Architecture of ML Systems 04 Operator Fusion and Runtime Adaptation

Matthias Boehm

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

Last update: Apr 05, 2019

Announcements/Org

- **#1 Programming/Analysis Projects**
  - **Apr 05:** Project selection
  - **3/9 projects** assigned so far
  - Discussion individual projects (first come, first served)

- **#1b Selected Projects**
  - **#1 Auto Differentiation**
  - ~~**#6 Reproduce Automated Label Generation**~~
  - **#12 Information Extraction from Unstructured PDF/HTML**
  - **#5 LLVM Code Generator**

Agenda

- **Runtime Adaptation**
- **Automatic Operator Fusion**

Runtime Adaptation

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**

- **Example ML Program Scenarios**
  - Conditional control flow
  - User-Defined Functions
  - Data-dependent operators
    ```
    Y = table( seq(1,nrow(X)), y )
    grad = t(X) %*% (P - Y);
    ```
  - Computed size expressions
  - Changing dimensions or sparsity

Ex: Stepwise LinregDS

```
while( continue ) {
   parfor( i in 1:n ) {
      if( fixed[1,i]==0 ) {
         X = cbind(Xg, Xorig[,i])
         AIC[1,i] = linregDS(X,y)
      }
   }
   #select & append best to Xg
}
```

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

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

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], **Mahout Samsara** [MLSystems'16]
  - Examples w/ control structures: **Weld** [CIDR'17], **OptiML** [ICML'11], **Emma** [SIGMOD'15]
- **#3 Program Compilation** (entire program)
  - Examples: **SystemML** [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");
```

Recompilation

Script

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

HOPs

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

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

Execution Plan

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

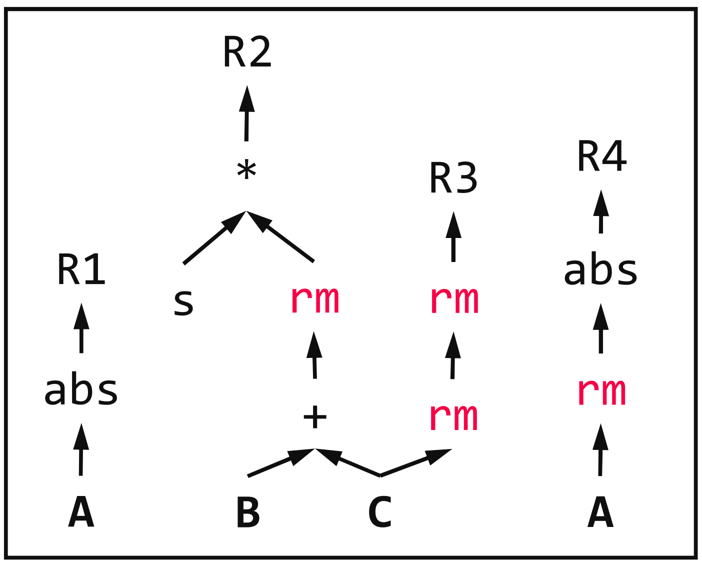
Dynamic Recompilation

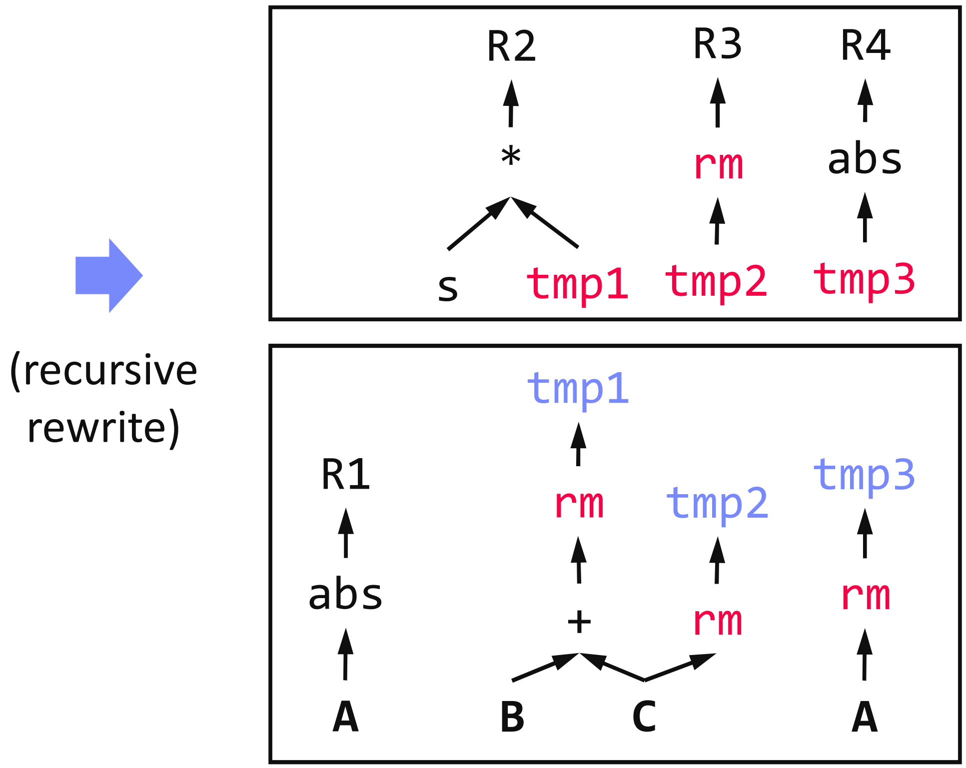
Other systems w/ recompile: SciDB, MatFast

Dynamic Recompilation

- **Optimizer Recompilation 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**

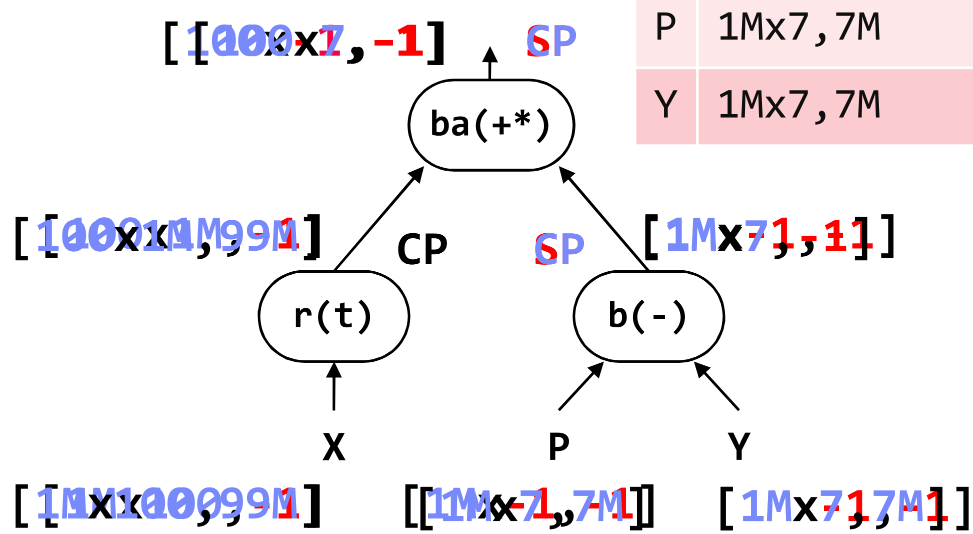




Dynamic Recompilation, cont.

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

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/ each entry 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]==0 ) {
         X = cbind(Xg,Xorig[,i])
         AIC[1,i] = linregDS(X,y)
      }
   }
   #select & append best to Xg
}
```

