TECHNISCHE UNIVERSITÄT BERLIN

Architecture of ML Systems (AMLS)
03 Compilation – Size Inference and Rewrites

Prof. Dr. Matthias Boehm
Technische Universität Berlin
Berlin Institute for the Foundations of Learning and Data
Big Data Engineering (DAMS Lab)

 Last update: May 03, 2023

BIFOLD

Announcements / Org

- **#1 Hybrid & Video Recording**
  - Hybrid lectures (in-person, zoom) with optional attendance
    https://tu-berlin.zoom.us/j/9529634787?pwd=R1ZsN1M3SC9BOU1OcFdmem9zT202UT09
  - Zoom **video recordings**, links from website
    https://mboehm7.github.io/teaching/ss23_amls/index.htm

- **#2 Reminder Project / Exercise Selection**
  - **Task description** and updated projects on course website
  - **Project Selection by May 10**, Submission by **July 04**

~254?

- **#3 Office Hours**
  - **Every Tuesday 3pm-4.30pm** (starting May 09), in **TEL-0811** seminar room + zoom (link on website)
  - Questions and answers on projects and exercises (e.g., task, approach, implementation)
  - TA Sebastian Baunsgaard

Agenda

- **Compilation Overview**
- **Size Inference and Cost Estimation**
- **Rewrites (and Operator Selection)**

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

DB
PL **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]
  - 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");
```

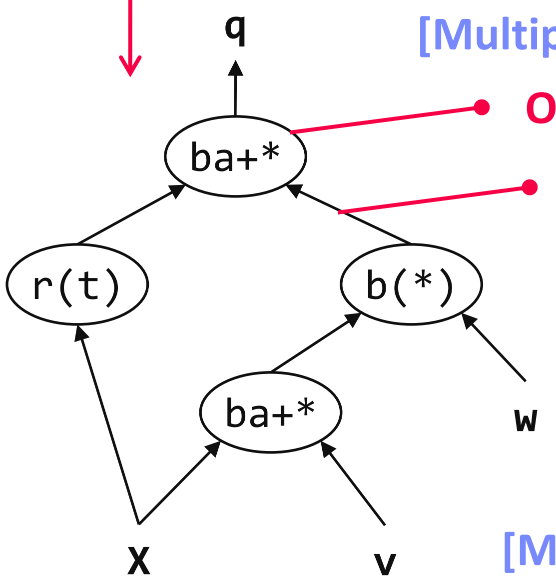
ML Program Compilation / Graphs

- **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

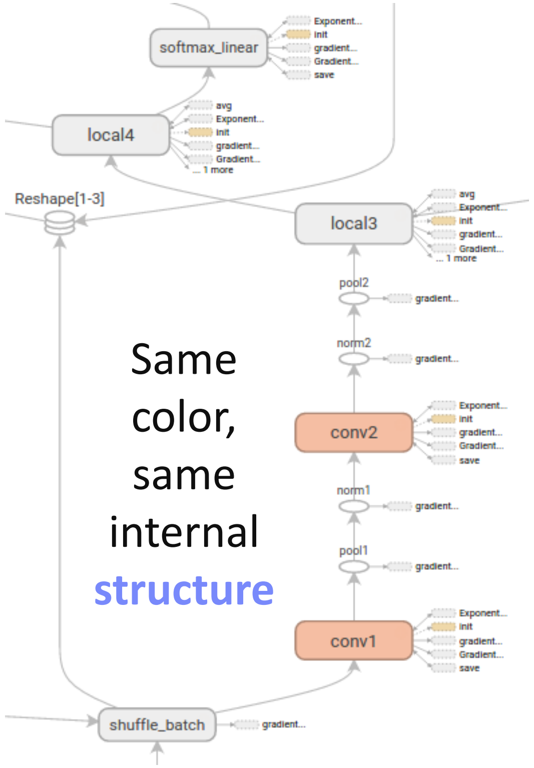
```
SPARK mapmmchain X.MATRIX.DOUBLE w.MATRIX.DOUBLE
  v.MATRIX.DOUBLE _mVar4.MATRIX.DOUBLE XtwXv
```

ML Program Compilation / Graphs, cont.

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

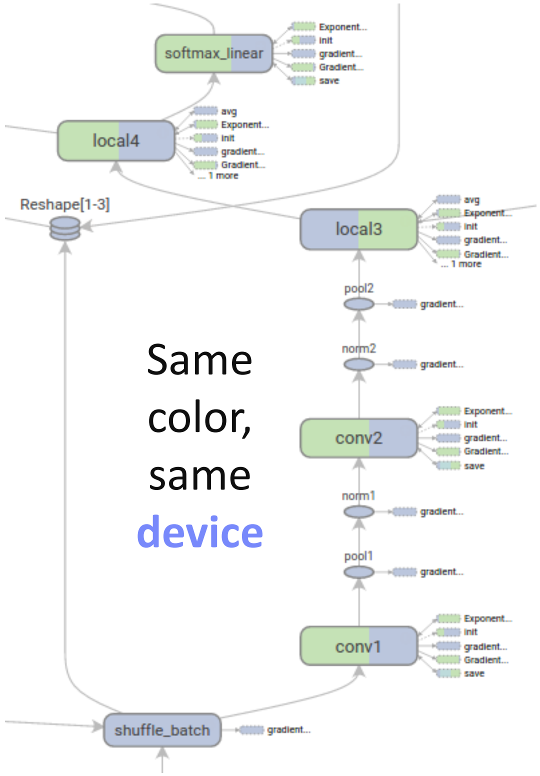
- **Example TF TensorBoard**

(Node) **Structure View**



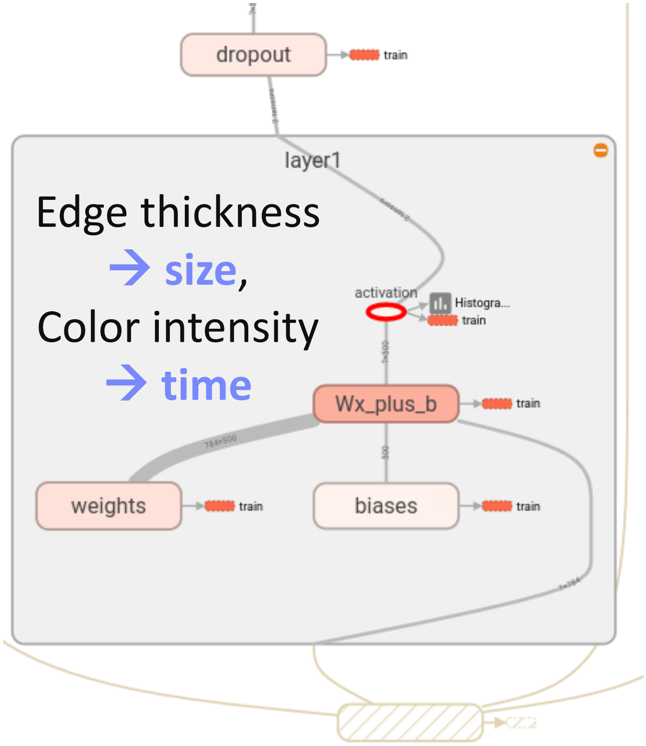
Same color, same internal **structure**

Device View (CPU, GPU)



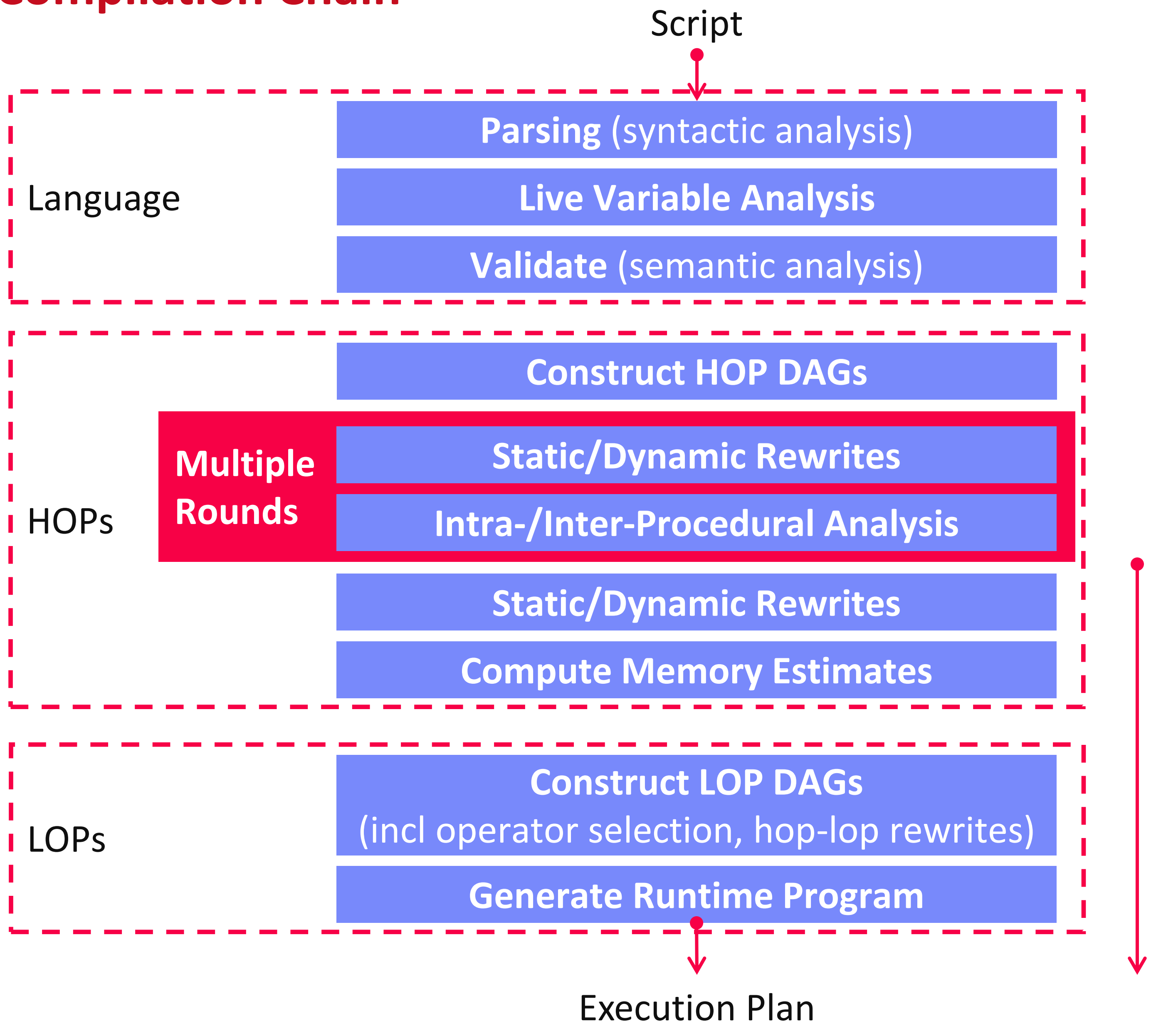
Same color, same **device**

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



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

Example SystemDS: Compilation Chain



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

Dynamic Recompilation (lecture 04)

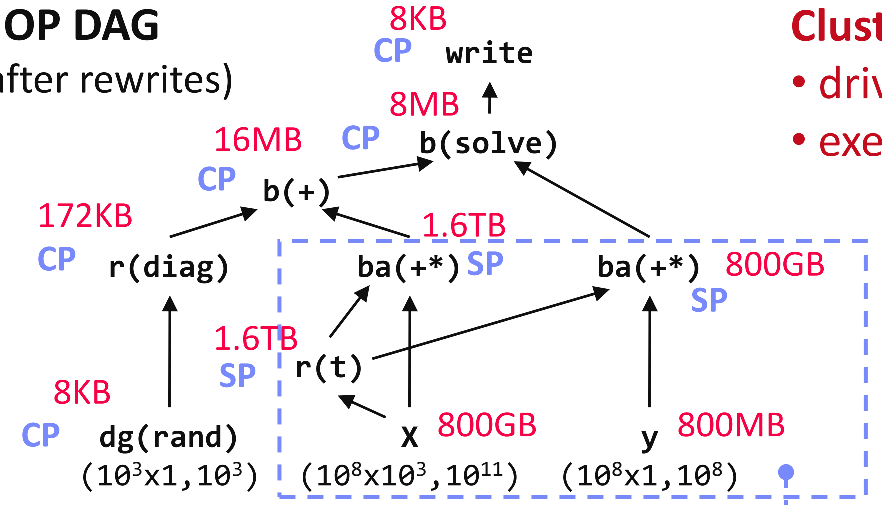
Example SystemDS: Basic HOP and LOP DAG Compilation

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);
```

