Welcome!

CS 744: SPLIT ANNOTATIONS

Shivaram Venkataraman

Fall 2020

ADMINISTRIVIA

Course Project Checkins – due tomorrow! → Hot CRP

In-class project presentations

- Dec 8th and Dec 10th
- Sign up sheet on Piazza → 5 min slot $\simeq$ 4 min presentation + 1 min Q & A

Slides upload

Cloud computing
↳ serverless computing

Compose and maintain
efficiency

NEW HARDWARE AND DATA MODELS

SETTING

✓ Intel MKL | Options Pricing workload

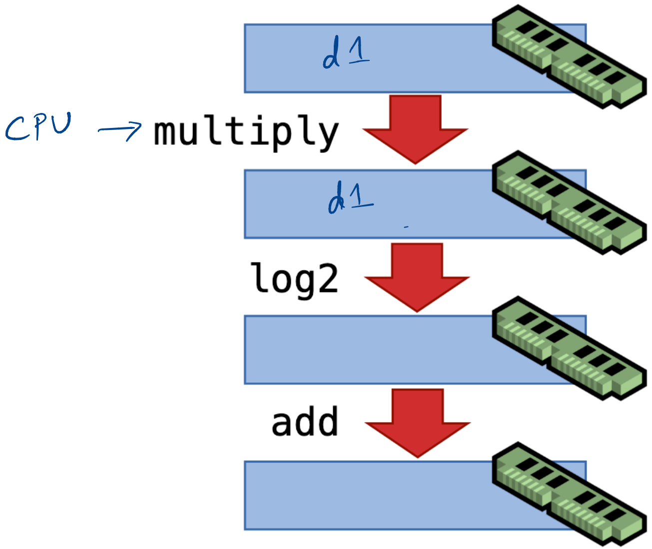
Multi-core machines

Multiple functions and libraries

```
// inputs are double arrays with `len` elems
vdLog1p(len, d1, d1);// d1 = log(d1)
vdAdd(len, d1, tmp, d1);// d1 = d1 + tmp
// d1 = d1 / vol_sqrt
vdDiv(len, d1, vol_sqrt, d1);
```

(1) Data movement is expensive even within a machine

(2) Arrays / data is larger than cache ⇒ streaming reads & writes to DRAM

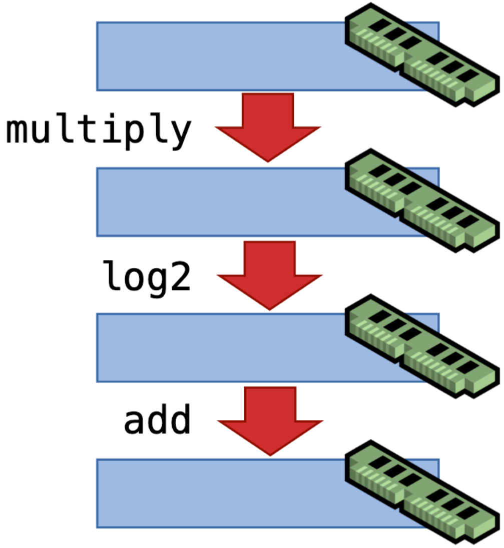


SCOPE
↳ optimizes across all operators

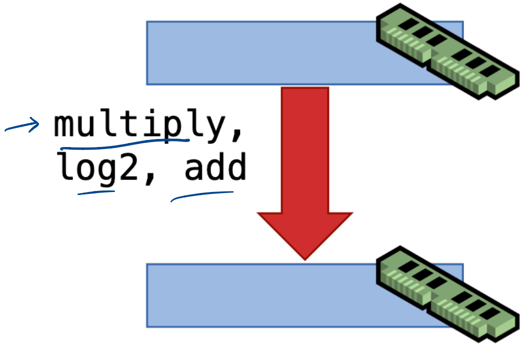
TVM
↳ layers of DNN

Spark
↳ cache if data fits in memory

COMPILER-BASED APPROACHES



(TVM)



we want to be here!

