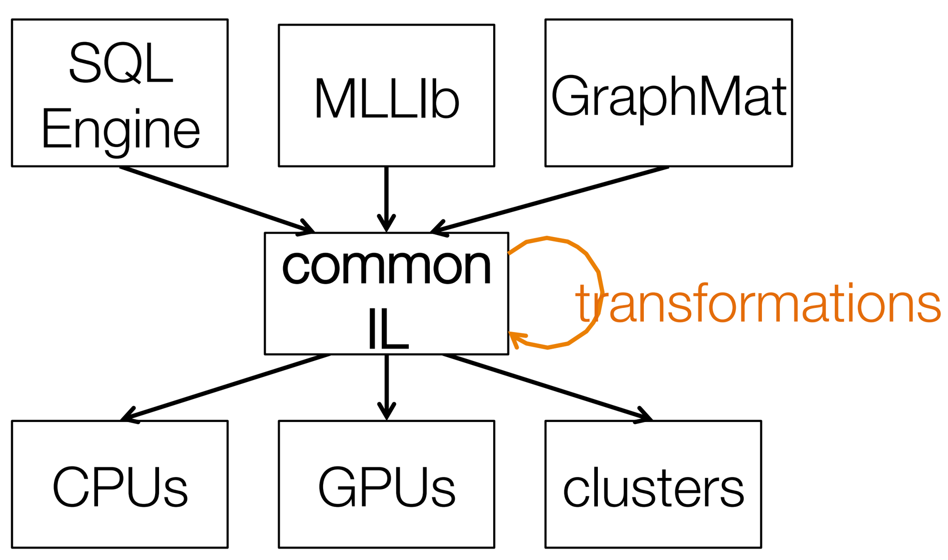
Nested Vector Language: Roofline Performance for Data Parallel Code

Shoumik Palkar, James Thomas, Matei Zaharia

Motivation

- Writing fast parallel code is **hard**.
  - *Numerous complex evolving platforms (GPUs, CPUs) and techniques (multicore, SIMD).*
- Many common algorithms can be written through "embarrassingly parallel" data operations.
  - *MapReduce is empirical example*
- *Libraries* like Numpy, Pandas, MLLib emit this language (programmers write high level code)

Focus on parallel operations.



An IL for optimizing data-parallel code using closed transformations

Overview and Examples

- Small language with closed transformations
- Few types: vectors, structs, dictionaries, primitives
- ***Builders*** compose partial results associatively
  - *like Cilk's reducers, Spark's Accumulator*
- ***Iteration*** is the only fundamental parallel construct
  - *Some specialization: SIMD, multicore, etc.*
- Functional ops implemented as library

Implementing map

```
map(v: vec[T], func: (x: T) => U) =>
    for(v, vecBuilder[T], func)
```

Merging and Inlining Loops

```
(v: vec[int]) =>
  map(map(
      v,
      (x: int) => x+1)
    ),
    (y: int) => y*10
  )
```

```
(v: vec[int]) =>
  map(v,
      (x: int) => (x+1)*10)
  )
```

