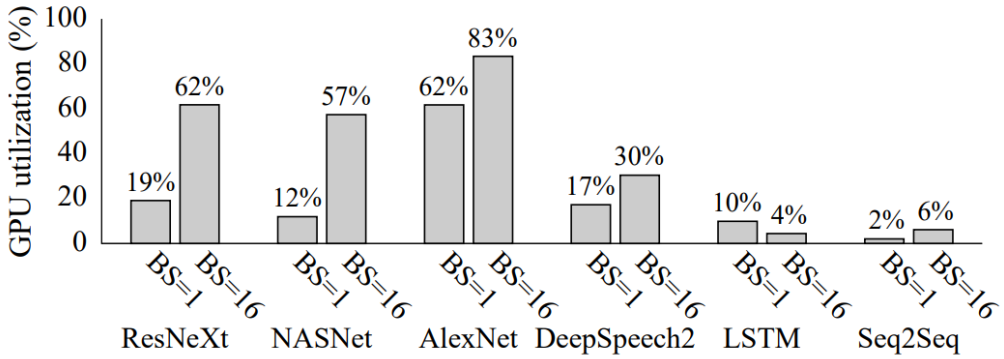
RAMMER: Enabling Holistic Deep Learning Compiler Optimizations with rTasks

Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, Lidong Zhou

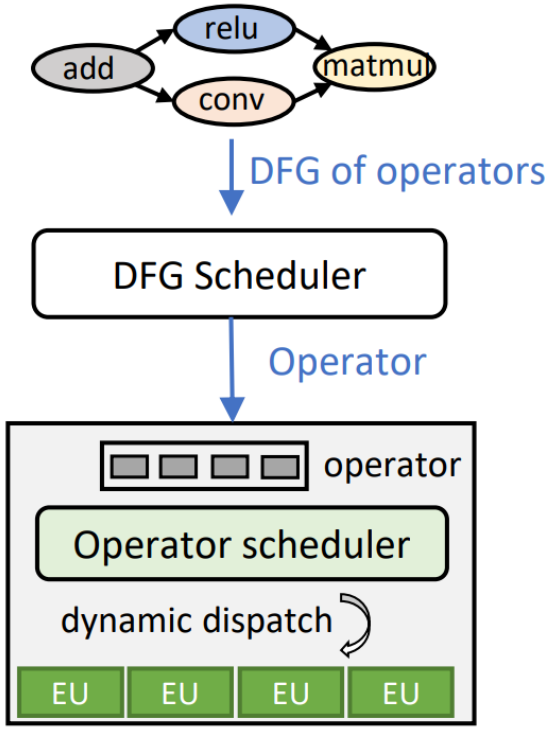
Presented by Tianrong Zhang

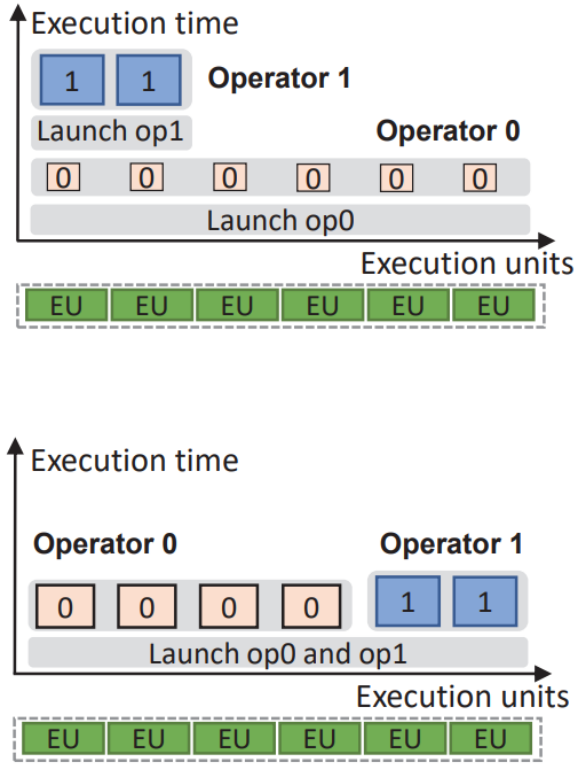
Motivations

- DNNs are often vastly parallelizable
  - Accelerators: GPU, FPGA, etc.
  - However, accelerators aren't fully utilized