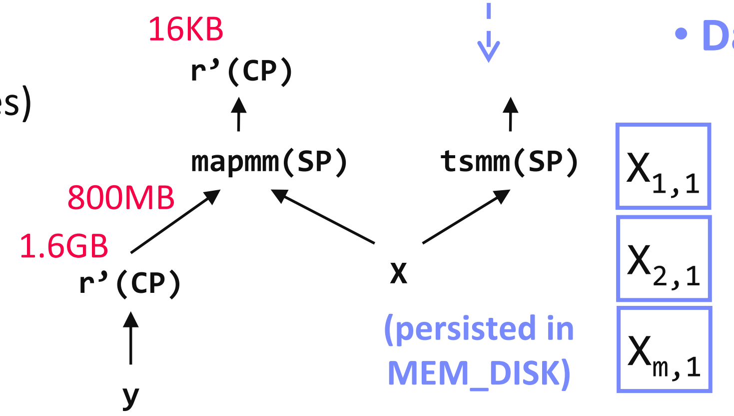
Scenario:
X: $10^8 \times 10^3$, 10^{11}
y: $10^8 \times 1$, 10^8

HOP DAG
(after rewrites)



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

LOP DAG
(after rewrites)



➔ Distributed Matrices
- Fixed-size matrix blocks
- Data-parallel operations

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

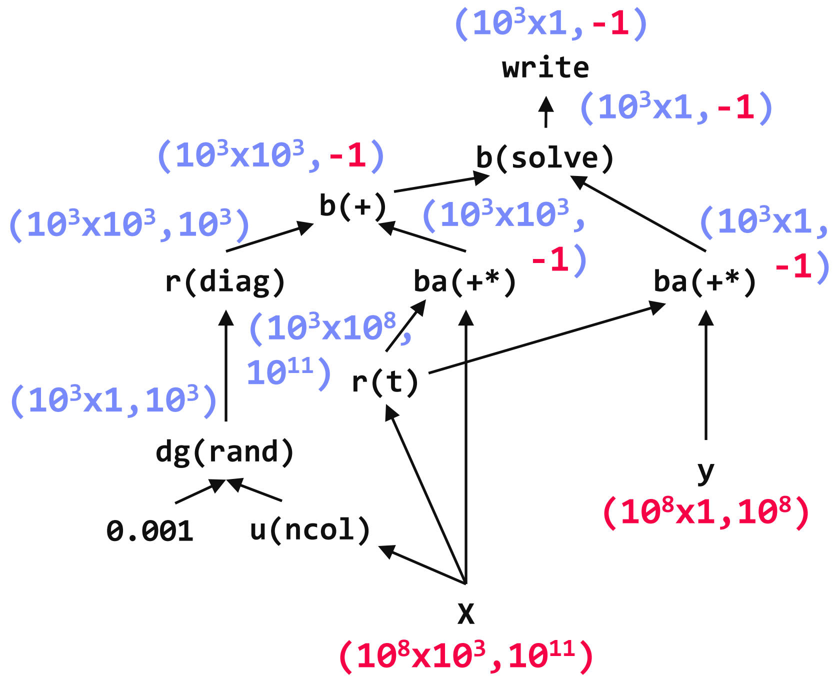
Size Inference and Cost Estimation

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
- **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)

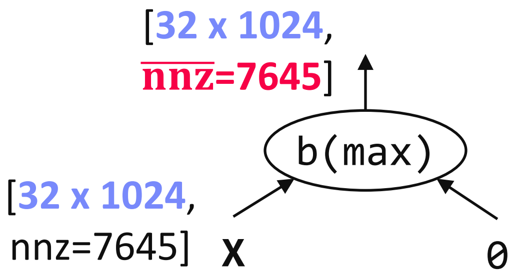
```
X = read($1);
y = read($2);
I = matrix(0.001, ncol(X), 1);
A = t(X) %*% X + diag(I);
b = t(X) %*% y;
beta = solve(A, b);
```



