We are a mission-based team

**Scientific Impact**

Deep computer science and mathematical expertise from several technical communities:
- Database systems and theory
- Machine learning
- Programming languages
- Operations research

2K+ publications

90K+ citations (35K+ in last 5 years)

37+ award-winning papers (3 this year!)

**AI and ML Industrial Impact**

- 42 core team members
- 6 former professors
- 22 PhDs
- 16 faculty network
- 250M direct value created
- 4 AI/ML companies Founded
- 2B total value created
- 42 core team members
- 6 former professors
- 22 PhDs
- 16 faculty network
- 250M direct value created
- 4 AI/ML companies Founded
- 2B total value created
The Case for Relational Artificial Intelligence

A New Technology Category
Databases should be Relational

What if I tell you
In the Navigational vs Relational DB wars of the 1980’s, Navigational DB’s were the incumbent and Relational DBs were the underdog!
Navigational 1974  Relational
Navigational

Charles Bachman

Weighing in with:
- Turing Award for Databases
- Integrated Data Store (IDS)
- Illustrious career at GE and Honeywell

Argument:
- Performance
  (it’s impossible to implement the relational model efficiently)
- Programmers won’t get it
  (Cobol programmers can’t possibly understand relational languages)

Relational

Ted Codd

Weighing in with:
- Researcher at IBM

Argument:
- Separation of the What from the How
  (Argument for declarativity)
- Domain experts will get it
  (and they are cheaper and more plentiful than programmers)
Navigational

Charles Bachman

Weighing in with:
- Turing Award for Databases
- Integrated Data Store (IDS)
- Illustrious career at GE and Honeywell

Argument:
- Performance
  (it’s impossible to implement the relational model efficiently)
- Programmers won’t get it
  (Cobol programmers can’t possibly understand relational languages)

1974

Relational

Ted Codd

Weighing in with:
- Researcher at IBM

Argument:
- Separation of the What from the How
  (Argument for declarativity)
- Domain experts will get it
  (and they are cheaper and more plentiful than programmers)

SO WHO WON?
Oracle (formerly Relational Software, Inc.)

- Launched RDBMS in 1979
- IPO in 1986
- Current Market Cap: $190.6B
Ingres (formerly Relational Technology, Inc.)

- Launched RDBMS in 1981
- IPO’d in 1988 (sold prematurely to ASK in 1989)
The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

**Relational DBMS**
1. Oracle
2. MySQL
3. Microsoft SQL Server
4. PostgreSQL

**DB-Engines Ranking May 2019**

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.
Analysts agree

Figure 1. Magic Quadrant for Operational Database Management Systems

Source: Gartner (October 2018)
Why?
What if I tell you

Business Intelligence should be Relational
In the Multidimensional (i.e. Tensor) vs Relational OLAP wars of the 1990’s, MOLAP was the incumbent and ROLAP was the underdog!
Tableau Software

- Launched in 2002
- IPO in 2013
- Current Market Cap: $11.6B
Analysts agree
Why?
What if I tell you

Artificial Intelligence should be Relational
What *if I tell you*

No way!!

Relational systems are **too slow**!

Tensors and linear algebra are the way we’ve always done it
I am here to tell you

**Relational Artificial Intelligence is Inevitable**
Why?

Rest of the talk
The Need for Speed

“We track about 47 different hardware startups that all have a unique approach” to accelerating AI.

Greg Brockman, CTO OpenAI, interviewed by Reid Hoffman, May 30, 2019

“13 private chip companies focused on the AI market have raised more than $1.2 billion in venture-capital funding”

- Barron’s article “AI Chip Market Will Soar to $34 Billion in Five Years”, Feb 20, 2019

“Today the job of training machine learning models is limited by compute, if we had faster processors we’d run bigger models...in practice we train on a reasonable subset of data that can finish in a matter of months. We could use improvements of several orders of magnitude – 100x or greater.”