  - Intra- and Inter- operator parallelism

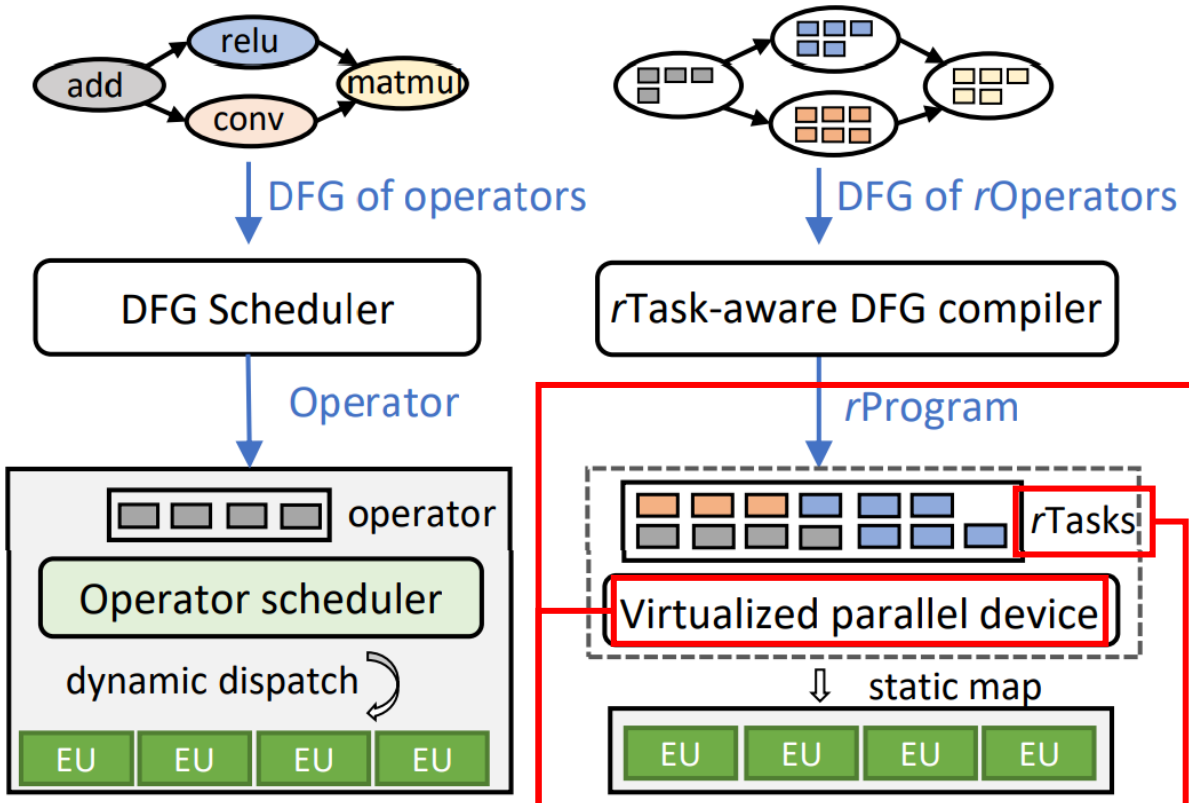




Challenges

- The operator implementations and their profiling details are opaque to DFG schedulers
  - Include operator implementations in DFG
  - Run profiler to gather details of each operation implementation
  - Use opensource libraries though less efficient in some cases
- The accelerators do not provide interface for software to manage the underlying scheduling
  - Define software required interfaces
  - Replace hardware scheduling with software
- Expensive for in time compilation
  - Apply only to freezed and static graph so that compilation can be done ahead of time

Compiler Workflow



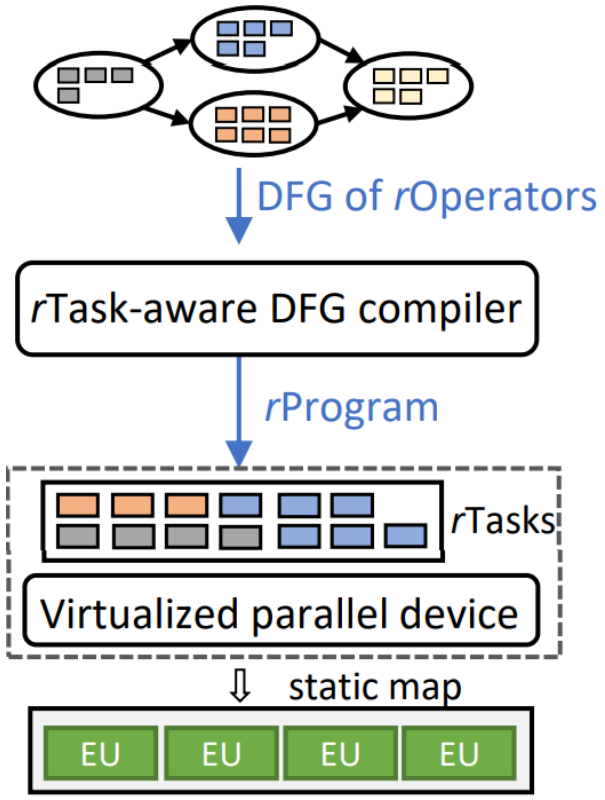

```
1  interface Operator { void compute(); };
2  interface rOperator {
3    void compute_rtask(size_t rtask_id);
4    size_t get_total_rtask_num();
5  };
```

```
1  __device__ void matmul_rTask(float *A, float *B,
2      float *C, size_t rtask_id) {
3    size_t tile_x = rtask_id / (M/32);
4    size_t tile_y = rtask_id % (N/32);
5    size_t i = threadIdx.x/32 + tile_x*32;
6    size_t j = threadIdx.x%32 + tile_y*32;
7    C[i][j] = 0;
8    for (size_t k = 0; k < K; k++)
9      C[i][j] += A[i][k] * B[k][j];
10 }
11
12 class MatmulROperator {
13   __device__ void compute_rtask(size_t rtask_id){
14     matmul_rTask(input0,input1,output0,rtask_id);}
15   size_t get_total_rtask_num(){return M/32*N/32);}
16 };
```

Expose potential profiling detail of each minimum job

Expose managing interface of hardware scheduling

Compiler Workflow



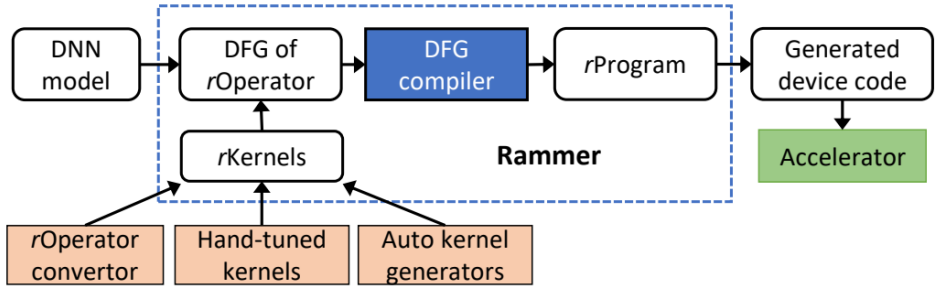

```
1  // config: <<<(vEU_size,1,1), (vEU#,1,1)>>>
2  __global__ void vdevice_run() {
3    if (Get_vEU_Id() == 0) { // vEU 0
4      MatmulrTaskOp.compute_rtask(0);
5      MatmulrTaskOp.compute_rtask(1);
6      // wait the rTask on vEU 1 with order=3
7      BarrierTask({<1, 3>}).compute_rtask();
8      Conv2DrTaskOp.compute_rtask(0);
9    }
10   else if (Get_vEU_Id() == 1) { // vEU 1
11     for (auto i : 4)
12       RelurTaskOp.compute_task(i);
13     // wait the rTask on vEU 0 with order=1
14     BarrierTask({<0, 1>}).compute_rtask();
15     Conv2DrTaskOp.compute_rtask(1);
16   }
17
```

wait mechanism

Store a step array in *r*Program's shared memory to keep track of of completed *r*Tasks.