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.”);
}
```

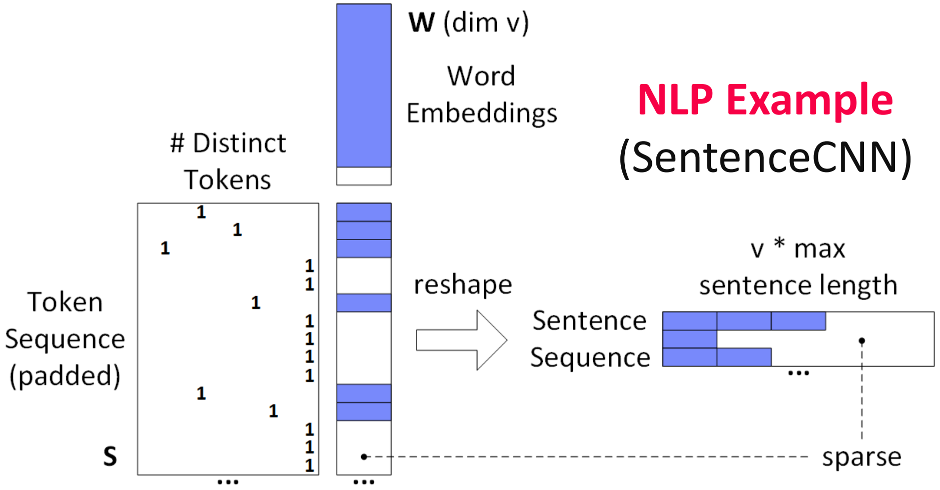
- **Additional IPA Passes (selection)**
  - **Inline functions** (single statement block, small)
  - **Dead code elimination** and simplification rewrites
  - Remove unused functions & flag recompile-once

```
//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

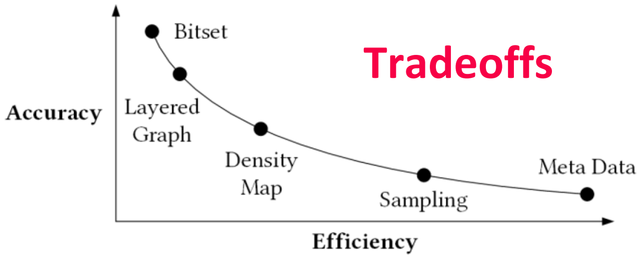
NLP Example (SentenceCNN)



- **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

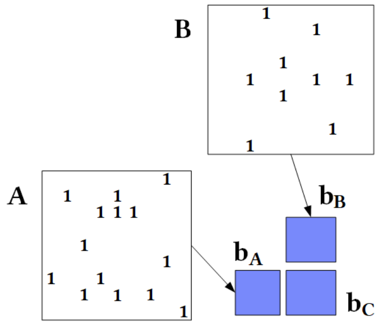


- **Overview**

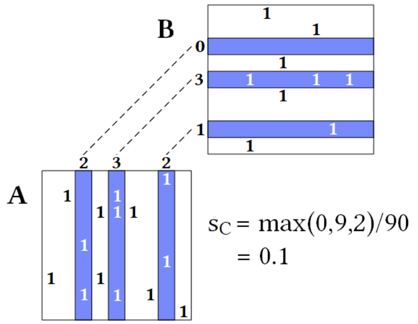
- **#1 Naïve Metadata Estimators**
  - Derive the output sparsity solely from the sparsity of inputs (e.g., SystemDS)

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

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

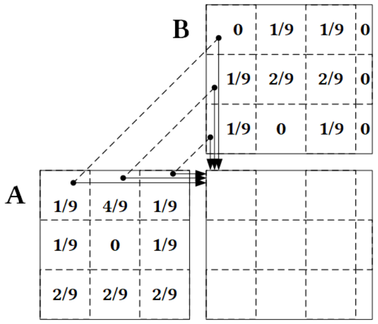


- **#3 Sampling**
  - Take a sample of aligned columns of A and rows of B
  - Sparsity estimated via max of count-products
  - Examples: MatFast [ICDE'17], improvements in paper

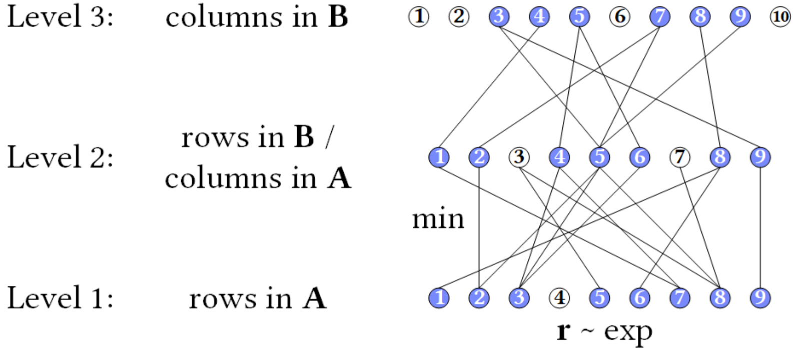


Sparsity Estimation – Estimators, cont.

- **#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]



- **#5 Layered Graph** [J.Comb.Opt.'98]
  - **Nodes:** rows/columns in mm chain
  - **Edges:** non-zeros connecting rows/columns
  - Assign r-vectors ~ exp (w/ λ=1) and propagate via elementwise min
  - **Intuition:** KMV (the more paths the larger the values)
  - Estimate over roots (output columns)

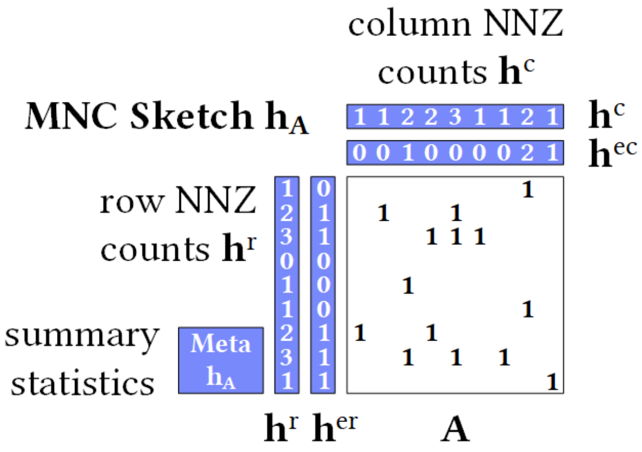


$$\hat{s}_C = \left(\sum_{v \in \text{roots}} \frac{|\mathbf{r}_v| - 1}{\text{sum}(\mathbf{r}_v)} \right) / (ml),$$

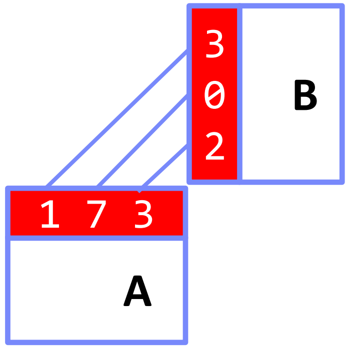
[Edith Cohen: Structure Prediction and Computation of Sparse Matrix Products. **Journal of Combinatorial Optimization 1998**]

Sparsity Estimation – Estimators, cont.

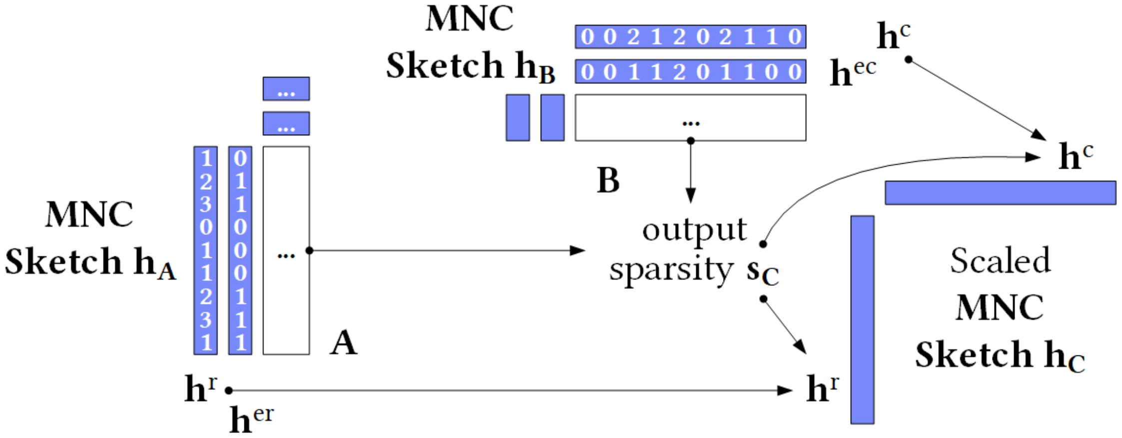
- **#6 MNC Sketch** (Matrix Non-zero Count)
  - Create MNC sketch for inputs A and B
  - **Exploitation of structural properties** (e.g., 1 non-zero per row, row sparsity)
  - **Support for matrix expressions** (reorganizations, elementwise ops)
  - **Construction:** O(d) space, O(nnz) time