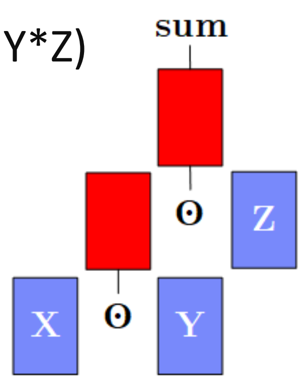
Automatic Operator Fusion

Motivation: Fusion Opportunities

- **State-of-the art ML systems**
  - DAGs of linear algebra (LA) operations and statistical functions
  - Materialized intermediates → **ubiquitous fusion opportunities**

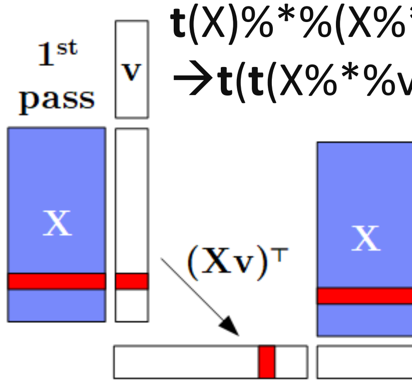
a) Intermediates

sum(X*Y*Z)

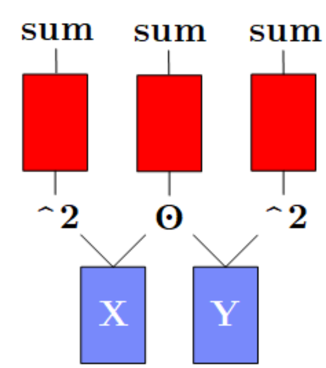


b) Single-Pass

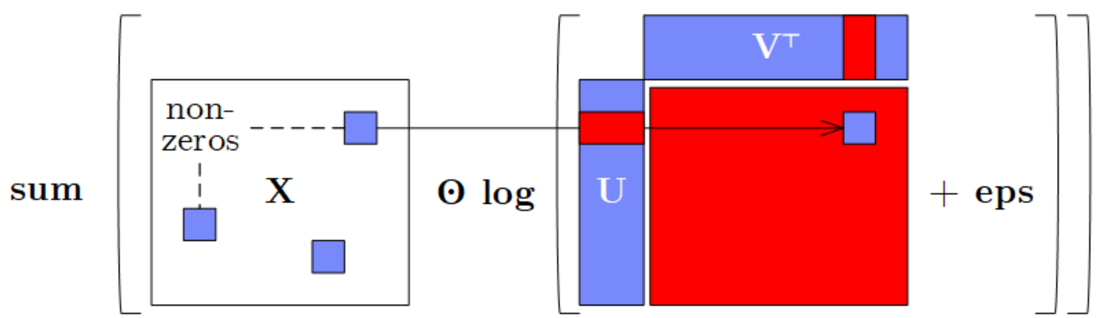
t(X)%*%(X%*%v)
→**t**(**t**(X%*%v)%*%X)