persistent threadblock { vEu1, vEu2

Scheduling Strategy



Algorithm 1: Wavefront Scheduling Policy

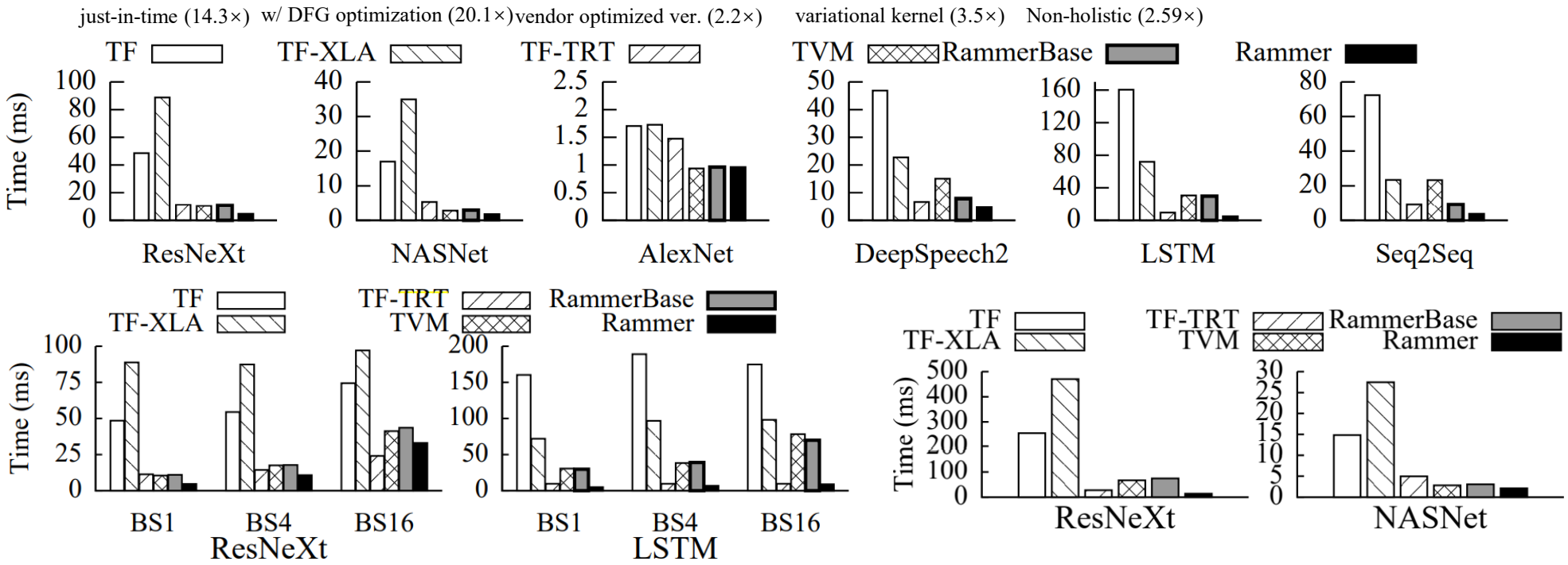
```
Data: G: DFG of rOperator, D: vDevice
Result: Plans: rPrograms
Function Schedule(G, D):
    P_curr = {};
    for W = Wavefront(G) do
        P_1 = ScheduleWave(W, P_curr, D);
        P_2 = ScheduleWave(W, {}, D);
        if time(P_1) ≤ time(P_curr) + time(P_2) then
            P_curr = P_1;
        else
            Plans.push_back(P_curr);
            P_curr = P_2;
    return Plans;
Function ScheduleWave(W, P, D):
    SelectRKernels(W, P);
    for op ∈ W do
        for r ∈ op.rTasks do
            vEU = SelectvEU(op, P, D);
            P.Wait(r, Predecessor(op).rTasks);
            P.Append(r, vEU);
    return P;
```

Corresponds to operator scheduler, implemented with BFS

Compile-time profiler

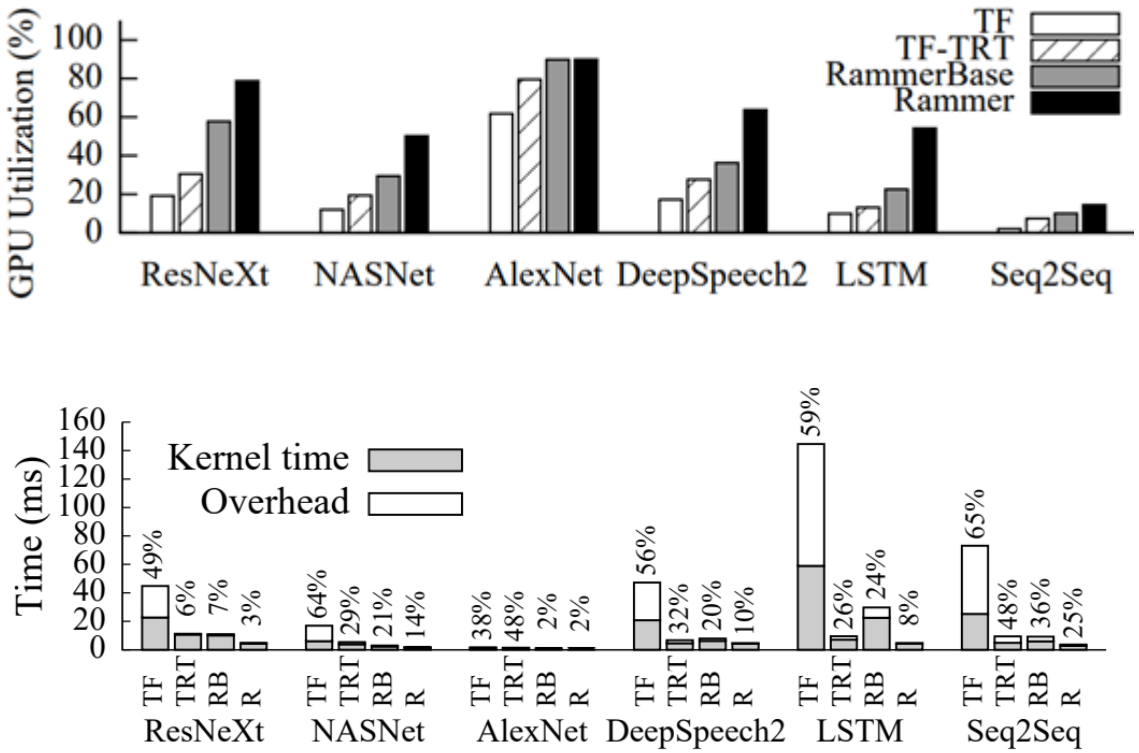
if most efficient implementations cannot saturate vDevice **then**
    Use most efficient implementations
else
    Try less efficient implementations for the most efficient operators
    Profile and accept if yield better running time

Experiments and Results

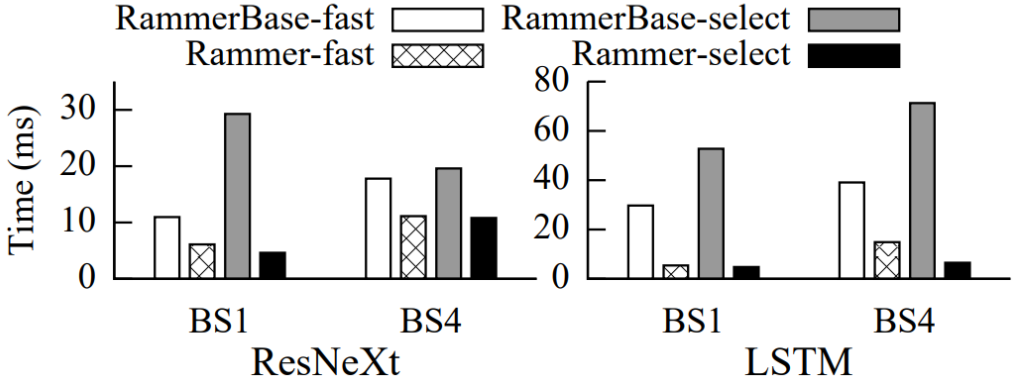


▲End2End results (overall inference time)

Experiments and Results



▲GPU utilization results and CPU runtime ratio results



▲Ablation study on *SelectRKernel*

Discussion

- Large data: Better performance boost when batch/input cannot saturate device.
- Dynamic graph: Since the schedule is determined at compile time, the model is required to be static for now.