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



  - Sketch propagation and estimation



[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

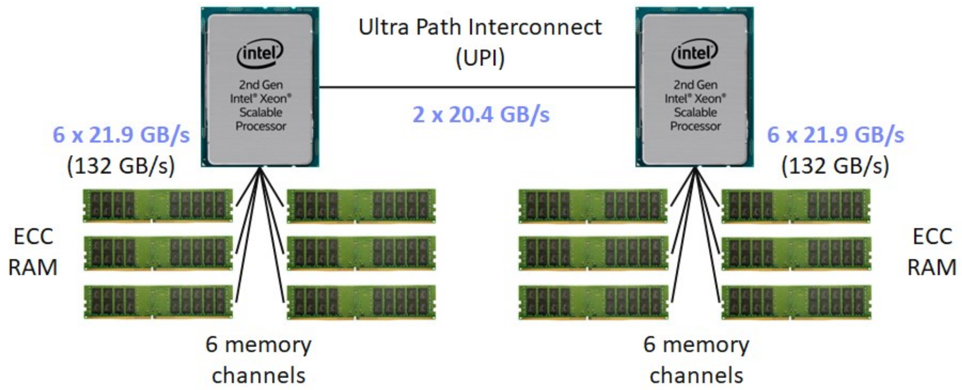
A Personal War Story

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

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

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

Example Analytical Cost Model



- **Hardware Setting**
  - **Max Memory Bandwidth:**
    2 sockets * 6 channels * 21.9 GB/s = **264 GB/s**
  - **Max Compute:**
    2 sockets * 28 cores * 2.2 GHz * 2 FMA units
    * 8 FP64 (AVX512) * 2 (FMA) = **3.85 TFLOP/s**
- **Workload**
  - Matrix-vector Multiplication (X %*% v)
  - X: 1,000,000 x 1,000 in dense FP64 (double)
- **Costs**
  - Determine data/compute workload, convert to time

```
C = max(8GB / 264GB/s, 2GFLOP / 3.85TFLOP/s)
  = max(30.3ms, 0.5ms) = 30.3ms
```

cpufetch output:

```
Name:                Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz
Microarchitecture:   Cascade Lake
Technology:          14nm
Max Frequency:       4.000 GHz
Sockets:             2
Cores:               28 cores (56 threads)
Cores (Total):       56 cores (112 threads)
AVX:                 AVX,AVX2,AVX512
FMA:                 FMA3
L1i Size:            32KB (1.75MB Total)
L1d Size:            32KB (1.75MB Total)
L2 Size:             1MB (56MB Total)
L3 Size:             38.5MB (77MB Total)
Peak Performance:    14.34 TFLOP/s
```

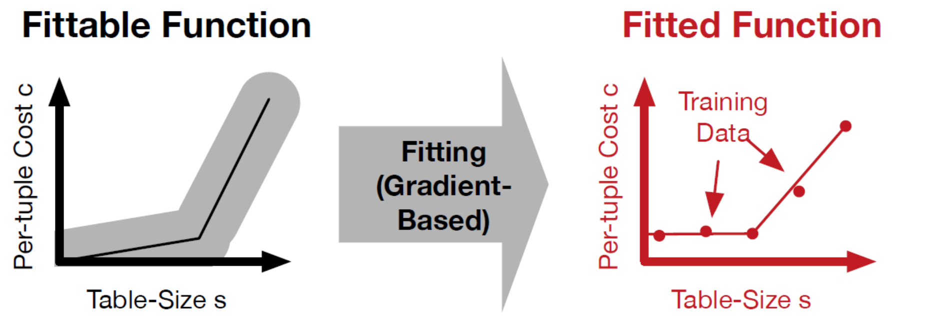
FP32 at max freq

Excursus: Differentiable Programming

- **Overview Differentiable Programming**
  - Adoption of auto differentiation concept from ML systems to PLs
  - Yann LeCun (Jan 2018)

"It's really very much like a regular prog[r]am, except it's parameterized, automatically differentiated, and trainable/optimizable."

- **Example DBMS Fitting**
  - Implement DBMS components as **differentiable functions**
  - E.g.: cost model components
  - Q: **What about guarantees** (memory, size)?



$$c(s) = \begin{cases} a\_in \cdot s + b\_in & \text{if } s < \text{cache-size} \\ a\_out \cdot s + b\_out & \text{if } s \geq \text{cache-size} \end{cases}$$

$$c(s) = \begin{cases} 0.0 \cdot s + 0.5 & \text{if } s < 30\text{MB} \\ 1.2 \cdot s - 35.5 & \text{if } s \geq 30\text{MB} \end{cases}$$

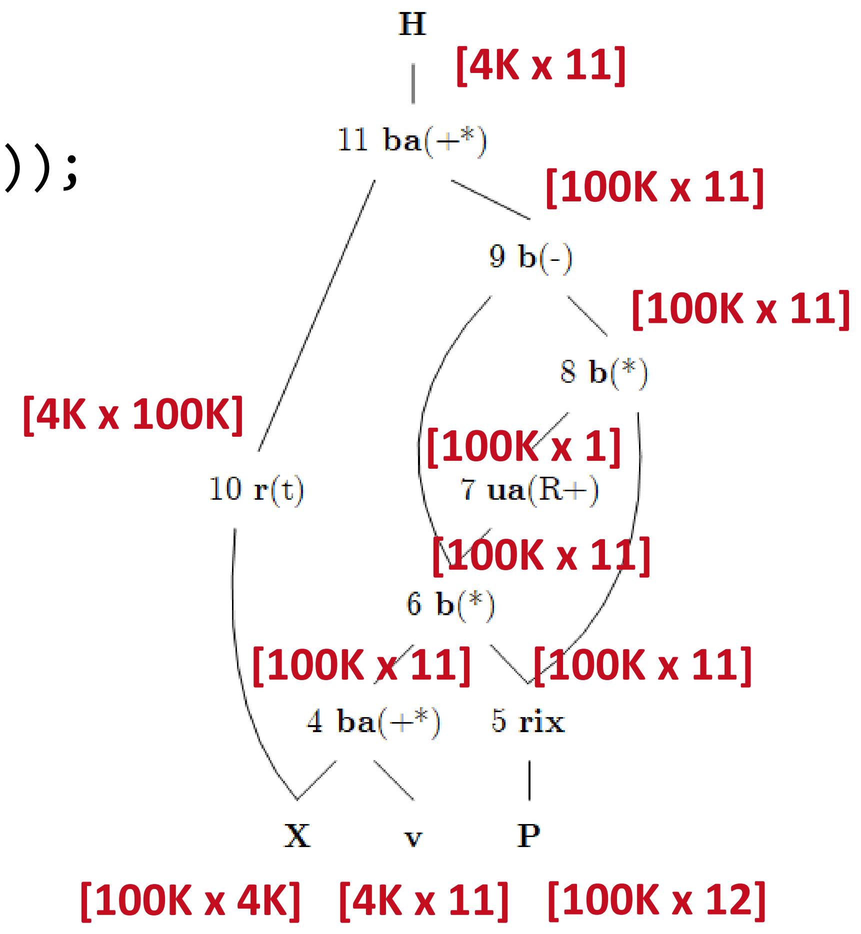
[Benjamin Hilprecht et al: DBMS Fitting: Why should we learn what we already know? **CIDR 2020**]

3min BREAK and TEST YOURSELF

- **Expression MLogReg**

```
Q = P[, 1:K] * (X %*% v);
H = t(X) %*% (Q - P[, 1:K] *
      (rowSums(Q) %*% matrix(1,1,K)));
```

- **Compiled DAG**
  - X: 100,000 x 4,000 / nnz = 1,365,000
  - v: 4,000 x 11
  - P: 100,000 x 12 (K=11, 12 classes)