Greg Diamos, Senior Researcher, SVAIL, Baidu, From EE Times – September 27, 2016
<table>
<thead>
<tr>
<th><strong>ACCURACY</strong></th>
<th><strong>VERSATILITY</strong></th>
<th><strong>ROBUSTNESS</strong></th>
<th><strong>SELF-SUPERVISION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Search for better</td>
<td>Reasoning and (generalized) inference: From observations to unknowns in any time period</td>
<td>Many “big data” problems are really a big collection of small data problems</td>
<td>“The future will be self-supervised” Yann LeCun</td>
</tr>
<tr>
<td>▪ Parameters</td>
<td>Inference of any property in the model (e.g., it’s just as easy to infer price from sales as it is to infer sales from price)</td>
<td>Overcome challenges with small, incomplete, and dirty data problems by incorporating prior knowledge and expertise</td>
<td>Build models of the world by observing it and searching model space for the models that have the most explanatory power</td>
</tr>
<tr>
<td>▪ Hyper parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Features</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Models</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Don’t make assumptions that you don’t need to make (e.g. i.i.d. assumption)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>INTERPRETABILITY</strong></th>
<th><strong>EXPLAINABILITY</strong></th>
<th><strong>FAIRNESS</strong></th>
<th><strong>CAUSALITY</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching for models that are accurate and interpretable is harder than searching for accurate models</td>
<td>Explainability typically implanted via separate shadow models that have to be learned</td>
<td>It’s not enough to exclude gender, ethnicity, race, age, etc as features to the models. Other features might be correlated.</td>
<td>Understanding causality beyond A/B testing</td>
</tr>
<tr>
<td>Interpretation in terms of prior knowledge and in language &amp; ontology that humans understand</td>
<td>Explanation in terms of prior knowledge and in language &amp; ontology that humans understand</td>
<td>Prejudice is a computational limitation: Reasoning about each person vs reasoning about the group</td>
<td>Computationally very expensive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Path to Performance: **Brawn**

Constant factors – Do same amount of work faster (i.e., brawn)

- **Latency hiding**: Memory hierarchy and network latencies (e.g., in memory and near-data computing)
- **Parallelization**: SIMD, multi-core, accelerators (e.g., GPU, TPU, FPGA)
- **Specialization**: Specialize for workload (e.g., JIT compilation), specialize for data
The Path to Performance: **Brains and Brawn**

**Asymptotics – Do less work (i.e., brains)**

- **Specialize algorithm** by exploiting problem structure
  - Algebraic (e.g., groups, semi rings, rings)
  - Combinatorial (e.g., fractional hypertree width)
  - Statistical (e.g., samples and sketches)
  - Geometric (e.g., fast multipole method)
- **Solve similar** but more tractable problem
  - Approximation (with error bars)
Brains

Do Less Work
The **relational model dominates data management**

- The last 40 years have witnessed massive adoption of the relational model
  - It's hard to find any examples today of enterprises whose data isn't in a relational database
- Millions of human hours invested in building relational models and populating them with data
- Relational databases are rich with knowledge of the underlying domains that they model
- The availability and accuracy of large amounts of curated data has made it possible for humans (BI) and machines (AI) to **learn** from the past and to **predict** the future
What’s the first thing we do when we build predictive models?

We work hard to throw away all relational structure (and semi-structure) we worked so hard to build.

We end up throwing away important domain knowledge that can help us build better AI models.
The wastefulness does not end there
The wastefulness does not end there.

- Avoid materializing the join
- Avoid filling in the zeros
- Avoid one-hot encoding
- Exploit relational structures to speed up learning
- Ideally, train models faster than the time it takes to produce the query output in the first place!
What **would a database do?**

1. Database

2. Feature extraction query

3. Model specification (e.g., “degree 2 ridge regression”)

- **Features**
  - ID
  - x1
  - x2
  - x3
  - ...
  - y

- **Examples**

---

S: Sufficient statistics generated from model spec and feature extraction query. Computed via aggregations.
### Number of Aggregates Varies By Model Class

#### Supervised

- **Regression**

<table>
<thead>
<tr>
<th>Model</th>
<th># features</th>
<th># params</th>
<th># aggregates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear regression</td>
<td>$n$</td>
<td>$n + 1$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Polynomial regression</td>
<td>$\Theta(n^d)$</td>
<td>$\Theta(n^d)$</td>
<td>$\Theta(n^{2d})$</td>
</tr>
<tr>
<td>Factorization machines</td>
<td>$\Theta(n^d)$</td>
<td>$\Theta(nr)$</td>
<td>$\Theta(n^{2d})$</td>
</tr>
</tbody>
</table>

- **Classification**

<table>
<thead>
<tr>
<th>Model</th>
<th># features</th>
<th># aggregates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision trees</td>
<td>$\Theta(n)$</td>
<td>$\Theta(nbh)$</td>
</tr>
</tbody>
</table>

#### Unsupervised

<table>
<thead>
<tr>
<th>Model</th>
<th># aggregates</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-means</td>
<td>$\Theta(kn)$</td>
</tr>
<tr>
<td>PCA</td>
<td>$\Theta(kn^2)$</td>
</tr>
</tbody>
</table>

$n$: # input features  
$d$: degree  
$r$: rank  
$b$: branching factor  
$h$: depth  
$k$: # clusters
We Efficiently Compute Those Aggregates
Case Study: Retail dataset

- Stores
  - store_id
  - zipcode
  - area_sq_ft
  - avghhi
  - distance_to_comp1
  - distance_to_comp2
  - ...

