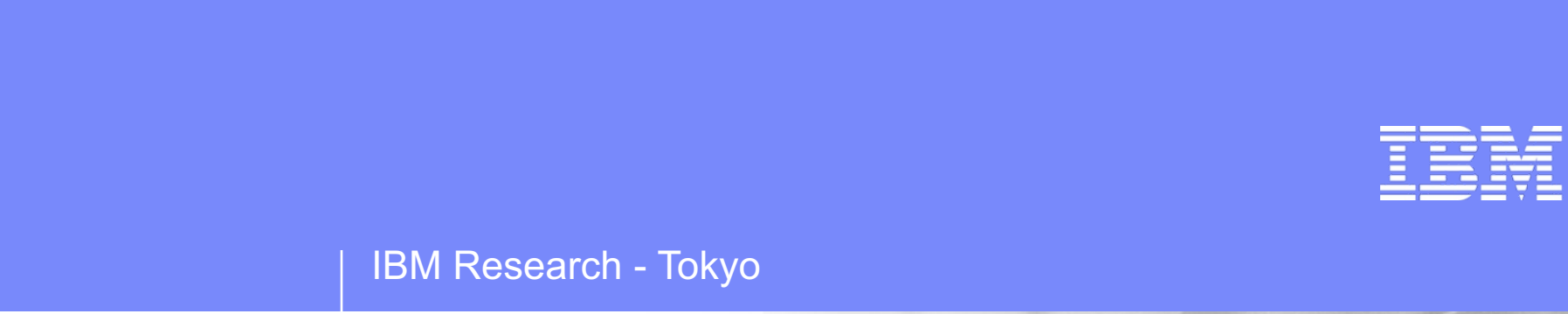


IBM Research - Tokyo

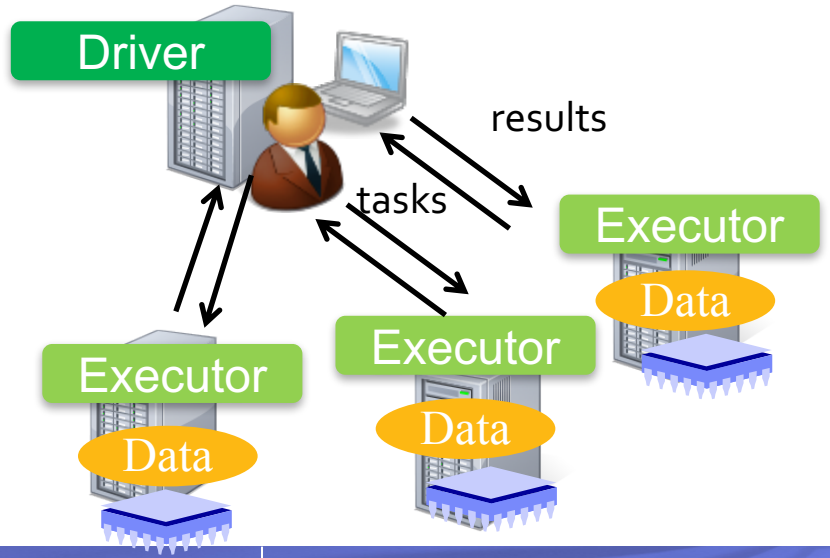
Bring Apache Spark Closer to Accelerators

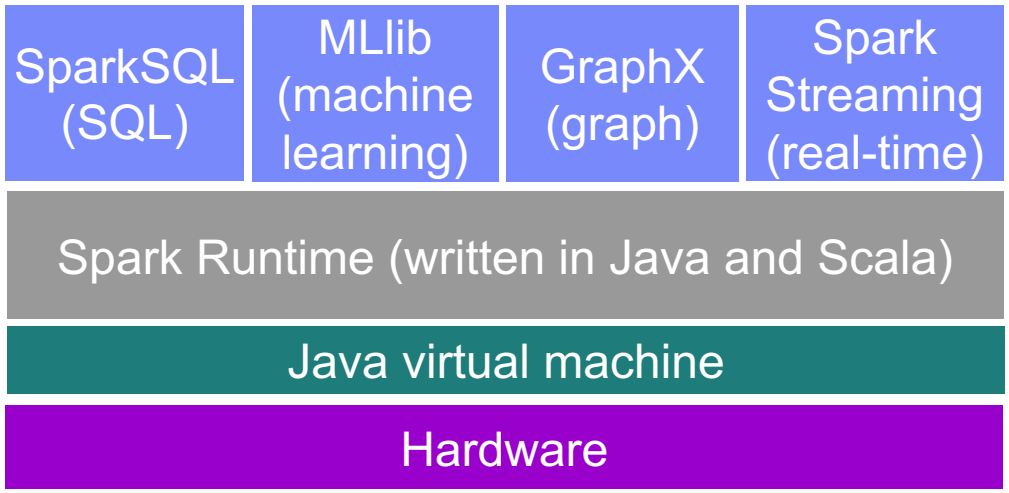
Hiroshi Inoue (IBM Research – Tokyo)
in collaboration with
Kazuaki Ishizaki (IBM Research – Tokyo)
Gita Koblents (IBM Toronto Software Lab)
Jan Wróblewski (University of Warsaw)