- **Q & A**
  - **What are the dimensions and sparsity of all intermediates?**
  - **What rewrites have been applied Expression → DAG?**
    → CSE P[,1:K], rm vector replication
  - **What other rewrites are possible if ncol(v)==1?**
    → rm rowsums(), factorize ((1-P[,1:K])*Q) & ((1-P[,1:K])*P[,1:K])

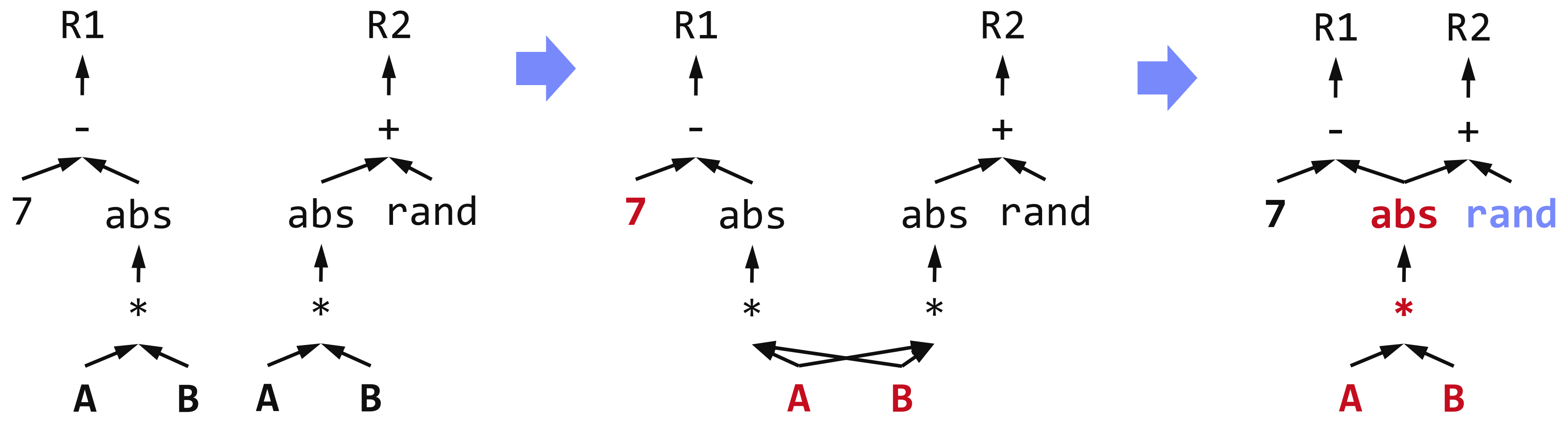
Rewrites (and Operator Selection)

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**)

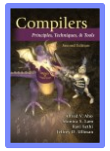
```
R1 = 7 – abs(A * B)
R2 = abs(A * B) + rand()
```

- **Example:**



Traditional PL Rewrites, cont.

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

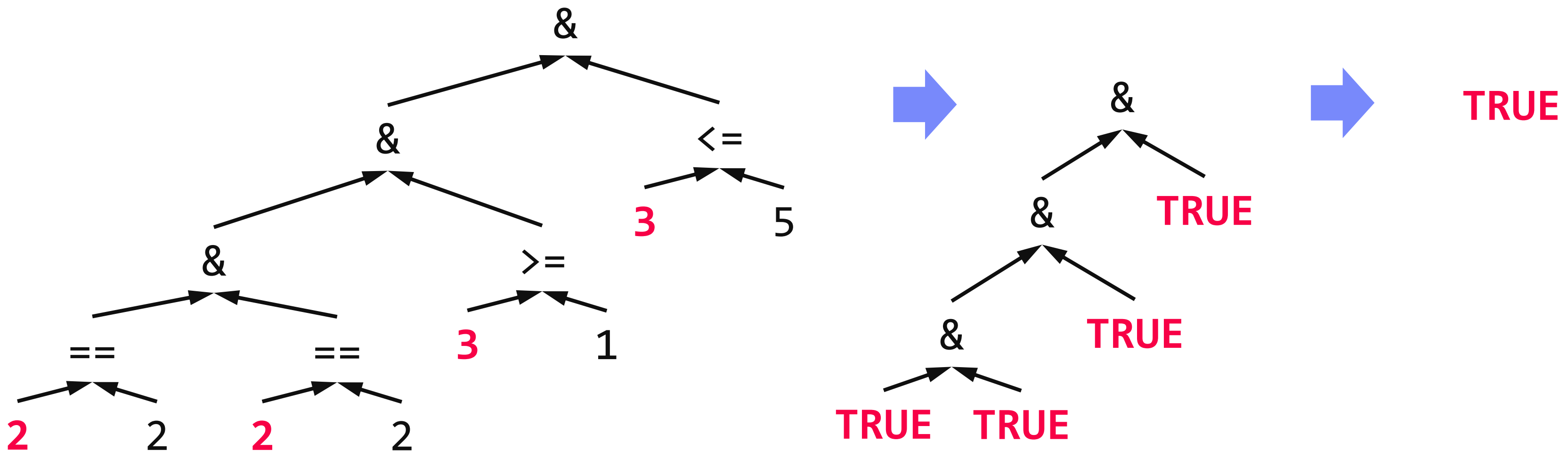


Turing Award '20

- **#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):

```
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

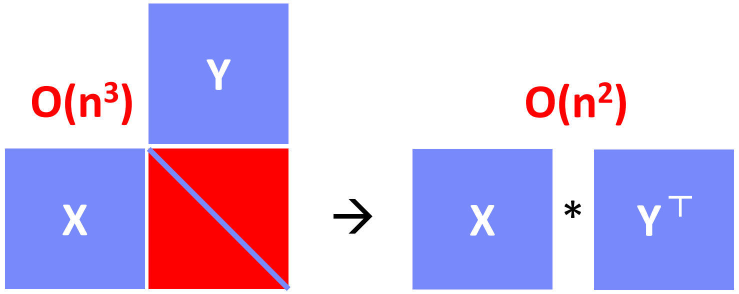
LinregDS (Direct Solve)

```
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

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

- **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);`



- **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 Logical: `!(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)`

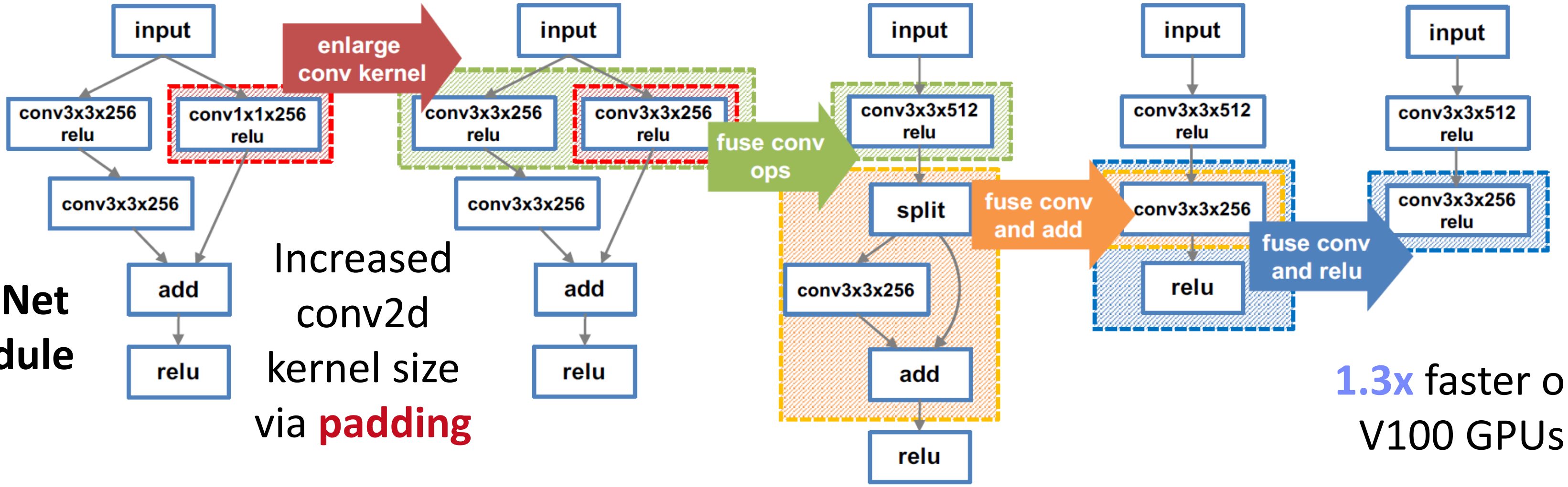
SystemML/SystemDS
`RewriteElementwise-MultChainOptimization`
(orders and collapses matrix, vector, scalar op chains)

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**]

ResNet module



1.3x faster on V100 GPUs

- **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.

