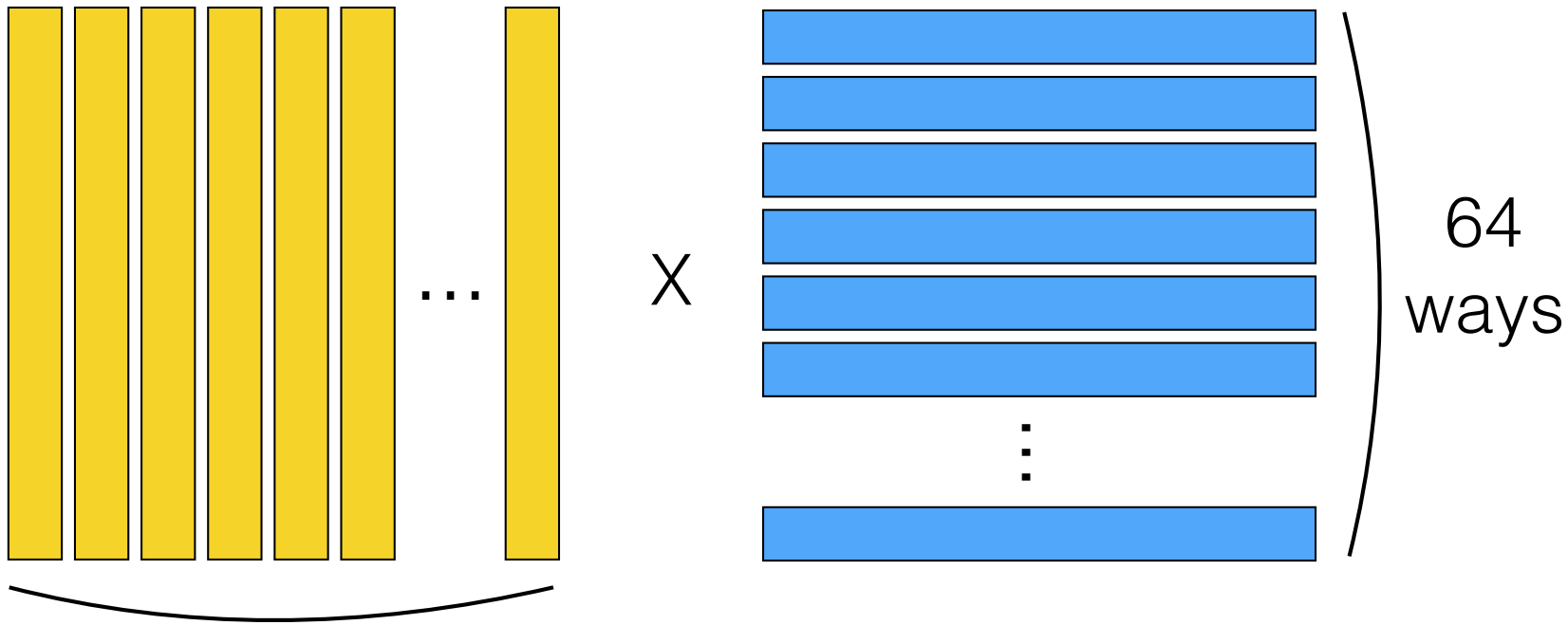



*First let's consider what a relational database could do*

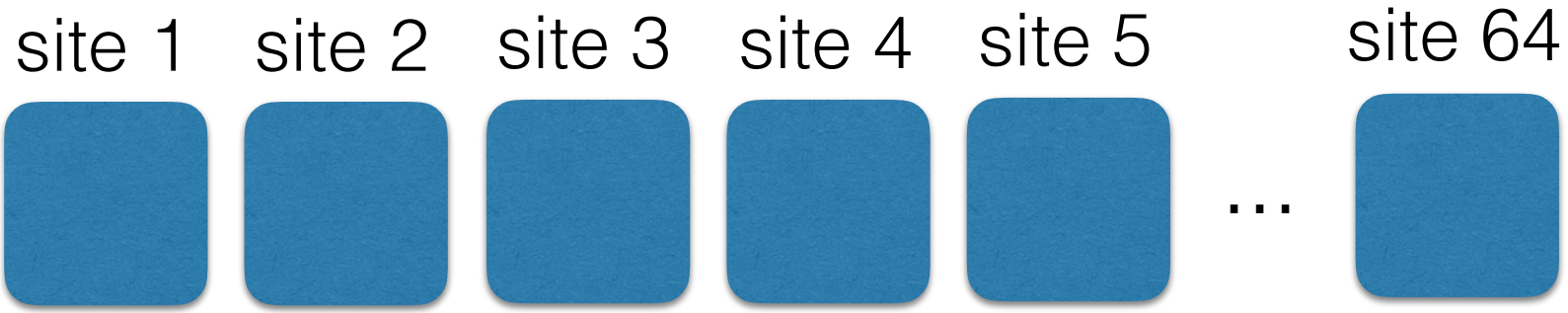


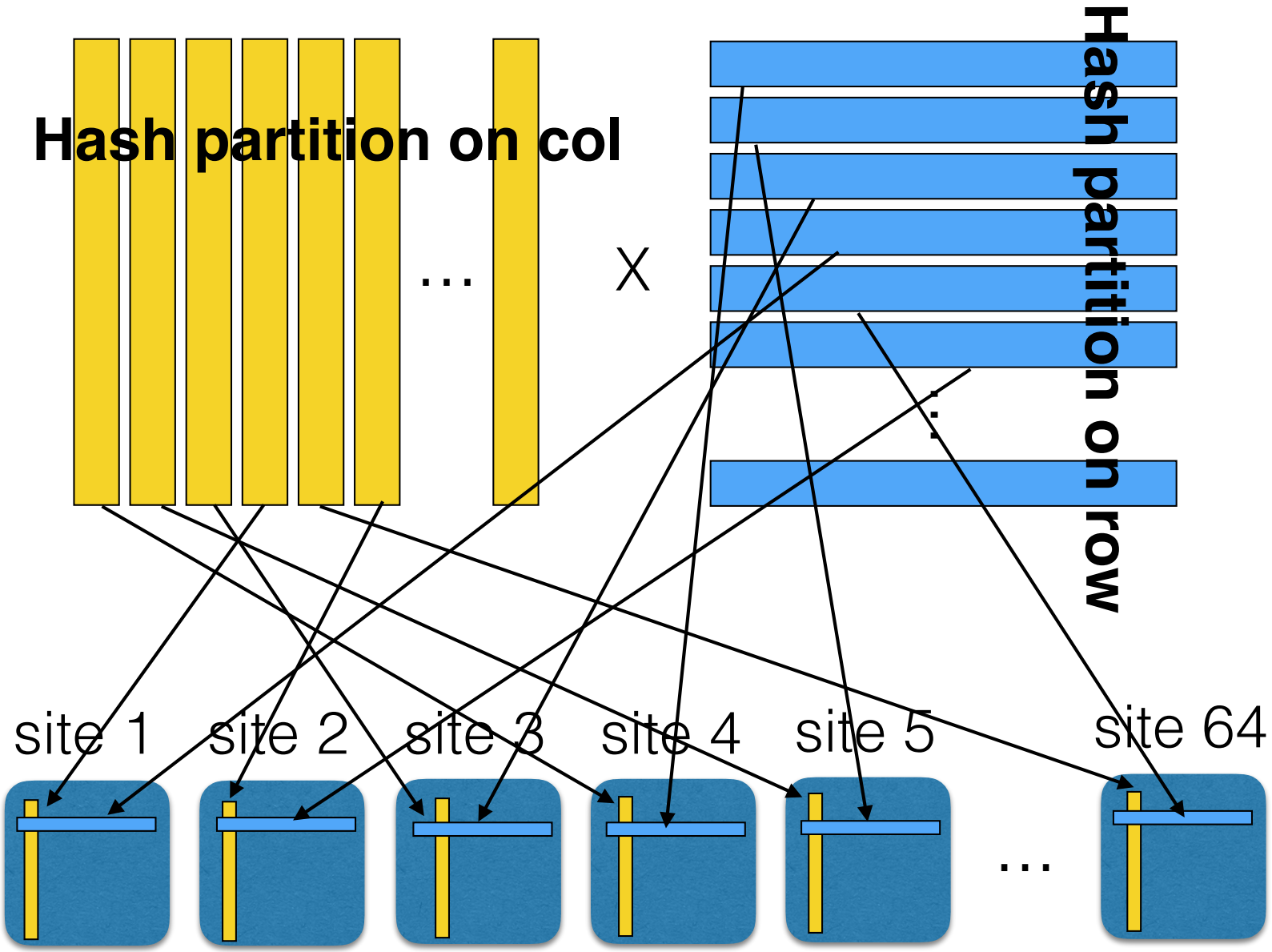

...
64 ways
X
64
ways

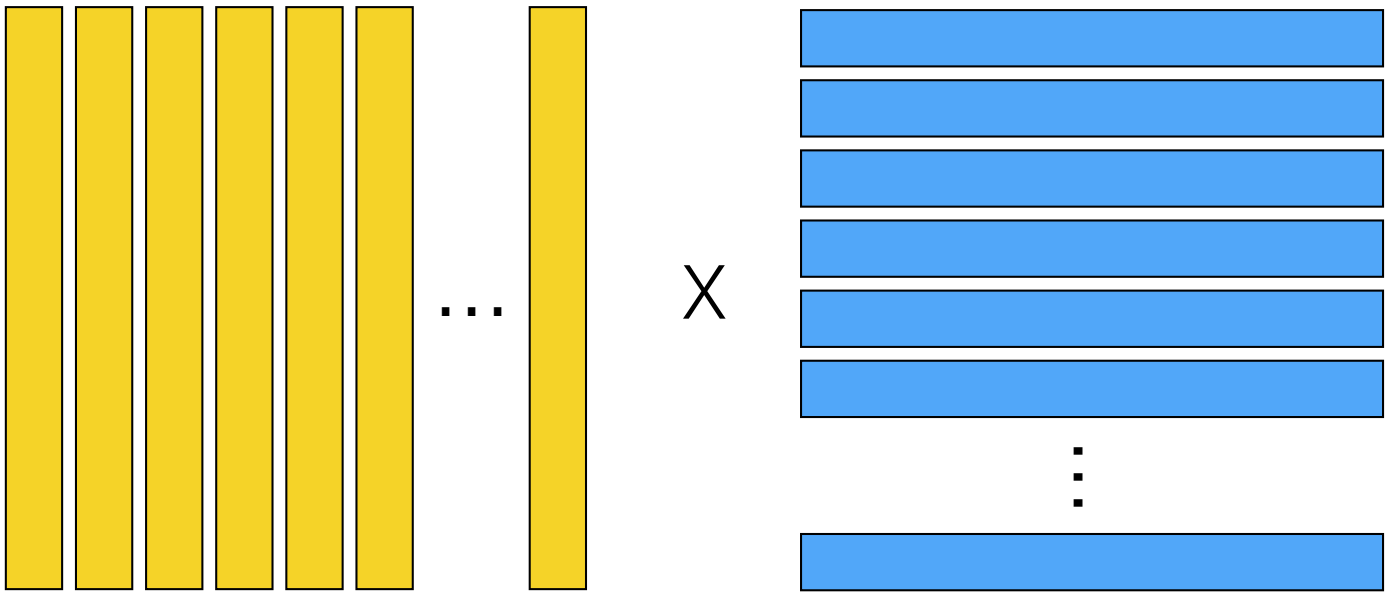