- Inventory
  - store_id
  - date
  - item_id
  - inventory_units

- Weather
  - store_id
  - date
  - rain
  - snow
  - thunder
  - min_temperature
  - max_temperature
  - mean_wind

- Demographics
  - zipcode
  - population
  - ethnicities
  - households
  - median_age
  - ...

- Items
  - item_id
  - category
  - subcategory
  - category_cluster
  - price
Case Study: **Retail dataset**

<table>
<thead>
<tr>
<th>Relation</th>
<th>Cardinality (# Tuples)</th>
<th>Degree (# k/v columns)</th>
<th>File size (csv)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inventory</td>
<td>84,055,817</td>
<td>3 &amp; 1</td>
<td>2 GB</td>
</tr>
<tr>
<td>Items</td>
<td>5,618</td>
<td>1 &amp; 4</td>
<td>129 KB</td>
</tr>
<tr>
<td>Stores</td>
<td>1,317</td>
<td>1 &amp; 14</td>
<td>139 KB</td>
</tr>
<tr>
<td>Demographics</td>
<td>1,302</td>
<td>1 &amp; 15</td>
<td>161 KB</td>
</tr>
<tr>
<td>Weather</td>
<td>1,159,457</td>
<td>2 &amp; 6</td>
<td>33 MB</td>
</tr>
</tbody>
</table>

**Total:** 2.1 GB
Case Study: Retail dataset – PostgreSQL & TensorFlow

- The design matrix is constructed by joining together all the relations
- Train a linear regression model to predict sales by item, store, date from all the other features

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality (# of tuples)</td>
<td>84,055,817</td>
</tr>
<tr>
<td>Degree (# of columns)</td>
<td>44 (3 &amp; 41)</td>
</tr>
<tr>
<td>Size</td>
<td>23 GB</td>
</tr>
<tr>
<td>Time to compute in PostgreSQL</td>
<td>217 secs</td>
</tr>
<tr>
<td>Time to export from PostgreSQL</td>
<td>373 secs</td>
</tr>
<tr>
<td>Time to learn parameters with GD</td>
<td>&gt; 12,000 secs</td>
</tr>
</tbody>
</table>
Case Study: **Retail dataset - comparison**

<table>
<thead>
<tr>
<th></th>
<th>Design matrix with PostgreSQL/TensorFlow</th>
<th>relationalAI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>Original</td>
<td>--</td>
<td>2.1 GB</td>
</tr>
<tr>
<td>Join Tables</td>
<td>217 secs</td>
<td>23 GB</td>
</tr>
<tr>
<td>Export DM</td>
<td>373 secs</td>
<td>23 GB</td>
</tr>
<tr>
<td>Aggregate</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Parameter learning with GD</td>
<td>&gt; 12 K secs</td>
<td>--</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>&gt; 12.5 K secs</td>
<td></td>
</tr>
<tr>
<td><strong>Improvement (1st Model)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Every model after</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Does it work for all model classes or methods?

Supported methods include

- Linear regression
- Polynomial regression
- Factorization machines
- Decision trees
- Linear SVM
- Deep sum-product networks
- Naive Bayes Classifier (discrete case)
- Hidden Markov Model (discrete case)
- K-Means & K-Median clustering
- Gaussian Discriminant Analysis
- Linear Discriminant Analysis
- Principal component analysis
- Frequent item set mining (with Apriori algorithm)
- Computing empirical mutual information and entropy

(with more on the way)
So what?

Some context:

Moore’s Law
gives us 2x speedup
every 1.5 years

According to Nvidia
GPUs give us a 2-10X
speed-up over CPUs

In other words, GPUs give us ~5 year advantage
So what?

What are the implications of 2-3 orders of magnitude speed-up?

256x is 8 doublings (i.e., $2^8$)

1024x is 10 doublings
So what?

What are the implications of 2-3 orders of magnitude speed-up?

Algorithms that exploit the domain structure give us a **12-15 YEAR ADVANTAGE**
### AI’s biggest challenges are computational!

