TU Graz

SCIENCE
PASSION
TECHNOLOGY

Architecture of ML Systems*
05 Compilation and Optimization

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMK endowed chair for Data Management

Last update: Aug 25, 2022

ISDS

Agenda

- **Compilation Overview**
- **Size Inference and Cost Estimation**
- **Rewrites (and Operator Selection)**
- **Runtime Adaptation**
- **Operator Fusion & JIT Compilation**

SystemDS, and several other ML systems

Compilation Overview

Recap: Linear Algebra Systems

- **Comparison Query Optimization**
  - **Rule- and cost-based rewrites and operator ordering**
  - **Physical operator selection and query compilation**
  - Linear algebra / other ML operators, DAGs, control flow, sparse/dense formats

PL **DB** **HPC**

Compilers for Large-scale ML

- **#1 Interpretation** (operation at-a-time)
  - Examples: **R**, **PyTorch**, **Morpheus** [PVLDB'17]
- **#2 Lazy Expression Compilation** (DAG at-a-time)
  - Examples: **RIOT** [CIDR'09], **TensorFlow** [OSDI'16] **Mahout Samsara** [MLSystems'16], **Dask**
  - Examples w/ control structures: **Weld** [CIDR'17], **OptiML** [ICML'11], **Emma** [SIGMOD'15]
- **#3 Program Compilation** (entire program)
  - Examples: **SystemML** [ICDE'11/PVLDB'16], **Julia**, **Cumulon** [SIGMOD'13], **Tupleware** [PVLDB'15]

Optimization Scope

```
X = read($1); # n x m matrix
y = read($2); # n x 1 vector
maxi = 50; lambda = 0.001;
intercept = $3;
...
r = -(t(X) %*% y);
norm_r2 = sum(r * r); p = -r;
w = matrix(0, ncol(X), 1); i = 0;
while(i<maxi & norm_r2>norm_r2_trgt)
{
   q = (t(X) %*% X %*% p)+lambda*p;
   alpha = norm_r2 / sum(p * q);
   w = w + alpha * p;
   old_norm_r2 = norm_r2;
   r = r + alpha * q;
   norm_r2 = sum(r * r);
   beta = norm_r2 / old_norm_r2;
   p = -r + beta * p; i = i + 1;
}
write(w, $4, format="text");
```

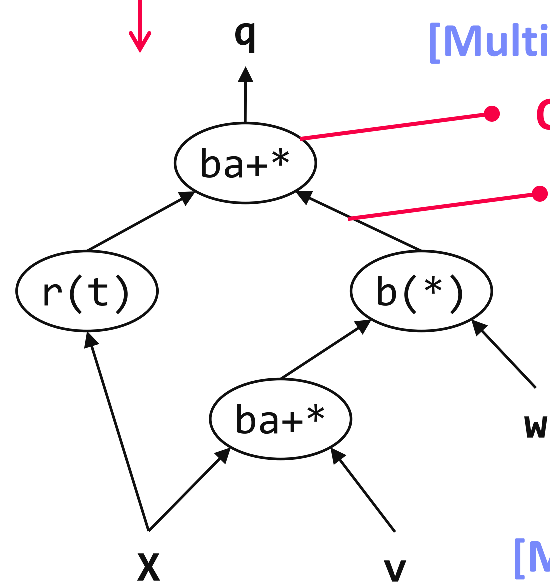
ML Program Compilation / Graphs

- **Script:**

```
while(...) {
    q = t(X) %*% (w * (X %*% v)) ...
}
```

Statement Block Hierarchy

- **Operator DAG** (today's lecture)
  - a.k.a. "graph" (data flow graph)
  - a.k.a. intermediate representation (IR)



[Multiple] DAG roots (outputs)

Operation

Data Dependency

[Multiple] Consumers of Intermediates

No cycles

[Multiple] DAG leafs (inputs)

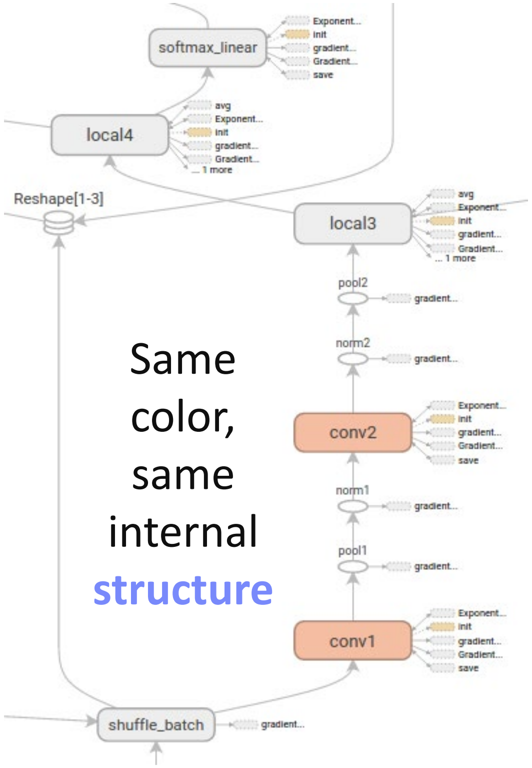
- **Runtime Plan**
  - Compiled runtime plans
    Interpreted plans

```
SPARK mapmmchain X.MATRIX.DOUBLE w.MATRIX.DOUBLE
  v.MATRIX.DOUBLE _mVar4.MATRIX.DOUBLE XtwXv
```

ML Program Compilation / Graphs, cont.

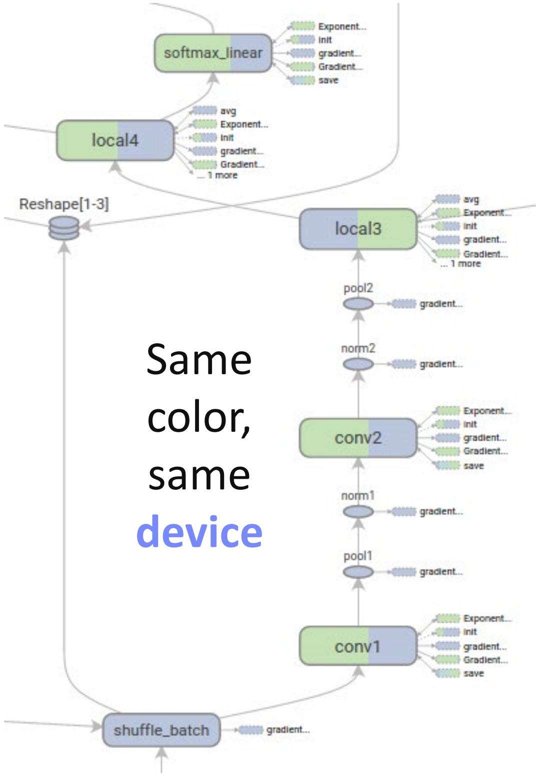
- **Example TF TensorBoard**

(Node) **Structure View**



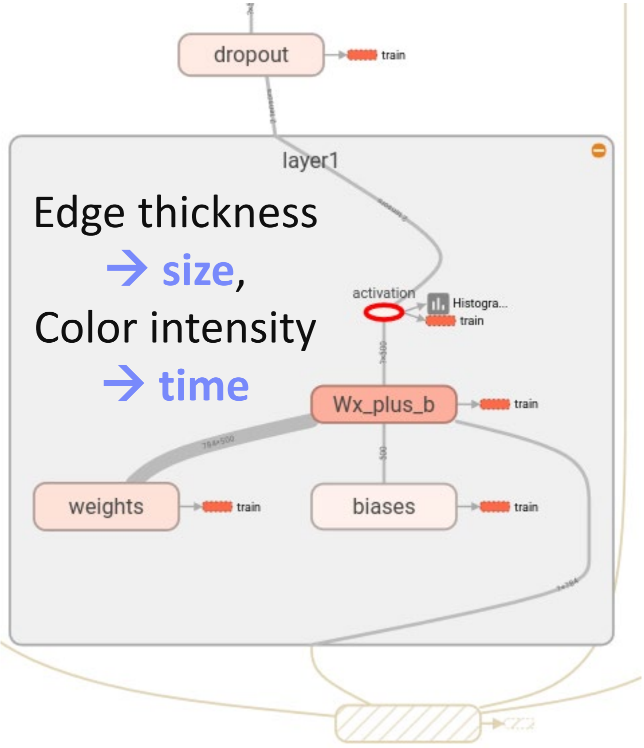
Same color, same internal **structure**

Device View (CPU, GPU)



Same color, same **device**

Tensor Shapes and **Runtime Statistics** (time, mem)



Edge thickness → **size**, Color intensity → **time**

[https://github.com/tensorflow/tensorboard/blob/master/docs/r1/graphs.md]

Compilation Chain

Script

Language
- **Parsing** (syntactic analysis)
- **Live Variable Analysis**
- **Validate** (semantic analysis)

HOPs
- **Construct HOP DAGs**
- **Multiple Rounds**
  - **Static/Dynamic Rewrites**
  - **Intra-/Inter-Procedural Analysis**
- **Static/Dynamic Rewrites**
- **Compute Memory Estimates**

LOPs
- **Construct LOP DAGs** (incl operator selection, hop-lop rewrites)
- **Generate Runtime Program**

Execution Plan

Dynamic Recompilation

[Matthias Boehm et al: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. **IEEE Data Eng. Bull 2014**]

Recap: Basic HOP and LOP DAG Compilation

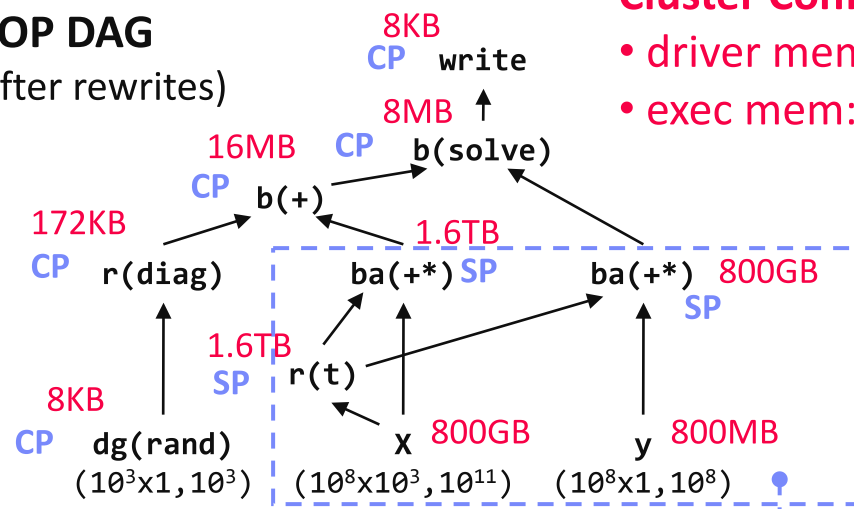
LinregDS (Direct Solve)

```
X = read($1);
y = read($2);
intercept = $3;
lambda = 0.001;
...
if( intercept == 1 ) {
  ones = matrix(1, nrow(X), 1);
  X = append(X, ones);
}
I = matrix(1, ncol(X), 1);
A = t(X) %*% X + diag(I)*lambda;
b = t(X) %*% y;
beta = solve(A, b);
...
write(beta, $4);
```

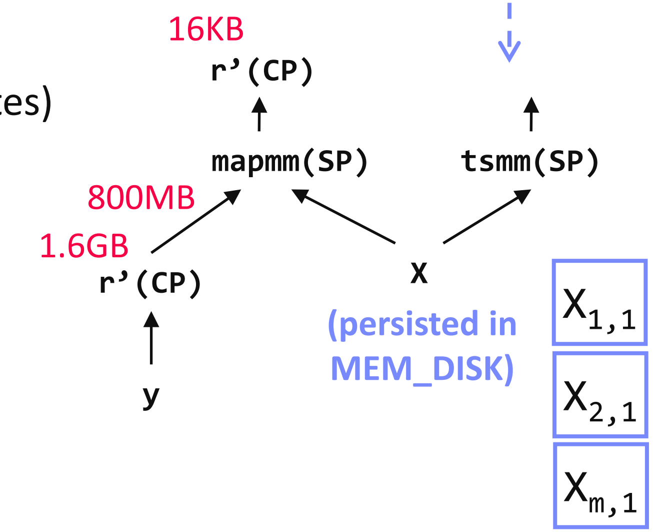
Scenario:
X: $10^8 \times 10^3$, 10^{11}
y: $10^8 \times 1$, 10^8

Cluster Config:
- driver mem: 20 GB
- exec mem: 60 GB

HOP DAG
(after rewrites)



LOP DAG
(after rewrites)



→ Hybrid Runtime Plans:
- Size propagation / memory estimates
- Integrated CP / Spark runtime
- Dynamic recompilation during runtime

→ Distributed Matrices
- Fixed-size (squared) matrix blocks
- Data-parallel operations