site 1
site 2
site 3
site 4
site 5
...
site 64



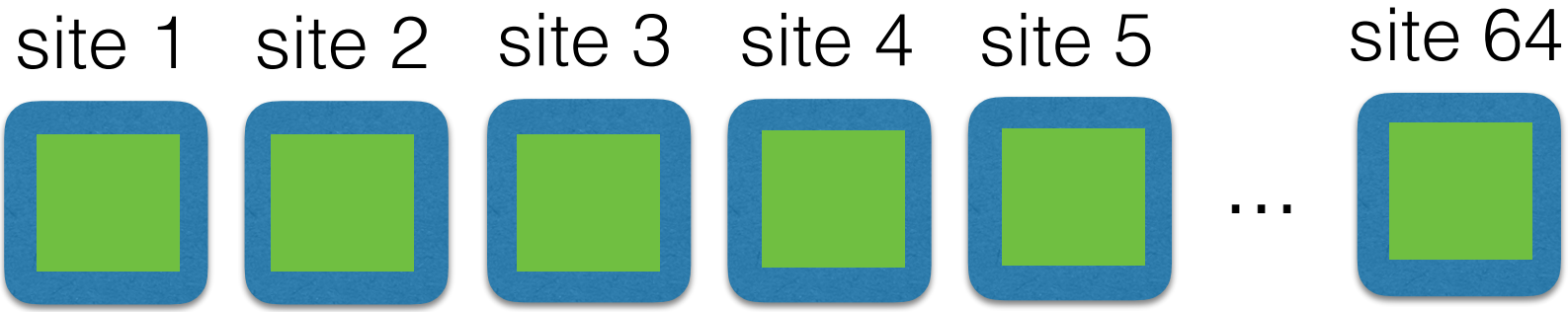

**Now run a hash join**

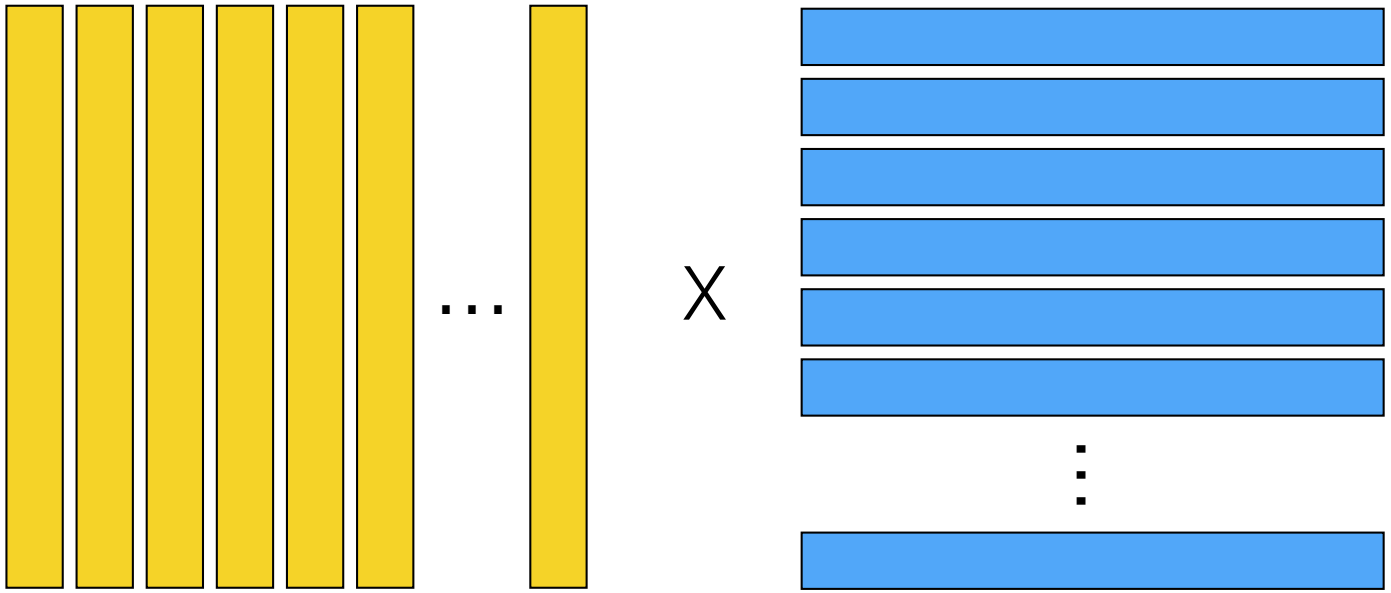



Hash partition on col
...
X
Hash partition on row
site 1
site 2
site 3
site 4
site 5
...
site 64

