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× 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 \( \frac{50}{2.9 \times 2} = \frac{50}{5.8} \approx 8 \) second
  - 8192 inference should take roughly \( \frac{50}{6.2 \times 2} = \frac{50}{12.4} \approx 4 \) 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

<table>
<thead>
<tr>
<th>Seq len</th>
<th>Time (s)</th>
<th>Mults/sec</th>
<th>Mult/sec per GPU</th>
<th>% peak performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>6.11</td>
<td>4.7E13</td>
<td>5.9E12</td>
<td>2%</td>
</tr>
<tr>
<td>8192</td>
<td>7.06</td>
<td>8.8E13</td>
<td>1.1E13</td>
<td>4%</td>
</tr>
<tr>
<td>16348</td>
<td>OOM</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
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
  - **X** is a 20K by 20K matrix
  - Compute **Y = X × X × ...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 automatically 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 $$

CMJ, Rice University
Could Software Make A100’s Obsolete?

- What if the system automatically partitioned?
  - And automatically 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 input matrix is data and which is model?
  ▶ Just automatically run it the best way
Data parallel makes sense here

Compute: $f(XW_1)$
Data parallel makes sense here.
Data parallel makes sense here
Data parallel makes sense here.
Total xfer: $|X| + 4|W_1|$
Node 1  Node 2  Node 3  Node 4

But here?

\[ w_1 \]
But here?

Silly, as weight matrix is massive
Instead: Do this!
Only \( \frac{5}{17} \) the cost
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: ???
(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 \( \mathbf{B} \):

\[
C_{ik} \leftarrow \sum_j A_{ij} \times B_{jk}
\]

- 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 $\text{SoftMax}(W^{(2)} \times \text{ReLU}(W^{(1)} \times X))$

\[
A_i \leftarrow \sum_j W_{i,j}^{(1)} \times X_j
\]
\[
B_i \leftarrow \sum_\emptyset \text{ReLU}(A_i)
\]
\[
C_i \leftarrow \sum_j W_{i,j}^{(2)} \times B_j
\]
\[
D_\emptyset \leftarrow \sum_i \exp(C_i)
\]
\[
E_i \leftarrow \sum_\emptyset \frac{\exp(C_i)}{D_\emptyset}
\]

CMJ, Rice University
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 $A$

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

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

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

```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:

```
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 \( A \) into row strips and \( B \) into col strips:

\[
\begin{align*}
\text{A} & \quad \text{B} \\
\end{align*}
\]

- Then TRA is:

\[
\begin{array}{l}
\text{SELECT} \quad \text{lhs.rowID, rhs.colID} \\
\quad \text{mat_mul (lhs.chunk, rhs.chunk)} \\
\text{FROM} \quad A \quad \text{AS} \quad \text{lhs, B AS rhs} \\
\text{WHERE} \quad \text{lhs.colID} = \text{rhs.rowID}
\end{array}
\]
Yet Another

• Or decompose \( A \) and \( B \) into chunks:

\[
\begin{array}{c|c|c|c|c|c|c|c}
& & & & & & & \\
& & & & & & & \\
& & & & & & & \\
& & & & & & & \\
\end{array}
\begin{array}{c|c|c|c|c|c|c|c}
& & & & & & & \\
& & & & & & & \\
& & & & & & & \\
& & & & & & & \\
\end{array}
\]

• Then TRA is:

\[
\begin{align*}
\text{SELECT} & \quad \text{lhs.rowID}, \text{rhs.colID} \\
& \quad \text{mat_sum} \ (\text{mat_mul} \ (\text{lhs.chunk}, \text{rhs.chunk})) \\
\text{FROM} & \quad \text{A AS} \ \text{lhs}, \ \text{B AS} \ \text{rhs} \\
\text{WHERE} & \quad \text{lhs.colID} = \text{rhs.rowID} \\
\text{GROUP BY} & \quad \text{lhs.rowID}, \ \text{rhs.colID}
\end{align*}
\]
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:

  $$\mathbf{U}_i$$

- Call $\mathbf{i}$ a vector of “inner” indices
Introduce the notion of “outer” indices

- $U$ could also be represented as $\bar{U}$ with outer indices and inner indices
- Let:

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

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

$$\bar{U}_i^o = U_{i+o\times\frac{b}{d}}$$
Tensor Relations Revisited

- Why do we do this?