Q&A

A Tensor Compiler for Unified Machine Learning Prediction Serving

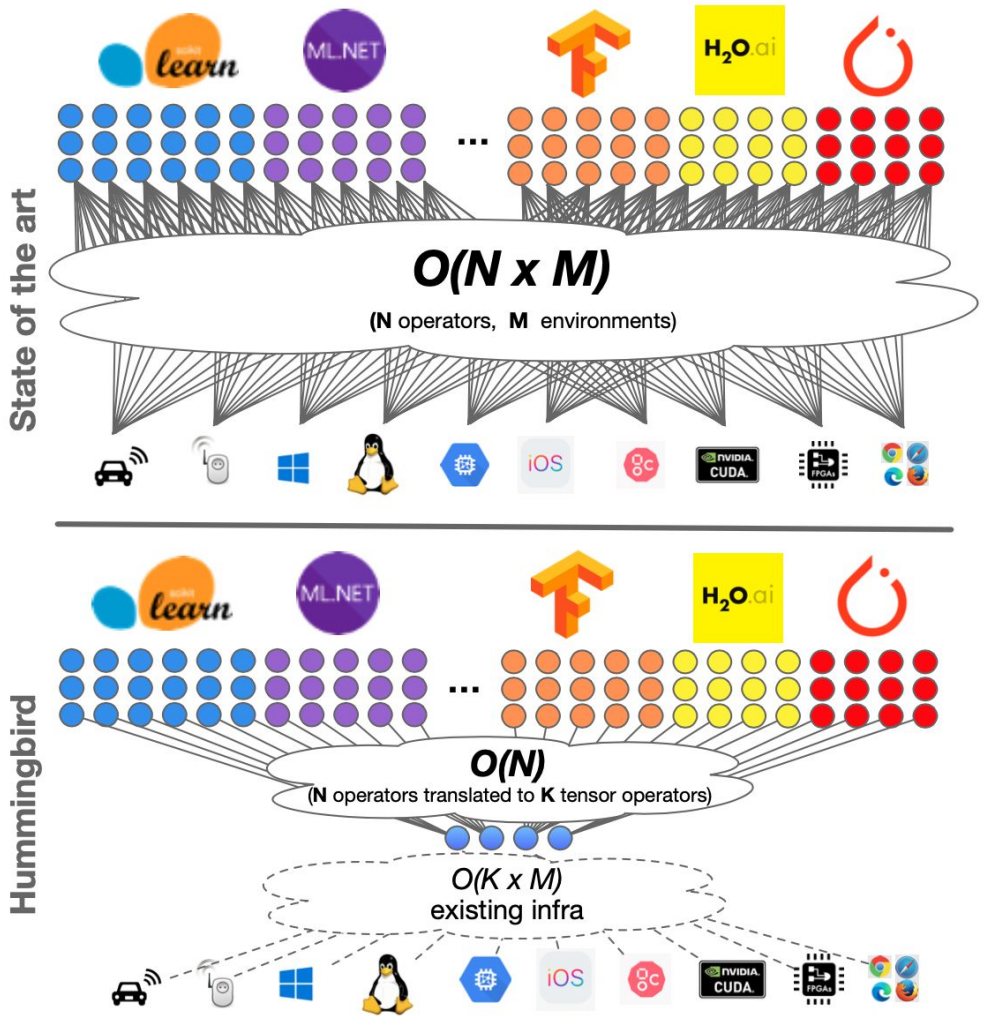
Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, Matteo Interlandi

14th USENIX Symposium on Operating Systems Design and Implementation

Motivation (1)

Traditional ML: Miss of abstraction leads to an O(N*M) explosion

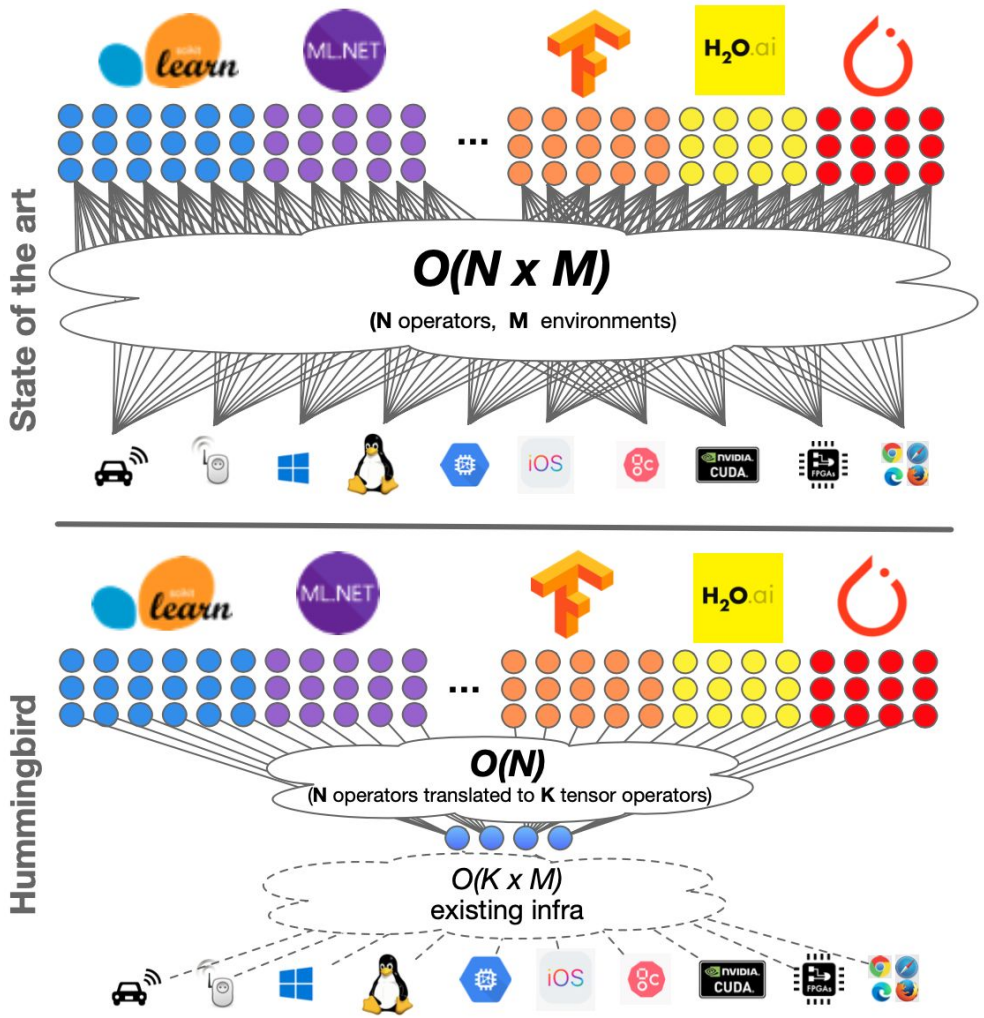
=> Hummingbird (HB) translates N operators into K core operators