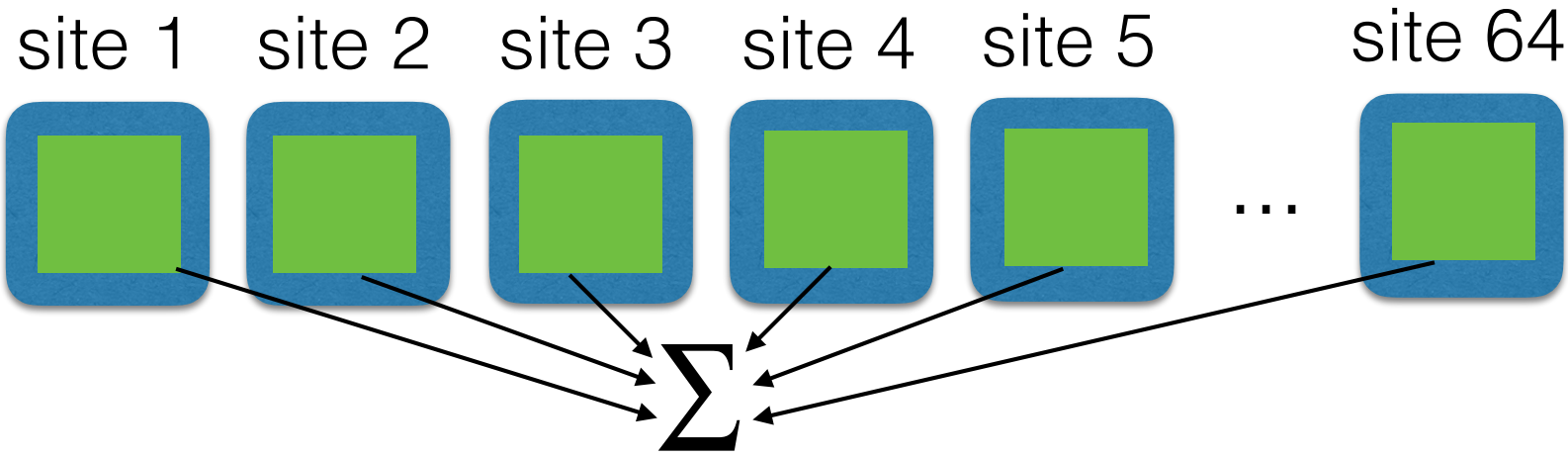


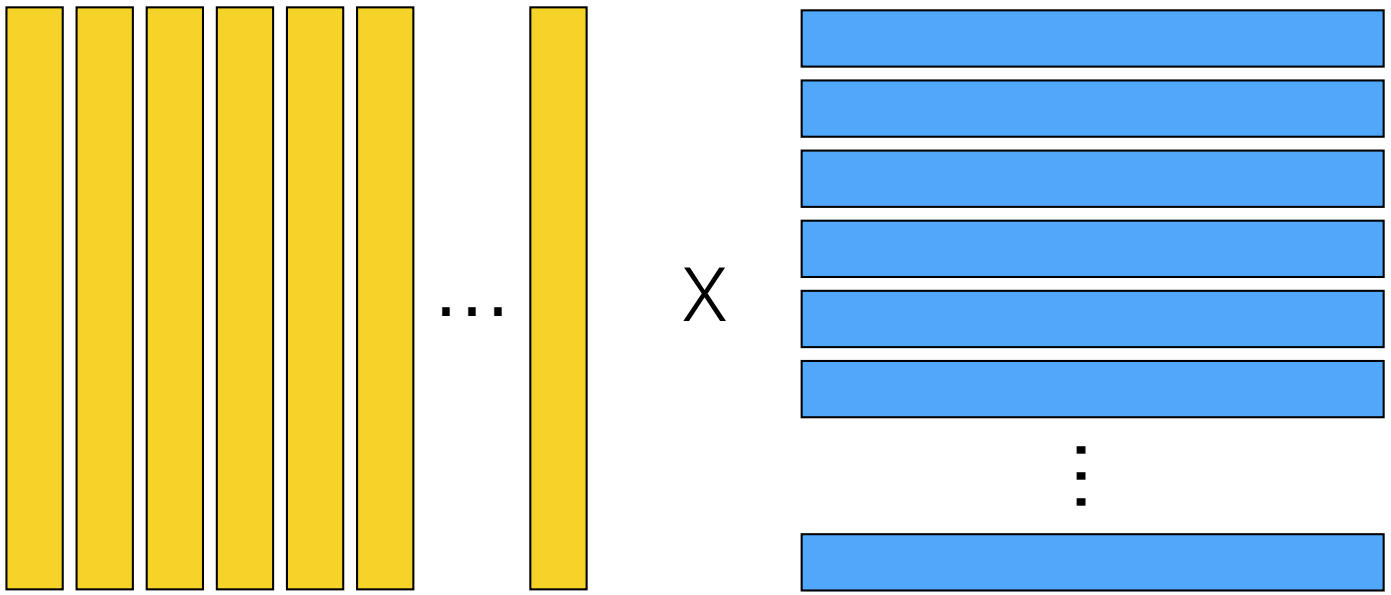
**Local multiply**





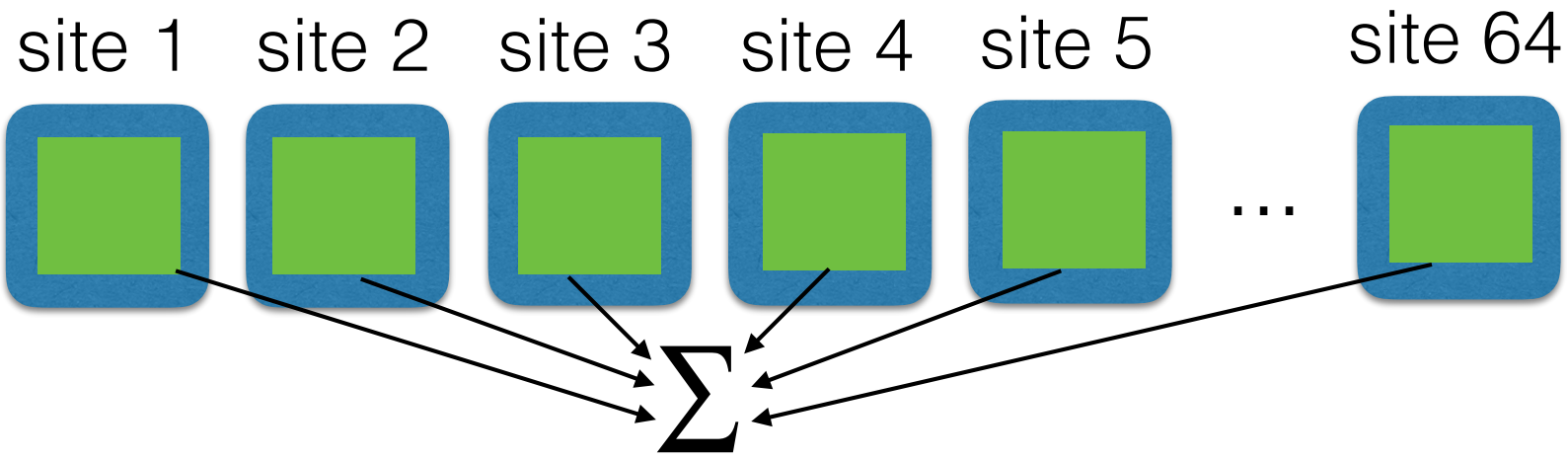

**And aggregate**

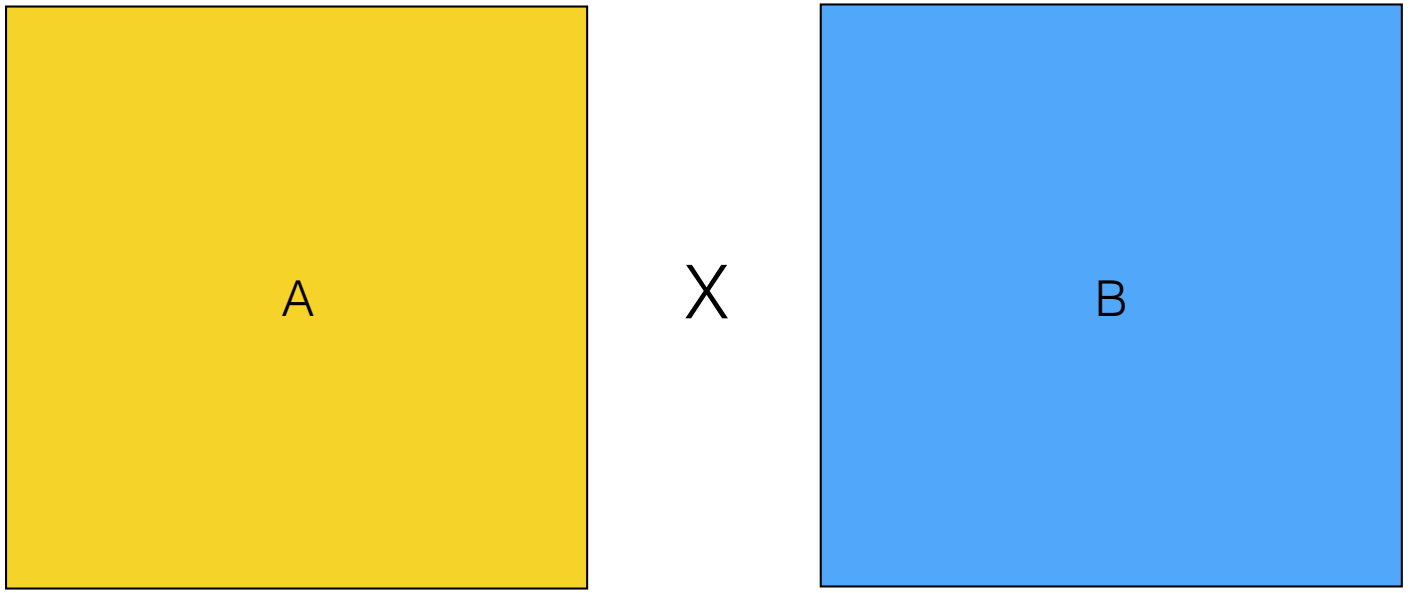




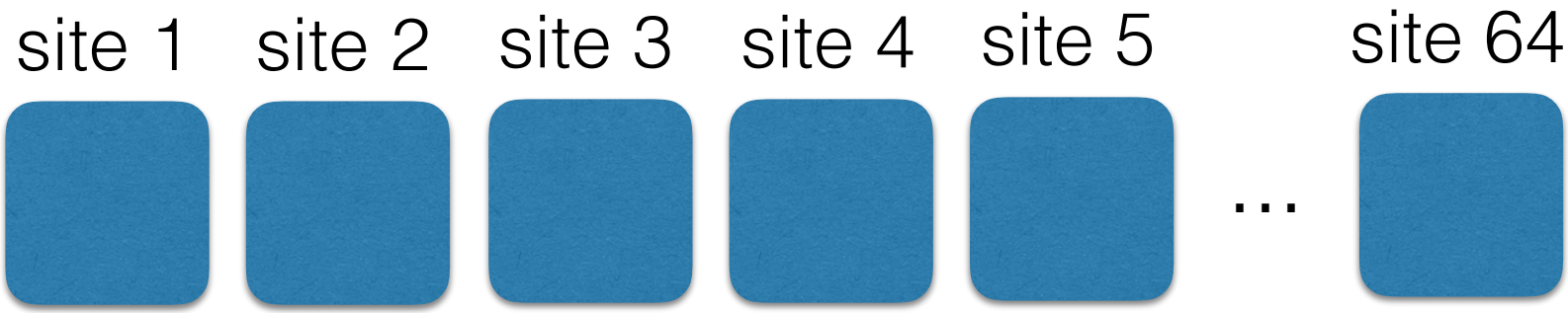

**Total cost:** |A| + |B| + 64|C|

**Not great**

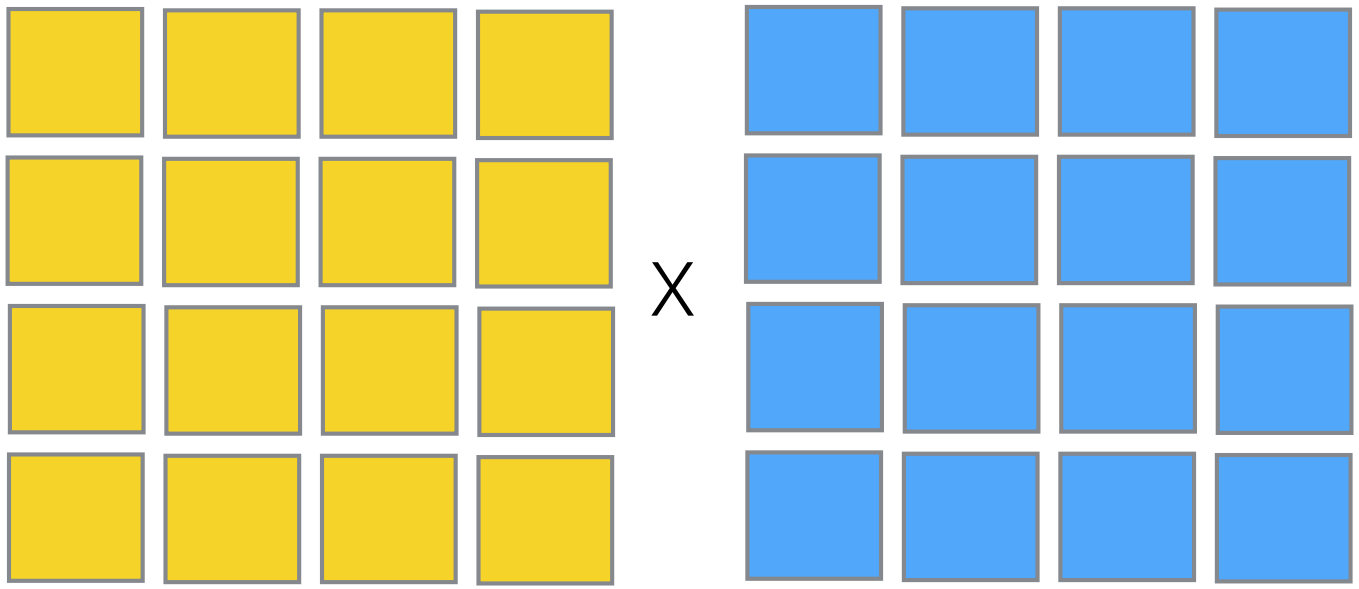




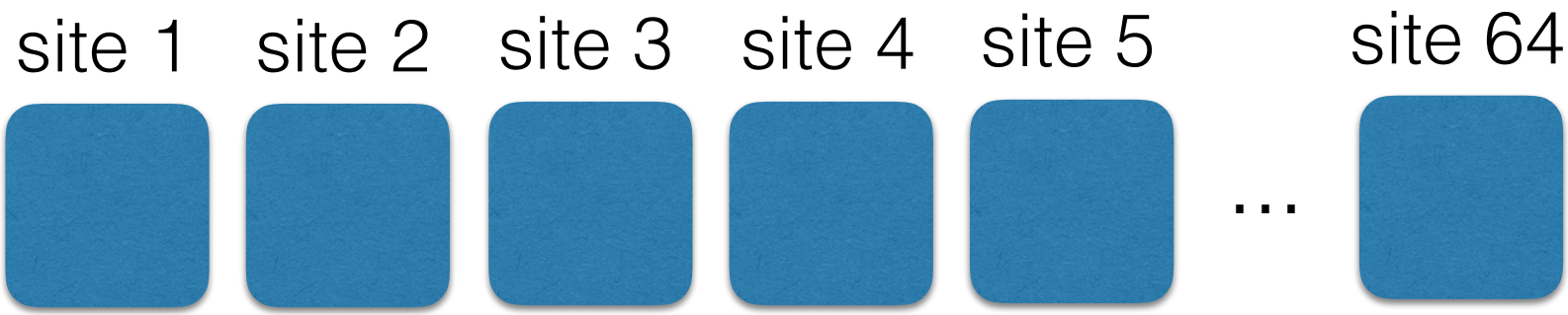

*Now let's consider a much better algorithm*