c) Multi-Aggregates



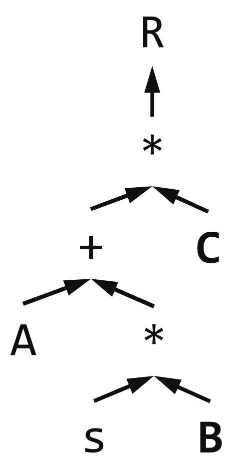
d) Sparsity Exploitation



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,i] + s*B[i,1]) * C[i,1];
```

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
Tensor Comprehensions	2018	Loop Fusion	No	No	Evolutionary, cost-based
TVM	2018	Loop Fusion	No	No	ML/cost-based

- **Challenge HW accelerators →** TVM / **tensorflow/mlir** (Apr 4, 2019)

Specific Fusion Techniques

- **#1 Micro Optimizations**
  - Hybrid tile-at-a-time loop fusion, predication, and result allocation
  - Examples: Tupleware
- **#2 Cross-Library Optimization**
  - Generic IR based on parallel loops and builders
  - Examples: Weld
- **#3 Sparsity Exploitation**
  - Exploit sparsity over chains of operations (compute, size of intermediates)
  - Examples: SystemML
- **#4 Iteration Schedules**
  - Decisions on loop ordering (e.g., tensor storage formats, join ordering)
  - Examples: Taco, TVM, Mateev et al
- **#5 Optimizing Fusion Plans**
  - Example: SystemML

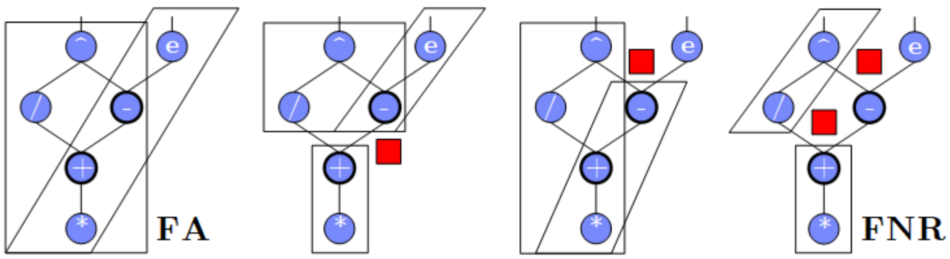
[Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, Niketan Pansare: On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. **PVLDB 2018**]

A Case for Optimizing Fusion Plans

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