<table>
<thead>
<tr>
<th><strong>ACCURACY</strong></th>
<th><strong>VERSATILITY</strong></th>
<th><strong>ROBUSTNESS</strong></th>
<th><strong>SELF-SUPERVISION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search for better</strong>&lt;br&gt;• Parameters&lt;br&gt;• Hyper parameters&lt;br&gt;• Features&lt;br&gt;• Models</td>
<td><strong>Reasoning and (generalized) inference: From observations to unknowns in any time period</strong>&lt;br&gt;<strong>Inference of any property in the model (e.g., it’s just as easy to infer price from sales as it is to infer sales from price)</strong></td>
<td><strong>Many “big data” problems are really a big collection of small data problems</strong>&lt;br&gt;<strong>Overcome challenges with small, incomplete, and dirty data problems by incorporating prior knowledge and expertise</strong></td>
<td><strong>“The future will be self-supervised” Yann LeCun</strong>&lt;br&gt;<strong>Build models of the world by observing it and searching model space for the models that have the most explanatory power</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>INTERPRETABILITY</strong></th>
<th><strong>EXPLAINABILITY</strong></th>
<th><strong>FAIRNESS</strong></th>
<th><strong>CAUSALITY</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Searching for models that are accurate and interpretable is harder than searching for accurate models</strong>&lt;br&gt;<strong>Interpretation in terms of prior knowledge and in language &amp; ontology that humans understand</strong></td>
<td><strong>Explainability typically implanted via separate shadow models that have to be learned</strong>&lt;br&gt;<strong>Explanation in terms of prior knowledge and in language &amp; ontology that humans understand</strong></td>
<td><strong>It’s not enough to exclude gender, ethnicity, race, age, etc as features to the models. Other features might be correlated.</strong>&lt;br&gt;<strong>Prejudice is a computational limitation: Reasoning about each person vs reasoning about the group</strong></td>
<td><strong>Understanding causality beyond A/B testing</strong>&lt;br&gt;<strong>Computationally very expensive</strong></td>
</tr>
</tbody>
</table>
Statistical Relational Learning

Relational generative models
What else do we throw away when we build the feature matrix?

Translation to feature matrix assumes each entity is independent of the others (iid assumption)

This is often not true - e.g. related sku’s or related people
What if we don’t make the i.i.d assumption?

<table>
<thead>
<tr>
<th>Pairs of Entities</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>...</td>
</tr>
</tbody>
</table>

Features
What if we **don’t make the i.i.d assumption?**

<table>
<thead>
<tr>
<th>ID</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>...</th>
<th>y</th>
<th>ID</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Features

<table>
<thead>
<tr>
<th>ID</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>...</th>
<th>y</th>
<th>...</th>
<th>ID</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Statistical Relational Learning

- Statistical Relational models generalize PGMs in the same way that first order logic generalizes propositional logic – they allow us to quantify over individuals/entities
  - Allows for generalization (e.g. item, sub-class, class, dept, etc.)
  - Ability to predict link-based patterns (e.g. inter item dependencies at sub-class, class, dept etc.)
  - Models a varied number of observations for each object/relation. (e.g. friends, colleagues, etc.)

- Variants
  - MLN in various flavors, PSL, RDN, BoostSRL, ProbLog, etc.
Statistical Relational Learning

- Inference
  - Unlike “traditional” methods where prediction is the input applied to the parameters of the model class, inference in SRL requires expensive optimization or (approximate) integration over possible worlds

- Learning
  - Unlike traditional learning algorithms, just one instance to learn from (the relational DB)
  - Structure learning uses inference during each step
Smoking and Quitting in Groups

Researchers studying a network of 12,067 people found that smokers and nonsmokers tended to cluster in groups of close friends and family members. As more people quit over the decades, remaining groups of smokers were increasingly pushed to the periphery of the social network.

1971 A sample of 1,000 people from the study includes many large groups of smokers.

2000 Nearly three decades later, groups of smokers tended to be smaller and more isolated.

Sources: New England Journal of Medicine; Dr. Nicholas A. Christakis; James H. Fowler

KEY
- Male smoker
- Male nonsmoker
- Female smoker
- Female nonsmoker
- Friendship, marriage or family tie
Circle size is proportional to the number of cigarettes smoked per day.
CERTAIN KNOWLEDGE WITH INTEGRITY CONSTRAINTS

A logical Knowledge Base is a set of Integrity Constraints that define a set of possible worlds:

\[
\begin{align*}
\text{person}(x) \\
\text{smokes}(x) & \rightarrow \text{person}(x) \\
\text{cancer}(x) & \rightarrow \text{person}(x) \\
\text{friends}(x, y) & \rightarrow \text{person}(x), \text{person}(y)
\end{align*}
\]
CERTAIN KNOWLEDGE WITH INTEGRITY CONSTRAINTS

