CS 744: SPLIT ANNOTATIONS

Shivaram Venkataraman

Fall 2020
Course Project Checkins – due tomorrow! → Hot CRP

In-class project presentations

Dec 8th and Dec 10th

Sign up sheet on Piazza → 5 min slot → 4 min presentation

Slides upload + 1 min Q&A
NEW HARDWARE AND DATA MODELS
Multi-core machines

Multiple functions and libraries

// inputs are double arrays with `len` elems
vdLog1p(len, d1, d1);// d1 = log(d1)
vAdd(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

Intel
MKL
Options
Pricing
workload

SCOPE
L3 optimizes across all operators

TVM
L3 layers of DNN

Spark
L2 cache if data fits in memory

CPU -> multiply

log2

add

$\vdots$
COMPILER-BASED APPROACHES

Replace every library call to emit intermediate representation (IR)

Compile all the IR together

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
APPROACH

\[ d_1 = \text{price} \times \text{strike} \]
\[ d_1 = \text{np.log2}(d_1) + \text{strike} \]

(1) Build execution graph

(2) Pass cache sized splits to every function
SPLIT ANNOTATIONS

@splittable

```java
size: SizeSplit(size), a: ArraySplit(size),
mut out: ArraySplit(size))
```

```java
void vdLog1p(long size, double*a, double*out)
```

Split types: N\langle V0...Vn\rangle 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

Given a library

- Fewer data types than operators

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

```java
vdLog1p(5, a, out)
vdlLog1p(5, a+5, out+5)
```
IMPLEMENTING SPLIT API

NameConstructor(A0,...An) => Parameters ArraySplit<10,2>
Split(D arg, int start, int end, Parameters) => D
Merge(Vector<D>, Parameters) => D

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);

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

merge operation implemented inside ReduceSplit class to combine partial outputs
MOZART DESIGN

Capture this execution graph:

- Lazily evaluate this graph, maximum opportunity to pipeline

User Application
\[
\begin{align*}
y &= \text{lib.f}() \\
z &= \text{lib.g}(y)
\end{align*}
\]

Annotations (§3)
Orig. library

Wrapped Library

Mozart Client Library (§4)
- Construct task graph
- Decide when to execute graph

Mozart Runtime (§5)
- Split and pipeline data, execute functions in parallel
Writing Annotations: Function decorators

```python
@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)

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 $\rightarrow$ 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
DISCUSSION

https://forms.gle/F2LJ21qFkBGWyyB7
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

**Differences**

→ Fault tolerance is not the objective

→ No checkpointing

→ Functions are black boxes

→ Merging vs. shuffling

→ Can't pick optimal join operator
Having more threads can lead to a mem bottleneck.

Relative Intensity (log10)

(a) Relative Intensity

Speedup over no SAs

(b) Speedup over no MKL

exp > 10x, more comp, expensive

Speedups increase with n-threads for add, mul

Speed for exp ~ 1lx

Compute intensive functions => not much speed up
NEXT STEPS

Next class: TPU
Project check-ins on HotCRP!