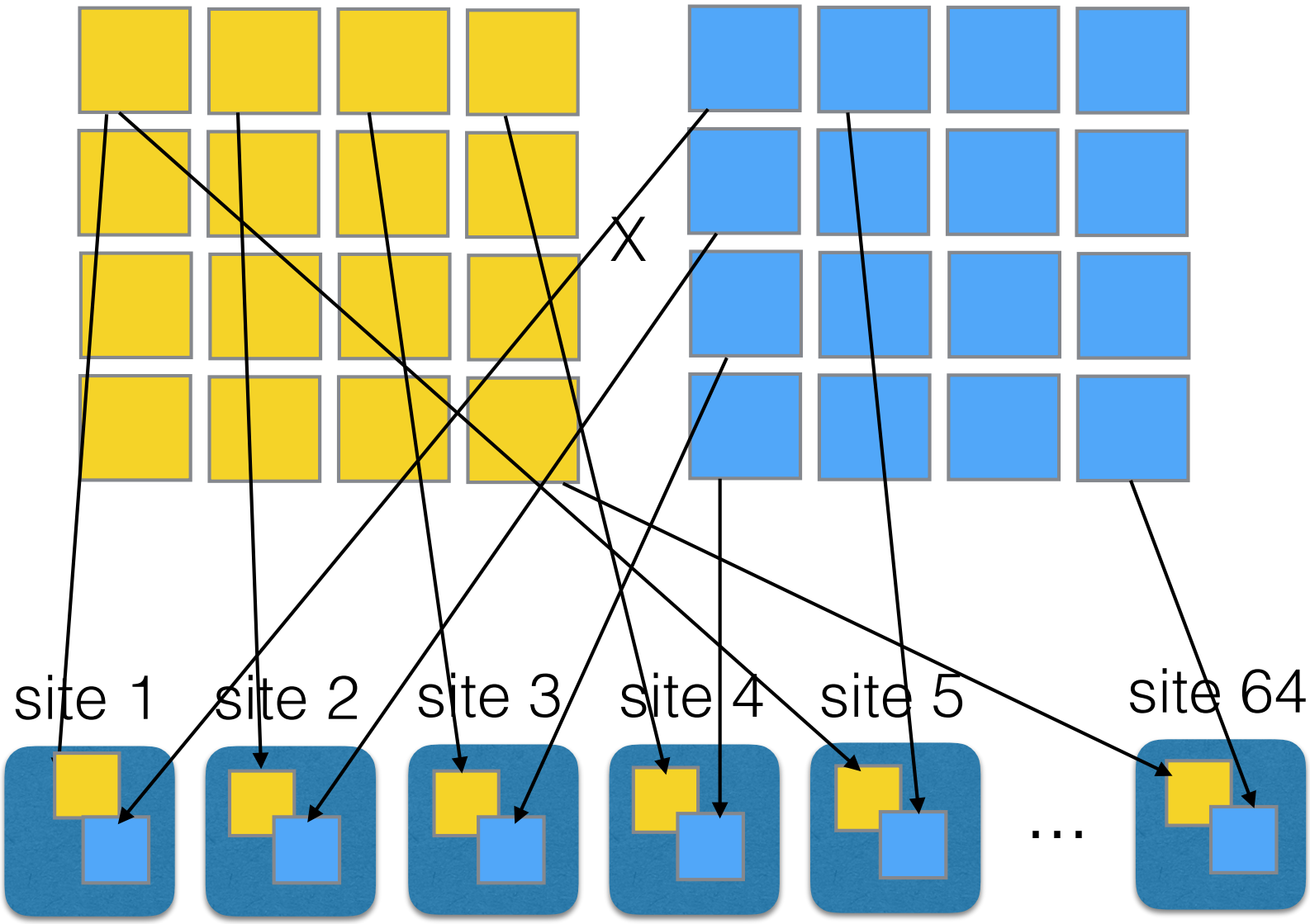


## 4 X 4 grid

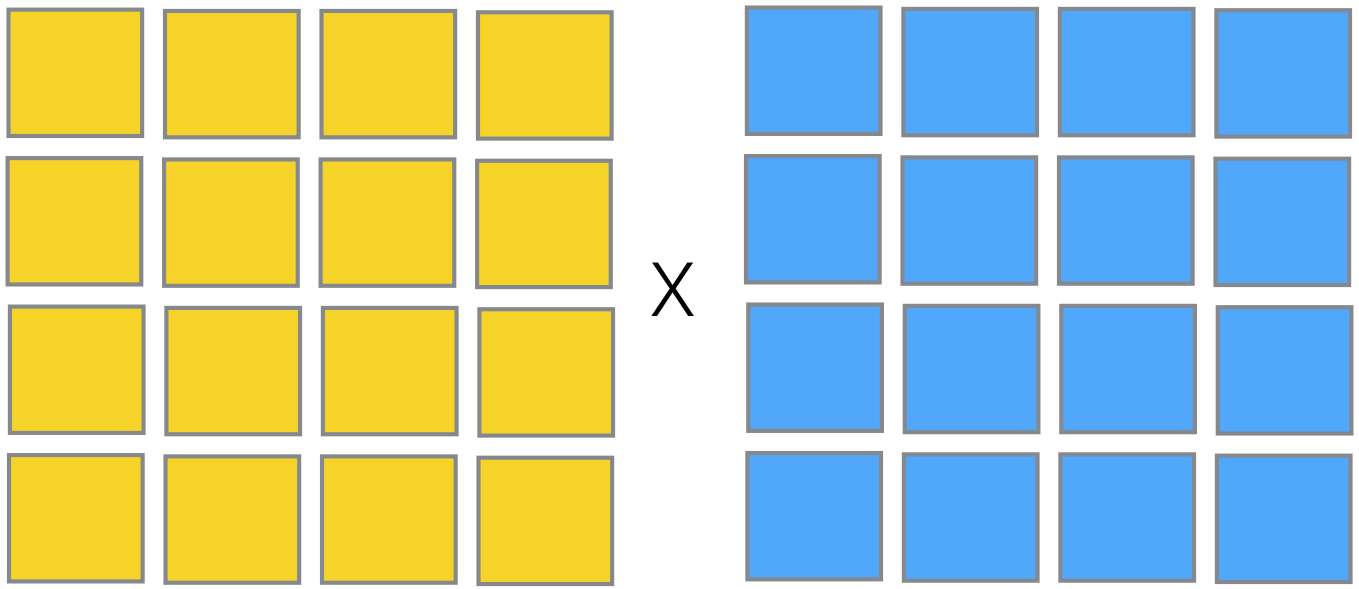


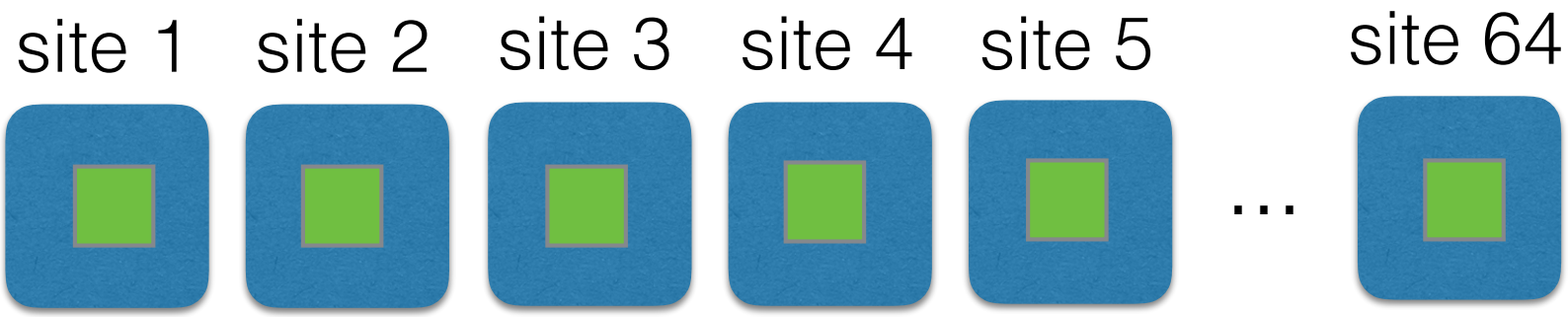
## Don’t partition keys: *place tuples*



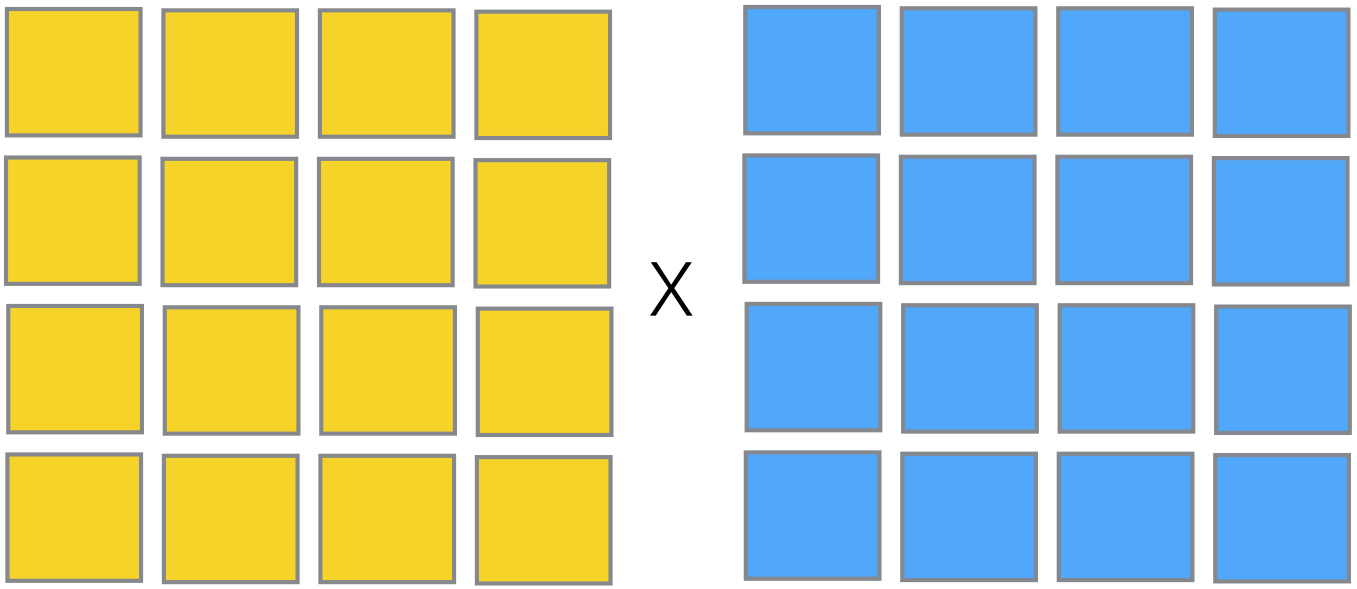



**Each tuple to four locations**

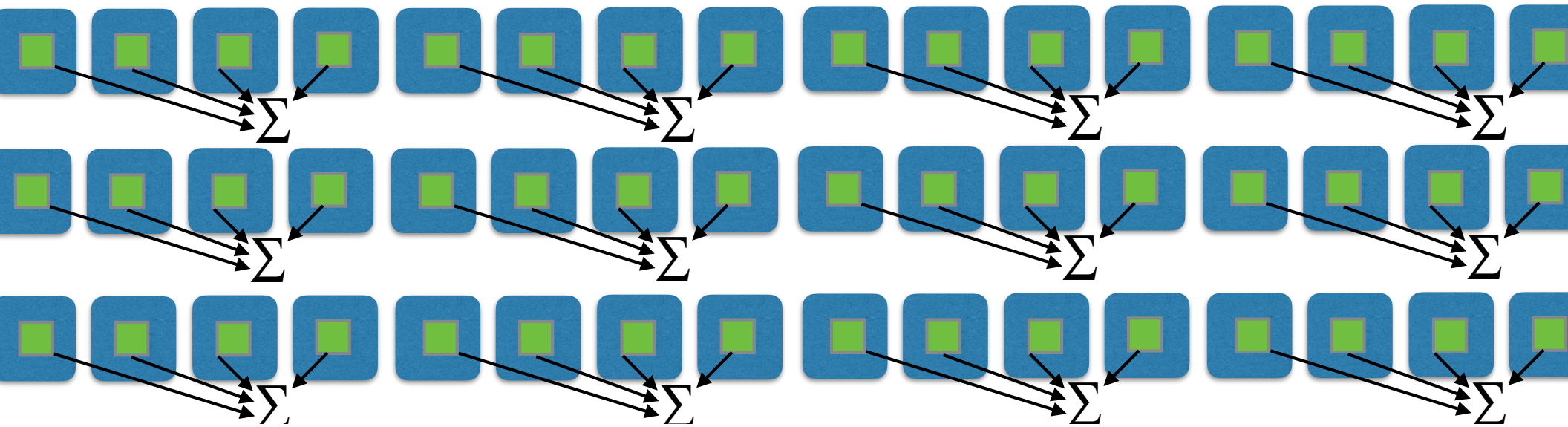


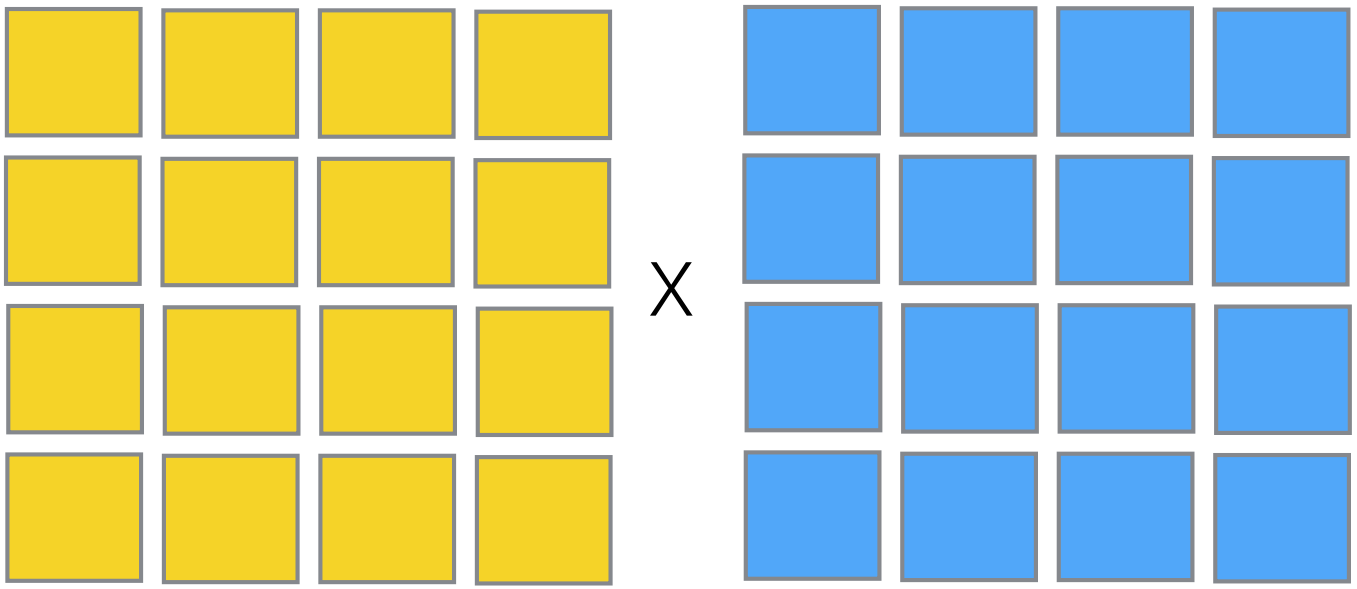

**Local multiply**



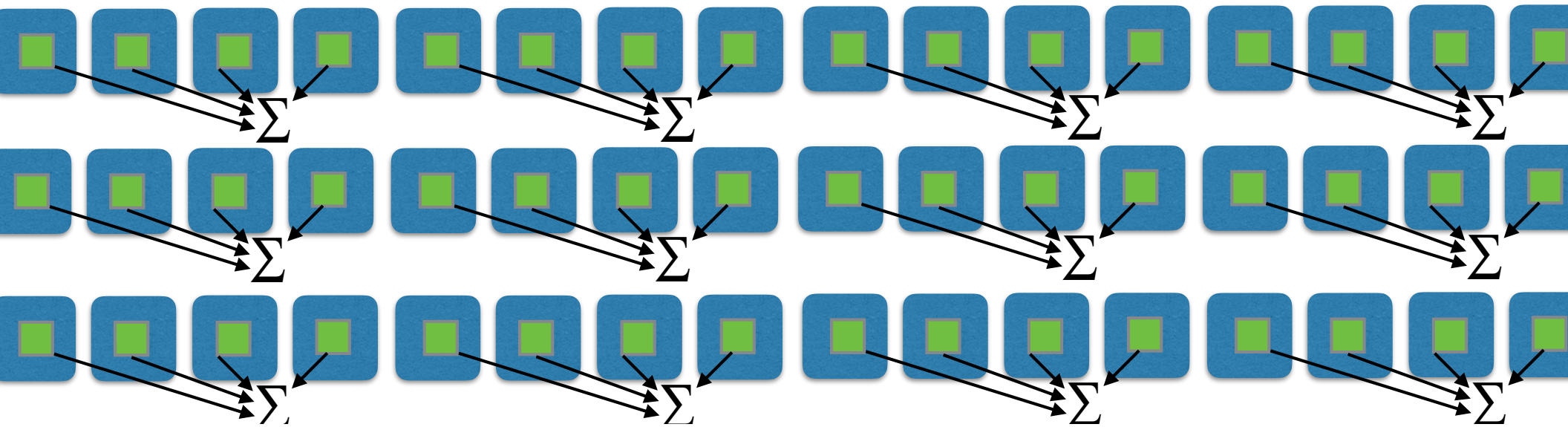