Size Inference and Cost Estimation

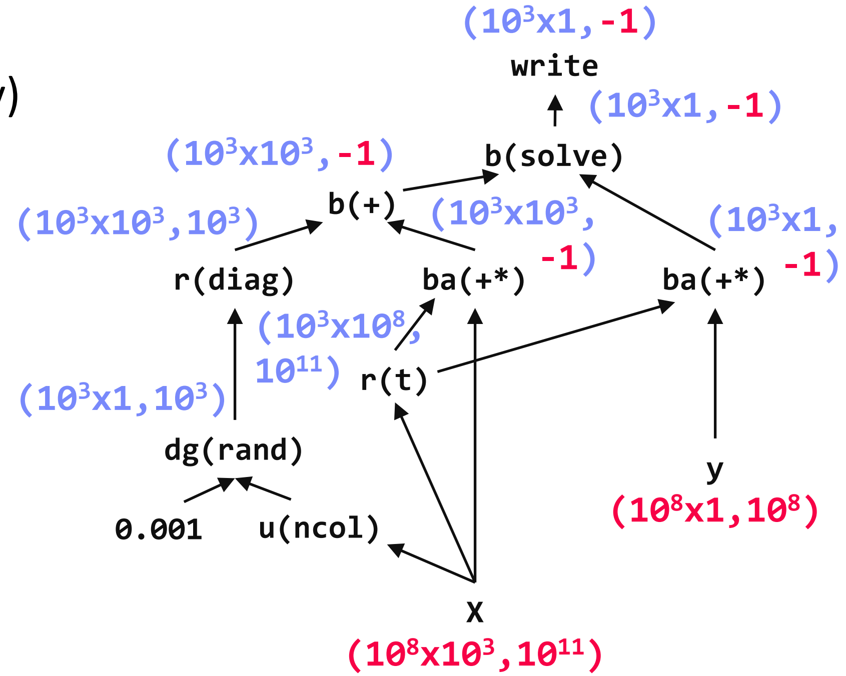
Crucial for Generating Valid Execution Plans
& Cost-based Optimization

Constant and Size Propagation

- **Size Information**
  - Dimensions (#rows, #columns)
  - Sparsity (#nnz/(#rows * #columns))
  - ➔ **memory estimates and costs**

```
X = read($1);
y = read($2);
I = matrix(0.001, ncol(X), 1);
A = t(X) %*% X + diag(I);
b = t(X) %*% y;
beta = solve(A, b);
```

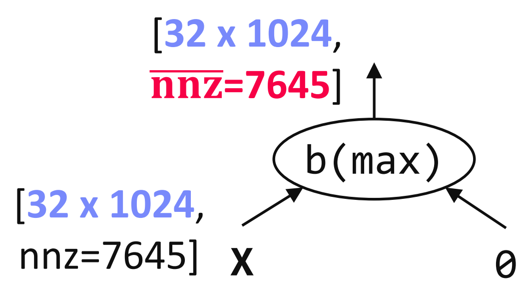
- **Principle: Worst-case Assumption**
  - Necessary for guarantees (memory)
- **DAG-level Size Propagation**
  - **Input:** Size information for leaves
  - **Output:** size information for all operators, -1 if still unknown
  - **Propagation based on operation semantics** (single bottom-up pass over DAG)



Constant and Size Propagation, cont.

- **Example SystemDS**
  - Hop `refreshSizeInformation()` (exact)
  - Hop `inferOutputCharacteristics()`
  - Compiler explicitly differentiates between exact and other size information
  - **Note:** ops like aggregate, ctable, rmEmpty challenging but w/ upper bounds

Example Relu
(rectified linear unit)



- **Example TensorFlow**
  - Operator registrations
  - Shape inference functions

```
REGISTER_OP("Relu")
    .Input("features: T")
    .Output("activations: T")
    .Attr("T: {realnumbertype, qint8}")
    .SetShapeFn(
      shape_inference::UnchangedShape)
```

[Alex Passos: Inside TensorFlow – Eager execution runtime, https://www.youtube.com/watch?v=qjx65mD6nrc, Dec 2019]

Constant and Size Propagation, cont.

- **Constant Propagation**
  - Relies on live variable analysis
  - Propagate constant literals into read-only statement blocks
- **Program-level Size Propagation**
  - Relies on **constant propagation** and **DAG-level size propagation**
  - **Propagate size information across conditional control flow:** size in leafs, DAG-level prop, extract roots
  - **if:** reconcile if and else branch outputs
  - **while/for:** reconcile pre and post loop, reset if pre/post different

```
X = read($1); # n x m matrix
y = read($2); # n x 1 vector
maxi = 50; lambda = 0.001;
if(...){ }
r = -(t(X) %*% y);
r2 = sum(r * r);
p = -r;                        # m x 1
w = matrix(0, ncol(X), 1);     # m x 1
i = 0;
while(i<maxi & r2>r2_trgt) {
   q = (t(X) %*% X %*% p)+lambda*p;
   alpha = norm_r2 / sum(p * q);
   w = w + alpha * p;          # m x 1
   old_norm_r2 = norm_r2;
   r = r + alpha * q;
   r2 = sum(r * r);
   beta = norm_r2 / old_norm_r2;
   p = -r + beta * p;          # m x 1
   i = i + 1;
}
write(w, $4, format="text");
```

Inter-Procedural Analysis

- **Intra/Inter-Procedural Analysis (IPA)**
  - Integrates all size propagation techniques (**DAG+program**, **size+constants**)
  - Intra-function and inter-function size propagation (**called once**, **consistent sizes**, **consistent literals**)

```
X = read($X1)
X = foo(X);
if( $X2 != “ ” ) {
  X2 = cbind(X,
    matrix(1,n,1));
  X2 = foo(X2);
}...
eval(“foo”, X)
```

1M x 1

1M x 2

? x ?

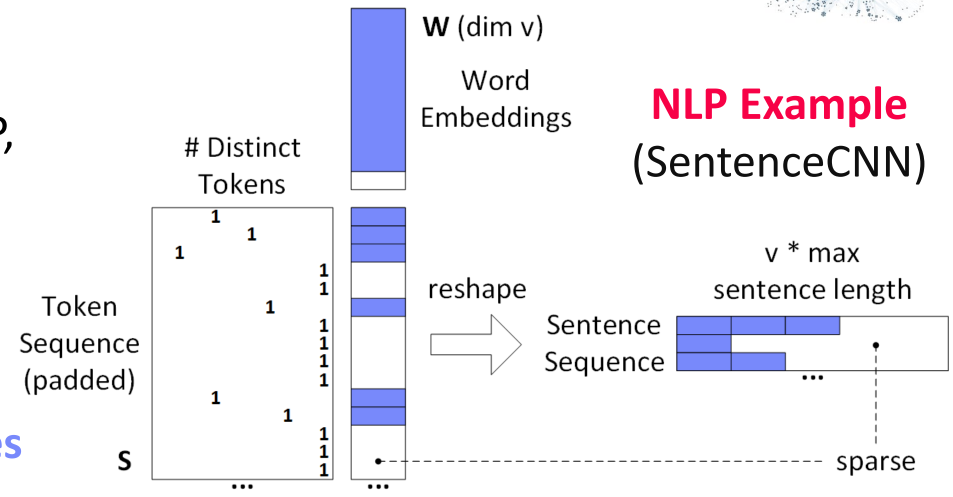
```
foo = function (Matrix[Double] A)
    return (Matrix[Double] B)
{
  B = A - colSums(A);
  if( sum(B!=B)>0 )
    print(“NaNs encountered.”);
}
```

- **Additional IPA Passes (selection)**
  - **Inline functions** (single statement block, small)
  - **Dead code elimination** and simplification rewrites
  - Remove unused functions & flag recompile-once

```
//create ordered list of IPA passes
_passes = new ArrayList<>();
_passes.add(new IPAPassRemoveUnusedFunctions());
_passes.add(new IPAPassFlagFunctionsRecompileOnce());
_passes.add(new IPAPassRemoveUnnecessaryCheckpoints());
_passes.add(new IPAPassRemoveConstantBinaryOps());
_passes.add(new IPAPassPropagateReplaceLiterals());
_passes.add(new IPAPassInlineFunctions());
_passes.add(new IPAPassEliminateDeadCode());
_passes.add(new IPAPassFlagNonDeterminism());
//note: apply rewrites last because statement block rewrites
//might merge relevant statement blocks in special cases, which
//would require an update of the function call graph
_passes.add(new IPAPassForwardFunctionCalls());
_passes.add(new IPAPassApplyStaticAndDynamicHopRewrites());
```

Sparsity Estimation Overview

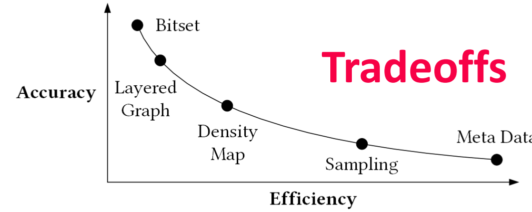
- **Motivation**
  - Sparse input matrices from NLP, graph analytics, recommender systems, scientific computing
  - Sparse intermediates (transform, selection, dropout)
  - Selection/permutation matrices



- **Problem Definition**
  - Sparsity estimates used for **format decisions**, **output allocation**, **cost estimates**
  - Matrix A with sparsity $s_A = nnz(A)/(mn)$ and matrix B with $s_B = nnz(B)/(nl)$
  - Estimate sparsity $s_C = nnz(C)/(ml)$ of matrix product C = A B; d=max(m,n,l)
  - **Assumptions**
    - **A1:** No cancellation errors
    - **A2:** No not-a-number (NaN)

Common assumptions
➔ **Boolean matrix product**

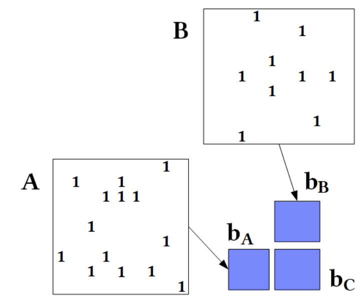
Sparsity Estimation – Estimators



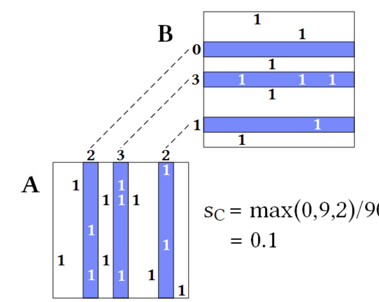
- **#1 Naïve Metadata Estimators**
  - Derive the output sparsity solely from the sparsity of inputs (e.g., SystemDS)

$$\hat{s}_C = 1 - (1 - s_A s_B)^n$$
$$\hat{s}_C = \min(1, s_A n) \cdot \min(1, s_B n)$$

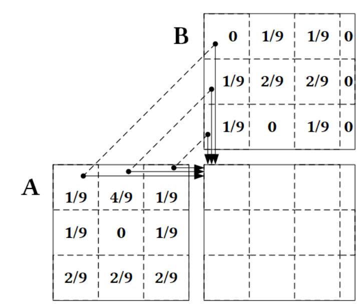
- **#2 Naïve Bitset Estimator**
  - Convert inputs to bitsets, perform Boolean mm (per row)
  - Examples: SciDB [SSDBM'11], NVIDIA cuSparse, Intel MKL



- **#3 Sampling**
  - Take a sample of aligned columns of A and rows of B
  - Sparsity estimated via max of count-products
  - Examples: MatFast [ICDE'17], improvements in paper

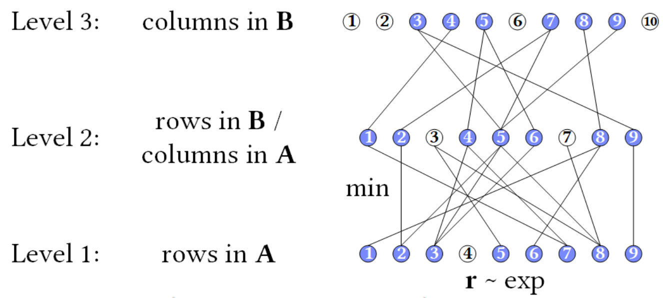


- **#4 Density Map**
  - Store sparsity per b x b block (default b = 256)
  - MM-like estimator (average case estimator for *, probabilistic propagation $s_A + s_B - s_A s_B$ for +)
  - Example: SpMacho [EDBT'15], AT Matrix [ICDE'16]



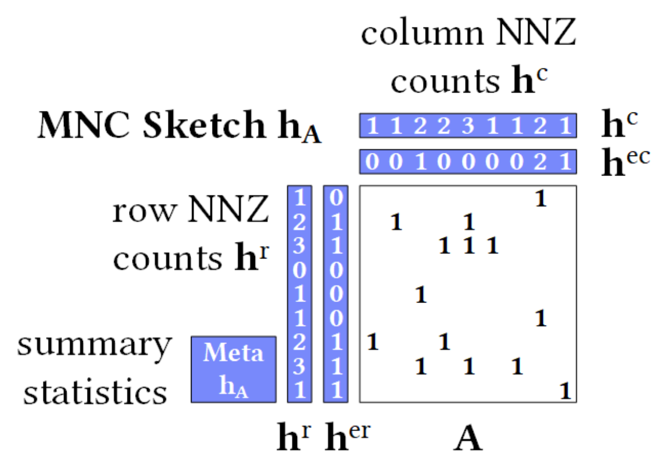
Sparsity Estimation – Estimators, cont.