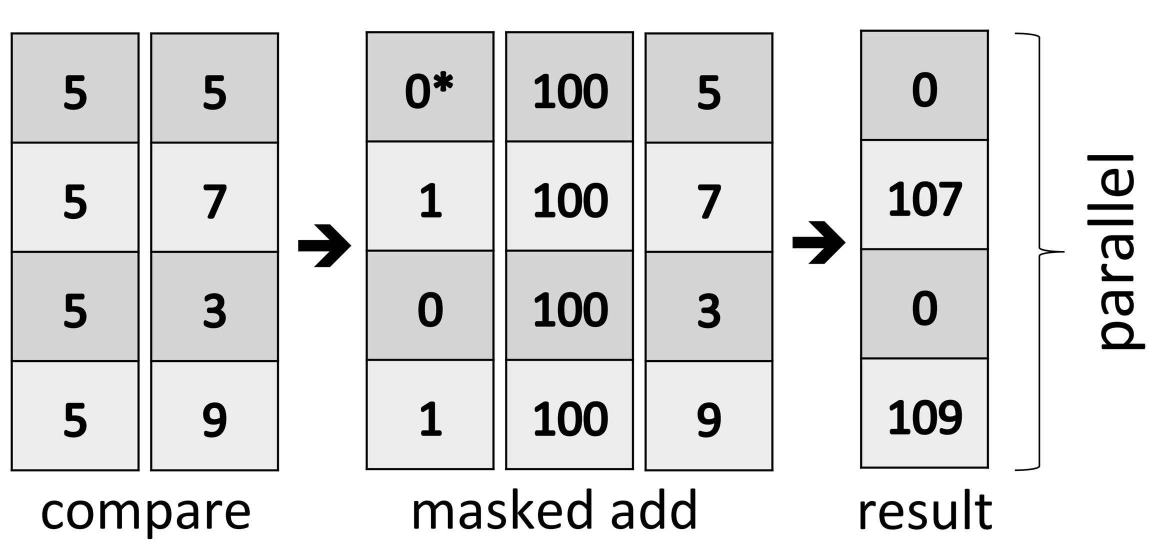
Vectorization

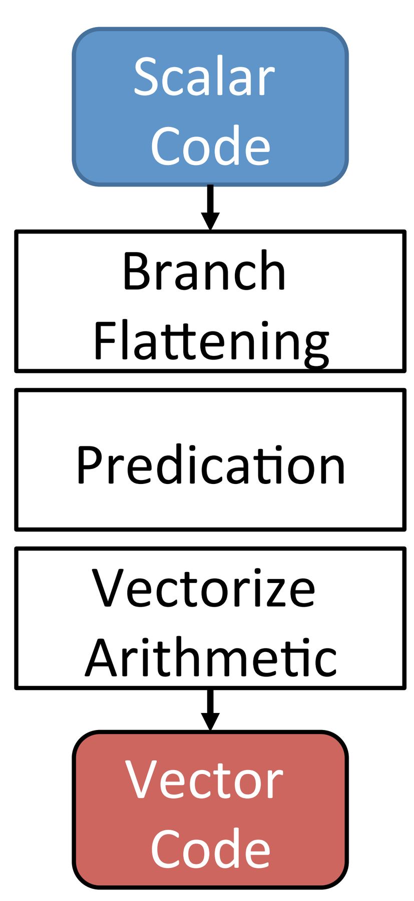
- Goal: leverage SIMD instructions to exploit data-parallelism on a single CPU

```
%res = add i32 %op1 %op2              Scalar
%res = add <8 x i32> %op1 %op2        Vectorized
```

Difficult with branches.
Transform to use predication:

```
if (5 < x) x += 100;     Branched
x += 100 * (5 < x);      Predicated
```