A logical Knowledge Base is a set of Integrity Constraints that define a set of possible worlds:

\[
\begin{align*}
\text{person}(x) \\
\text{smokes}(x) & \rightarrow \text{person}(x) \\
\text{cancer}(x) & \rightarrow \text{person}(x) \\
\text{friends}(x, y) & \rightarrow \text{person}(x), \text{person}(y)
\end{align*}
\]

Smoking causes cancer
Friends have similar smoking habits
A logical Knowledge Base is a set of Integrity Constraints that define a set of possible worlds:

- `person(x)`
- `smokes(x) -> person(x)`
- `cancer(x) -> person(x)`
- `friends(x, y) -> person(x), person(y)`

Smoking causes cancer
Friends have similar smoking habits

- `w1  smokes(x) -> cancer(x)`
- `w2  smokes(x), friends(x, y) -> smokes(y)`
Approximate answer by converting into convex continuous optimization problem

Exploit group symmetry $\rightarrow$ lifted inference and approximate lifted inference

Avoid grounding altogether $\rightarrow$ in-database learning

Leveraging database semantics to avoid having to cluster $\rightarrow$ in-database SPNs

Stay tuned
Brawn

Do same amount of work faster
The Path to Performance: Brawn

Constant factors – Do same amount of work faster (i.e., brawn)

▪ **Latency hiding**: Memory hierarchy and network latencies (e.g., in memory and near-data computing)
▪ **Parallelization**: SIMD, multi-core, accelerators (e.g., GPU, TPU, FPGA)
▪ **Specialization**: Specialize for workload (e.g., JIT compilation), specialize for data
Motivation for implementation strategy

3 to 5 years building something similar in prior lives using C++ without ability to specialize for queries or data sets
**Julia in a nutshell**

“Looks like Python, feels like LISP, runs like C”

Julia is fast, dynamic, optionally typed, and multi-dispatched

- Feels like Lisp: Hygienic macros, code quoting, generated functions
- Runs like C: Specialization based on type inference, inlining, unboxing, LLVM to gen assembly
Brains and Brawn: **Systems Programming in Julia**

- **Specialization**
  - **Query evaluation**: Just-in-time compiled query plans

- Specialization
  - **Data types**: e.g., fixed-precision decimals
Just-in-Time Query Compilation

- Query compilation has only recently replaced interpretation in modern database systems

```
select A, B, C
from R, S, T
where ...
group by ...
```

- But, state of the practice is surprisingly primitive
  - Typically: variations on template expansion in C/C++
  - Ad-hoc methods to generate code: e.g., write a text file and invoke gcc
  - Cumbersome engineering effort

- Better: use a language with proper staged metaprogramming support
  - e.g., LegoDB using Scala/LMS/Squid

- Julia is very appealing from this point of view!
Simplified TPC-H Q1: from SQL to Julia to Native Code

```
select
  sum(l_extprice * (100 - l_discount) * (100 + l_tax))
from
lineitem
```

From SQL to Julia with runtime code generation

```
sum = 0
for i in 1:size
  sum += l_extprice[i] * (100 - l_discount[i]) * (100 + l_tax[i])
end
return sum
```

From Julia to LLVM to optimized x86-64 *

```
testq %rcx, %rcx
jle L71
movq (%rdi), %r8
movq (%rsi), %r9
movq (%rdx), %r10
xorl %edi, %edi
xorl %eax, %eax
L32:
  movl $100, %esi
  subq (%r9,%rdi,8), %rsi
  movq (%r10,%rdi,8), %rdx
  addq $100, %rdx
  imulq (%r8,%rdi,8), %rsi
  imulq %rdx, %rsi
  addq %rsi, %rax
  addq $1, %rdi
  cmpq %rdi, %rcx
  jne L32
  retq
L71:
  xorl %eax, %eax
  retq
```

(*) The loop actually even gets vectorized, but we produced simpler code here for presentation purposes.
BI benchmark: **vs Tableau/Hyper and Databricks Spark**

Spark numbers based on Databricks hardware and TPCH setup. Snowflake benchmarks closer to Spark than Hyper.
Brains and Brawn Together: **3-Clique Graph benchmark vs Databricks Spark**

Triangle Count on graph500 dataset

- **Spark GraphFrames built-in**: 364.45717
- **Spark GraphFrames RDD**: 185.3177
- **Delve**: 10.22252