- **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:

PYTORCH

[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

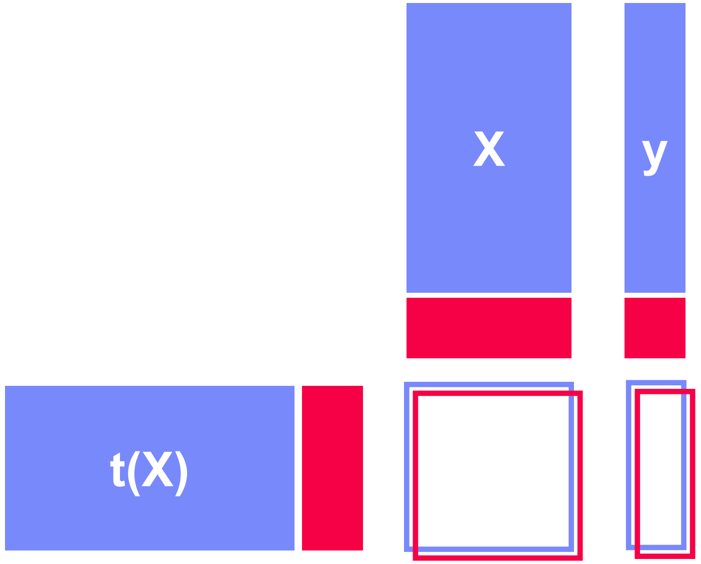
- **Loop Transformations** (e.g., OptiML, SystemML)
  - **Loop vectorization**
  - Loop hoisting

```
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
```



- **"Decremental"/Unlearning** (GDPR)

[Sebastian Schelter: "Amnesia" – Machine Learning Models That Can Forget User Data Very Fast. **CIDR 2020**]

[Sebastian Schelter, Stefan Grafberger, Ted Dunning: HedgeCut: Maintaining Randomised Trees for Low-Latency Machine Unlearning. **SIGMOD 2021**]

Update-in-place

- **Example: Cumulative Aggregate via Strawman Scripts**
  - **But:** R, Julia, Matlab, SystemDS, NumPy all provide `cumsum(X)`, etc

```
cumsumN2 = function(Matrix[Double] A)
  return(Matrix[Double] B)
{
  B = A; csums = matrix(0,1,ncol(A));
  for( i in 1:nrow(A) ) {
    csums = csums + A[i,];
    B[i,] = csums;
  }
}
```

copy-on-write → O(n^2)

```
cumsumNlogN = function(Matrix[Double] A)
  return(Matrix[Double] B)
{
  B = A; m = nrow(A); k = 1;
  while( k < m ) {
    B[(k+1):m,] = B[(k+1):m,] + B[1:(m-k),];
    k = 2 * k;
  }
}
```

→ O(n log n)

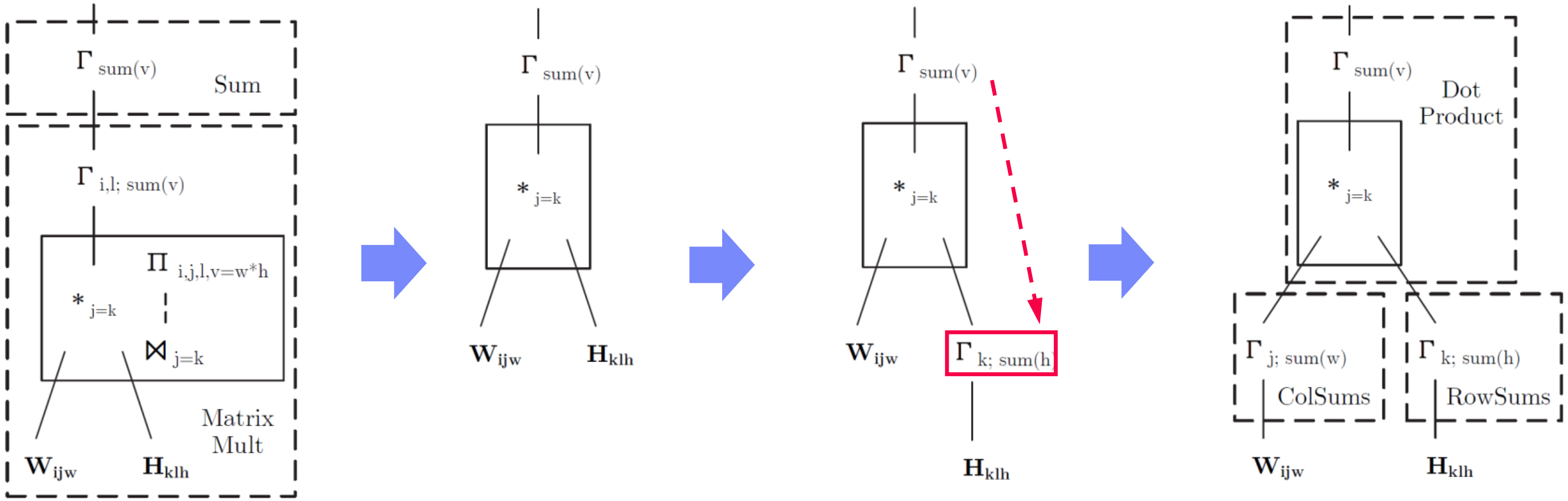
- **Update in place** (w/ O(n))
  - **SystemDS:** via rewrites (**why do the above scripts apply?**)
  - **R:** via reference counting
  - **Julia:** by default, otherwise explicit **B = copy(A)** necessary

Excursus: Automatic Rewrite Generation

[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**]

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

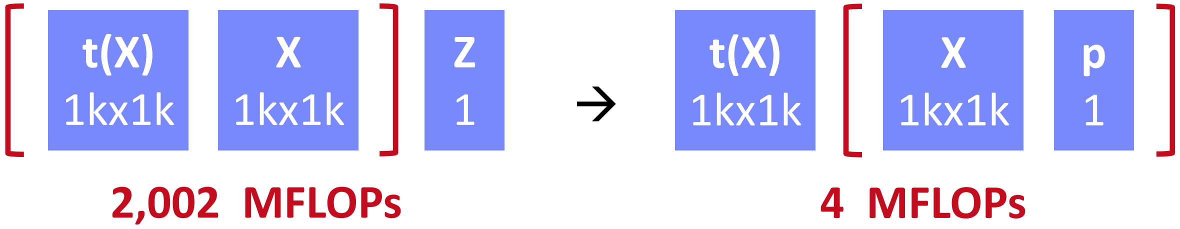


- **TASO (Super Optimization)**
  - List of operator specifications and properties
  - Automatic generation/verification of graph substitutions and data layouts via cost-based backtracking search

[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 $M_1, M_2, \ldots M_n$ (associative)
  - Optimal parenthesization of the product $M_1 M_2 \ldots M_n$