- $\bar{U} \equiv U/d$ is a tensor-relational representation of $U$
  
  - The tensor $U$ is broken into $\prod_i d_i$ “tiles”
  - Each tile is a tuple in the relation
  - The outer indices of $\bar{U}$ serve as keys for the relation
  - The inner indices of $\bar{U}$ serve index into tensors in a relation

- Example:
  
  $$(U_{/\langle 4,4,4 \rangle})_{1,0,1}^{3,2,1}$$
  
  - 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 $b = \langle 8, 4, 8 \rangle$)
Trivial Indices

- Convention: drop trivial (useless) indices
  - The $i$th outer index is trivial when $d_i = 1$ (no slicing)
  - The $i$th inner index is trivial when $d_i = b_i$ (full slicing)
  - So we write $(U_{\langle 4,4,4 \rangle})^{3,2,1}_{1,0,1}$ as $(U_{\langle 4,4,4 \rangle})^{3,2,1}_{1,1}$
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:

\[
C_{i,j} \leftarrow \sum_k A_{i,k} \times B_{k,j}
\]

- This is the same as:

\[
(C_{/\langle 1,1 \rangle})_{i,j} \leftarrow \sum_k (A_{/\langle 1,1 \rangle})_{i,k} \times (B_{/\langle 1,1 \rangle})_{k,j}
\]
This Re-Write Gives Us a New TRA Expression

- Now split 2nd dim of $B$:

$$
(C_{/\langle1,2\rangle})_{i,j}^a \leftarrow \sum_k (A_{/\langle1,1\rangle})_{i,k} \times (B_{/\langle1,2\rangle})_{k,j}^a
$$

- Let the kernel function $K(A, B)$ compute $C_{i,j} \leftarrow \sum_k A_{i,k} \times B_{k,j}$

- Then the above Einstein notation can be re-written to:

$\triangleright$ SELECT $B.a$, SUM ($K(A.array, B.array)$) FROM $A$, $B$ GROUP BY $B.a$.

- Is a mult of the following two tensor relations:
We Can Keep Going!

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

\[
\left( C/\langle 1,2 \rangle \right)^a_{i,j} \leftarrow \sum_b \sum_k \left( A/\langle 1,2 \rangle \right)^b_{i,k} \times \left( B/\langle 2,2 \rangle \right)^b_{k,j}
\]

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

- Is a mult of the following two tensor relations:
Another re-write gives:

\[
\left( \frac{C}{\langle 2,2 \rangle} \right)_{i,j}^{c,a} \leftarrow \sum_b \sum_k \left( \frac{A}{\langle 2,2 \rangle} \right)_{i,k}^{c,b} \times \left( \frac{B}{\langle 2,2 \rangle} \right)_{k,j}^{b,a}
\]

Again uses the same kernel function

Equivalent to the SQL:

\[
\text{SELECT } A.c, B.a, \text{SUM} \left( K(A.array, B.array) \right) \text{ FROM } A, B \text{ WHERE } A.b = B.b \text{ GROUP BY } A.c, B.a.
\]

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 $d$ so $|U/d| = 2^n$ is small
  - If we assume entries in $d$ are powers of 2
  - Powers of 2 not a big restriction
  - 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/ags
  - 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 relational system: the TOS???
Can We Just Re-Use Relational RDBMS

- Nope! RDBMS is operator-centric
- Not data-centric: problem for ML
- Consider matrix multiplication
  - One join followed by aggregation
  - Say we have $n$ processors
  - How would a hash join/hash-based agg work?
First let’s consider what a relational database could do.