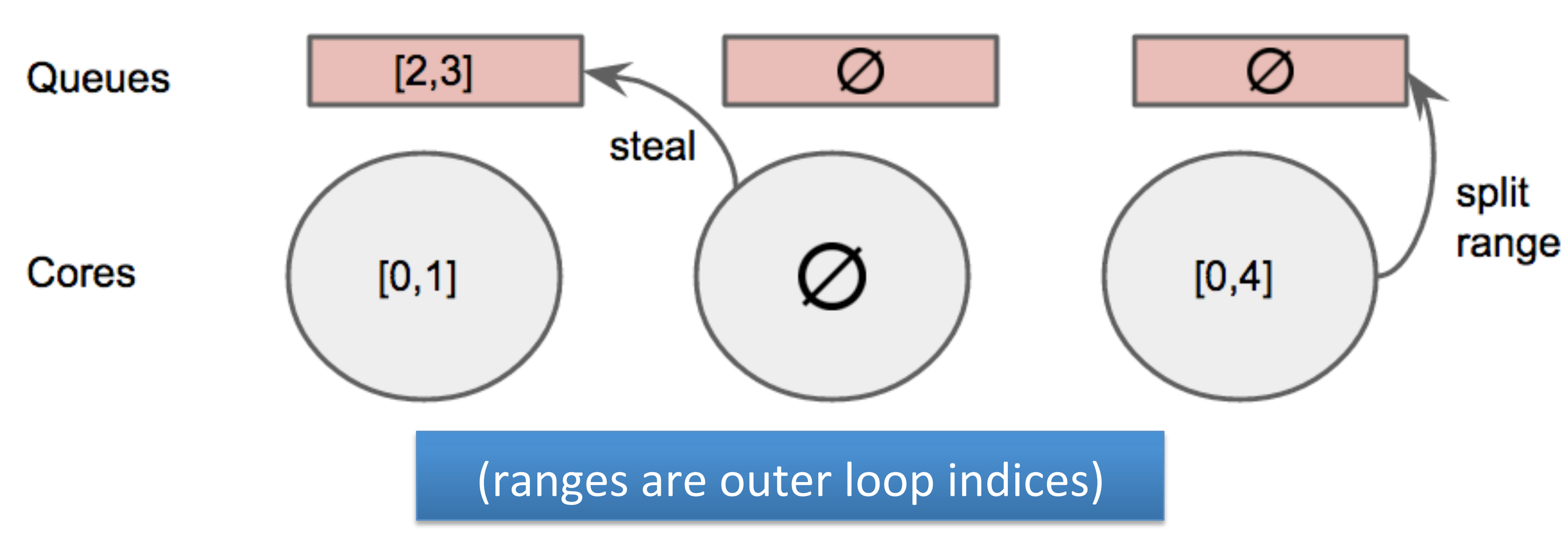




*Identity element inferred from builder type. Implemented using select instr.

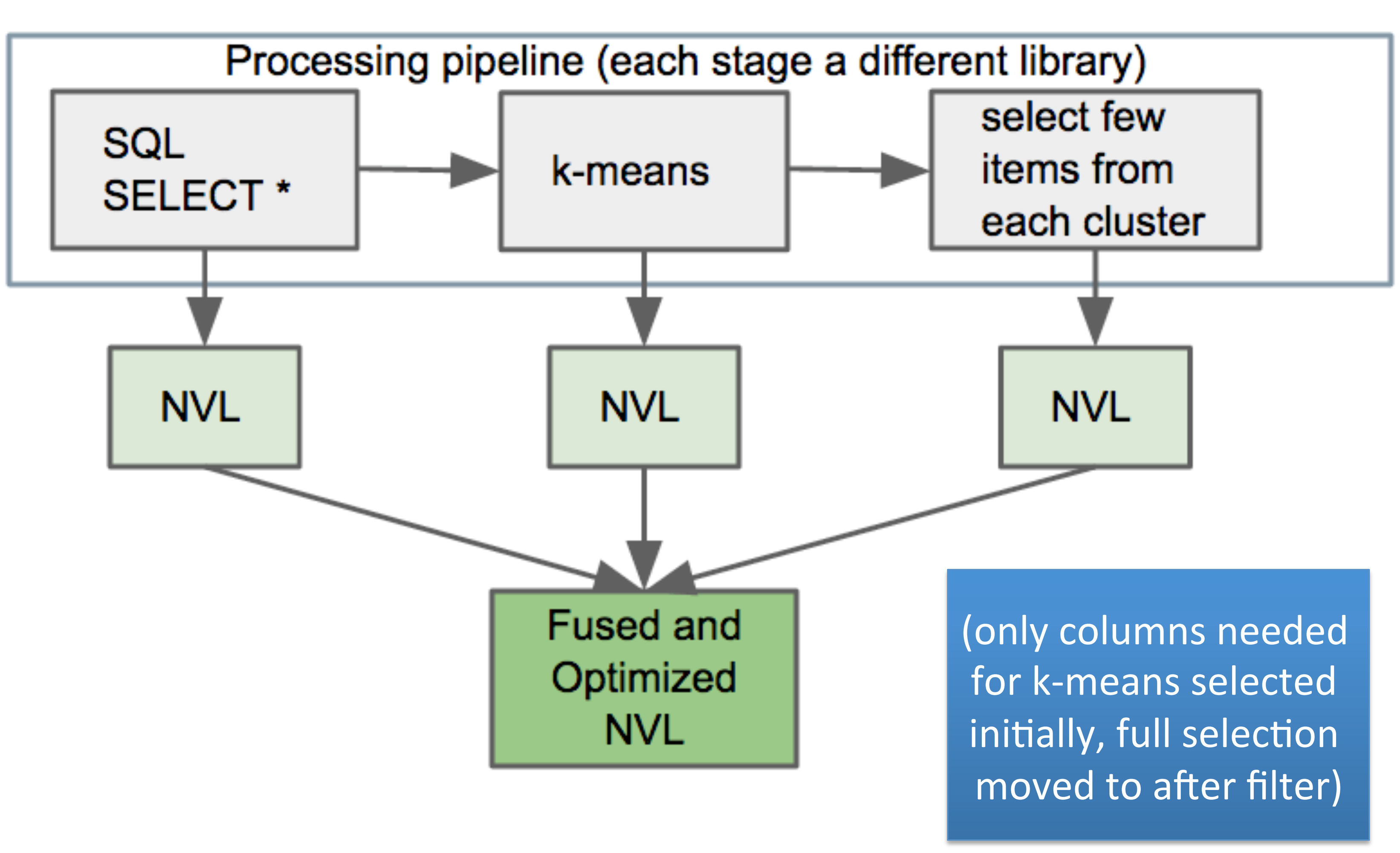
Parallelization

- Single data-parallel construct to parallelize – for loop
- Two main challenges
  1. Dynamic load balancing among cores
  2. Parallel state construction with builders
- Solutions
  1. Steal queued work from outermost loop of other cores
  2. Per-core state & merge into global state when size threshold crossed