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

- **#1 Materialization Points** (e.g., for multiple consumers)



- **#2 Sparsity Exploitation** (and ordering of sparse inputs)

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

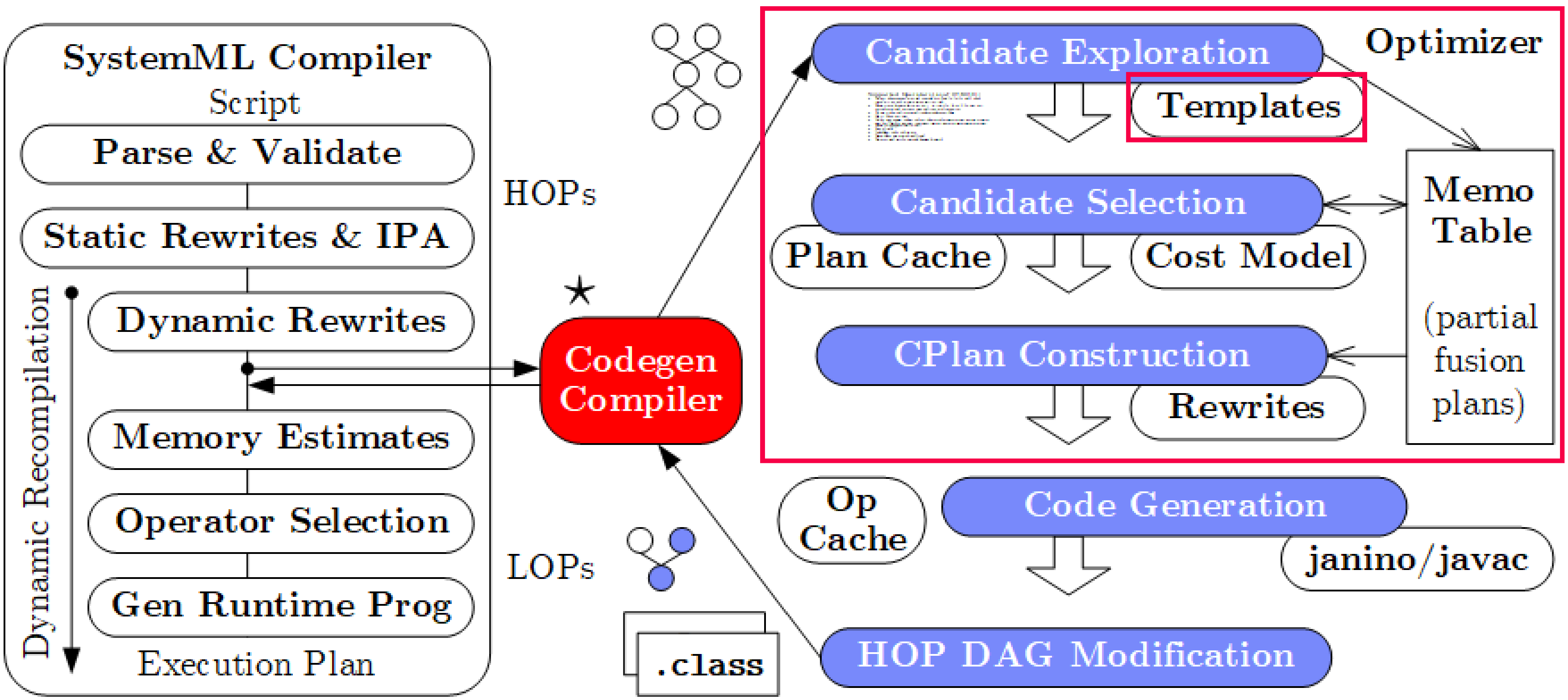
sparse-safe over X

- **#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

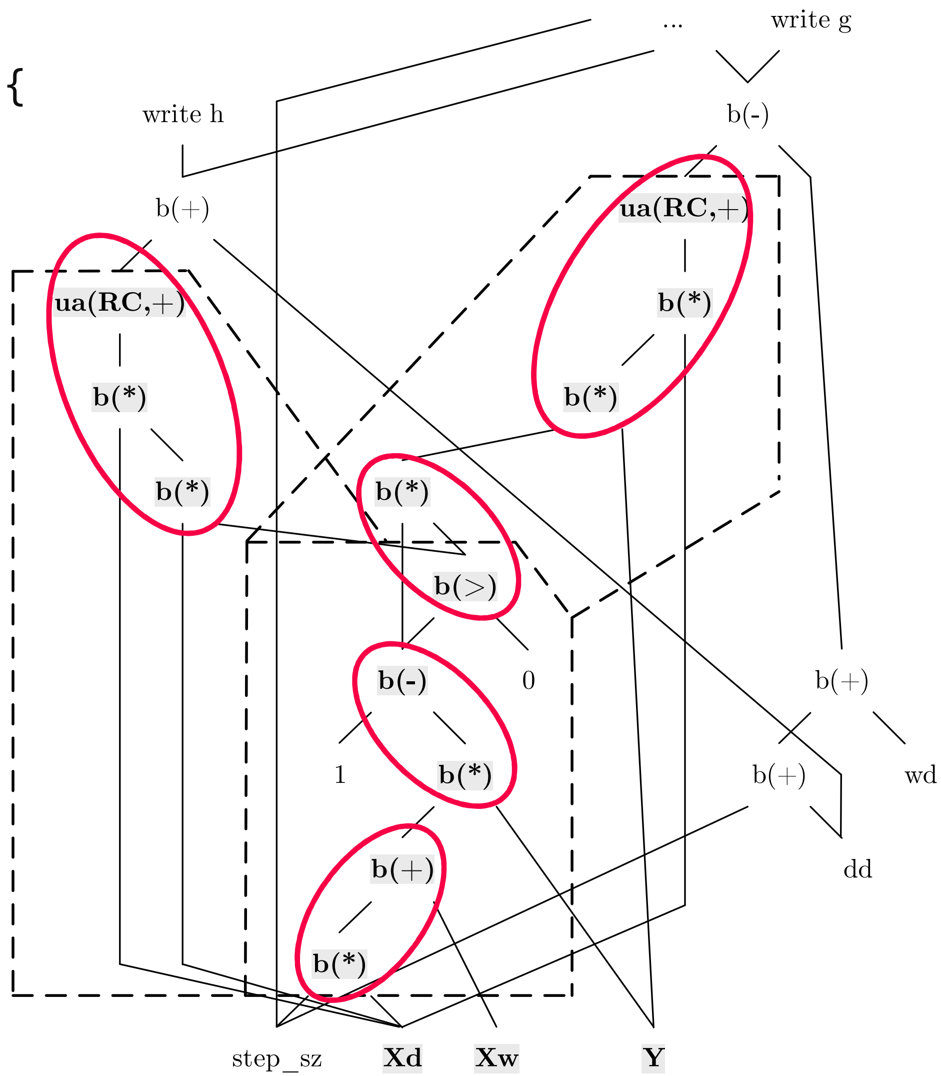


- **Templates:** Cell, Row, MAgg, Outer w/ different data bindings

Codegen Example L2SVM (Cell/MAgg)

- **L2SVM Inner Loop**

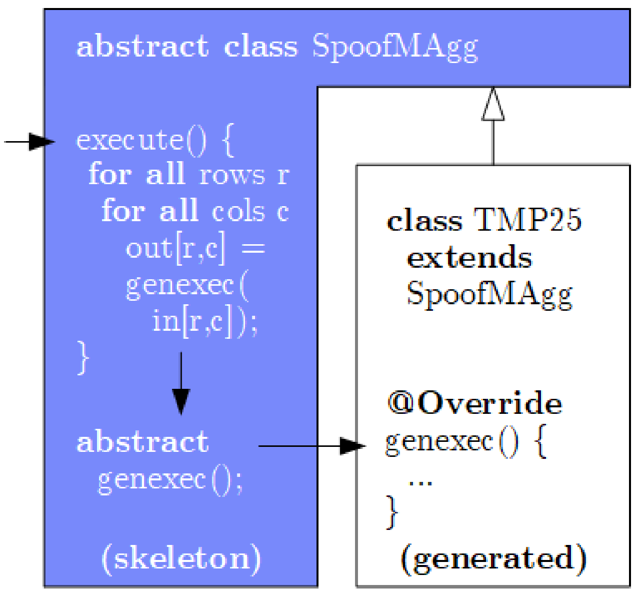
```
while(continueOuter & iter < maxi) {
  #...
  while(continueInner) {
    out = 1-Y* (Xw+step_sz*Xd);
    sv = (out > 0);
    out = out * sv;
    g = wd + step_sz*dd
      - sum(out * Y * Xd);
    h = dd + sum(Xd * sv * Xd);
    step_sz = step_sz - g/h;
}} ...
```



- **# 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

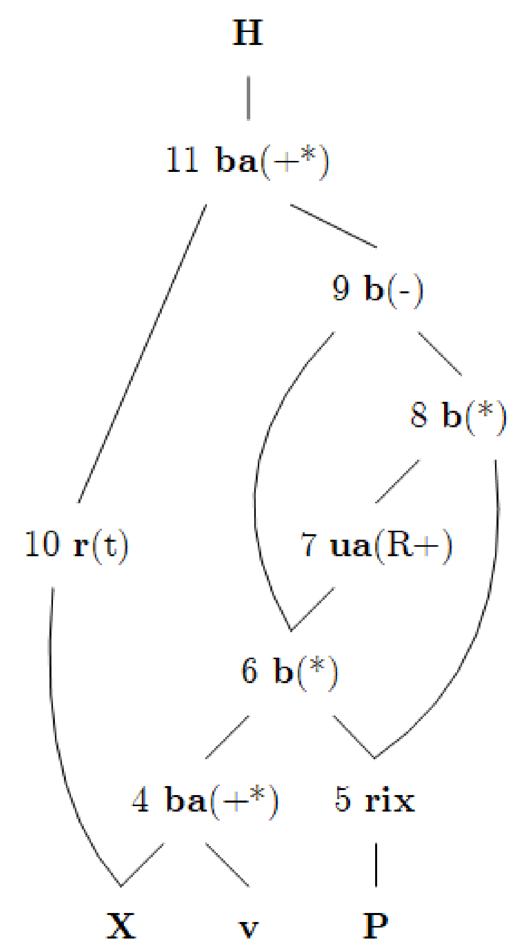
```
public final class TMP25 extends SpoofMAgg {
  public TMP25() {
    super(false, AggOp.SUM, AggOp.SUM);
  }
  protected void genexec(double a, SideInput[] b,
   double[] scalars, double[] c, ...) {
    double TMP11 = getValue(b[0], rowIndex);
    double TMP12 = getValue(b[1], rowIndex);
    double 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)