Motivation (2)

Deep Neural Network (DNN): tensor transformation provides an algebraic abstraction

=> HB reduces traditional ML operators to tensor computations



Background

Predictive pipeline: the *iterative* process of designing and training traditional ML models

- a Directed Acyclic Graph (DAG) of typically tens of ***operators***

2 categories of operators:

- ***featurizers*** (e.g. string tokenizer, data normalization)
- ***models*** (e.g. linear models, decision tree ensembles)

Runtimes for DNN Inference: execute by implementing *a small set* of highly optimized operators on *multiple hardwares*

- e.g. ONNX Runtime (ORT) and TVM

How does HB work?

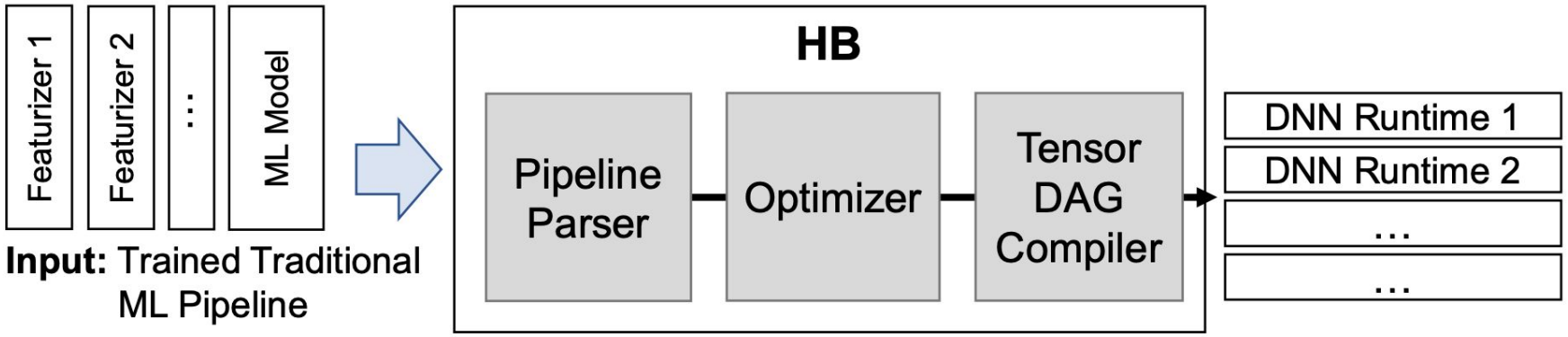
High-level idea:

introduces redundancies (both memory and *computation*) but utilize the efficient tensor operators

3 kinds of optimizations:

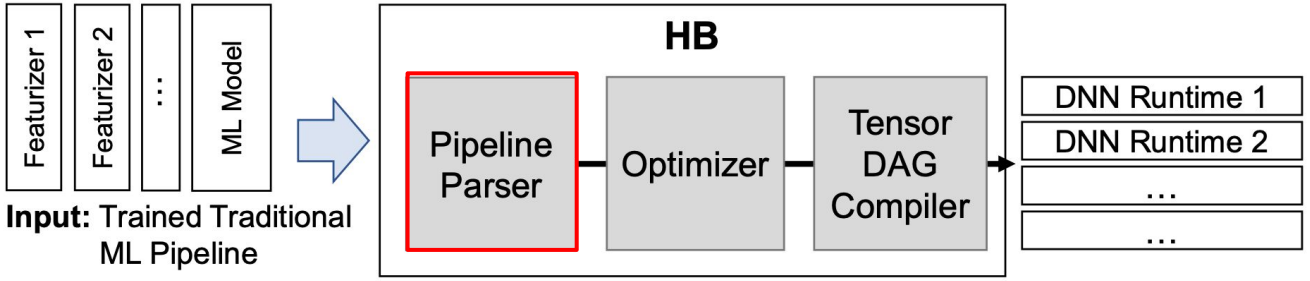
1. heuristics on compiler strategies selection (OPT1)
2. runtime-independent optimizations (OPT2)
3. runtime-specific optimizations (OPT3)

How does HB work?



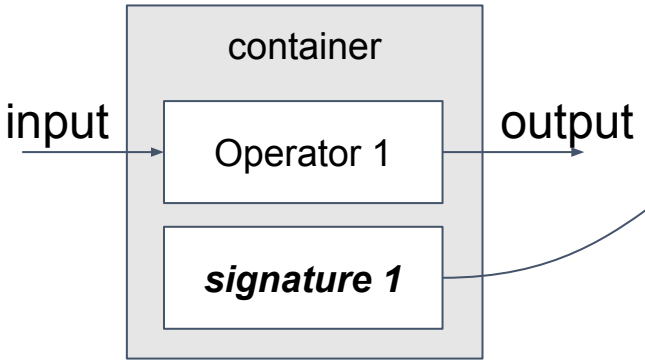
- **Pipeline parser**: parse operators into a DAG of containers
- **Optimizer**: (1) select compiler strategy; (2) apply runtime-independent optimizations **=> OPT1 and OPT2**
- **Tensor DAG Compiler**: convert to PyTorch *neural network module*, and export to target runtime format **=> OPT3**

Pipeline Parser



- parse operators into a DAG of ***containers***
- over 40 scikit-learn operators are supported in HB

For each operator:

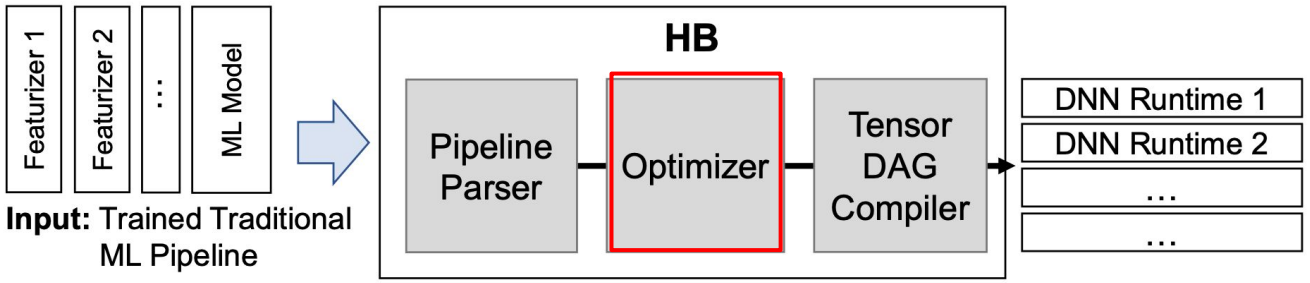


Build on start time:

extractor hash table	
signature 1	extractor 1
signature 2	extractor 2
...	...

extractor:
extract the parameters of each operator (e.g. thresholds of a decision tree).