Replace every library call to emit intermediate representation (IR)

Compile all the IR together

loop fusion
pipelining

Lots of code change required!

Existing rich libraries NumPy, Pandas ....

GOALS

Provide data movement optimizations across libraries

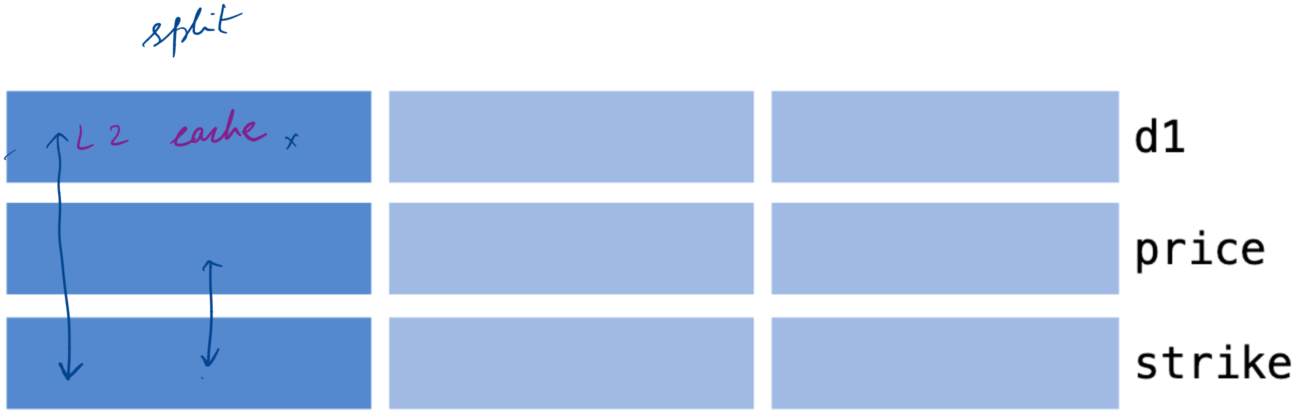
Require minimal or no changes to existing libraries → not be very intrusive

Leverage existing hand-tuned code for speedups

matrix multiply

FFT

APPROACH




```
d1 = price * strike
d1 = np.log2(d1) + strike
```

(1) Build execution graph

(2) Pass cache sized splits to every function

SPLIT ANNOTATIONS

easier to provide than changing code

Given a library
↳ fewer data types than operators

```
@splittable(
    size: SizeSplit(size), a: ArraySplit(size),
    mut out: ArraySplit(size))
void vdLog1p(long size, double*a, double*out)
```

vdScale (long size, int scalar, double *a)

you can pipeline these functions

a : [size 10]

vdLog1p(5, a, out)

vdLog1p(5, a+5, out+5)

Split types: N⟨V0…Vn⟩ e.g,: ArraySplit⟨10, 2⟩ for 10 element array, 2 pieces

Split annotation:

Name and split type to each argument and return value

Output is split in the same fashion as input

IMPLEMENTING SPLIT API

```
NameConstructor(A0,...An)=> Parameters
Split(D arg, int start, int end, Parameters)=> D
Merge(Vector<D>, Parameters)=> D
```