All benchmarks run on 1 core laptop.
Brains and Brawn: **Systems Programming in Julia**

- Specialization
  - **Query evaluation**: Just-in-time compiled query plans

- **Specialization**
  - Data types: e.g., fixed-precision decimals
Abstraction **without regret by example**: Fixed-precision decimals

Fixed-precision decimals are an important data type in database systems (e.g., for currencies), and avoid the inexact representation problems of floats:

```
julia> 0.3333 + 0.3333
0.6666300000000001  # oops
```

The Julia ecosystem has a FixedPointDecimal package for this purpose

```
julia> T = FixedDecimal{Int64,5}
FixedDecimal{Int64,5}

julia> T(0.3333) + T(0.3333)
FixedDecimal{Int64,5}(0.66663)  # much better!
```

But... is this really going to be efficient enough? (Most database systems need special code to “compile away” fixed precision decimal operations into simple operations on integers...)
Here's the `FixedDecimal` datatype and its addition operation...

```julia
default FixedDecimal\{T <: Integer, f} <: Real
  i::T

  function Base.reinterpret\{::Type\{FixedDecimal\{T, f\}\}, i::Integer\} where \{T, f\}
      n = max_\text{exp\_10}(T)
      if f >= 0 && (n < 0 || f <= n)
          new{T, f}(i \% T)
      else
          _\text{throw\_storage\_error}(f, T, n)
      end
  end
end

+(x::FixedDecimal\{T, f\}, y::FixedDecimal\{T, f\}) where \{T, f\} =
  reinterpret\{FD\{T, f\}, x.i+y.i\}
```

... and lo, the Julia compiler produces a tiny # of ops on integers, just as required!

```julia
julia> @code_native +(T(0.3333),T(0.33333))
decl %eax
movl (%esi), %eax
decl %eax
addl (%edi), %eax
reml
```

Moreover, this will be inlined at the call site in any practical example!
What about Parallelization and Accelerators?

Parallel Computing

For newcomers to multi-threading and parallel computing it can be useful to first appreciate the different levels of parallelism offered by Julia. We can divide them in three main categories:

1. Julia Coroutines (Green Threading)
2. Multi-Threading
3. Multi-Core or Distributed Processing

We will first consider Julia Tasks (aka Coroutines) and other modules that rely on the Julia runtime library, that allow us to suspend and resume computations with full control of the task communication without having to manually interfere with the operating system's scheduler. Julia also supports communication between Tasks through operations like `wait` and `fetch`. Communication and data synchronization is managed through Channels, which are the conduits that provide inter-task communication.

Julia also supports experimental multi-threading, where execution is forked and an anonymous function is run across all threads. Known as the fork-join approach, parallel threads execute independently, and must ultimately be joined in Julia's main thread to allow serial execution to continue. Multi-threading is supported using the `Base.Threads` module that is still considered experimental, as Julia is not yet fully thread-safe. In particular segfaults seem to occur during I/O operations and task switching. As an up-to-date reference, keep an eye on the Issue tracker. Multi-Thread short should only be used if you take into consideration global variables, locks and atomic, all of which are explained later.

In the end we will present Julia's approach to distributed and parallel computing. With scientific computing in mind, Julia natively implements interfaces to distribute a process across multiple cores or machines. Also we will mention useful external packages for distributed programming like MPI.jl and DistributedArrays.jl.

---

High-level GPU programming in Julia

Tim Revoredo
Computer Systems Lab
Ghent University, Belgium

Piotr Vicentru
Ghent University, Belgium

Ryden De Sitter
Computer Systems Lab
Ghent University, Belgium

Abstract

GPUs are popular devices for accelerating scientific calculations. However, as GPU code is usually written in low-level languages, it breaks the abstraction of high-level languages popular with scientific programmers. To overcome this, we present a framework for CUDA GPU programming in the high-level Julia programming language. This framework complements Julia's code generation techniques to avoid run-time overhead. Evaluating the framework and its APIs on a case study comparing the trace transforms from the field of image processing, our framework shows a 27-fold speedup over the CUDA API on a mainstream, while greatly increasing programmer productivity. The metaprogramming capabilities of the Julia language proved invaluable for enabling this. Our framework significantly improves stability of GPU, making it accessible for a wide range of programmers. It is available as free and open-source software licensed under the MIT License.