We want to run this multiply on 64 processors.
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

site 1  site 2  site 3  site 4  site 5  site 64
And aggregate

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

\[ \sum \]
Total cost: $|A| + |B| + 64|C|$

Not great
Now let's consider a much better algorithm
Don’t partition keys: *place tuples*

site 1  site 2  site 3  site 4  site 5  site 64
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 expression

\[ C_{i,j} \leftarrow \sum_k A_{i,k} \times B_{k,j} \]

- 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:

  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.

CMJ, Rice University
Step 2: Pilot Run

- Input data

\[
\begin{align*}
A &= \{(\langle 0, 0 \rangle, T1), (\langle 0, 1 \rangle, T2), (\langle 1, 0 \rangle, T3), (\langle 1, 1 \rangle, T4)\} \\
B &= \{(\langle 0, 0 \rangle, T5), (\langle 0, 1 \rangle, T6), (\langle 1, 0 \rangle, T7), (\langle 1, 1 \rangle, T8)\}
\end{align*}
\]

- Query is run and lineage is collected

▷ Are converted into relations without payloads
▷ But with keys for tiles ("tensor IDs")
Step 3: Analyze Lineage

- Executing this query gives as the following lineage:

  ![Diagram showing lineage with MatAdd and MatMul operations with arrows between them]

  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
Step 4: Execute Resulting Graph

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

<table>
<thead>
<tr>
<th>Seq. Len.</th>
<th>Time (secs)</th>
<th>Mults/Sec</th>
<th>Mults/sec/mach.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2K</td>
<td>17.4</td>
<td>7.94e12</td>
<td>4.97e11</td>
</tr>
<tr>
<td>4K</td>
<td>38.2</td>
<td>7.53e12</td>
<td>4.71e11</td>
</tr>
<tr>
<td>8K</td>
<td>81.7</td>
<td>7.61e12</td>
<td>4.75e11</td>
</tr>
<tr>
<td>16K</td>
<td>230</td>
<td>6.17e12</td>
<td>3.86e11</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>16 CPU machines, Einsum.</th>
<th>PyTorch A100 GPU Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>2K</td>
<td>17.4</td>
</tr>
<tr>
<td>4K</td>
<td>38.2</td>
</tr>
<tr>
<td>8K</td>
<td>81.7</td>
</tr>
<tr>
<td>16K</td>
<td>230</td>
</tr>
</tbody>
</table>

High-end GPU server 4× 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

<table>
<thead>
<tr>
<th>16 CPU machines, Einsum.</th>
<th>PyTorch A100 GPU Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>2K</td>
<td>17.4</td>
</tr>
<tr>
<td>4K</td>
<td>38.2</td>
</tr>
<tr>
<td>8K</td>
<td>81.7</td>
</tr>
<tr>
<td>16K</td>
<td>230</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Seq. Len. (secs)</th>
<th>Time</th>
<th>Mults</th>
<th>Mults/sec</th>
<th>Mults/sec /mach.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2K</td>
<td>17.4</td>
<td>7.94e12</td>
<td>4.97e11</td>
<td></td>
</tr>
<tr>
<td>4K</td>
<td>38.2</td>
<td>7.53e12</td>
<td>4.71e11</td>
<td></td>
</tr>
<tr>
<td>8K</td>
<td>81.7</td>
<td>7.61e12</td>
<td>4.75e11</td>
<td></td>
</tr>
<tr>
<td>16K</td>
<td>230</td>
<td>6.17e12</td>
<td>3.86e11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seq. Len. (secs)</th>
<th>Time</th>
<th>Mults</th>
<th>Mults/sec</th>
<th>Mults/sec /GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>2K</td>
<td>4.00</td>
<td>3.46e13</td>
<td>4.32e12</td>
<td></td>
</tr>
<tr>
<td>4K</td>
<td>6.11</td>
<td>4.71e13</td>
<td>5.89e12</td>
<td></td>
</tr>
<tr>
<td>8K</td>
<td>7.06</td>
<td>8.80e13</td>
<td>1.10e13</td>
<td></td>
</tr>
<tr>
<td>16K</td>
<td>OOM</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

**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