**Then aggregate (16 separate results)**



**Total cost:** 4|A| + 4|B| + 4|C|



**Less than 18% the communication** (if |A| = |B| = |C|)



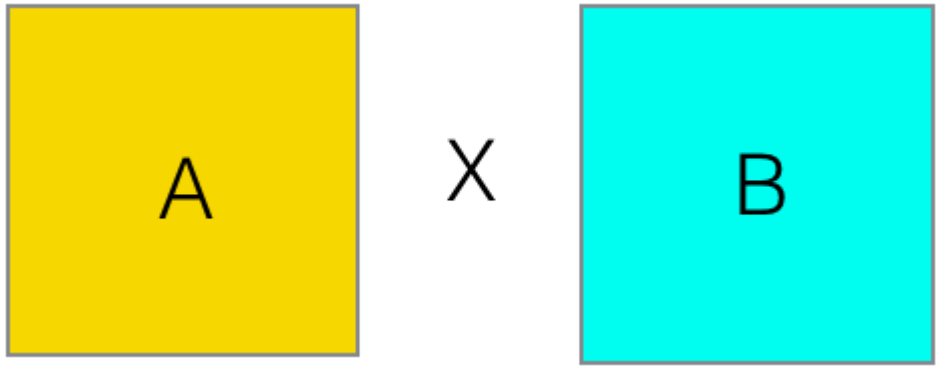
# This Will Require a New Kind of Relational Engine

# The `Einsummable` Database system

- We are given an EinSum expresson

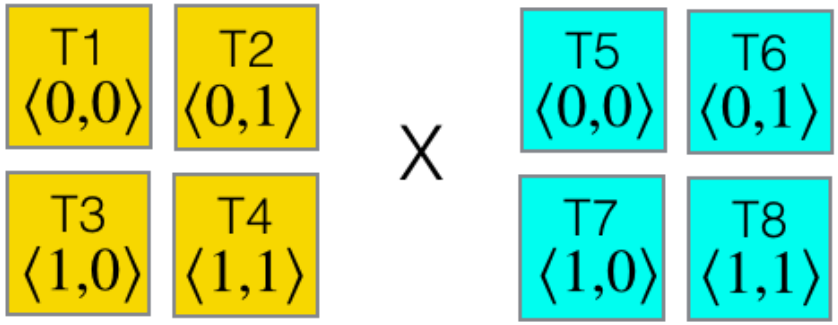
$$\mathbf{C}_{i,j} \leftarrow \sum_k \mathbf{A}_{i,k} \times \mathbf{B}_{k,j}$$