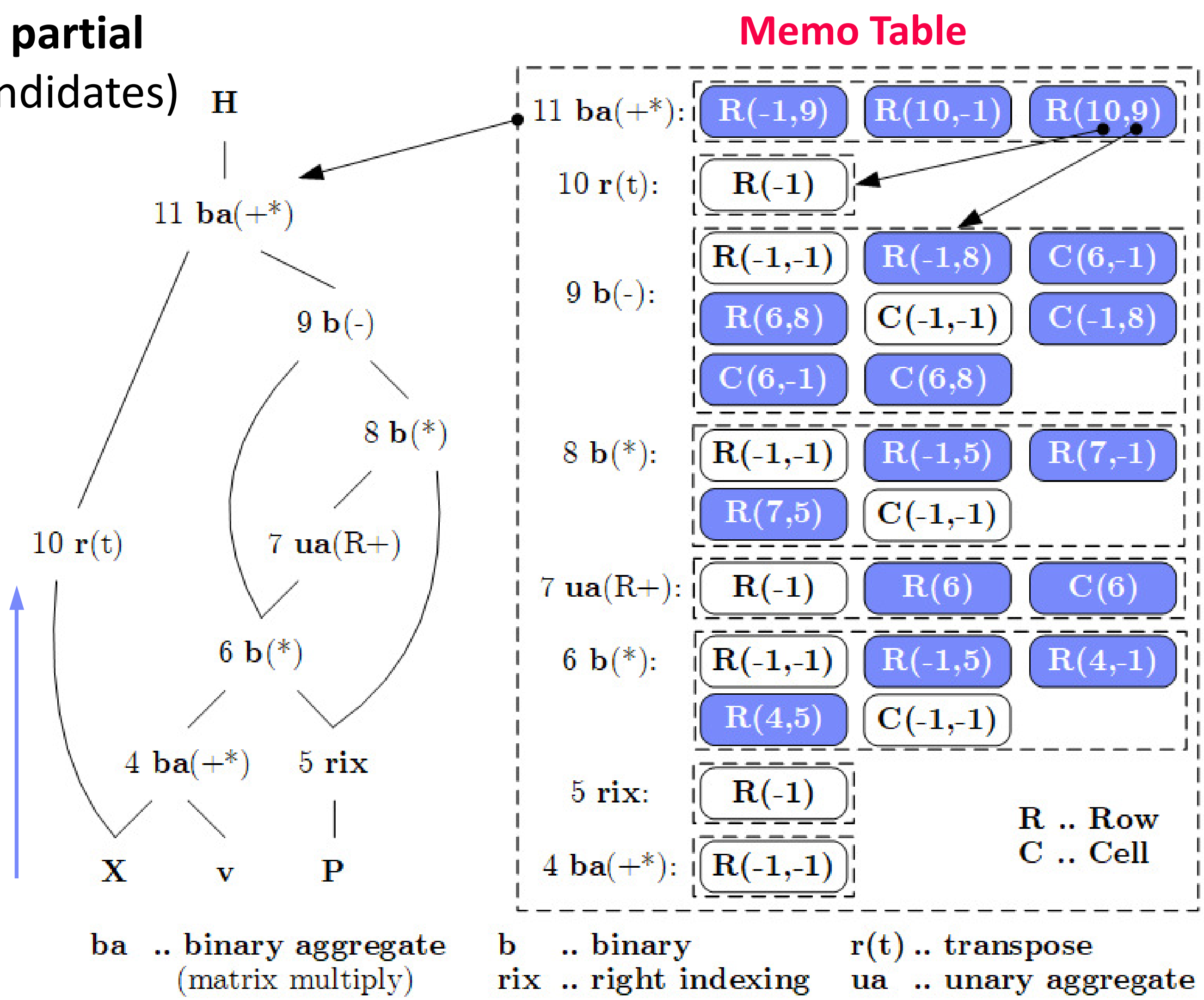
```
1: Q = P[, 1:k] * (X %*% v)
2: H = t(X) %*% (Q - P[, 1:k] * rowSums(Q))
```

```
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) {...}
}
```