Size propagation and sparsity estimation

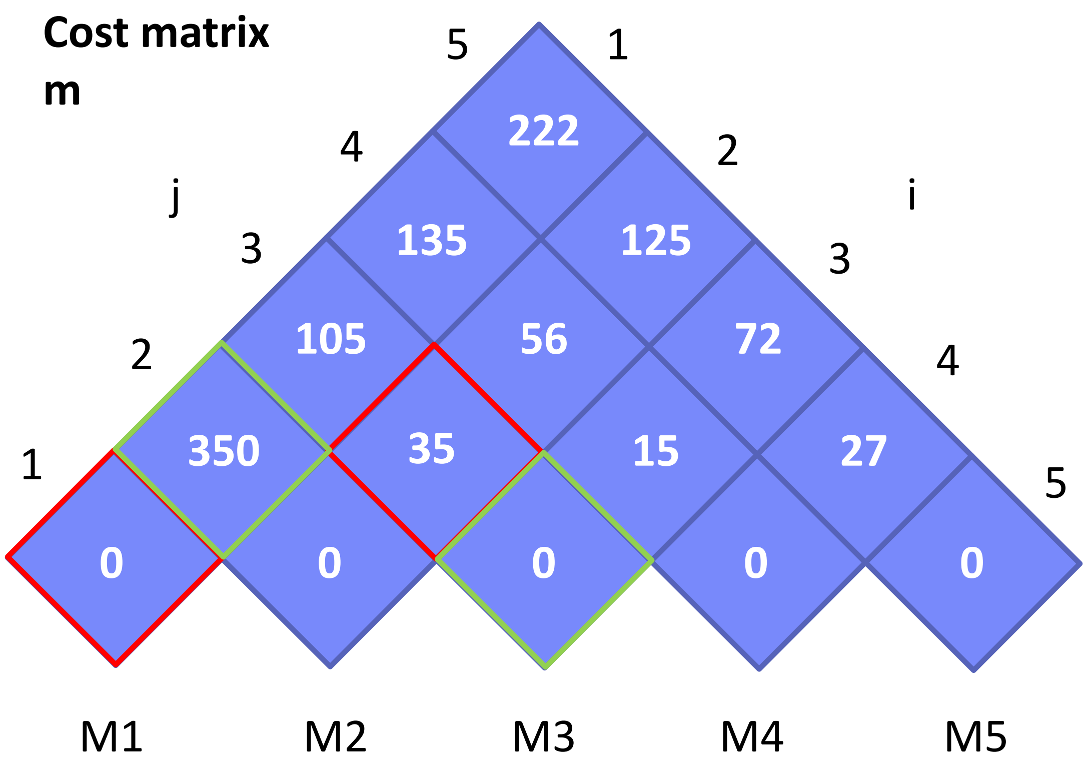
- **Search Space Characteristics**
  - Naïve exhaustive: Catalan numbers → $\Omega(4^n / n^{3/2})$
  - DP applies: (1) optimal substructure, (2) overlapping subproblems
  - Textbook DP algorithm: $\Theta(n^3)$ time, $\Theta(n^2)$ space
    - Examples: SystemML '14, RIOT ('09 I/O costs), SpMachO ('15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

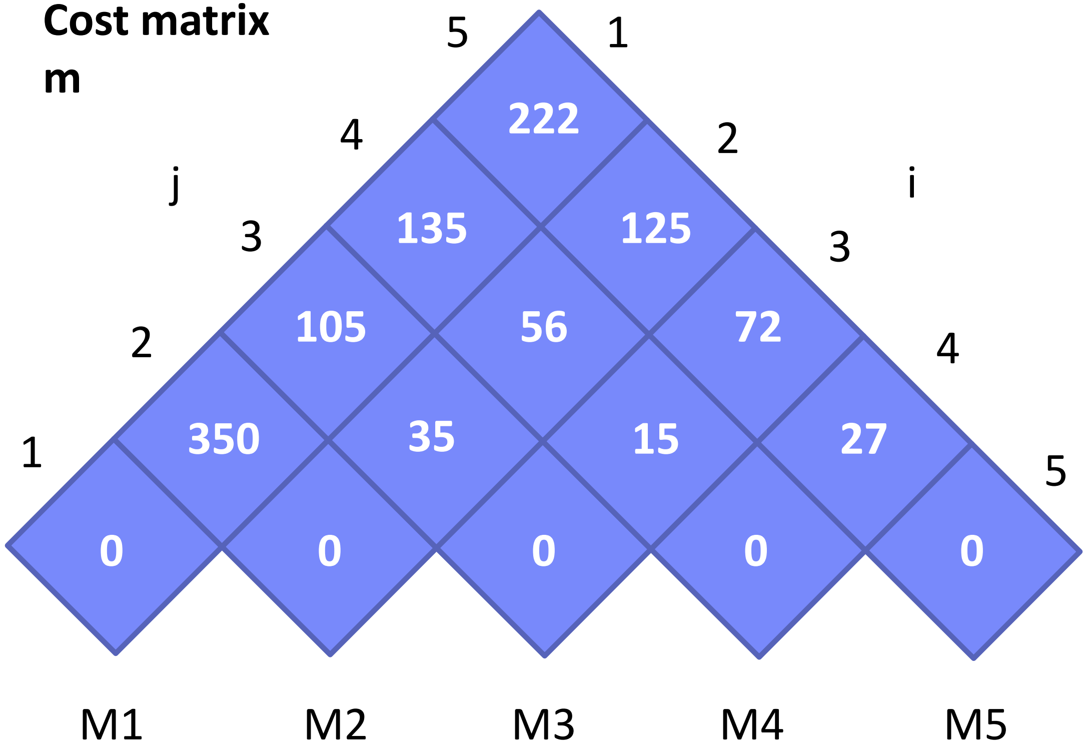
= min(
105,
400)

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

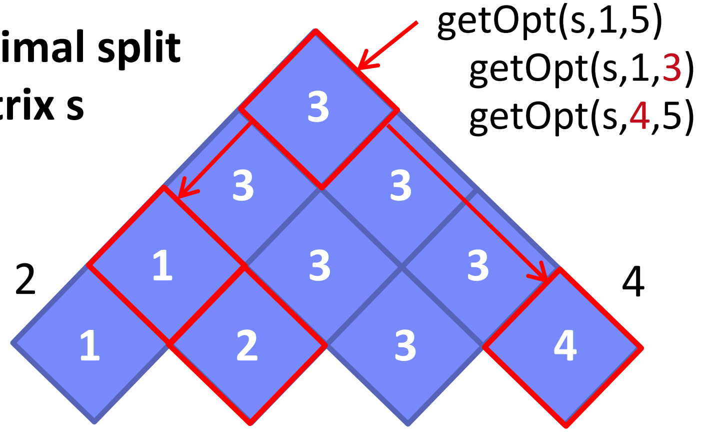
Matrix Multiplication Chain Optimization, cont.

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

Cost matrix m



Optimal split matrix s



(M1 M2 M3 M4 M5)

((M1 M2 M3) (M4 M5))

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

➔ ((M1 (M2 M3)) (M4 M5))

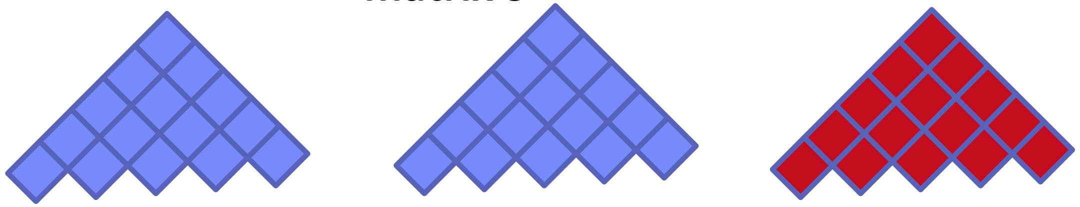
➔ Open questions: DAGs; other operations, sparsity, joint opt w/ rewrites, CSE, fusion, and physical operators

Sparsity-aware MMChain Optimization

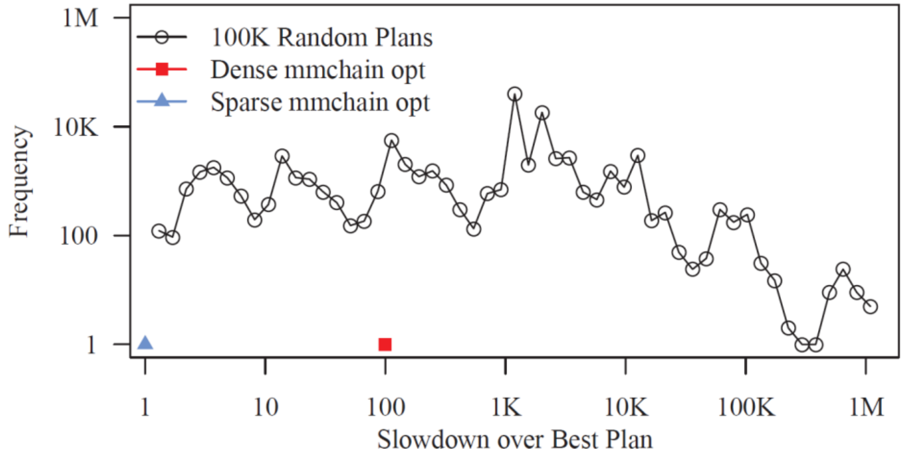
- **Sparsity-aware MMChain Opt**
  - Additional n x n sketch matrix e
  - Sketch propagation for optimal subchains (currently for all chains)
  - Modified cost computation via MNC sketches (**number FLOPs for sparse** instead of dense mm)