Arraysplit<10, 2>

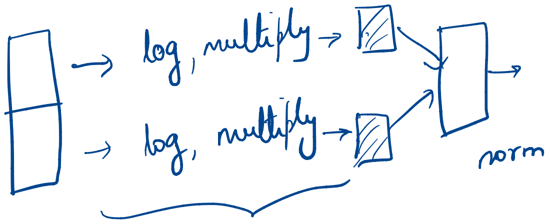
Split (double *a, start (5), int end (10), Parameters) =>

return a+5

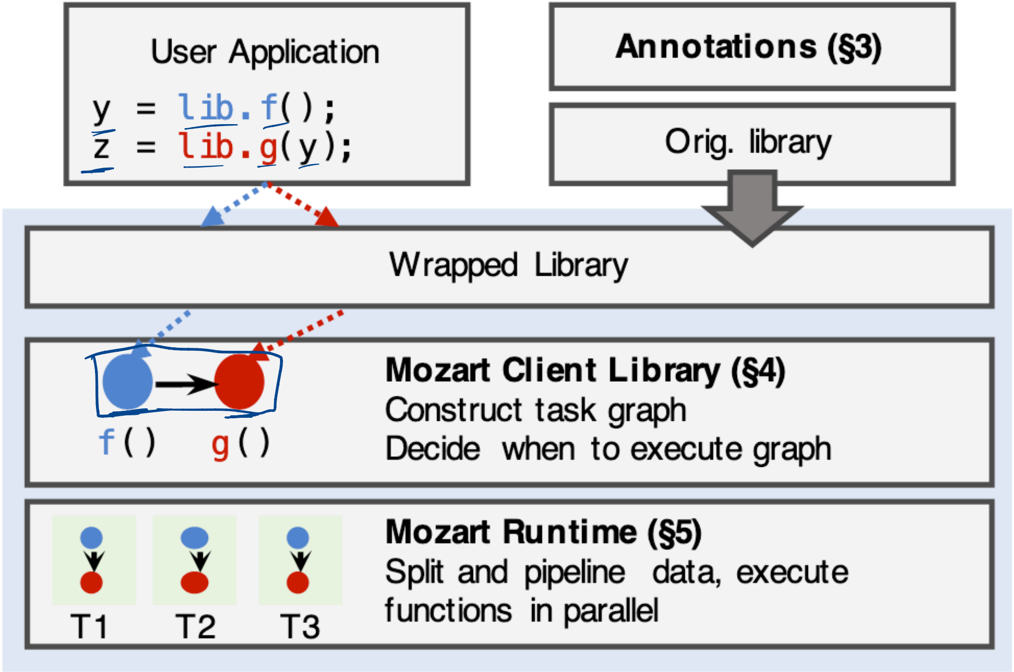
```
@splittable(m:MatrixSplit(m, axis), axis:_)
    -> ReduceSplit(axis)
vector sumReduceToVector(matrix m, int axis);
```

merge operation implemented inside ReduceSplit class to combine partial outputs

- If data shares same split type => you can safely pipeline
- If you cannot pipeline merge prior results call next function



MOZART DESIGN



→ Capture this execution graph

→ lazily evaluate this graph, maximum opportunity to pipeline

PYTHON CLIENT LIBRARY

Already exists

Writing Annotations: Function decorators

```
@sa((DataFrameSplit(), DataFrameSplit()), {}, DataFrameSplit())
def divide(series, value):
```

Pandas library

Capturing the graph

Wraps original Python function and registers in graph

Returns a Future object → (Ray, PyWren)

If somebody calls "divide", can be intercepted by decorator

Graph is constructed internally

Evaluation Points

Lazily evaluate by overriding __getattribute__

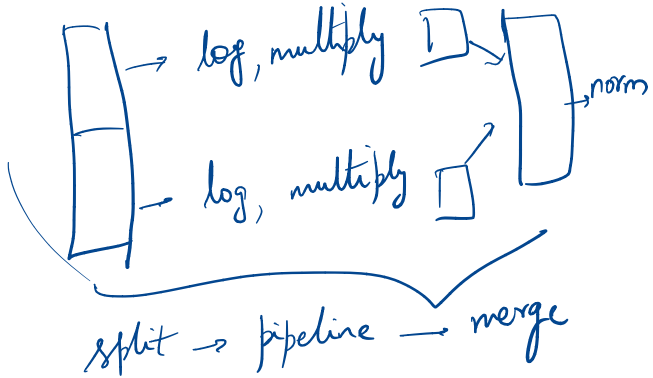
Future [Dataframe] : print (10) → internally do the eval and call print on the result

MOZART RUNTIME

Take dataflow graph → execution plan

Series of **stages** each stage split, pipeline and merge

Execute one stage at a time



Choosing a batch size

Set number of elements per batch using L2 cache size

Compute number of elements that will fit in L2 cache.