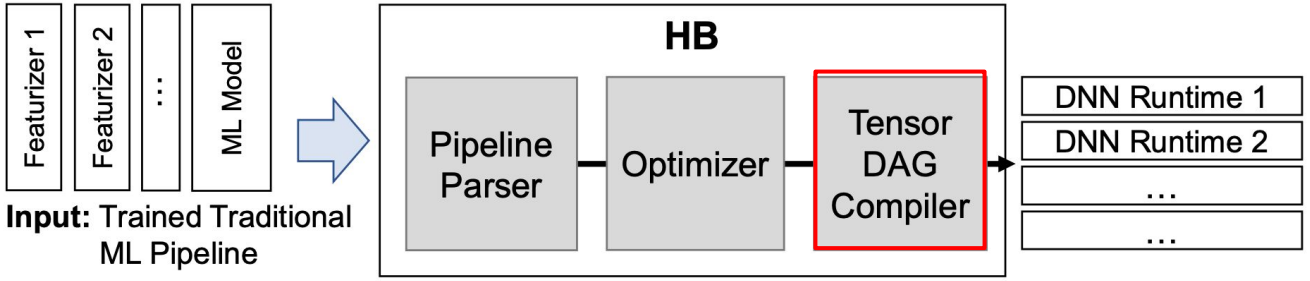
Optimizer



consist of 2 passes (traverse the DAG):

1. (prepare for the next step):
   a. extract parameters and store in container
   b. select the compiler strategy
2. apply runtime-independent optimizations
   a. **feature selection push-down**: similar to projection push-down in SQL
   b. **feature selection injection**: inject feature selection if not exist
      i. linear models: prune the zero weights
      ii. tree models: update the indices of the decision variables

Compiler



convert to PyTorch *neural network module*, and export to target runtime format (include runtime-specific optimizations)

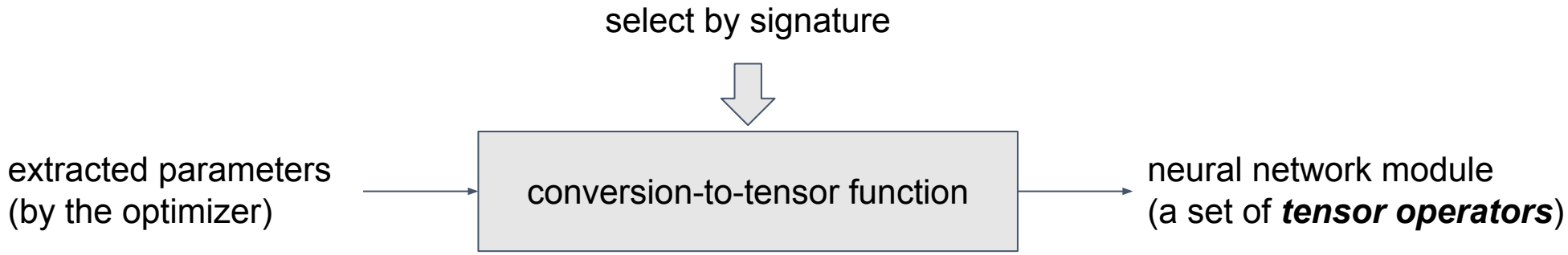


Table 2: PyTorch tensor operators used by the Tensor DAG Compiler.

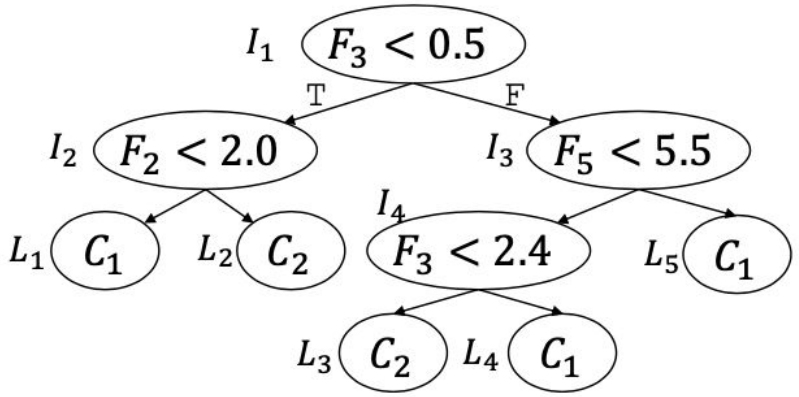
matmul, add, mul, div, lt, le, eq, gt, ge, &, |, ≪, ≫, bitwise_xor, gather, index_select, cat, reshape, cast, abs, pow, exp, arxmax, max, sum, relu, tanh, sigmoid, logsumexp, isnan, where

Compiler Example

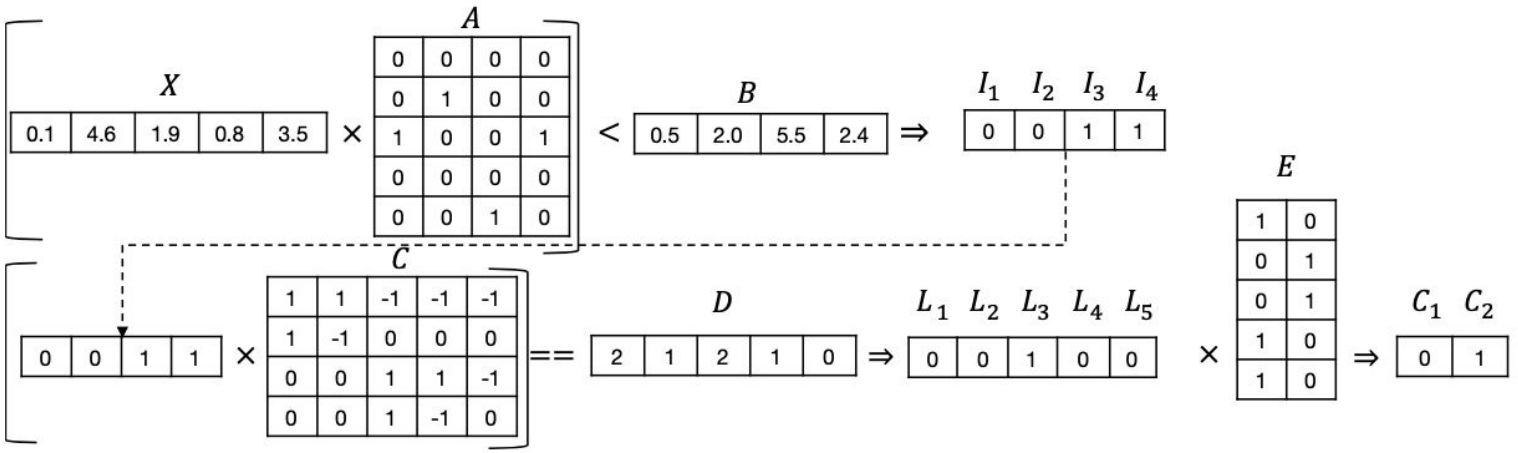
Strategies to compile tree-based model:

1. GEneric Matrix Multiplication (GEMM)
2. Tree Traversal (TT)
3. Perfect Tree Traversal (PTT)

GEneric Matrix Multiplication (GEMM)



What is the worst-case space/time complexity?



GEneric Matrix Multiplication (GEMM)

batch processing for trees of different sizes:

- pick the maximum size as the tensor dimensions
- pad the smaller tensor with zeros

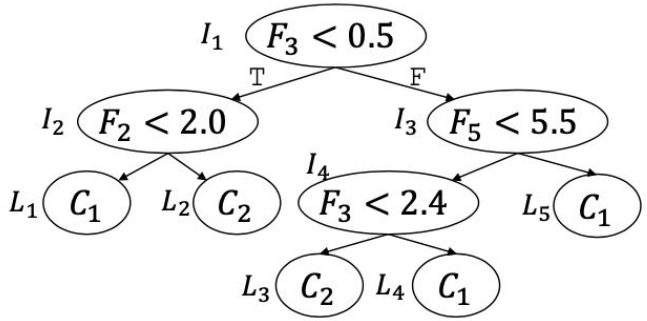
Input : $X \in \mathbb{R}^{n \times |F|}$, Input records
Output : $R \in \{0,1\}^{n \times |C|}$, Predicted class labels

```
/* Evaluate all internal nodes
```
$T \leftarrow \texttt{GEMM}(\mathbf{X}, \mathbf{A})$
$T \leftarrow T < B$

```
/* Find the leaf node which gets selected
```
$T \leftarrow \texttt{GEMM}(\mathbf{T}, \mathbf{C})$
$T \leftarrow T == D$

```
/* Map selected leaf node to class label
```
$R \leftarrow \texttt{GEMM}(\mathbf{T}, \mathbf{E})$

Tree Traversal (TT)



N_L:	2	4	6	4	5	8	7	8	9
N_R:	3	5	7	4	5	9	7	8	9
N_F:	3	2	5	1	1	3	1	1	1
N_T:	0.5	2.0	5.5	0	0	2.4	0	0	0
N_C:	0	0							
	0	0							
	0	0							
	1	0							
	0	1							
	0	0							
	1	0							
	0	1							
	1	0							

Input : $X \in \mathbb{R}^{n \times |F|}$, Input records
Output : $R \in \{0,1\}^{n \times |C|}$, Predicted class labels

```
/* Initialize all records to point to k, with k the index
   of Root node.                                          */
T_I ← {k}^n                                   // T_I ∈ Z^n
for i ← 1 to TREE_DEPTH do
    /* Find the index of the feature evaluated by the
       current node. Then find its value.                 */
    T_F ← Gather(N_F, T_I)                    // T_F ∈ Z^n
    T_V ← Gather(X, T_f)                      // T_V ∈ R^n
    /* Find the threshold, left child and right child     */
    T_T ← Gather(N_T, T_I)                    // T_T ∈ R^n
    T_L ← Gather(N_L, T_I)                    // T_L ∈ Z^n
    T_R ← Gather(N_R, T_I)                    // T_R ∈ Z^n
    /* Perform logical evaluation. If true pick from T_L;
       else from T_R.                                     */
    T_I ← Where(T_V < T_T, T_L, T_R)          // I ∈ Z^n
end
/* Find label for each leaf node                          */
R ← Gather(N_C, T_I)                          // R ∈ Z^n
```

Perfect Tree Traversal (PTT)

Assume a perfect binary tree

=> Remove N_L and N_R

=> reduce 2 Gather() per iteration

Input : $X \in \mathbb{R}^{n \times |F|}$, Input records
Output : $R \in \{0,1\}^{n \times |C|}$, Predicted class labels

```
/* Initialize all records to point to the root node.      */
```
$T_I \leftarrow \{1\}^n$ // $T_I \in \mathbb{Z}^n$

for $i \leftarrow 1$ **to** TREE_DEPTH **do**

```
    /* Find the index of the feature evaluated by the
       current node. Then find its value.                 */
```
    $T_F \leftarrow$ Gather(N_F, T_I) // $T_F \in \mathbb{Z}^n$
    $T_V \leftarrow$ Gather(X, T_f) // $T_V \in \mathbb{R}^n$

```
    /* Find the threshold                                 */
```
    $T_T \leftarrow$ Gather(N_T, T_I) // $T_T \in \mathbb{R}^n$

```
    /* Perform logical evaluation. If true pick left child;
       else right child.                                  */
```
    $T_I \leftarrow 2 \times T_I +$ Where$(T_V < T_T, 0, 1)$ // $I \in \mathbb{Z}^n$

end

```
/* Find label for each leaf node                          */
```
$R \leftarrow$ Gather(N'_C, T_I) // $R \in \mathbb{Z}^n$

Analysis of the Strategies

Strategy	Memory	Runtime
GEMM	$O(\|F\|\|N\| + \|N\|^2 + \|C\|\|N\|)$	$O(\|F\|\|N\| + \|N\|^2 + \|C\|\|N\|)$
TT	$O(\|N\|)$	$O(\|N\|)$
PTT	$O(2^{\|N\|})$	$O(\|N\|)$

=> Heuristics-based strategy selection

Other Techniques in the Compiler

1. Exploit Automatic Broadcasting
   a. broadcast: make two tensors shape compatible (reshape)
2. Minimize Operator Invocations
3. Avoid Generating Large Intermediate Results
4. Fixed Length Restriction on String Features

Performance

Task 1: Micro-benchmarks

- Tree ensembles
- Various other operators

Task 2: Optimizer

Task 3: End-to-end pipelines

Performance :Tree Ensembles (1)

3 algorithms, 6 datasets, CPU & GPU, HB on 3 different runtimes

Baseline:

- CPU: Sklearn, ONNX-ML
- GPU: RAPIDS FIL

conduct 8 experiments (batch inference, request/response, scaling batch size, etc.)