- **#5 Layered Graph** [J.Comb.Opt.'98]
  - **Nodes:** rows/columns in mm chain
  - **Edges:** non-zeros connecting rows/columns
  - Assign r-vectors ~ exp and propagate via min
  - Estimate over roots (output columns)

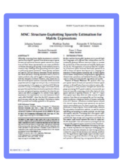


$$\hat{s}_C = \left(\sum_{v \in \text{roots}} \frac{|\mathbf{r}_v| - 1}{\text{sum}(\mathbf{r}_v)} \right) / (ml),$$

- **#6 MNC Sketch** (Matrix Non-zero Count)
  - Create MNC sketch for inputs A and B
  - **Exploitation of structural properties** (e.g., 1 non-zero per row, row sparsity)
  - **Support for matrix expressions** (reorganizations, elementwise ops)
  - Sketch propagation and estimation



$$s_C = \hat{s}_C = h_A^c h_B^r / (ml)$$
$$\text{if } \max(h_A^r) \le 1 \vee \max(h_B^c) \le 1$$



[Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, Peter J. Haas: **MNC:** Structure-Exploiting Sparsity Estimation for Matrix Expressions. **SIGMOD 2019**]

Memory Estimates and Costing

- **Memory Estimates**
  - **Matrix memory estimate :=** based on the dimensions and sparsity, decide the format (sparse, dense) and estimate the size in memory
  - **Operation memory estimate :=** input, intermediates, output
  - **Worst-case sparsity estimates** (**upper bound**)
- **#1 Costing at Logical vs Physical Level**
  - Costing at physical level takes physical ops and rewrites into account but is much more costly
- **#2 Costing Operators/Graphs vs Plans**
  - Costing plans requires heuristics for **# iterations**, **branches** in general
- **#3 Analytical vs Trained Cost Models**
  - Analytical: **estimate I/O and compute workload**
  - Training: **build regression models** for individual ops

A Personal War Story

Physical, Plans,
Trained
[**PVLDB 2014**]

Physical, Plans,
Analytical
[**SIGMOD 2015**]

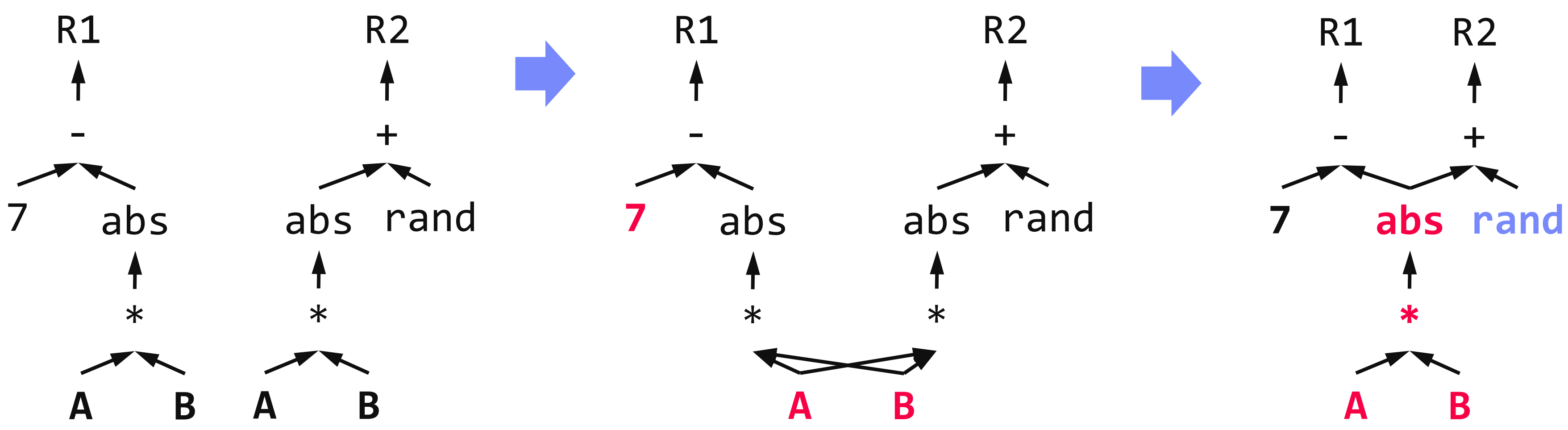
Logical, Graphs,
Analytical
[**PVDLB 2018**]

Rewrites and Operator Selection

Traditional PL Rewrites

- **#1 Common Subexpression Elimination (CSE)**
  - **Step 1:** Collect and **replace leaf nodes** (variable reads and literals)
  - **Step 2:** recursively **remove CSEs bottom-up** starting at the leafs by merging nodes with same inputs (**beware non-determinism**)
  - **Example:**

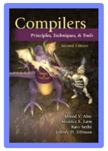
```
R1 = 7 - abs(A * B)
R2 = abs(A * B) + rand()
```



Traditional PL Rewrites, cont.

- **#2 Constant Folding**
  - After constant propagation, fold sub-DAGs over literals into a single literal
  - **Approach:** recursively compile and execute runtime instructions with special handling of one-side constants
  - **Example** (GLM Binomial probit):

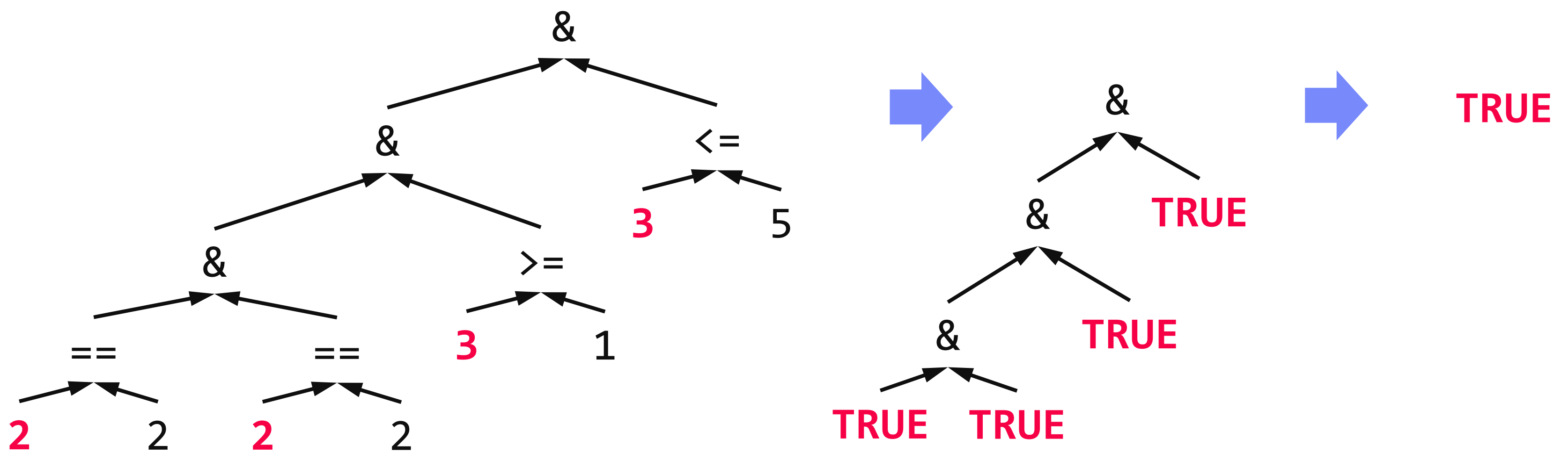
[A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. Compilers – Principles, Techniques, & Tools. Addison-Wesley, 2007]



Turing Award '20

```
ncol_y == 2 & dist_type == 2
  & link_type >= 1 & link_type <= 5
```

```
2 == 2 & 2 == 2 & 3 >= 1 & 3 <= 5
```



Traditional PL Rewrites, cont.

- **#3 Branch Removal**
  - Applied after **constant propagation** and **constant folding**
  - **True predicate:** replace if statement block with if-body blocks
  - **False predicate:** replace if statement block with else-body block, or remove

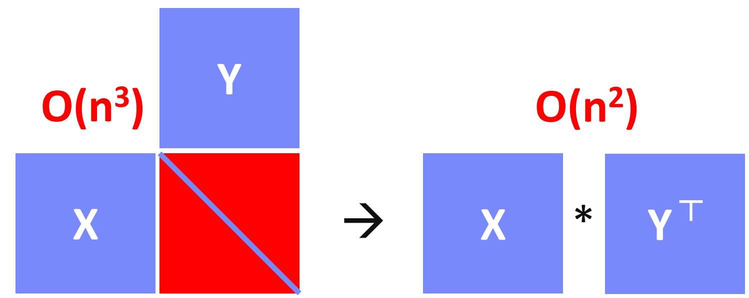
- **#4 Merge of Statement Blocks**
  - **Merge sequences of unconditional blocks** (s1,s2) into a single block
  - Connect matching DAG roots of s1 with DAG inputs of s2

LinregDS (Direct Solve)

```
X = read($1);
y = read($2);
intercept = 0;
lambda = 0.001;
...          FALSE
if( intercept == 1 ) {
  ones = matrix(1, nrow(X), 1);
  X = cbind(X, ones);
}
I = matrix(1, ncol(X), 1);
A = t(X) %*% X + diag(I)*lambda;
b = t(X) %*% y;
beta = solve(A, b);
...
write(beta, $4);
```

Static/Dynamic Simplification Rewrites

- **Examples of Static Rewrites**
  - `trace(X%*%Y)   → sum(X*t(Y))`
  - `sum(X+Y)       → sum(X)+sum(Y)`
  - `(X%*%Y)[7,3]   → X[7,]%*%Y[,3]`
  - `sum(t(X))      → sum(X)`
  - `rand()*7       → rand(,min=0,max=7)`
  - `sum(lambda*X)  → lambda * sum(X);`



[Matthias Boehm et al: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. **IEEE Data Eng. Bull 2014**]

- **Examples of Dynamic Rewrites**
  - `t(X) %*% y      → t(t(y) %*% X) s.t. costs`
  - `X[a:b,c:d]=Y → X = Y iff dims(X)=dims(Y)`
  - `(...) * X     → matrix(0, nrow(X), ncol(X)) iff nnz(X)=0`
  - `sum(X^2)      → t(X)%*%X; rowSums(X) → X iff ncol(X)=1`
  - `sum(X%*%Y)    → sum(t(colSums(X))*rowSums(Y)) iff ncol(X)>t`

Static/Dynamic Simplification Rewrites, cont.

[Rasmus Munk Larsen, Tatiana Shpeisman: TensorFlow Graph Optimizations, **Guest Lecture Stanford 2019**]

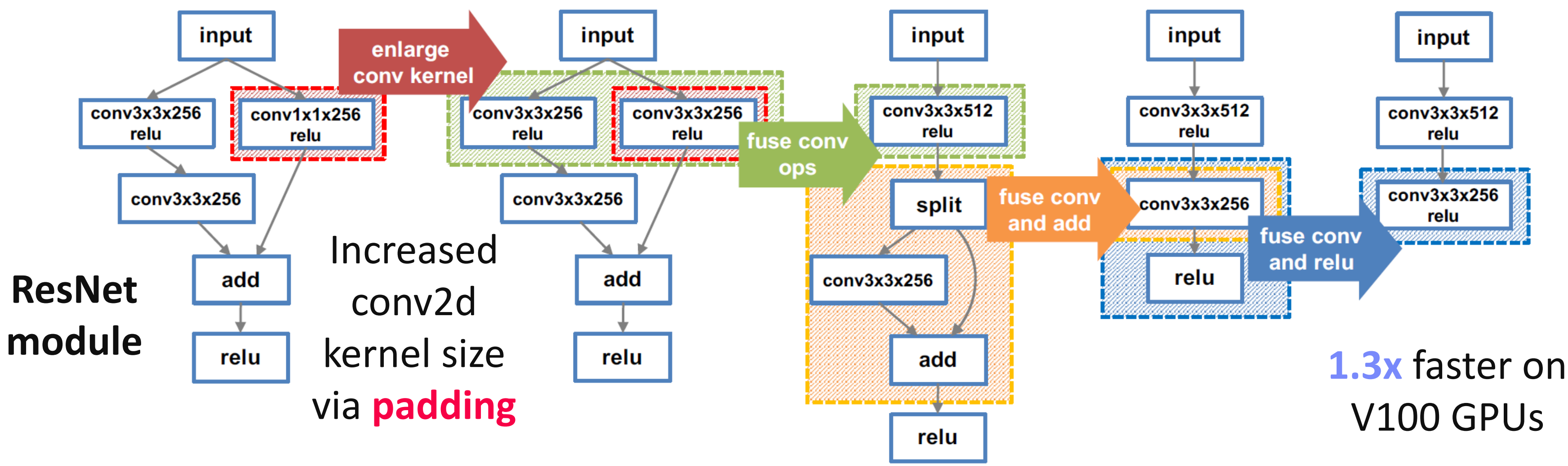
- **TF Constant Push-Down**
  - `Add(c1,Add(x,c2)) → Add(x,c1+c2)`
  - `ConvND(c1*x,c2) → ConvND(x,c1*c2)`
- **TF Arithmetic Simplifications**
  - Flattening: a+b+c+d → `AddN(a, b, c, d)`
  - Hoisting: `AddN(x * a, b * x, x * c) → x * AddN(a+b+c)`
  - Reduce Nodes Numeric: `x+x+x → 3*x`
  - Reduce Nodes Logicial: `!(x > y) → x <= y`
- **TF Broadcast Minimization**
  - `(M1+s1) + (M2+s2) → (M1+M2) + (s1+s2)`
- **TF Better use of Intrinsics**
  - `Matmul(Transpose(X), Y) → Matmul(X, Y, transpose_x=True)`

SystemML/SystemDS
`RewriteElementwise-MultChainOptimization`
(orders and collapses matrix, vector, scalar op chains)

Static/Dynamic Simplification Rewrites, cont.