Spark is Becoming Popular for Parallel Computing

- Write a Scala/Java/Python program using parallel functions with distributed in-memory data structures on a cluster

```
val dataset = …((x1, y1), (x2, y2), …)…  // input points
val model = KMeans.fit(dataset)          // train k-means model
...
val vecs = model.clusterCenters.map(vec => (vec(0)*2, vec(1)*2))
                                         // x2 to all centers
```

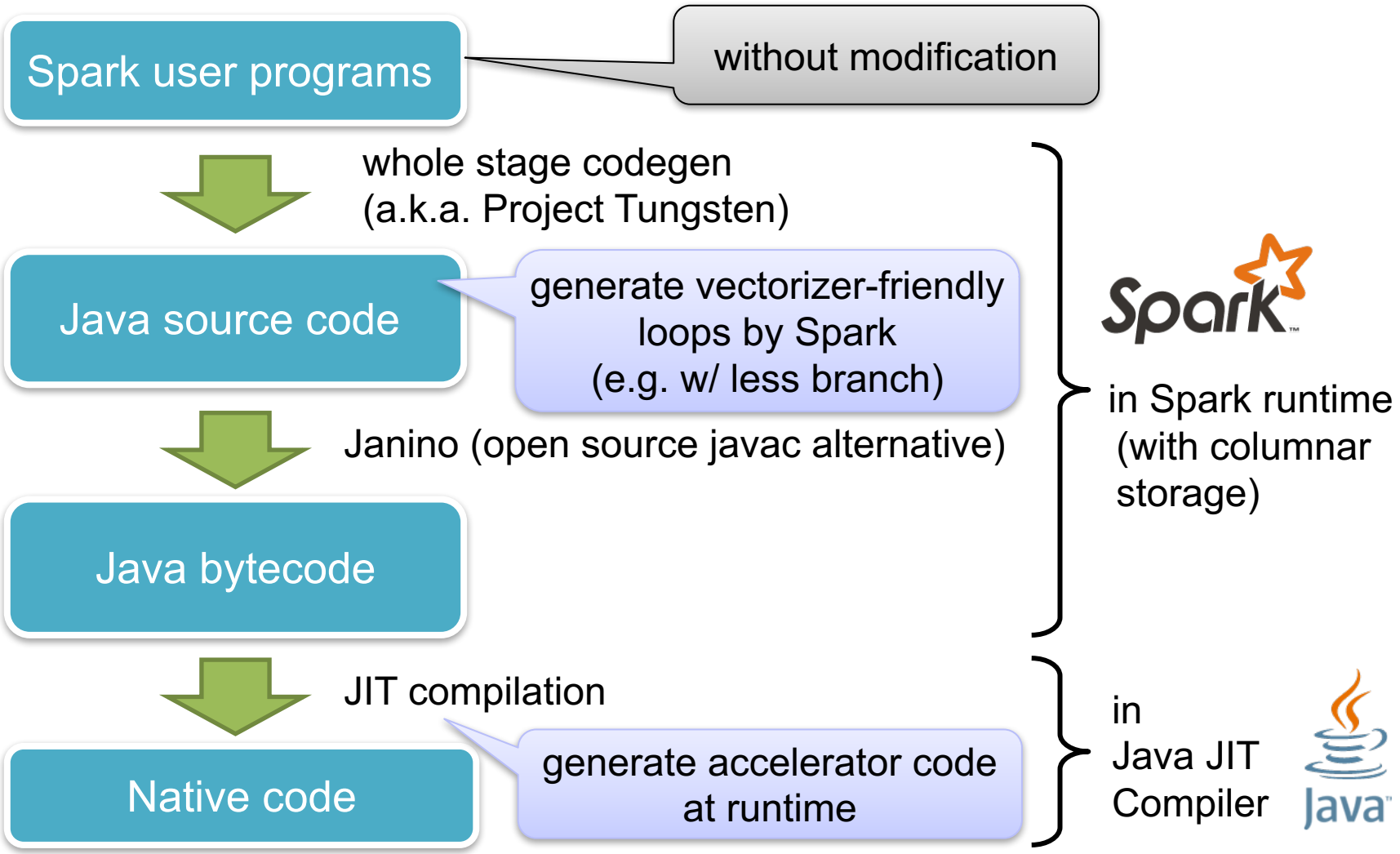




Opportunities and Challenges

- **Spark programs explicitly show data parallelism for distributed execution**
  - ➔We want to exploit accelerators, such as GPU and SIMD instruction of CPUs, based on the same parallelism
- But, **JVM hides details of underlying hardware**
  - We can call optimized native libraries (e.g. written with CUDA), but it is not easy to accelerate user Spark code; we need to generate accelerator code at runtime by JIT compiler