- And a set of input data



# Step 1: Auto Schema Design

- Run optimization problem to choose optimal schema for each tensor
  - Input data are decomposed into:

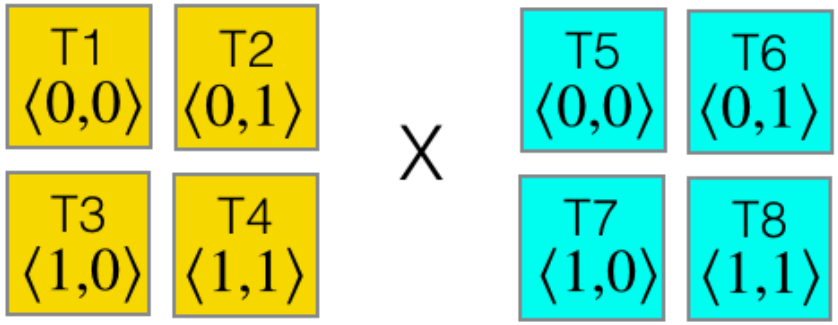



  - With assocated relational query:

    SELECT A.$c$, B.$a$, SUM ($K$(A.array, B.array)) FROM A, B WHERE A.$b$ = B.$b$ GROUP BY A.$c$, B.$a$.

# Step 2: Pilot Run

- Input data




  - ▷ Are converted into reltions without payloads