- **Relaxed DNN Graph Substitutions**
  - Allow substitutions that preserve semantics, no matter if faster/slower
  - Backtracking search

[Zhihao Jia, James J. Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, Alex Aiken: Optimizing DNN Computation with Relaxed Graph Substitutions. **MLSys 2019**]



- **Additional Algorithms**
  - Partial order of substitutions w/ pruning
  - Dynamic programming → substitutions

[Jingzhi Fang, Yanyan Shen, Yue Wang, Lei Chen: Optimizing DNN Computation Graph using Graph Substitutions. **PVLDB 13(11) 2020**]

Static/Dynamic Simplification Rewrites, cont.

PYTORCH

- **Rewrites in PyTorch** (Torch Script JIT)
  - Misc: Canonicalization, erase number types and no-ops
  - Fuse linear, fuse relu, fuse graph pipeline
  - Peephole simplifications (e.g., for dtype management)
  - Inlining and loop unrolling
  - Concatenation and fusion rewrites:

[https://github.com/pytorch/pytorch/blob/master/torch/csrc/jit/passes/subgraph_rewrite.cpp]

```
void SubgraphRewriter::RegisterDefaultPatterns() {
  // TODO: Add actual patterns (like Conv-Relu).
  RegisterRewritePattern(
      R"IR(
graph(%x, %w, %b):
  %c = aten::conv(%x, %w, %b)
  %r = aten::relu(%c)
  return (%r))IR",
      R"IR(
graph(%x, %w, %b):
  %r = aten::convrelu(%x, %w, %b)
  return (%r))IR",
      {{"r", "c"}});
}
```

subgraph_rewrite.cpp
(extracted Mar 17, 2022)

Vectorization and Incremental Computation

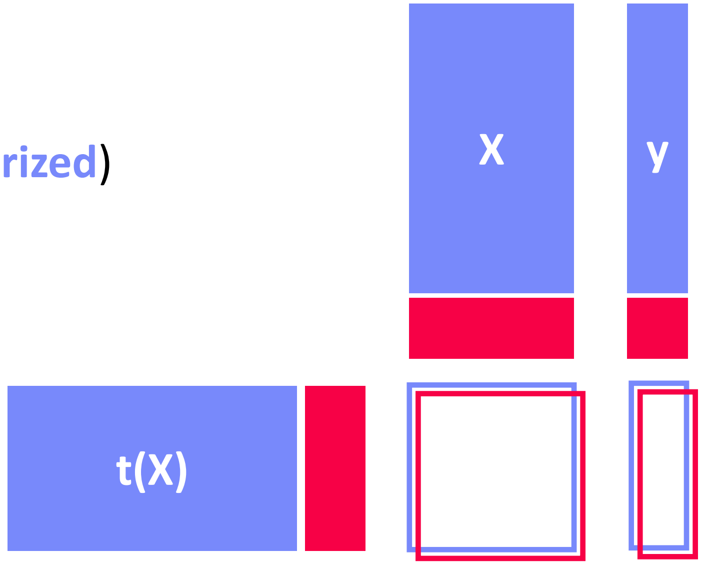
- **Loop Transformations** (e.g., **OptiML**, **SystemML**)
  - **Loop vectorization**
  - Loop hoisting

```
for(i in a:b)
   X[i,1] = Y[i,2] + Z[i,1]
```

```
→ X[a:b,1] = Y[a:b,2] + Z[a:b,1]
```

- **Incremental Computations**
  - **Delta update rules** (e.g., **LINVIEW**, **factorized**)
  - Incremental iterations (e.g., **Flink**)

```
A = t(X) %*% X + t(ΔX) %*% ΔX
b = t(X) %*% y + t(ΔX) %*% Δy
```



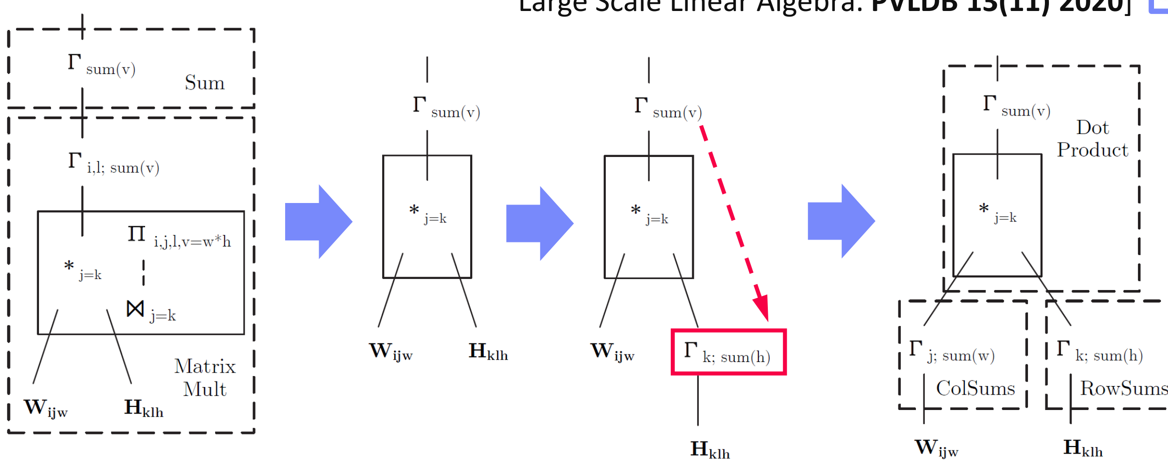
- **Update In-Place**
  - **SystemDS:** via rewrites (**guaranteed applicability**)
  - **R:** via reference counting
  - **Julia:** by default, otherwise explicit **B = copy(A)** necessary

Excursus: Automatic Rewrite Generation

- **SPOOF/SPORES (Sum-Product Optim.)**
  - **Break up** LA ops into basic ops (RA)
  - **Elementary sum-product/RA rewrites**
  - **Example:** `sum(W%*%H)`

[Tarek Elgamal et al: SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. **CIDR 2017**]

[Yisu Remy Wang et al: SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. **PVLDB 13(11) 2020**]



- **TASO (Super Optimization)**
  - List of operator specifications and properties
  - Automatic **generation/verification of graph substitutions** and **data layouts** via cost-based backtracking search

[Zhihao Jia et al: TASO: optimizing deep learning computation with automatic generation of graph substitutions. **SOSP 2019**]

Matrix Multiplication Chain Optimization

- **Optimization Problem**
  - Matrix multiplication chain of n matrices M_1, M_2, …M_n (associative)
  - Optimal parenthesization of the product $M_1 M_2 \ldots M_n$

[t(X) 1kx1k   X 1kx1k]   Z 1   →   t(X) 1kx1k   [X 1kx1k   p 1]

2,002 MFLOPs   4 MFLOPs

Size propagation and sparsity estimation

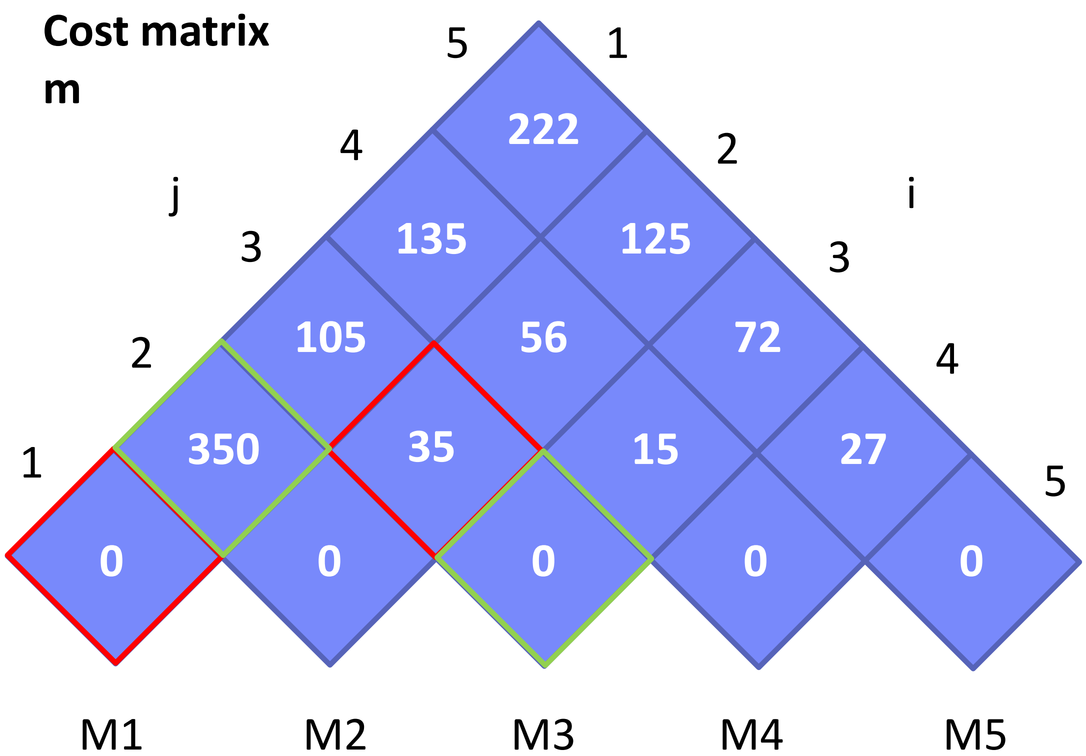
- **Search Space Characteristics**
  - Naïve exhaustive: Catalan numbers → $\Omega(4^n / n^{3/2})$)
  - DP applies: (1) optimal substructure, (2) overlapping subproblems
  - Textbook DP algorithm: $\Theta(n^3)$ time, $\Theta(n^2)$ space
    - Examples: SystemML ‘14, RIOT (‘09 I/O costs), SpMachO (‘15 sparsity)
  - Best known: O(n log n)

n	C_{n-1}
5	14
10	4,862
15	2,674,440
20	1,767,263,190
25	1,289,904,147,324

[T. C. Hu, M. T. Shing: Computation of Matrix Chain Products. Part II. **SIAM J. Comput.** 13(2): 228-251, 1984]

Matrix Multiplication Chain Optimization, cont.

M1	M2	M3	M4	M5
10x7	7x5	5x1	1x3	3x9



m[1,3] = min(
m[1,1] + m[2,3] + p1p2p4,
m[1,2] + m[3,3] + p1p3p4)

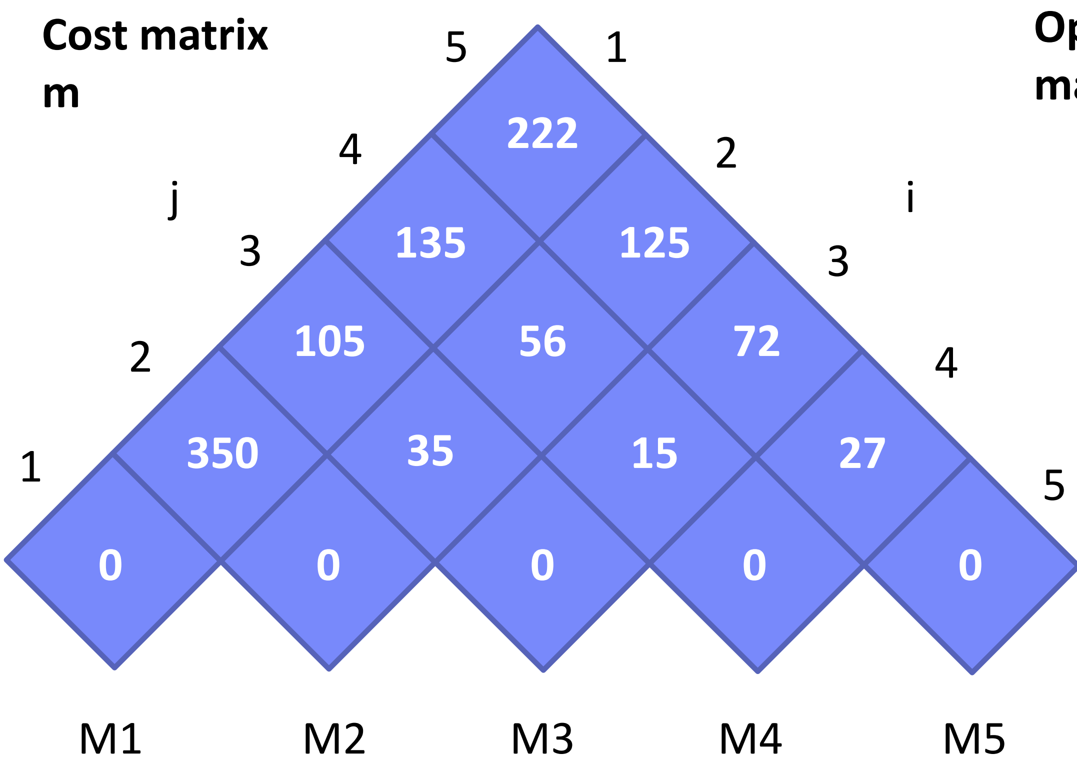
= min(
0 + 35 + 10*7*1,
350 + 0 + 10*5*1)