Our approach: end-to-end software stack optimization



Example: Vectorization of simple reduction code

- Even a simple reduction user program, we need to enhance Spark to emit vectorizer-friendly Java code

```
data.selectExpr("sum(value)")
```

- Conditional branches in the loop disturb vectorization

```
int inputadapter_rowIdx = columnar_batchIdx;
    while (inputadapter_rowIdx < columnar_numRows) {
      boolean inputadapter_isNull1 =
inputadapter_col0.isNullAt(inputadapter_rowIdx);
      double inputadapter_value1 = inputadapter_isNull1 ? -1.0 :
(inputadapter_col0.getDouble(inputadapter_rowIdx));

      // do aggregate
      // common sub-expressions

      // evaluate aggregate function
      boolean agg_isNull13 = true;
      double agg_value13 = -1.0;

      boolean agg_isNull14 = agg_bufIsNull1;
      double agg_value14 = agg_bufValue1;
      if (agg_isNull14) {
        boolean agg_isNull16 = false;
        double agg_value16 = -1.0;
        if (!false) {
          agg_value16 = (double) 0;
        }
        if (!agg_isNull16) {
          agg_isNull14 = false;
          agg_value14 = agg_value16;
        }
      }
      boolean agg_isNull18 = inputadapter_isNull1;
      double agg_value18 = -1.0;
      if (!inputadapter_isNull1) {
        agg_value18 = inputadapter_value1;
      }
      if (!agg_isNull18) {
        agg_isNull13 = false;
        agg_value13 = agg_value14 + agg_value18;
      }
      boolean agg_isNull12 = agg_isNull13;
      double agg_value12 = agg_value13;
      if (agg_isNull12) {
        if (!agg_bufIsNull1) {
          agg_isNull12 = false;
          agg_value12 = agg_bufValue1;
        }
      }
      // update aggregation buffer
      agg_bufIsNull1 = agg_isNull12;
      agg_bufValue1 = agg_value12;
      inputadapter_rowIdx++;
      if (shouldStop()) return;
    }
```

Example: Vectorization of simple reduction code

- Even a simple reduction user program, we need to enhance Spark to emit vectorizer-friendly Java code

```
data.selectExpr("sum(value)")
```

- Conditional branches in the loop disturb vectorization

➔ We eliminate conditional branches as much as possible by enhancing Spark code generator

  - **Nullcheck** of input is skipped if scheme assure non-null
  - **Buffer initialization** is moved outside the loop
  - **Output buffer overflow check** is not required for reduction

JIT Compiler Enhancements

- Java JIT cannot reorder floating point arithmetic not to affect the final results as required by language spec.
- But Spark programming model does not guarantee the order of computation due to its inherent nature of parallel and distributed execution
  - ➔ So we can selectively optimize FP operations for Spark
- We put a special annotation for Spark generated Java loop to inform vectorizable loops with floating point arithmetic to JIT compiler

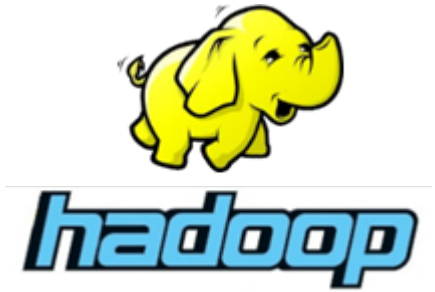
Still we have lots of challenges..

- Example: Overhead of calling user-defined functions (lambda) [1]
  - Problem: A user-defined function takes plain Java objects as input; so Spark need boxing/unboxing to call user-defined functions
  - Our Solution: To analyze and rewrite bytecode sequence of user-defined function (at runtime!) to directly access Spark's internal data representation

[1] Jan Wróblewski, Kazuaki Ishizaki, Hiroshi Inoue and Moriyoshi Ohara, "Accelerating Spark Datasets by inlining deserialization", *IPDPS 2017*

Summary

- Apache Spark is becoming an important infrastructure for bigdata analytics and machine learning tasks
- To fully exploit computing resource based on the data parallelism available in user programs, we need optimization technologies in the software stack including Spark itself and also Java runtime environment



References for more detail of our work

- "Bringing Apache Spark Closer to SIMD and GPU", *Blog post at Spark Technology Center*, Dec. 2016 http://www.spark.tc/simd-and-gpu/
- Jan Wróblewski, Kazuaki Ishizaki, Hiroshi Inoue and Moriyoshi Ohara, "Accelerating Spark Datasets by inlining deserialization", *IPDPS 2017*
- Kazuaki Ishizaki, "Leverage GPU Acceleration for your Program on Apache Spark", *GPU Technology Conference (GTC) 2017*