Performance: Tree Ensembles (2)

batch inference:

- CPU: HB-TVM outperforms on 15/18
- GPU: HB-TVM outperforms on 14/18

request/response (batch size = 1): HB-TVM outperforms on 11/15, 3x speedup

output validation:

- HB has no mismatch on 9/18 (12/18 for ONNX-ML)
- For Epsilon dataset: mismatch 0.1% records (11.5% for ONNX-ML)

Performance: Other Operators

test on 13 operators supported by both ONNX-ML and HB (Logistic Regression, Normalizer, etc.)

batch inference:

- CPU: HB-TVM outperforms on 8/13
- GPU: HB-TorchScript outperforms on 11/13

request/response:

- ONNX-ML outperform on 9/13
- all frameworks are within a factor of 2

Performance: Optimizer

Between GEMM, TT & PTT:

- GEMM outperforms for small batch sizes
- TT/PTT are better over large batch sizes, PTT is usually better
- PTT fails for very deep trees

Feature selection push-down

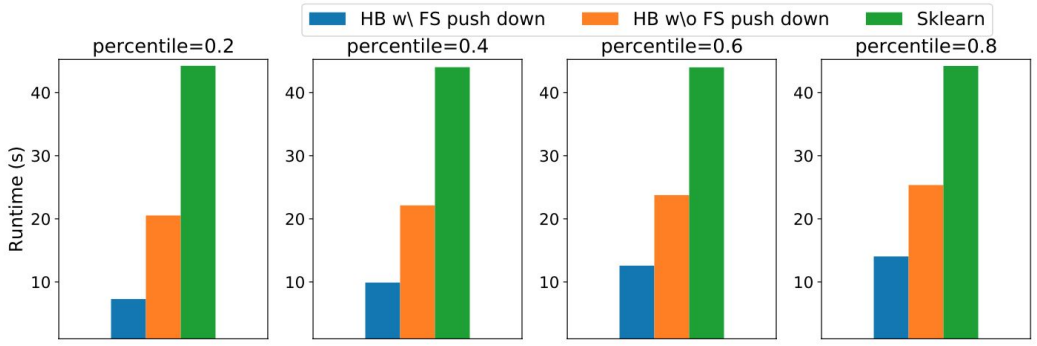


Figure 9: Feature selection push down.

Feature selection injection

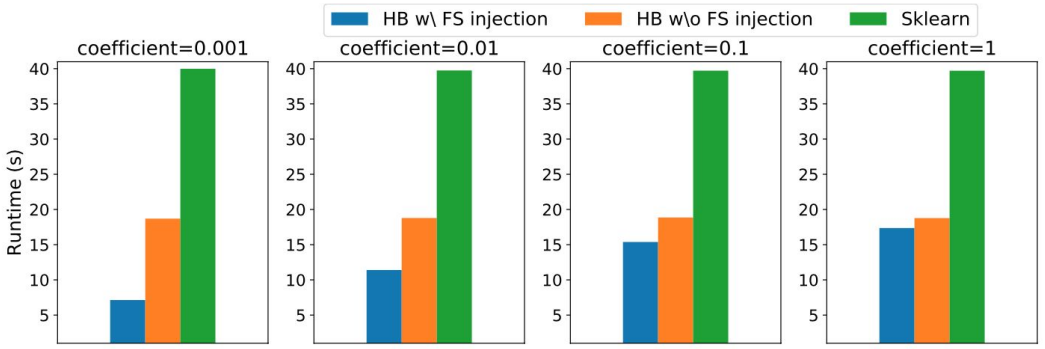


Figure 10: Feature selection injection.

Performance: End-to-end pipeline

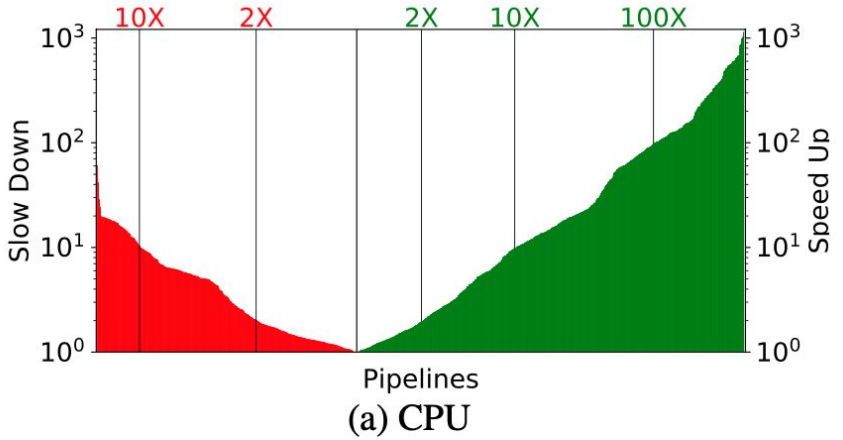
2317 pipelines (avg 3.3 operators)

CPU: 60% pipelines see speedup

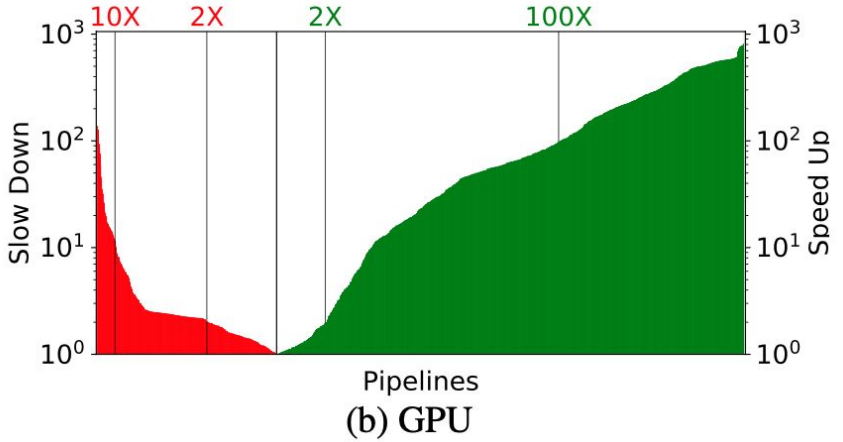
GPU: 73% pipelines see speedup

reasons for slowdown:

1. dataset is too small
2. sparse operators
3. pipeline does not require much computation



(a) CPU



(b) GPU

How it contributes?

Use two-level optimization approach

1. runtime-independent optimizer (OPT2)
2. strategy selection (OPT1) and other physical rewrites (supported by DNN runtimes, OPT3)

Provide 3 tree inference strategies

Q&A