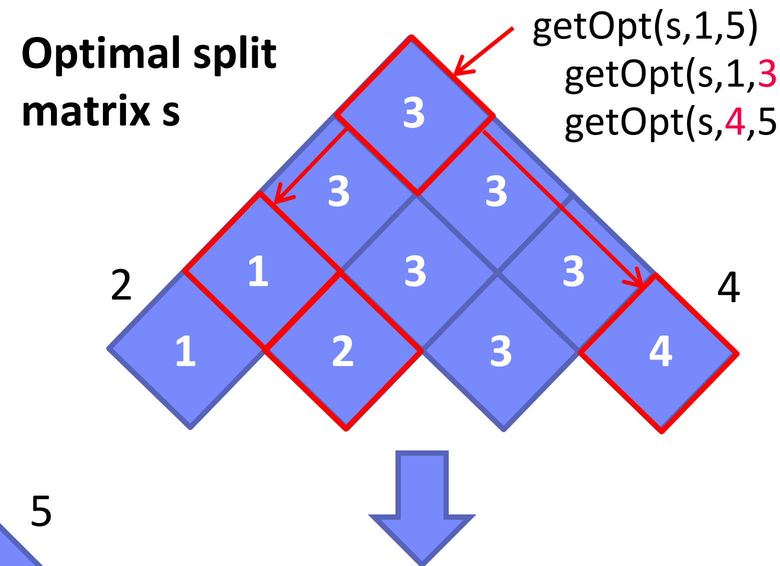
= min(
105,
400)

[T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, Third Edition, **The MIT Press**, pages 370-377, 2009]

Matrix Multiplication Chain Optimization, cont.

M1	M2	M3	M4	M5
10x7	7x5	5x1	1x3	3x9





(M1 M2 M3 M4 M5)

((M1 M2 M3) (M4 M5))

((M1 (M2 M3)) (M4 M5))

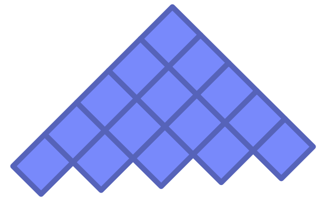
➔ ((M1 (M2 M3)) (M4 M5))

➔ Open questions: DAGs; other operations, sparsity joint opt w/ rewrites, CSE, fusion, and physical operators

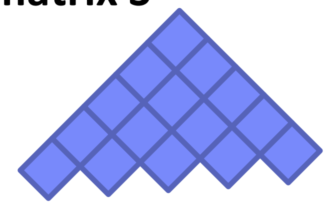
Matrix Multiplication Chain Optimization, cont.

- **Sparsity-aware mmchain Opt**
  - Additional n x n sketch matrix e

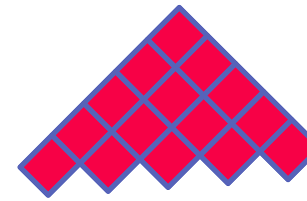
Cost matrix M



Optimal split matrix S



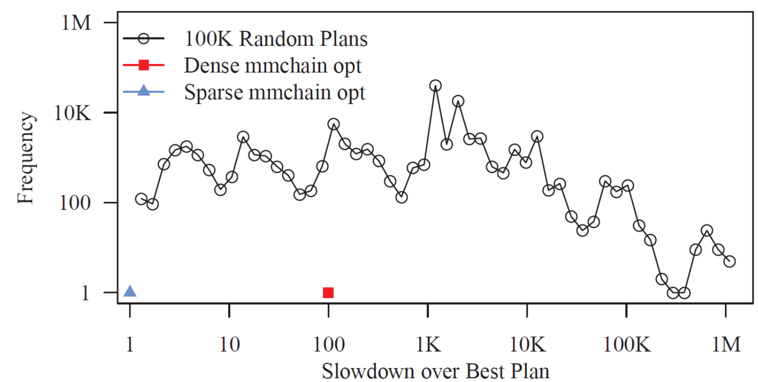
Sketch matrix E

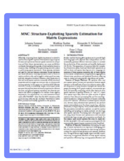


  - Sketch propagation for optimal subchains (currently for all chains)
  - Modified cost computation via MNC sketches (**number FLOPs for sparse** instead of dense mm)

$$C_{i,j} = \min_{k \in [i,j-1]} \begin{pmatrix} C_{i,k} + C_{k+1,j} \\ + \boldsymbol{E_{i,k}.h^c E_{k+1,j}.h^r} \end{pmatrix}$$

Example: n=20 matrices

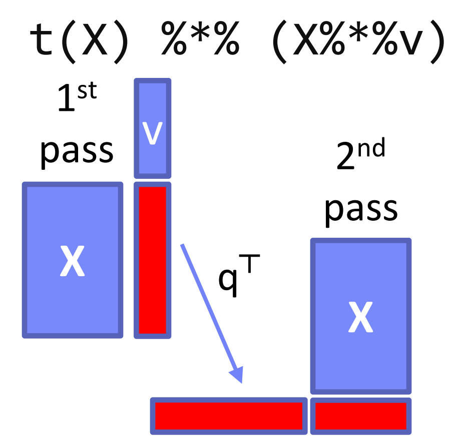




[Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, Peter J. Haas: **MNC:** Structure-Exploiting Sparsity Estimation for Matrix Expressions. **SIGMOD 2019**]

Physical Operator Selection

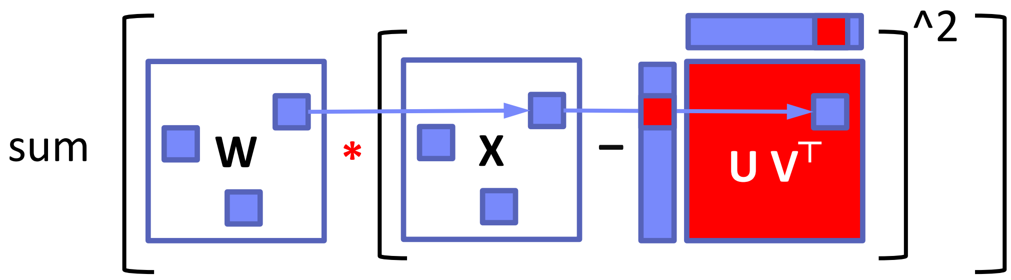
- **Common Selection Criteria**
  - **Data and cluster characteristics** (e.g., data size/shape, memory, parallelism)
  - **Matrix/operation properties** (e.g., diagonal/symmetric, sparse-safe ops)
  - **Data flow properties** (e.g., co-partitioning, co-location, data locality)
- **#0 Local Operators**
  - SystemML mm, tsmm, mmchain; Samsara/Mllib local
- **#1 Special Operators** (special patterns/sparsity)
  - SystemML **tsmm**, **mapmmchain**; Samsara AtA
- **#2 Broadcast-Based Operators** (aka broadcast join)
  - SystemML **mapmm**, **mapmmchain**
- **#3 Co-Partitioning-Based Operators** (aka improved repartition join)
  - SystemML **zipmm**; Emma, Samsara OpAtB
- **#4 Shuffle-Based Operators** (aka repartition join)
  - SystemML **cpmm**, **rmm**; Samsara OpAB



Sparsity-Exploiting Operators

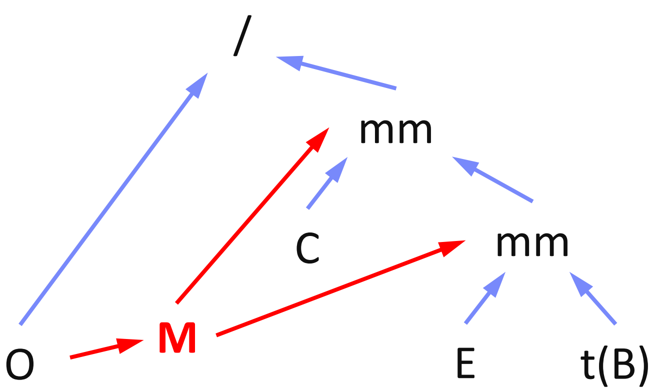
- **Goal:** Avoid dense intermediates and unnecessary computation
- **#1 Fused Physical Operators**
  - E.g., SystemML [PVLDB'16] wsloss, wcemm, wdivmm
  - Selective computation over non-zeros of "sparse driver"

`sum(W * (X – U %*% t(V))^2)`



- **#2 Masked Physical Operators**
  - E.g., Cumulon MaskMult [SIGMOD'13]
  - Create mask of "sparse driver"
  - Pass mask to single masked matrix multiply operator

`O / (C %*% E %*% t(B))`



Runtime Adaptation

ML Systems w/ Optimizing Compiler

Apache SystemML™

Terminology Ahead-of-Time / Just-in-Time

- **Ahead-of-Time Compilation**
  - Originating from compiled languages like C, C++
  - **#1** **Program compilation** at different abstraction levels
  - **#2** **Inference program compilation** & packaging
- **Just-In-Time Compilation** (at runtime for specific data/HW)
  - Originating from JIT-compiled languages like Java, C#
  - **#1** **Lazy expression evaluation** + optimization
  - **#2** Program/function compilation **with recompilation**

(LLVM)

- **Excursus: Java JIT**
  - **#1** Start w/ Java bytecode interpretation by JVM → **fast startup**
  - **#2** **Tiered JIT compile** (cold, warm, hot, very hot, scorching) → **performance**
  - Trace statistics (frequency, time) at method granularity
  - Note: `-XX:+PrintCompilation`

PL

Issues of Unknown or Changing Sizes

- **Problem of unknown/changing sizes**
  - Unknown or changing sizes and sparsity of intermediates
    These unknowns lead to very conservative fallback plans (distributed ops)

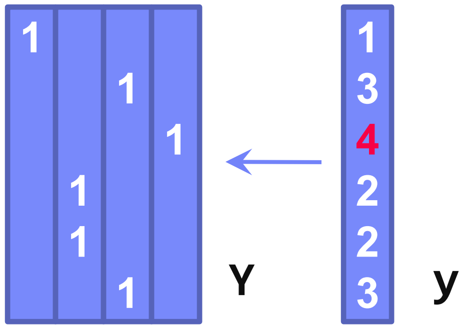
- **#1 Control Flow**
  - Branches and loops
  - Complex function call graphs
  - User-Defined Functions

```
X = read('/tmp/X.csv');
if( intercept )
  X = cbind(X, matrix(1,nrow(X),1));

Z = foo(X) + X; # size of + and Z?
```

- **#2 Data-Dependencies**
  - Data-dependent operators
    (e.g., table, rmEmpty, aggregate)
  - Computed size expressions

```
d = dout[,(t-2)*M+1:(t-1)*M];

cur_Q = matrix (0, 1, 2*ncur);
cur_S = matrix (0, 1, ncur*dist);
```

```
Y = table(seq(1,nrow(X)), y);
grad = t(X) %*% (P - Y);
```



Ex.: Multinomial Logistic Regression

Issues of Unknown or Changing Sizes, cont.