  - ▷ But with keys for tiles ("tensor IDs")

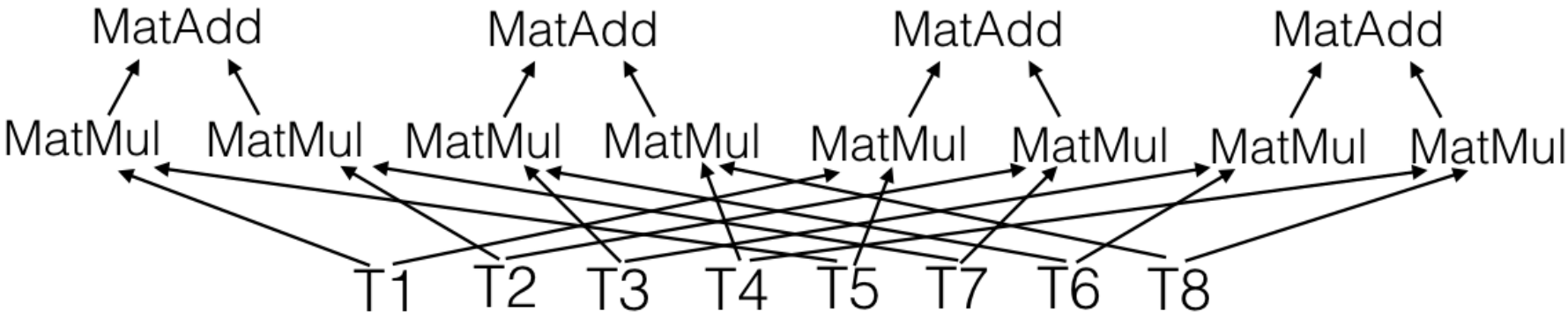
$$\mathbf{A} = \{(\langle 0,0\rangle, \mathrm{T1}), (\langle 0,1\rangle, \mathrm{T2}), (\langle 1,0\rangle, \mathrm{T3}), (\langle 1,1\rangle, \mathrm{T4})\}$$

$$\mathbf{B} = \{(\langle 0,0\rangle, \mathrm{T5}), (\langle 0,1\rangle, \mathrm{T6}), (\langle 1,0\rangle, \mathrm{T7}), (\langle 1,1\rangle, \mathrm{T8})\}$$