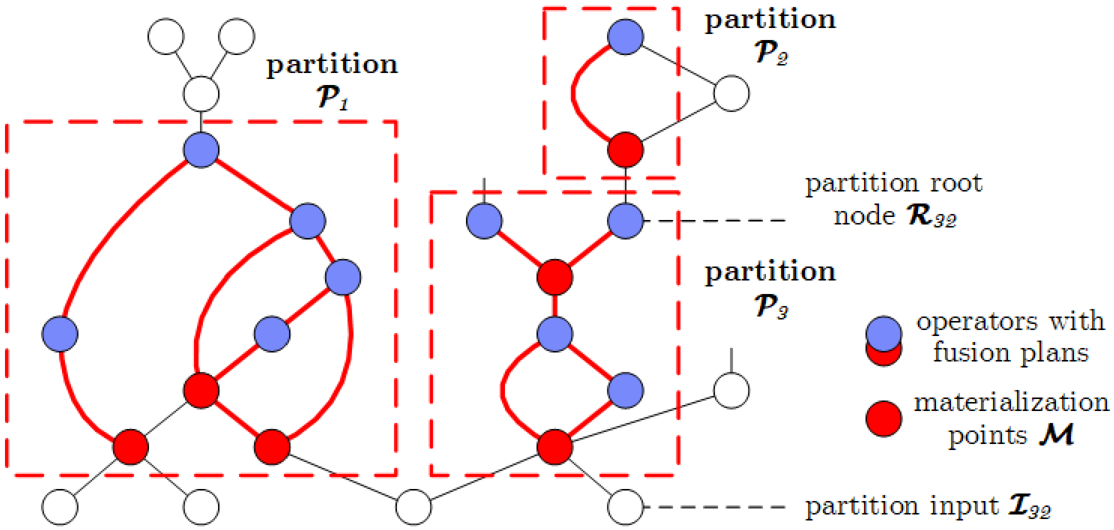
Candidate Exploration (by example MLogreg)

- **Memo Table for partial fusion plans** (candidates)
- OFMC **Template Fusion API**
  - Open
  - Fuse, Merge
  - Close
- OFMC **Algorithm**
  - **Bottom-up Exploration** (single-pass, template-agnostic)
  - Linear space and time



Candidate Selection (Partitions and Interesting Points)

- **#1 Determine Plan Partitions**
  - Materialization Points M
  - Connected components of fusion references
  - Root and input nodes
  - ➔ **Optimize partitions independently**



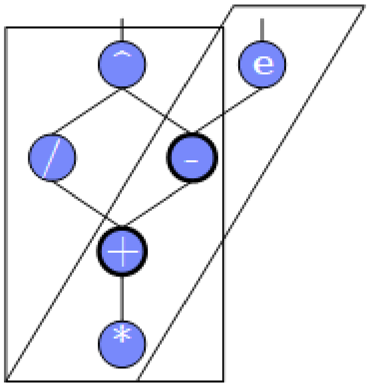
- **#2 Determine Interesting Points**
  - **Materialization Point Consumers:** Each data dependency on materialization points considered separately
  - **Template / Sparse Switches:** Data dependencies where producer has templates that are non-existing for consumers
  - ➔ **Optimizer considers all $2^{|M'i|}$ plans** (with $|M'_i| \geq |M_i|$) per partition

Candidate Selection, cont. (Costs and Constraints)

- **Overview Cost Model**
  - Cost partition with **analytical cost model** based on peak memory and compute bandwidth
  - Plan comparisons / fusion errors don't propagate / dynamic recompilation

$$C(\mathcal{P}_i|\mathbf{q}) = \sum_{p \in \mathcal{P}_i|\mathbf{q}} \left(\hat{T}_p^w + \max\left(\hat{T}_p^r, \hat{T}_p^c\right)\right)$$

- **#3 Evaluate Costs**
  - #1: Memoization of already processed sub-DAGs
  - #2: Account for shared reads and CSEs within operators
  - #3: Account for redundant computation (overlap)
  - ➔ **DAG traversal** and **cost vectors** per fused operator (with memoization of pairs of operators and cost vectors)

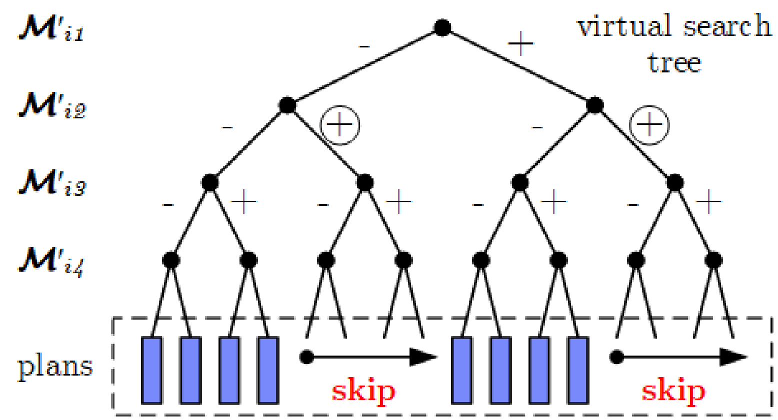


- **#4 Handle Constraints**
  - **Prefiltering** violated constraints (e.g., row template in distributed ops)
  - Assign **infinite costs for violated constraints** during costing

Candidate Selection, cont. (MPSkipEnum and Pruning)

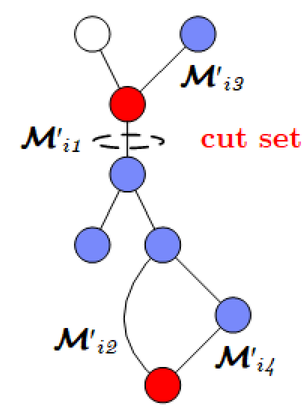
- **#5 Basic Enumeration**
  - Linearized search space: from - to *

```
for( j in 1:pow(2,|M'i|) )
   q = createAssignment(j)
   C = getPlanCost(Pi, q)
   maintainBest(q, C)
```