- **#3 Changing Dims and Sparsity**
  - Iterative feature selection workloads
  - Changing dimensions or sparsity
  - ➔ Same code with different data
- **#4 API Limitations**
  - Precompiled scripts/programs (inputs unavailable)
- **(#5 Compiler Limitations)**

Ex: Stepwise LinReg

```
while( continue ) {
   parfor( i in 1:n ) {
      if( !fixed[1,i] ) {
         Xi = cbind(Xg, X[,i])
         B[,i] = lm(Xi,y)
      }
   }
   # add best to Xg (AIC)
}
```

➔ **Dynamic recompilation techniques** as robust fallback strategy

- Shares goals and challenges with adaptive query processing
- However, ML domain-specific techniques and rewrites

Recompilation

[Matthias Boehm et al: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. **IEEE Data Eng. Bull 2014**]

Script

~100 ms — Language
- **Parsing** (syntactic analysis)
- **Live Variable Analysis**
- **Validate** (semantic analysis)

~10 ms — HOPs — **Multiple Rounds**
- **Construct HOP DAGs**
- **Static/Dynamic Rewrites**
- **Intra-/Inter-Procedural Analysis**

~1 ms
- **Static/Dynamic Rewrites**
- **Compute Memory Estimates**

LOPs
- **Construct LOP DAGs** (incl operator selection, hop-lop rewrites)
- **Generate Runtime Program**

Execution Plan

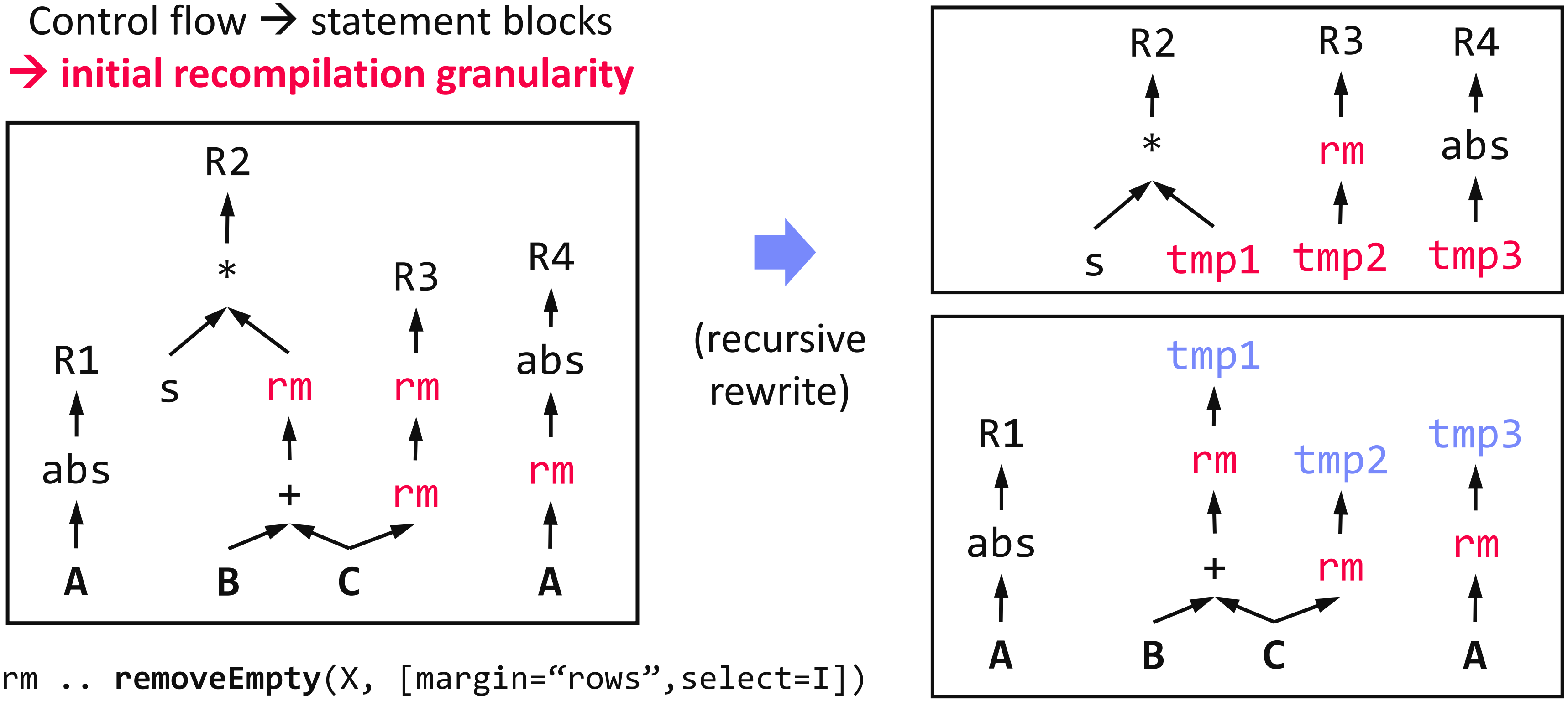
Dynamic Recompilation

Other systems w/ recompile: **SciDB**, **MatFast**

Dynamic Recompilation

- **Compile-time Decisions**
  - **Split HOP DAGs for recompilation:** prevent unknowns but keep DAGs as large as possible; split after reads w/ unknown sizes and specific operators
  - **Mark HOP DAGs for recompilation:** Spark due to unknown sizes / sparsity

Control flow → statement blocks
→ **initial recompilation granularity**



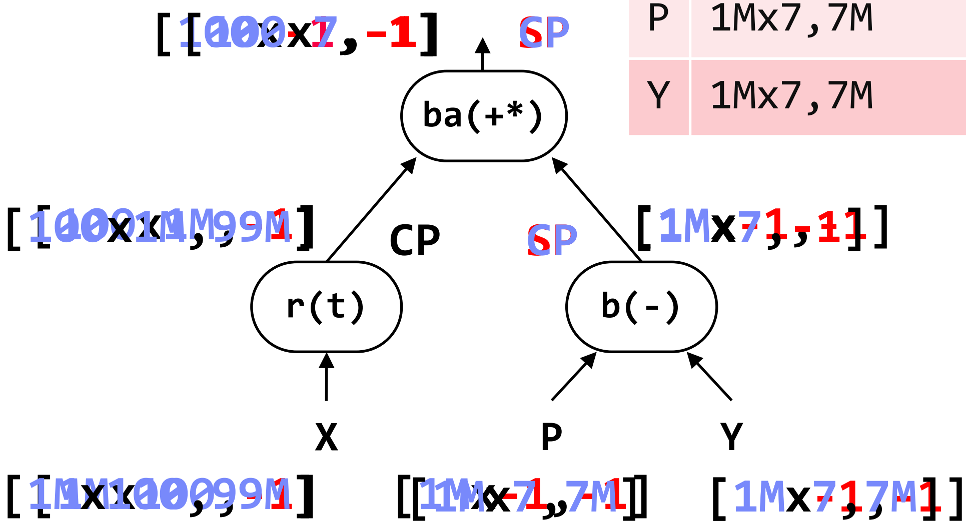
`rm .. removeEmpty(X, [margin="rows",select=I])`

Dynamic Recompilation, cont.

- **Dynamic Recompilation at Runtime** on recompilation hooks (last level program blocks, predicates, recompile once functions)
  - **Deep Copy DAG**
  - Replace Literals
  - **Update DAG Statistics**
  - **Dynamic Rewrites**
  - **Recompute Memory Estimates**
  - **[Codegen]**
  - **Generate Runtime Instructions**

Symbol Table

X	1Mx100,99M
P	1Mx7,7M
Y	1Mx7,7M



Dynamic Recompilation, cont.

- **Recompile Once Functions**
  - Unknowns due to inconsistent or unknown call size information
  - IPA marks functions as "recompile once", if it contains loops
  - **Recompile the entire function on entry + disable unnecessary recompile**

```
foo = function(Matrix[Double] A)
    # recompiled w/ size of A
    return (Matrix[Double] C)
{
    C = rand(nrow(A),1) + A;
    while(...)
      C = C / rowSums(C) * s
}
```

- **Recompile parfor Loops**
  - Unknown sizes and iterations
  - **Recompile parfor loop on entry + disable unnecessary recompile**
  - Create independent DAGs for individual parfor workers

```
while( continue ) {
   parfor( i in 1:n ) {
      if( !fixed[1,i] ) {
         Xi = cbind(Xg, X[,i])
         B[,i] = lm(Xi,y)
      }
   }
   # add best to Xg (AIC)
}
```

Operator Fusion & JIT Compilation

(aka Code Generation)

Many State-of-the-Art ML Systems,
especially for DNNs and numerical computation

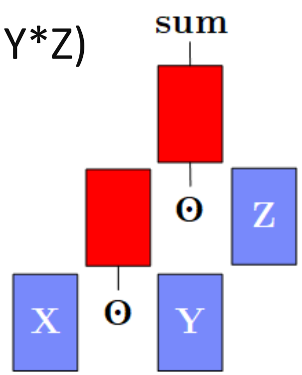
Motivation: Fusion

[Matthias Boehm et al.: On Optimizing Operator Fusion Plans for Large-Scale ML in SystemML. **PVLDB 2018**]

- **Data Flow Graphs** (better data access)
  - DAGs of linear algebra (LA) operations and statistical functions
  - Materialized intermediates → ubiquitous fusion opportunities

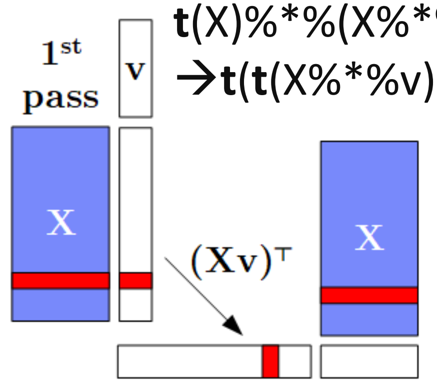
a) Intermediates

sum(X*Y*Z)

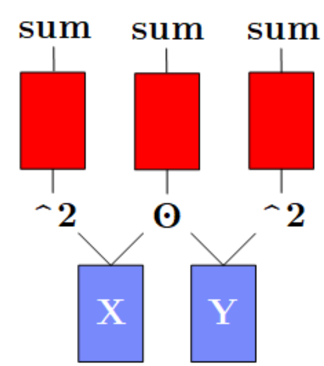


b) Single-Pass

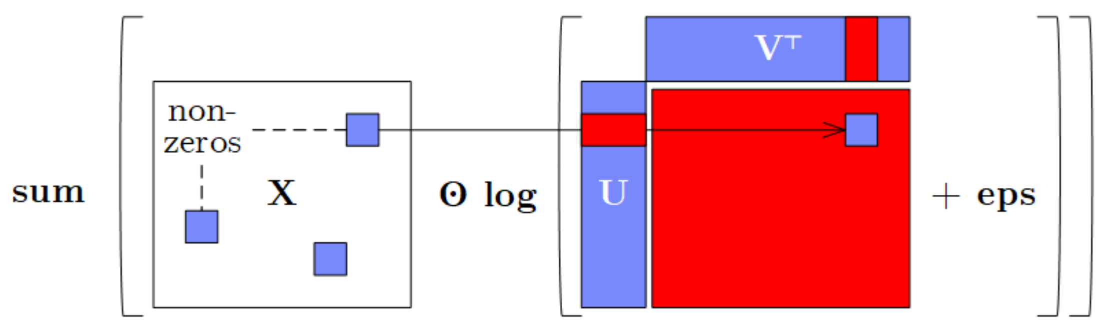
t(X)%*%(X%*%v)
→**t**(**t**(X%*%v)%*%X)



c) Multi-Aggregates



d) Sparsity Exploitation

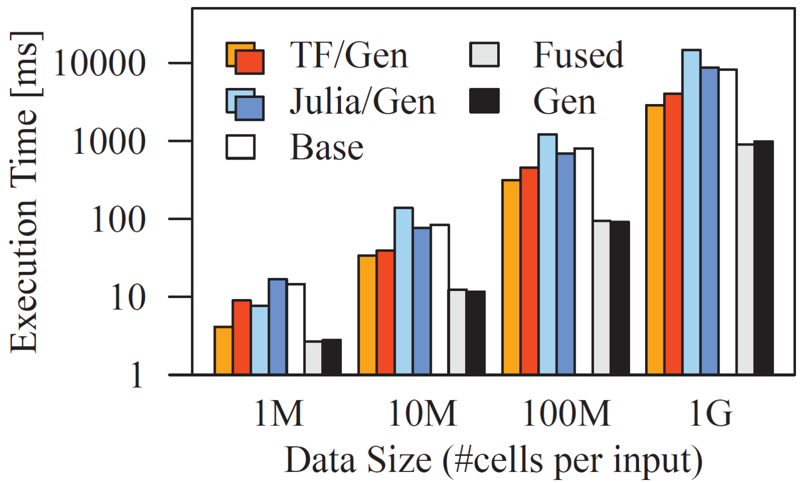


Motivation: Fusion, cont.

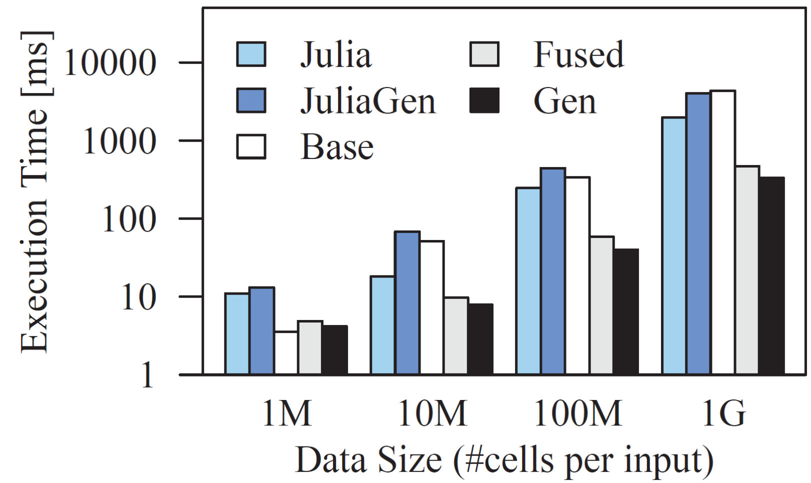
Beware: SystemML 1.0, Julia 0.6.2, TensorFlow 1.5

Cell Template: `sum(X*Y*Z)`

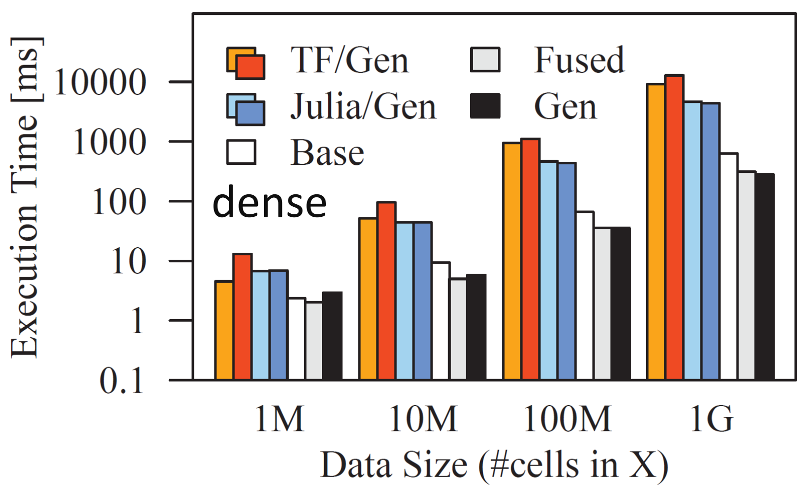
dense



sparse (0.1)