SUMMARY

Applications compose data processing libraries

Data movement is bottleneck on multi-core machines

Key idea: Split and pipeline data across functions

Split Annotations to reduce programmer effort

Mozart: Client library and runtime for lazy evaluation

Iterative workload

↳ will add stages to graph

↳ pipeline across iterations?

DISCUSSION

https://forms.gle/F2LJ21qFkBGWyypB7

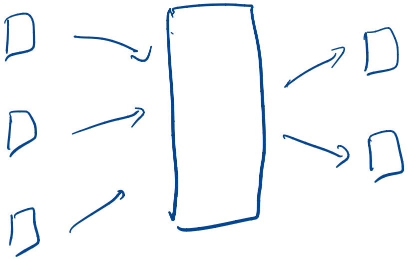
How does the dataflow graph that is executed by Mozart compare to dataflow graphs we have seen in other systems like Spark/PyTorch etc.

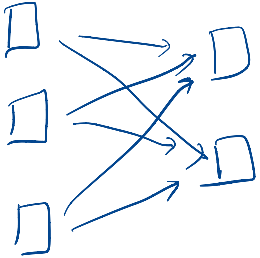
Similarities

→ Lazy execution

→ narrow dependencies
= pipelined by Mozart.

→ merging vs. shuffling





Differences

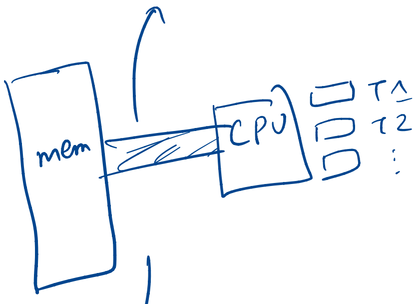
→ Fault tolerance is not the objective

→ No checkpointing

→ Functions are blackboxes

↳ Can't pick optimal join operator

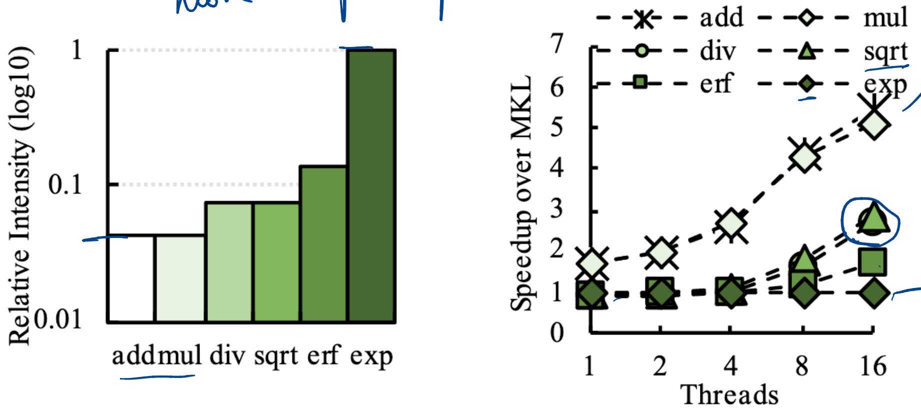
mem bandwidth



having more threads can lead to mem bw bottleneck

exp > 10x more comp. expensive

speedups increase with n_threads for add, mul



(a) Relative Intensity **(b)** Speedup over no SAs

speed for exp ~ 1x

compute intensive functions ⇒ not much speed up

NEXT STEPS

Next class: TPU

Project check-ins on HotCRP!