Categories and Subject Descriptions D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Runtime environments

Keywords: Julia, GPU, CUDA, LLVM, Metaprogramming

1. Introduction

GPUs are specifically designed to work on matrix operations. However, targeting GPUs requires some effort. Specialized machine code needs to be generated through the use of a vendor-supplied compiler. Because of the architectural setup, initiating executions on the coprocessor is often quite complex as well. Even though the vendors try to simplify block sizes for different developers environments and often come with a support for some optimization routines.

We propose a framework for CUDA GPU programming in the high-level Julia programming language. The framework provides a transparent interface to the CUDA API, making it accessible for a wide range of programmers. It is available as free and open-source software licensed under the MIT License.

In Section 2 we describe relevant technologies and the motivation for our work. Section 3 provides an overview of our framework, each component explained in detail in Section 4. Finally, we evaluate our work in Section 5.

Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs

Kazu Fuchita, Effiz Yalba

ABSTRACT

Google’s Cloud TPUs are a promising new hardware architecture for machine learning workloads. They have powered many of Google’s machine learning achievements in recent years. Google has made TPUs available for general use on their cloud platform and at a very recently has opened them up further to allow use by non-TensorFlow frameworks. We describe a method and implementation for offloading suitable sections of Julia programs to TPUs via the Julia API and the Google SLA compiler. Our method is able to completely offload the forward pass of a VGG19 model expressed as a Julia program into a single TPU executable to be offloaded to the device. We have composed code with existing compile-time automated differentiation techniques on Julia code, and we first show in detail how to automatically obtain the VGG19 backward pass and instantly offload it to the TPU. Targeting TPUs using our compiler, we are able to evaluate the VGG19 forward pass on a batch of 100 images in 0.2s, which compares favorably to the 52.6s reported for the original model on the CPU. Our implementation in less than 1500 lines of Julia, with no TPU specific changes made to the Julia compiler or any other Julia packages.

1 Introduction

One of the fundamentals changes that has enabled the steady progress of machine learning technology over the past several years has been the availability of vast amounts of compute power to train and optimize machine learning models. Many fundamental technologies are decades old, but only the compute power available in recent years was able to deliver sufficiently good results to be interesting for real world problems. A significant chunk of this compute power is delivered through off-the-shelf accelerator cards by companies like Nvidia and AMD, which have a direct impact on the number of compute units available. The desire for higher performance has prompted the development of new hardware accelerators like TPUs and FPGAs.

In this paper, we present our work to compile Julia code to TPUs using this interface. This approach is in contrast to the approach taken by TensorFlow (Abadi et al., 2015), which does not compile Python code, but rather uses Python to build a computational graph, which is then compiled. It is also not compile-time optimized (Abadi et al., 2016), which aims to offload computations written in Python proper by running and offloading high-level array operations. Crucially, however, we do not rely on this, instead leverage Julia’s static analysis and compilation capabilities to compile the full program, including any control flow into the device. In particular, our approach allows users to take advantage of the full expressiveness of the Julia programming language in writing their models.

This includes higher-level features such as multiple dispatch, higher order functions and existing libraries such as those for differential equation solvers (Balaauc et al., 2017) and geometric algebra engines. Since it operates in parallel with the CPU, this allows for better utilization of the TPU while leaving the CPU free to handle other tasks.
Closing

One more time
AI’s **biggest opportunities are relational!**

<table>
<thead>
<tr>
<th>ACCURACY</th>
<th>VERSATILITY</th>
<th>ROBUSTNESS</th>
<th>SELF-SUPERVISION</th>
</tr>
</thead>
</table>
| Search for better  
  - Parameters  
  - Hyper parameters  
  - Features  
  - Models  | Reasoning and (generalized) inference: From observations to unknowns in any time period  
Inference of any property in the model (e.g., it’s just as easy to infer price from sales as it is to infer sales from price)  | Many “big data” problems are really a big collection of small data problems  
Overcome challenges with small, incomplete, and dirty data problems by incorporating prior knowledge and expertise  | “The future will be self-supervised” Yann LeCun  
Build models of the world by observing it and searching model space for the models that have the most explanatory power |

<table>
<thead>
<tr>
<th>INTERPRETABILITY</th>
<th>EXPLAINABILITY</th>
<th>FAIRNESS</th>
<th>CAUSALITY</th>
</tr>
</thead>
</table>
| Searching for models that are accurate and interpretable is harder than searching for accurate models  
Interpretation in terms of prior knowledge and in language & ontology that humans understand  | Explainability typically implanted via separate shadow models that have to be learned  
Explanation in terms of prior knowledge and in language & ontology that humans understand  | It’s not enough to exclude gender, ethnicity, race, age, etc as features to the models. Other features might be correlated.  
Prejudice is a computational limitation: Reasoning about each person vs reasoning about the group  | Understanding causality beyond A/B testing  
Computationally very expensive |
Why hasn’t this happened yet?