Row: `t(X)%*%(w*(X%*%v))`

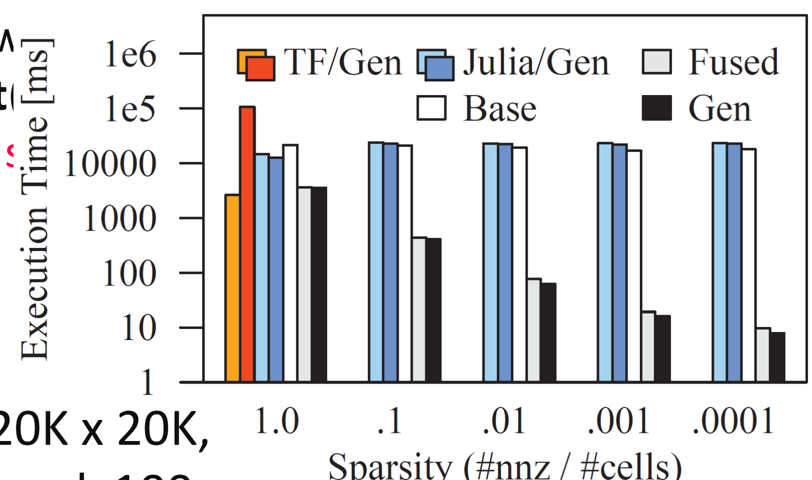


Outer: `sum(X*log(U%*%t(V)+1e-15))`

TF w

→ t(

9.2



20K x 20K, rank 100

Motivation: Just-In-Time Compilation

- **Operator Kernels** (**better code**)
  - Specialization opportunities: data types, shapes, and operator graphs
  - Heterogeneous hardware: CPUs, GPUs, FPGAs, ASICs x architectures

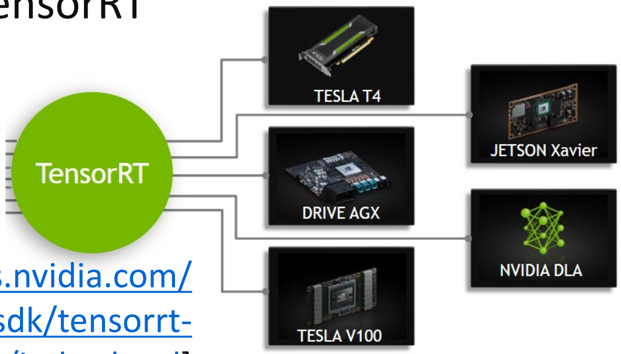
- **#1 CPU Architecture**
  - Specialize to available instructions sets
  - Register allocation and assignment, etc

Examples: x86-64, sparc, amd64, arm, ppc

- **#2 Heterogeneous Hardware**
  - JIT compilation for custom-build ASICs with HW support for ML ops
  - Different architectures of devices

Example: NVIDIA TensorRT

GPU Platforms



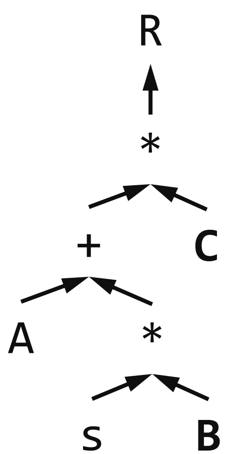
- **#3 Custom ML Program**
  - Operator graphs and sizes

[https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html]

Operator Fusion Overview

- **Related Research Areas**
  - DB: **query compilation**
  - HPC: **loop fusion, tiling, and distribution** (**NP complete**)
  - ML: **operator fusion** (dependencies given by data flow graph)

- **Example Operator Fusion**




```
for( i in 1:n )
  tmp1[i,1] = s * B[i,1];
for( i in 1:n )
  tmp2[i,1] = A[i,1] + tmp1[i,1];
for( i in 1:n )
  R[i,1] = tmp2[i,1] * C[i,1];
```

```
for( i in 1:n )
  R[i,1] = (A[i,1] + s*B[i,1]) * C[i,1];
```

Memory Bandwidth:
L1 core: 1TB/s
L3 socket: 400GB/s
Mem: 100 GB/s

[https://software.intel.com/en-us/articles/memory-performance-in-a-nutshell]

Automatic Operator Fusion System Landscape

System	Year	Approach	Sparse	Distr.	Optimization
BTO	2009	Loop Fusion	No	No	k-Greedy, cost-based
Tupleware	2015	Loop Fusion	No	Yes	Heuristic
Kasen	2016	Templates	(Yes)	Yes	Greedy, cost-based
SystemML	**2017**	**Templates**	**Yes**	**Yes**	**Exact, cost-based**
Weld	2017	Templates	(Yes)	Yes	Heuristic
Taco	2017	Loop Fusion	Yes	No	Manuel
Julia	2017	Loop Fusion	Yes	No	Manuel
Tensorflow XLA	2017	Loop Fusion	No	No	Manuel/Heuristic
Tensor Comprehensions	2018	Loop Fusion	No	No	Evolutionary, cost-based
TVM	2018	Loop Fusion	No	No	ML/cost-based
PyTorch	2019	Loop Fusion	No	No	Manual/Heuristic
JAX	2019	N/A	No	No	See TF XLA

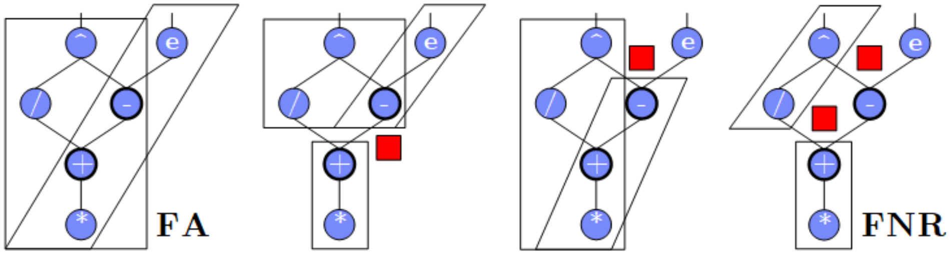
JIT

A Case for Optimizing Fusion Plans

Apache SystemML™

- **Problem:** Fusion heuristics → **poor plans** for complex DAGs (cost/structure), sparsity exploitation, and local/distributed operations
- **Goal: Principled approach for optimizing fusion plans**

$$\mathbf{C} = \mathbf{A} + \mathrm{s} * \mathbf{B}$$
$$\mathbf{D} = (\mathbf{C}/2)\hat{\ }(\mathbf{C}\text{-}1)$$
$$\mathbf{E} = \exp(\mathbf{C}\text{-}1)$$



- **#1 Materialization Points** (e.g., for multiple consumers)
- **#2 Sparsity Exploitation** (and ordering of sparse inputs)

```
Y + X * (U %*% t(V))
```

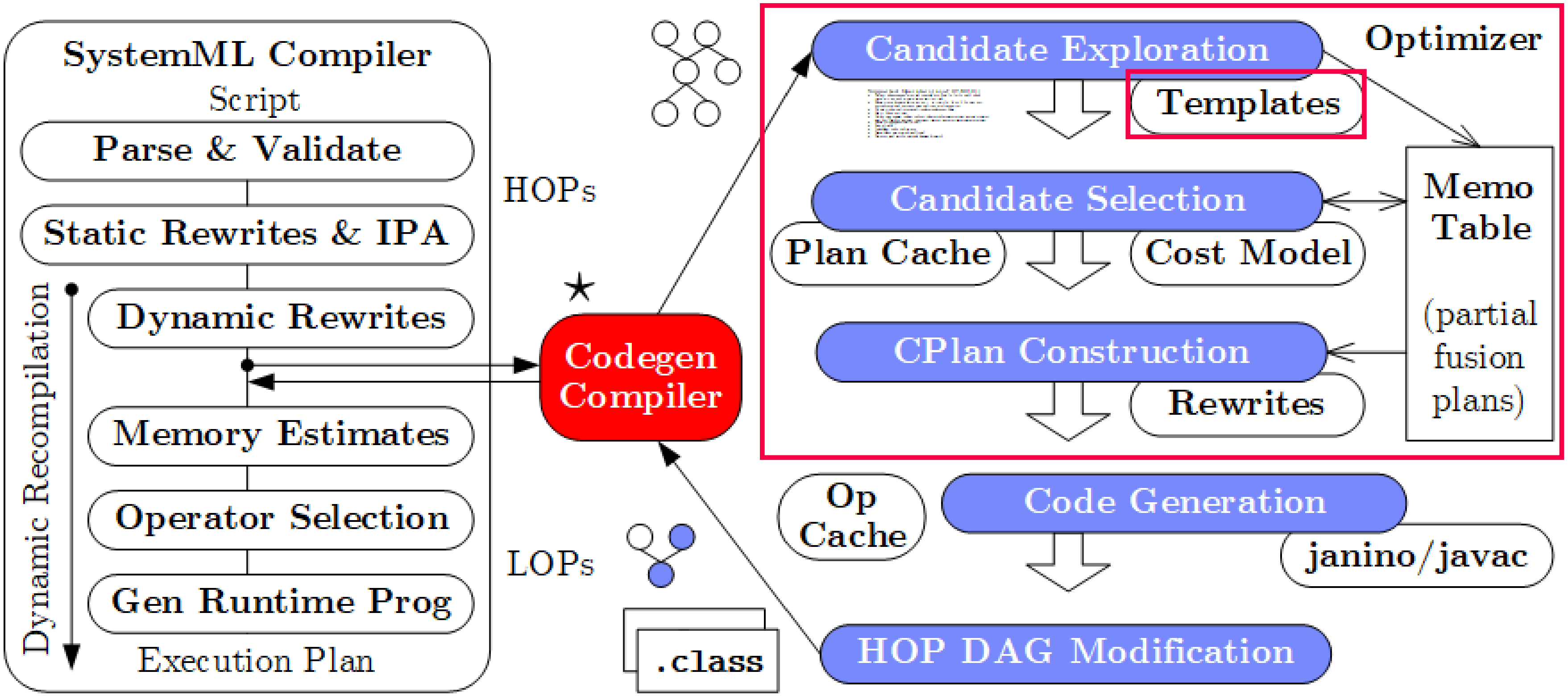
sparse-safe over X

- **#3 Decisions on Fusion Patterns** (e.g., template types)
- **#4 Constraints** (e.g., memory budget and block sizes)

→ **Search Space that requires optimization**

System Architecture (Compiler & Codegen Architecture)

Practical, exact, cost-based optimizer

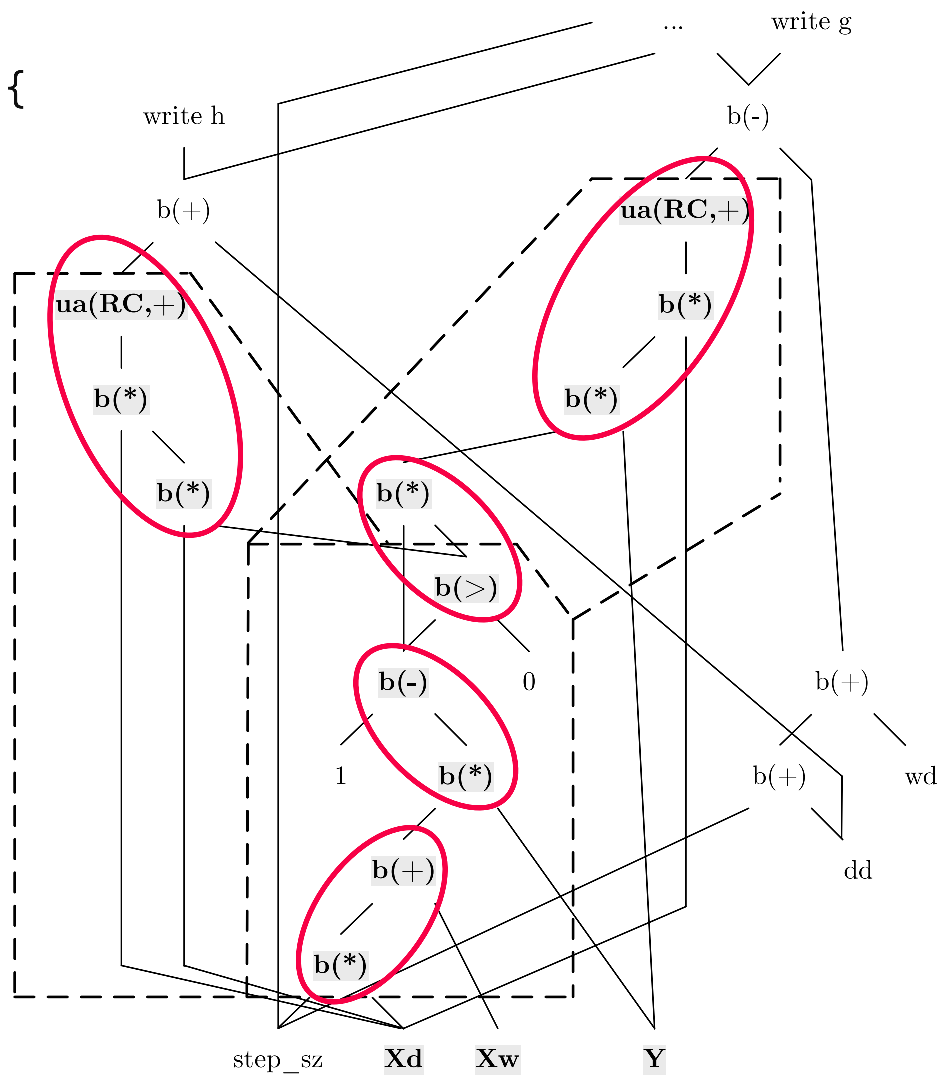


- CPlan representation/construction and codegen similar in TF XLA (HLO primitives, pre-clustering of nodes, caching, LLVM codegen)
- **Templates:** Cell, Row, MAgg, Outer w/ different data bindings