- Query is run and lineage is collected

# Step 3: Analyze Lineage

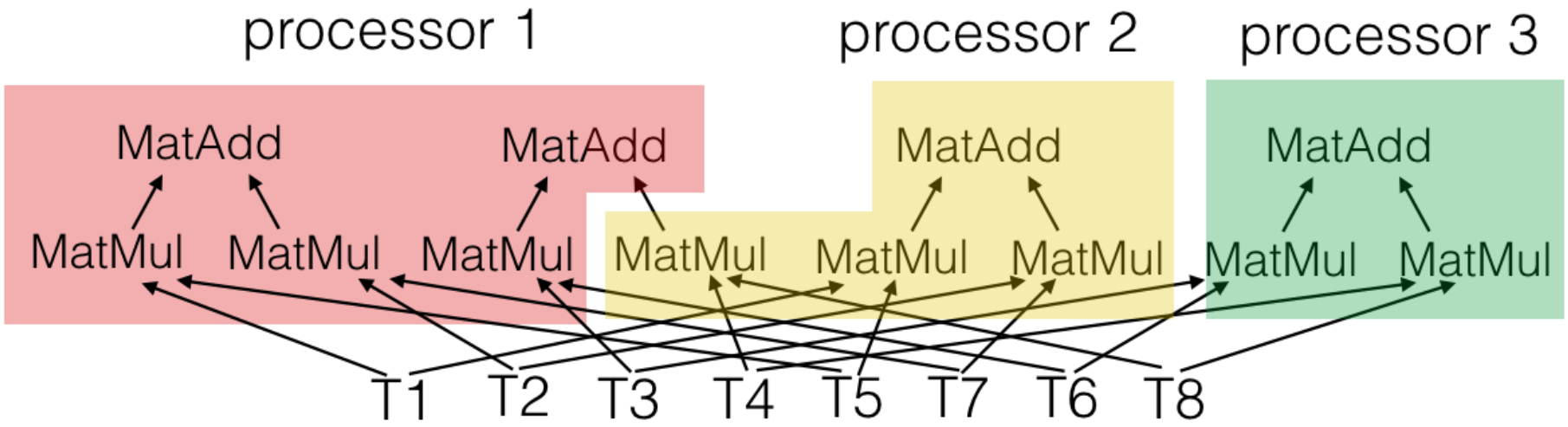
- Executing this query gives as the following lineage:




- Which is input into an opt problem:
  - Where to place kernels (nodes)
  - So has to min communication
  - But ensure everyone has work (no starvation)

# Step 3: Analyze Lineage

- Results in a mapping of kernel calls to processors



# Step 4: Execute Resulting Graph

- The underlying implementation is a dataflow system
  - The "Tensor Operating System" (TOS)
  - Dataflow systems: not a new idea

SIAM J. APPL. MATH.
Vol. 14, No. 6, November, 1966
*Printed in U.S.A.*

**PROPERTIES OF A MODEL FOR PARALLEL COMPUTATIONS: DETERMINACY, TERMINATION, QUEUEING***

RICHARD M. KARP AND RAYMOND E. MILLER†

**Abstract.** This paper gives a graph-theoretic model for the description and analysis of parallel computations. Within the model, computation steps correspond to nodes of a graph, and dependency between computation steps is represented by branches with which queues of data are associated. First, it is shown that each such *computation graph G* represents a unique computation, determined independently of operation times. Next, methods of determining whether such a computation terminates and of

# Step 4: Execute Resulting Graph

- The underlying implementation is a dataflow system
  - The "Tensor Operating System" (TOS)
  - Dataflow systems: not a new idea

SIAM J. APPL. MATH.
Vol. 14, No. 6, November, 1966
*Printed in U.S.A.*

**PROPERTIES OF A MODEL FOR PARALLEL COMPUTATIONS: DETERMINACY, TERMINATION, QUEUEING***

RICHARD M. KARP AND RAYMOND E. MILLER†

**Abstract.** This paper gives a graph-theoretic model for the description and analysis of parallel computations. Within the model, computation steps correspond to nodes of a graph, and dependency between computation steps is represented by branches with which queues of data are associated. First, it is shown that each such *computation graph G* represents a unique computation, determined independently of operation times. Next, methods of determining whether such a computation terminates and of

# How Well Does This Work?

- We've implemented this in a system called `Einsummable`
- Reconsider inference for the LLaMA 65B model
  - Run on 16 AWS `m6in.16xlarge` machines
  - Each has 256GB of RAM, 100Gb network, 32 cores
  - First-token inference

16 CPU machines, `Einsum`.

| Seq. Len. | Time (secs) | Mults /sec | Mults/sec /mach. |
|---|---|---|---|
| 2K | 17.4 | 7.94e12 | 4.97e11 |
| 4K | 38.2 | 7.53e12 | 4.71e11 |
| 8K | 81.7 | 7.61e12 | 4.75e11 |
| 16K | 230 | 6.17e12 | 3.86e11 |

# How Well Does This Work?

- We've implemented this in a system called `Einsummable`
- Reconsider inference for the LLaMA 65B model
  - Run on 16 AWS `m6in.16xlarge` machines
  - Each has 256GB of RAM, 100Gb network, 32 cores
  - First-token inference

16 CPU machines, `Einsum`.

| Seq. Len. | Time (secs) | Mults /sec | Mults/sec /mach. |
|---|---|---|---|
| 2K | 17.4 | 7.94e12 | 4.97e11 |
| 4K | 38.2 | 7.53e12 | 4.71e11 |
| 8K | 81.7 | 7.61e12 | 4.75e11 |
| 16K | 230 | 6.17e12 | 3.86e11 |

PyTorch A100 GPU Machine

| Seq. Len. | Time (secs) | Mults /sec | Mults/sec /GPU |
|---|---|---|---|
| 2K | 4.00 | 3.46e13 | 4.32e12 |
| 4K | 6.11 | 4.71e13 | 5.89e12 |
| 8K | 7.06 | 8.80e13 | 1.10e13 |
| 16K | OOM | N/A | N/A |

High-end GPU server $4\times$ the throughput of `Einsummable` CPU

# How Well Does This Work?

- We've implemented this in a system called `Einsummable`
- Reconsider inference for the LLaMA 65B model
  - Run on 16 AWS `m6in.16xlarge` machines
  - Each has 256GB of RAM, 100Gb network, 32 cores
  - First-token inference

16 CPU machines, `Einsum`.

| Seq. Len. | Time (secs) | Mults /sec | Mults/sec /mach. |
|---|---|---|---|
| 2K | 17.4 | 7.94e12 | 4.97e11 |
| 4K | 38.2 | 7.53e12 | 4.71e11 |
| 8K | 81.7 | 7.61e12 | 4.75e11 |
| 16K | 230 | 6.17e12 | 3.86e11 |

PyTorch A100 GPU Machine

| Seq. Len. | Time (secs) | Mults /sec | Mults/sec /GPU |
|---|---|---|---|
| 2K | 4.00 | 3.46e13 | 4.32e12 |
| 4K | 6.11 | 4.71e13 | 5.89e12 |
| 8K | 7.06 | 8.80e13 | 1.10e13 |
| 16K | OOM | N/A | N/A |

GPU 11× the throughput of `Einsummable` CPU here

# How Well Does This Work?

- We've implemented this in a system called `Einsummable`
- Reconsider inference for the LLaMA 65B model
  - Run on 16 AWS `m6in.16xlarge` machines
  - Each has 256GB of RAM, 100Gb network, 32 cores
  - First-token inference

16 CPU machines, `Einsum`.

| Seq. Len. | Time (secs) | Mults /sec | Mults/sec /mach. |
|---|---|---|---|
| 2K | 17.4 | 7.94e12 | 4.97e11 |
| 4K | 38.2 | 7.53e12 | 4.71e11 |
| 8K | 81.7 | 7.61e12 | 4.75e11 |
| 16K | 230 | 6.17e12 | 3.86e11 |

PyTorch A100 GPU Machine

| Seq. Len. | Time (secs) | Mults /sec | Mults/sec /GPU |
|---|---|---|---|
| 2K | 4.00 | 3.46e13 | 4.32e12 |
| 4K | 6.11 | 4.71e13 | 5.89e12 |
| 8K | 7.06 | 8.80e13 | 1.10e13 |
| 16K | OOM | N/A | N/A |

`Einsummable` has no OOM errors here!

# Thanks!

"I gotta skitty, dude. Catch you on the flip side."

- Special thanks to my collaborators
  - Daniel Bourgeois, Zhimin Ding, Dimitrije Jankov, Jiehui Li, Sleem Abdelghafar, Ge Huang, Yuxin Tang, Sarah Yao, Xin Yao