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

Cost matrix M | **Optimal split matrix S** | **Sketch matrix E**



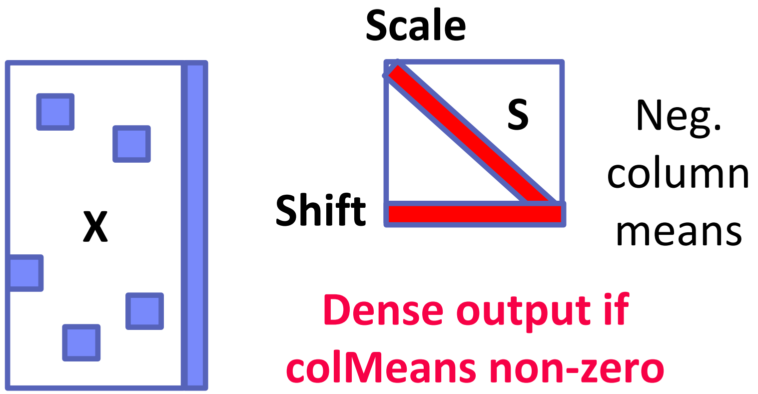
Example: n=20 matrices



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

Sparsity-aware MMChain Optimization, cont.

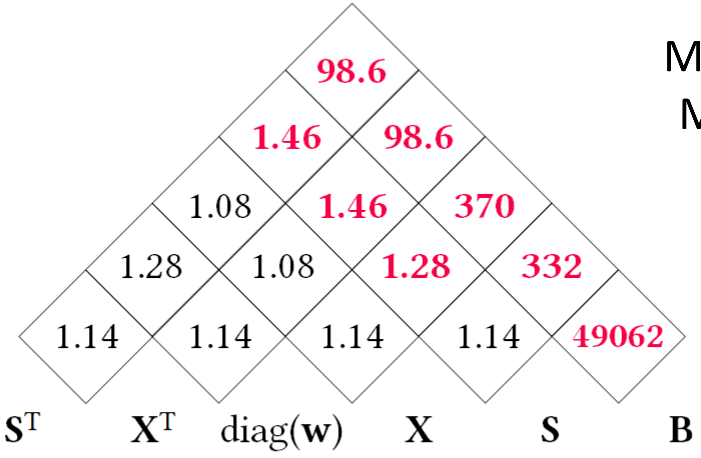
- **Example: Deferred Standardization**
  - Mean subtraction is densifying
  - Substitute standardized X with X %*% S and optimize mm chain



- **Accuracy of Sparsity Estimation**
  - Example Expression (substitute X with `X %*% S`)
  - Mnist1m
  - Error Density Map vs MNC Sketch

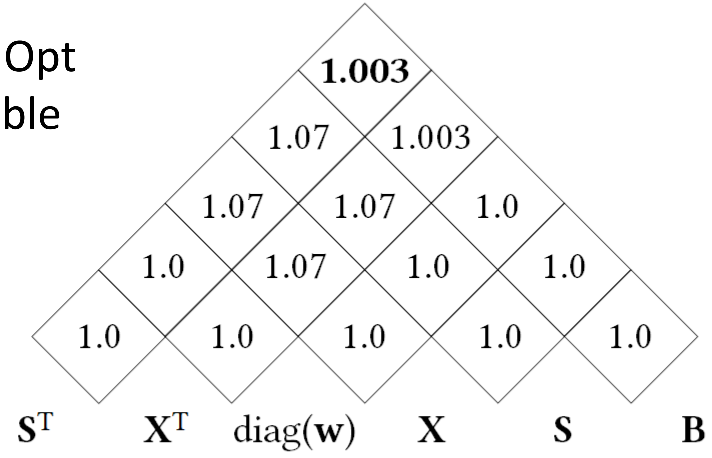
```
t(X) %*% diag(w) %*% X %*% B →
t(S) %*% t(X) %*% diag(w) %*% X %*% S %*% B
```

Density Map



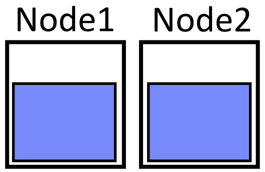
MMChain Opt Memo Table

MNC Sketch



Physical Rewrites and Optimizations

- **Distributed Caching**
  - Redundant compute vs. memory consumption and I/O
  - **#1 Cache intermediates w/ multiple refs** (Emma)
  - **#2 Cache initial read and read-only loop vars** (SystemML)



- **Partitioning**
  - Many frameworks exploit co-partitioning for efficient joins
  - **#1 Partitioning-exploiting operators** (SystemML, Emma, Samsara)
  - **#2 Inject partitioning to avoid shuffle per iteration** (SystemML)
  - **#3 Plan-specific data partitioning** (SystemML, Dmac, Kasen)

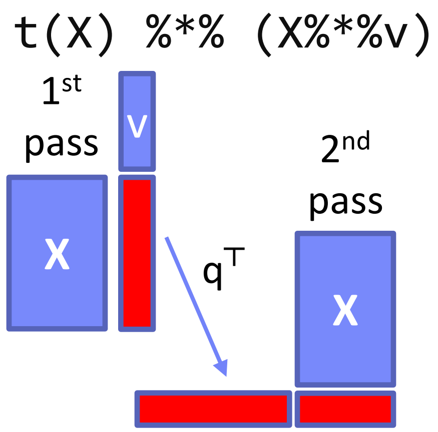
Example Hash Partitioning:
For all (k,v) of R:
hash(k) % numPartitions → pid

- **Other Data Flow Optimizations** (Emma)
  - **#1 Exists unnesting** (e.g., filter w/ broadcast → join)
  - **#2 Fold-group fusion** (e.g., groupByKey → reduceByKey)

- **Physical Operator Selection**

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

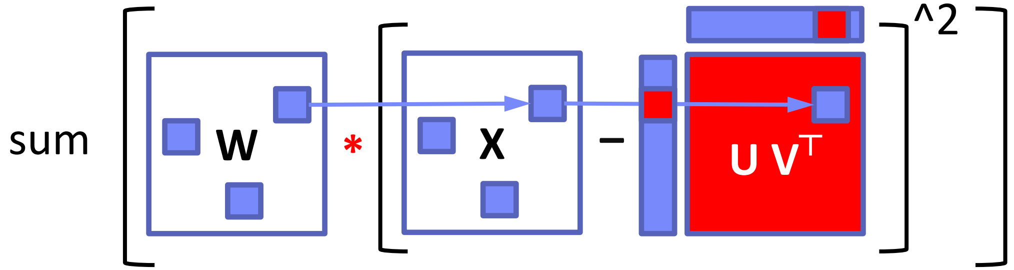


Sparsity-Exploiting Operators

- **Goal:** Avoid dense intermediates and unnecessary computation

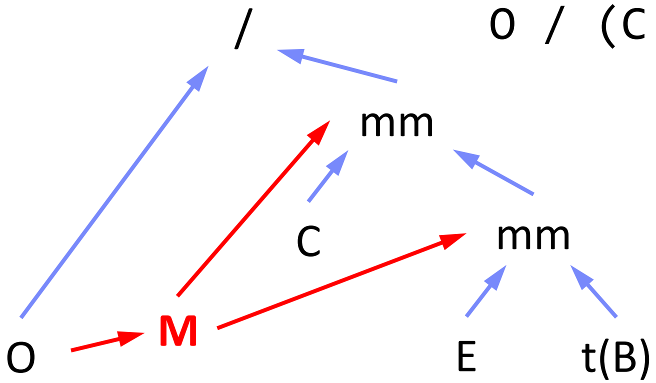
- **#1 Fused Physical Operators**
  - E.g., SystemML [PVLDB'16] wsloss, wcemm, wdivmm
  - Selective computation over non-zeros of **"sparse driver"**

`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))`



Summary & QA

- **Basic compilation overview**
- **Size inference and cost estimation**
- **Rewrites and operator selection**

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

➔ **Ubiquitous Rewrite Opportunities**
  - Linear algebra programs have plenty of room for optimization
  - Potential for changed asymptotic behavior

- **Next Lectures**
  - 04 Compilation – Operator Fusion and Runtime Adaptation [May 11]
    (advanced compilation, operator scheduling, JIT compilation, operator fusion / codegen, MLIR)