Codegen Example L2SVM (Cell/MAgg)

- **L2SVM Inner Loop**

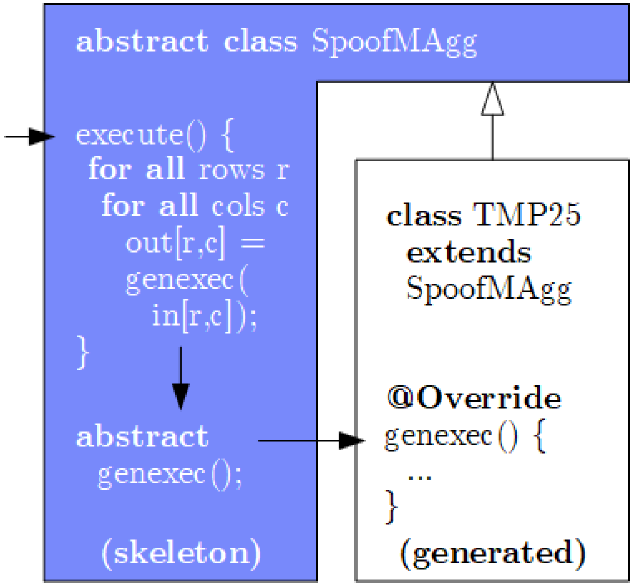
```
while(continueOuter & iter < maxi) {
  #...
  while(continueInner) {
    out = 1-Y* (Xw+step_sz*Xd);
    sv = (out > 0);
    out = out * sv;
    g = wd + step_sz*dd
      - sum(out * Y * Xd);
    h = dd + sum(Xd * sv * Xd);
    step_sz = step_sz - g/h;
}} ...
```



- **# of Vector Intermediates**
  - Base (w/o fused ops): **10**
  - Fused (w/ fused ops): **4**

Codegen Example L2SVM, cont. (Cell/MAgg)

- **Template Skeleton**
  - Data access, blocking
  - Multi-threading
  - Final aggregation



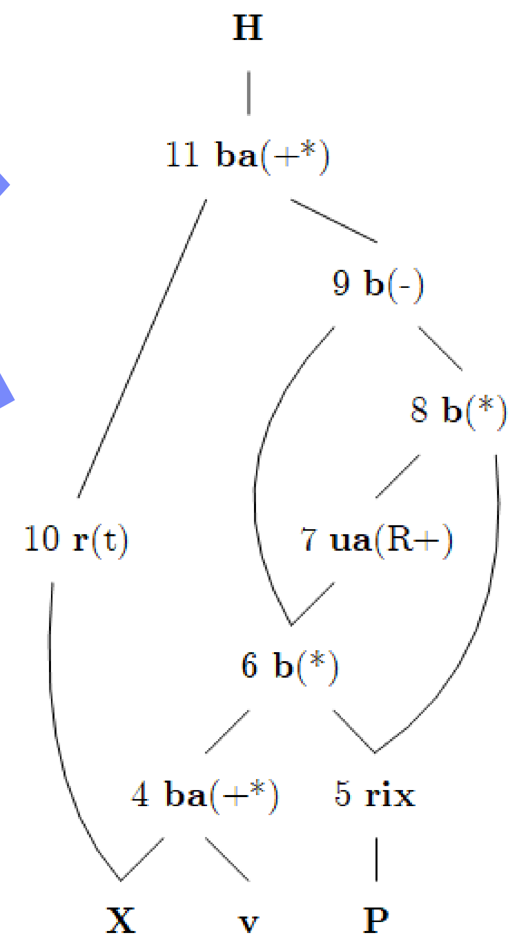
- **# of Vector Intermediates**
  - Gen (codegen ops): 0

```
public final class TMP25 extends SpoofMAgg {
  public TMP25() {
    super(false, AggOp.SUM, AggOp.SUM);
  }
  protected void genexec(double a, SideInput[] b,
   double[] scalars, double[] c, ...) {
    double TMP11 = getValue(b[0], rowIndex);
    double TMP12 = getValue(b[1], rowIndex);
    double TMP13 = a * scalars[0];
    double TMP14 = TMP12 + TMP13;
    double TMP15 = TMP11 * TMP14;
    double TMP16 = 1 - TMP15;
    double TMP17 = (TMP16 > 0) ? 1 : 0;
    double TMP18 = a * TMP17;
    double TMP19 = TMP18 * a;
    double TMP20 = TMP16 * TMP17;
    double TMP21 = TMP20 * TMP11;
    double TMP22 = TMP21 * a;
    c[0] += TMP19;
    c[1] += TMP22;
  }
}
```

Codegen Example MLogreg (Row)

- **MLogreg Inner Loop**
  (main expression on feature matrix X)

```
1: Q = P[, 1:k] * (X %*% v)
2: H = t(X) %*% (Q - P[, 1:k] * rowSums(Q))
```




```
public final class TMP25 extends SpoofRow {
  public TMP25() {
    super(RowType.COL_AGG_B1_T, true, 5);
  }
  protected void genexecDense(double[] a, int ai,
   SideInput[] b, double[] c,..., int len) {
    double[] TMP11 = getVector(b[1].vals(rix),...);
    double[] TMP12 = vectMatMult(a, b[0].vals(rix),...);
    double[] TMP13 = vectMult(TMP11, TMP12, 0, 0,...);
    double TMP14 = vectSum(TMP13, 0, TMP13.length);
    double[] TMP15 = vectMult(TMP11, TMP14, 0,...);
    double[] TMP16 = vectMinus(TMP13, TMP15, 0, 0,...);
    vectOuterMultAdd(a, TMP16, c, ai, 0, 0,...); }
  protected void genexecSparse(double[] avals, int[] aix,
   int ai, SideInput[] b, ..., int len) {...}
}
```

Ahead-of-Time Compilation

- **TensorFlow `tf.compile`**
  - Compile entire TF graph into binary function w/ low footprint
  - **Input:** Graph, config (feeds+fetches w/ fixes shape sizes)
  - **Output:** x86 binary and C++ header (e.g., inference)
  - **Specialization for frozen model and sizes**

[Chris Leary, Todd Wang: XLA – TensorFlow, Compiled!, **TF Dev Summit 2017**]

- **PyTorch Compile**
  - Compile Python functions into ScriptModule/ScriptFunction
  - Lazily collect operations, optimize, and JIT compile
  - Explicit `jit.script` call or `@torch.jit.script`

```python
a = torch.rand(5)
def func(x):
  for i in range(10):
    x = x * x # unrolled into graph
  return x

jitfunc = torch.jit.script(func) # JIT
jitfunc.save("func.pt")
```

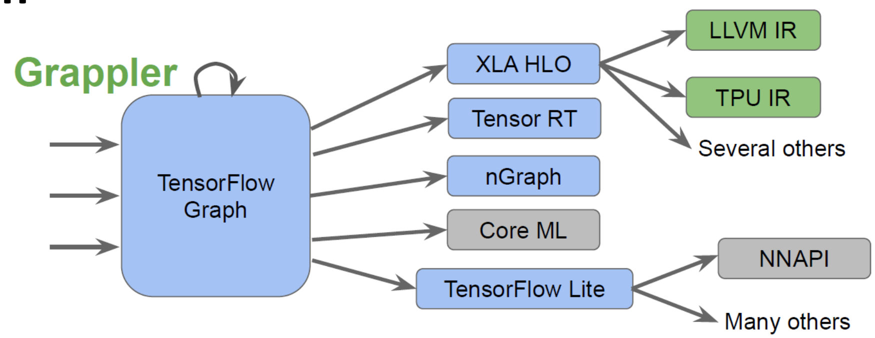
[Vincent Quenneville-Bélair: How PyTorch Optimizes Deep Learning Computations, **Guest Lecture Stanford 2020**]

Excursus: MLIR

TensorFlow

[Rasmus Munk Larsen, Tatiana Shpeisman: TensorFlow Graph Optimizations, **Guest Lecture Stanford 2019**]

- **Motivation TF Compiler Ecosystem**
  - Different IRs and compilation chains for runtime backends
  - **Duplication of infrastructure** and fragile error handling
  - Adoption: PYTORCH [https://github.com/llvm/torch-mlir]



- **MLIR (Multi-level, Machine Learning IR)**
  - SSA-based IR, similar to LLVM
  - Hierarchy of modules, functions, regions, blocks, and operations
  - **Dialects for different backends** (defined ops, customization)
  - **Systematic lowering**

```
func @testFunction(%arg0: i32) {
  %x = call @thingToCall(%arg0)
     : (i32) -> i32
  br ^bb1
^bb1:
  %y = addi %x, %x : i32
  return %y : i32
}
```

[Chris Lattner et al.: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. **CGO 2021,** https://arxiv.org/pdf/2002.11054.pdf]

Excursus: MLIR, cont.

(DAPHNE pre-project prototype)

```
while(i < max_iter) { # PageRank
  p = alpha*(G%*%p) + (1-alpha)*(e%*%u%*%p);
  i += 1;
}
```

After Several Optimization Passes

```
module  {
  func @main() {
    %0 = daphne.constant 5.000000e-01 : f64
    %1 = daphne.constant 0 : i64
    %2 = daphne.constant 1.000000e+00 : f64
    %3 = daphne.constant 1 : i64
    %4 = daphne.constant 10 : i64
    %5 = daphne.rand  {cols = 50 : i64, rows = 50 : i64, seed = -1 : i64, sparsity = 7.000000e-02 : f64} : () -> ...
    %6, %7, %8 = ...
    %9 = daphne.sub %2, %0 : (f64, f64) -> f64
    %10:2 = daphne.while (%arg0 = %6, %arg1 = %1) : (!daphne.matrix<50x1xf64>, i64) -> (same) condition: {
      %11 = cmpi "ult", %arg1, %4 : i64
      daphne.yield %11 : i1
    } body: {
      %11 = daphne.mat_mul %5, %arg0 : (!daphne.matrix<50x50xf64>, !daphne.matrix<50x1xf64>) -> !daphne.matrix<50x1xf64>
      %12 = daphne.mul %11, %0 : (!daphne.matrix<50x1xf64>, f64) -> !daphne.matrix<50x1xf64>
      %13 = daphne.mat_mul %8, %arg0 : (!daphne.matrix<1x50xf64>, !daphne.matrix<50x1xf64>) -> !daphne.matrix<1x1xf64>
      %14 = daphne.mat_mul %7, %13 : (!daphne.matrix<50x1xf64>, !daphne.matrix<1x1xf64>) -> !daphne.matrix<50x1xf64>
      %15 = daphne.mul %9, %14 : (f64, !daphne.matrix<50x1xf64>) -> !daphne.matrix<50x1xf64>
      %16 = daphne.add %12, %15 : (!daphne.matrix<50x1xf64>, !daphne.matrix<50x1xf64>) -> !daphne.matrix<50x1xf64>
      %17 = daphne.add %arg1, %3 : (i64, i64) -> i64
      daphne.yield %16, %17 : !daphne.matrix<50x1xf64>, i64
    }
    daphne.print %10#0 : !daphne.matrix<50x1xf64>
    daphne.return
  }
}
```

3) Code motion outside loop

1) Shape inference of dimensions

2) Matrix multiplication chain reordered

Summary and Q&A

- **Compilation Overview**
- **Size Inference and Cost Estimation**
- **Rewrites (and Operator Selection)**
- **Runtime Adaptation**
- **Operator Fusion & JIT Compilation**

Recommended Reading

[Chris Leary, Todd Wang: XLA – TensorFlow, Compiled!, **TF Dev Summit 2017,** https://www.youtube.com/watch?v=kAOanJczHA0]

➔ **Impact of Size Inference and Costs**
- Advanced optimization of LA programs requires size inference for cost estimation and validity constraints

➔ **Ubiquitous Rewrite Fusion, and Codegen/JIT Opportunities**
- Linear algebra programs have plenty of room for optimization
- Potential for changed asymptotic behavior