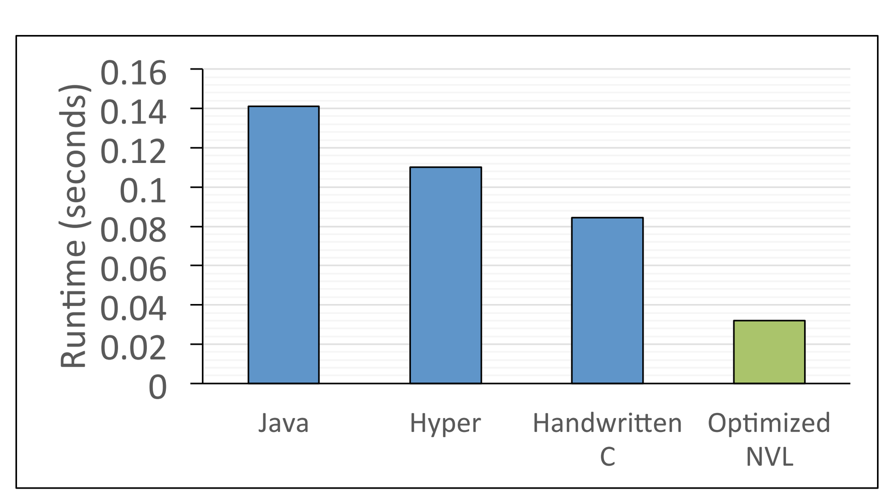
Future Work

- More transformations like loop blocking
- GPU backend
- Joint optimization over pipelined workloads



Preliminary Results

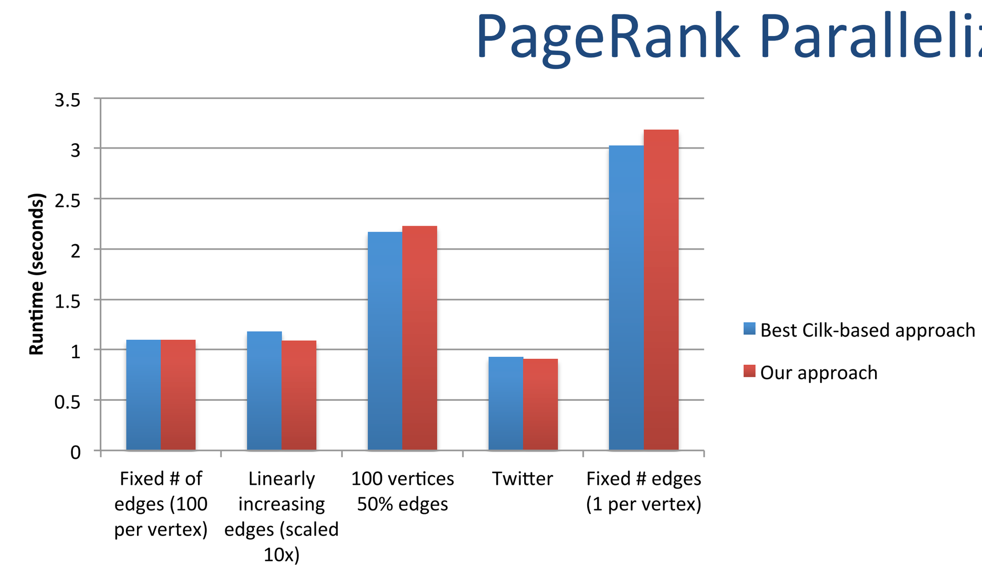
Vectorization, Branch Flattening, Predication



- TPC-H Query 6
- 5GB dataset
- Python implementation: 0.533s
- 2.5x speedup even on simple code!

Ongoing work: which branches shouldn't be vectorized? Based on selectivity of branches, complexity of predicated code.

PageRank Parallelization



- Cilk must be tuned differently for each graph type
- Our approach is competitive without tuning