- **#6 Cost-Based Pruning**
  - **Upper bound:** cost C^U of best plan q* (monotonically decreasing)
  - **Opening heuristic:** evaluate FA and FNR heuristics first
  - **Lower bound:** C^{LS} (read input, write output, min compute) + dynamic C^{LD} (materialize intermediates q) ➔ **skip subspace** if $C^U \leq C^{LS} + C^{LD}$

- **#7 Structural Pruning**
  - **Observation:** Assignments can create independent sub problems
  - Build **reachability graph** to determine **cut sets**
  - During enum: probe cut sets, recursive enum, combine, and skip



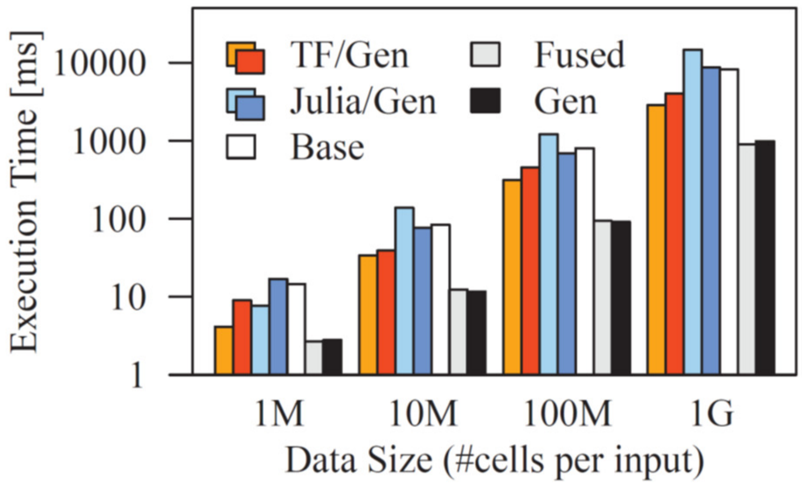
Experimental Setting

- **Setup**
  - 1+6 node cluster (head 2x4 Intel Xeon E5530, 64GB RAM; 6workers 2x6 Intel Xeon E5-2440, 96GB RAM, peak 2x32GB/s 2x115GFLOP/s, 10Gb Ethn)
  - Modern scale-up server (2x20 Intel Xeon Gold 6138, 768GB RAM, peak 2x119 GB/s 2x1.25TFLOP/s)
  - Java 1.8.0, Hadoop 2.7.3, Spark 2.2.0 (client w/ 35GB driver, 6 executors w/ 65 GB and 24 cores, aggregate cluster memory: 234 GB)
- **Baselines**
  - SystemML 1.0++ (Feb 2018): **Base**, **Fused** (hand-coded, default), **Gen** (optimizer), and heuristics **FA** (all) and **FNR** (no redundancy)
  - Julia 0.6.2 (Dec 13 2017): LLVM code generation, **Julia** (without fusion) and **JuliaGen** (fusion via dot syntax)
  - TensorFlow 1.5 (Jan 26 2018): **TF** (without fusion), and **TFGen** (fusion via TensorFlow XLA), limited support for sparse

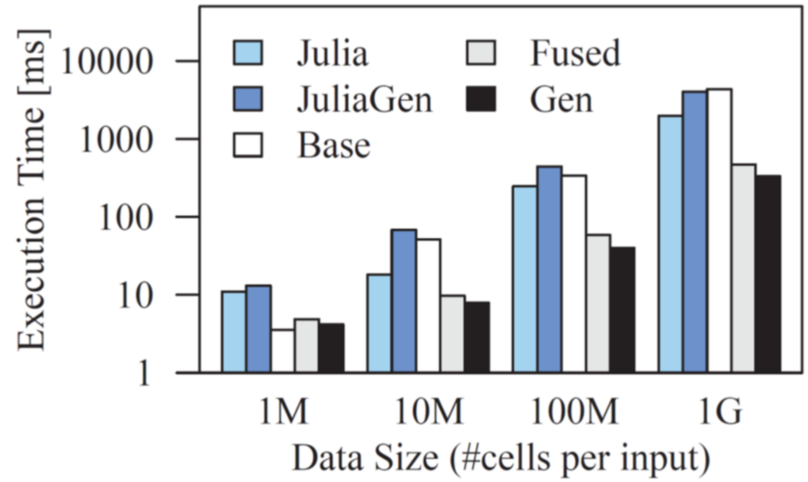
Operations Performance

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

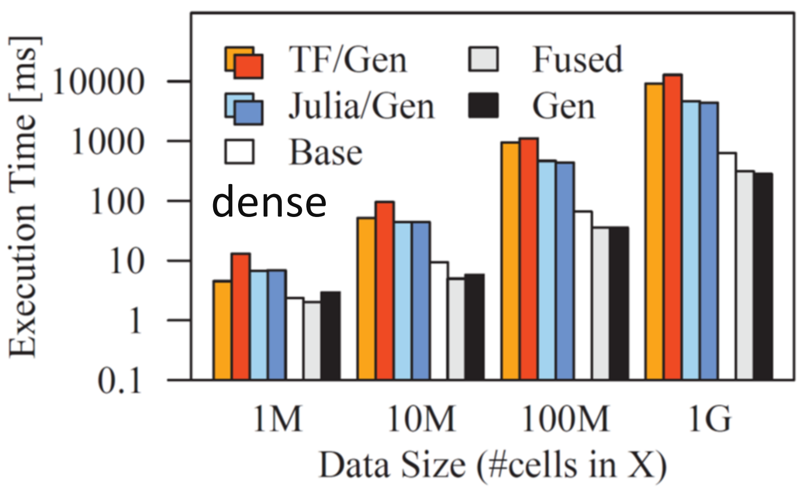
dense



sparse (0.1)