**AI investment is focused on consumer AI**
- Deep learning for images, speech, text → not relational data (yet)

**Weaknesses of implementations of relational data management systems**
- Abstraction leads to regret
- Can guarantee correct answer but can’t guarantee optimal path to get there
- Limitations on expressiveness, i.e. I can’t always ask the question I want to ask

**Inertia** — we have something that (sort of) works and we’re getting by. “you can’t expect us to rewrite all this code and retrain all those data scientists and programmers”
- The number of models that haven’t been built is >>> the number of models that have
- The number of future modelers is >>> the number of current modelers
- The number of domain experts is >>> the number of modelers and data scientists
Why Now?

- We invented a new generation of (meta) algorithms that provide optimal solutions to large problem classes
  - OOM **more power** for OOM **better intelligence**

- New generation of compilers that eliminate the cost of abstraction
  - Allow us to specialize for workload
  - Allow us to specialize for datasets

- Backlash against Hadoop (Map-Reduce), NoSQL, ML Frameworks – “the emperor has no clothes” is in the air
  - Require you to sell your soul for scalability and/or performance
  - Harder to program and operate
What are we doing about it?

We built a system that gives you abstraction without regret

How are we going to do that?
- Constant factors
- Asymptotic factors

We’re going to meet people where they are:
- Tables and SQL if you are an analyst
- Tensors & Linear Algebra if you are a data scientist

We’re going to simplify and consolidate analytics:
- The building blocks for next gen AI (e.g. fast aggregation, factoring, multi-way evaluation, JIT, accelerators) building blocks for all enterprise analytics: BI, graphs, rules, planning, mathematical optimization.

We’re going to stage it. We’re going to consolidate and checkpoint our gains as we go.
- AutoML (with automatic feature engineering and relational statistics) -> Data scientist
- Data Management Systems for Analytics (aka data lakes) -> Data scientist
- Business Intelligence & Data Warehouses -> Analyst & End User
Product: **Never have to start from scratch again**

**Data**
- General: e.g. Weather, Events, Consumer, Sentiment
- Domain and industry specific: e.g. securities, crypto currencies
- Competitor: e.g. price

**Templates**
- Industry: retail, financial services, technology & software.
- Problem class: (product) knowledge graphs, recommender systems, anomaly detection, portfolio optimization

**Tools**
- Data scientists: Notebooks (e.g. Jupyter)
- Domain modelers: e.g. ontology editors (e.g. Jupyter, NORMA, Protégé)
- Analysts: e.g. BI and spreadsheets

**Engine**
- Database
- AI and Analytics
References

Incomplete list
Underlying magic: **Worst-case optimal join algorithms**

- What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? Abo Khamis, Ngo, Suciu, (PODS 2017 - Invited to Journal of ACM)
- Beyond Worst-Case Analysis for Joins with Minesweeper. Abo Khamis, Ngo, Re, Rudra. (PODS 2014)
Underlying magic: **Optimal query plans for worst-case optimal joins**

- Juggling functions inside a database, Abo Khamis, Ngo, Suciu (Invited to SIGMOD Record)
- On Functional Aggregate Queries with Additive Inequalities. Abo Khamis, Curtin, Moseley, Ngo, Nguyen, Olteanu, Schleich. PODS 2019
- What do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog have to do with one another? Abo Khamis, Ngo, Suciu, (PODS 2017 - Invited to Journal of ACM)
Underlying magic: **In-database relational learning**

- In-Database Learning with Sparse Tensors, Abo Khamis, Ngo, Nguyen, Olteanu, Schleich (PODS 2018 - Invited to Journal of TODS)
- AC/DC: In-Database Learning Thunderstruck, Abo Khamis, Ngo, Nguyen, Olteanu, Schleich (DEEM 2018)
- In-Database Factorized Learning, Ngo, Nguyen, Olteanu, Schleich (AMW 2017)
- Data Science with Linear Programming. Makrynioti, Vasiloglou, Pasalic, Vassalos. (DeLBP 2017)
Underlying magic: Julia

- Julia Subtyping: A Rational Reconstruction, Francesco Zappa Nardelli, Julia Belyakova, Artem Penelitsyn, Benjamin Chung, Jeff Bezanson, Jan Vitek (OOPSLA 2018)
THANK YOU