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

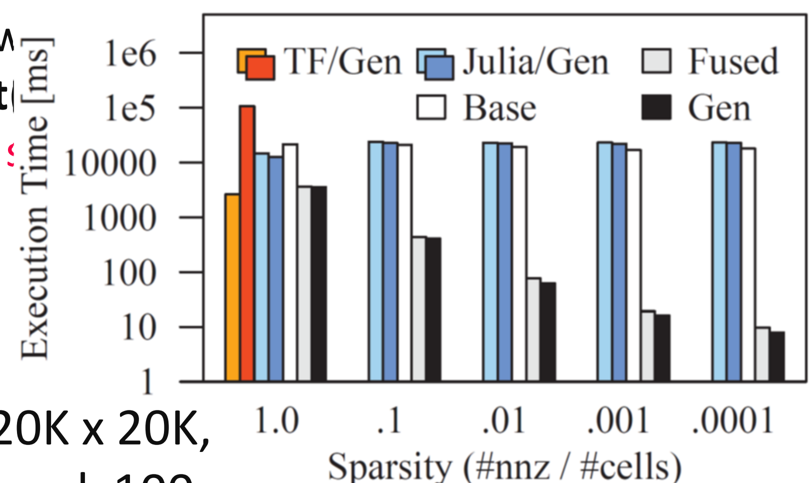


Outer: `sum(X*log(U%*%t(V)+1e-15))`

TF w

→ t(

9.2 s



20K x 20K,
rank 100

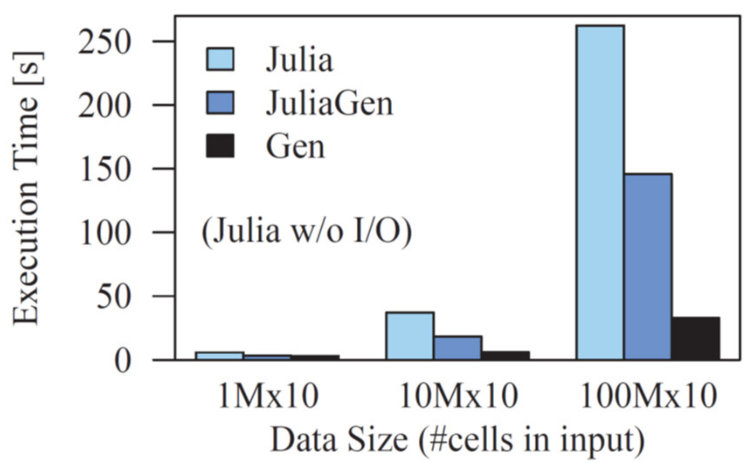
L2SVM End-to-End Performance (20 outer/∞ inner)

- **Local and Distributed** [seconds]

Data	Base	Fused*	Gen	FA	FNR
10^8 x 10, D	446	276	**37**	44	92
Airline78, D	151	105	**24**	26	45
Mnist8m, S	203	156	**113**	115	116
$2*10^8$ x 100, D	1218	895	**347**	1433	539
$2*10^8$ x 10^3, S	1481	1066	**373**	2205	575
Mnist80m, S	1593	1114	**552**	1312	896

#1 Heuristics struggle w/ hybrid plans

- **Julia Comparison**
  - Dataset: 10^8 x 10 (8GB)
  - Hand-tuned fusion script for JuliaGen



ALS-CG End-to-End Performance (20 outer/20 inner, rk 20)

- **ALG-CG**
  - Representative for many matrix factorization algorithms
  - Requires sparsity exploitation in loss computation and update rules
- **Local single node** [seconds]

Data	Base	Fused*	Gen	FA	FNR
$10^4 \times 10^4$, S (0.01)	426	**20**	25	215	226
$10^5 \times 10^5$, S (0.01)	23,585	96	**80**	13,511	12,353
$10^6 \times 10^6$, S (0.01)	N/A	860	**722**	N/A	N/A
Netflix	N/A	1,026	**789**	N/A	N/A
Amazon Books	N/A	17,335	**7,420**	N/A	N/A

(8,026,324 x 2,330,066; sparsity=0.0000012)

#2 Heuristics struggle w/ sparsity exploitation

#3 Heuristics struggle w/ complex DAGs

Backup: Programming/Analysis Projects

Example Projects

- **#1 Auto Differentiation**
  - Implement auto differentiation for deep neural networks
  - Integrate auto differentiation framework in compiler or runtime
- **#2 Sparsity-Aware Optimization of Matrix Product Chains**
  - Extend DP algorithm for DAGs and other operations
- **#3 Parameter Server Update Schemes**
  - New PS update schemes: e.g., stale-synchronous, Hogwild!
  - Language and local/distributed runtime extensions
- **#4 Extended I/O Framework for Other Formats**
  - Implement local readers/writers for NetCDF, HDF5, libsvm, and/or Arrow
- **#5 LLVM Code Generator**
  - Extend codegen framework by LLVM code generator
  - Native vector library, native operator skeletons, JNI bridge
- **#6 Reproduce Automated Label Generation** (analysis)

Example Projects, cont.

- **#7 Data Validation Scripts**
  - Implement recently proposed integrity constraints
  - Write DML scripts to check a set of constraints on given dataset
- **#8 Data Cleaning Primitives**
  - Implement scripts or physical operators to perform data imputation and data cleaning (find and remove/fix incorrect values)
- **#9 Data Preparation Primitives**
  - Extend `transform` functionality for distributed binning
  - Needs to work in combination w/ dummy coding, recoding, etc
- **#10 Common Subexpression Elimination & Constant Folding**
  - Exploit commutative common subexpressions
  - One-shot constant folding (avoid compile overhead)
- **#11 Repartition joins and binary ops without replication**
  - Improve repartition mm and binary ops by avoiding unnecessary replication