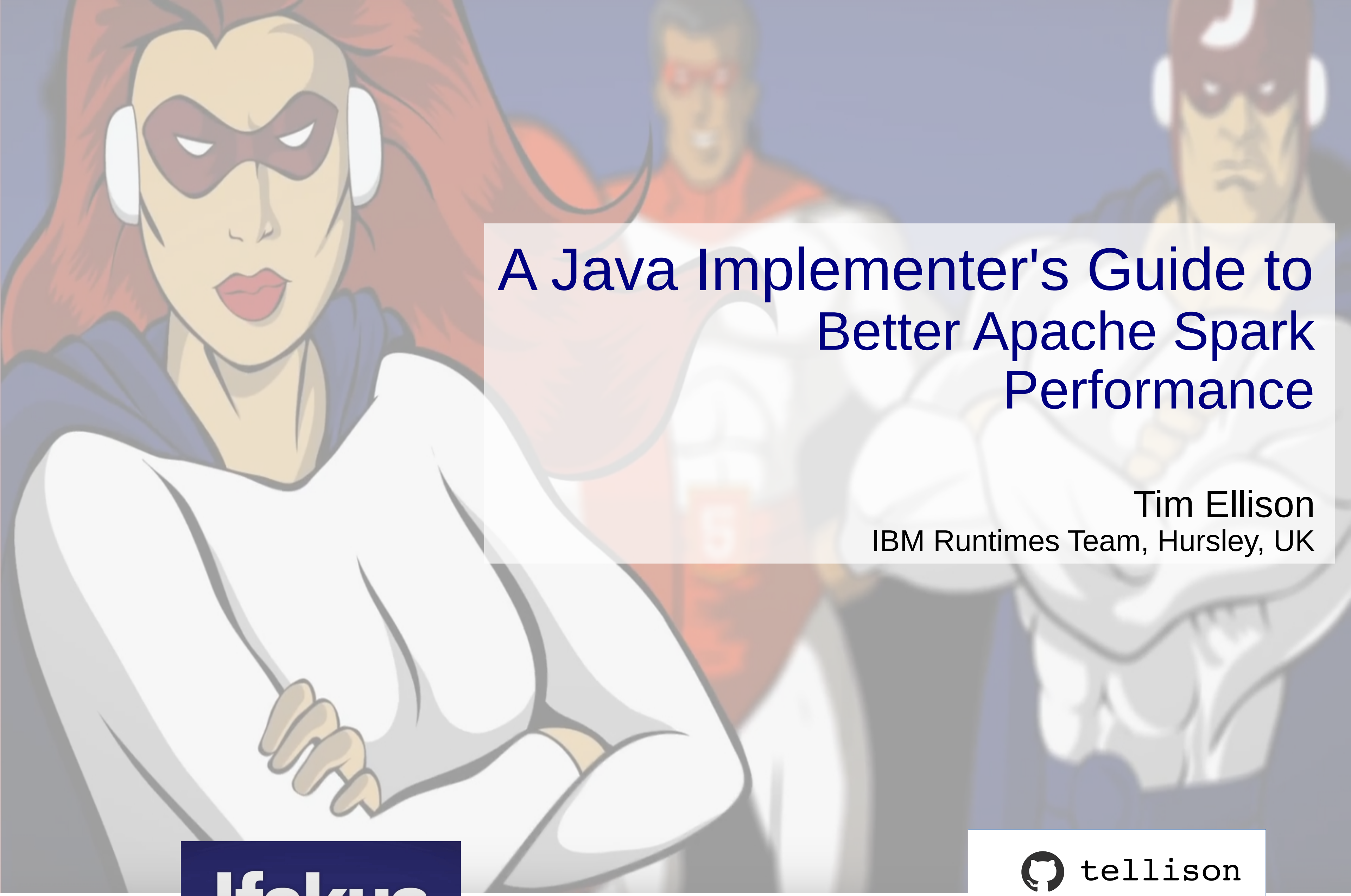


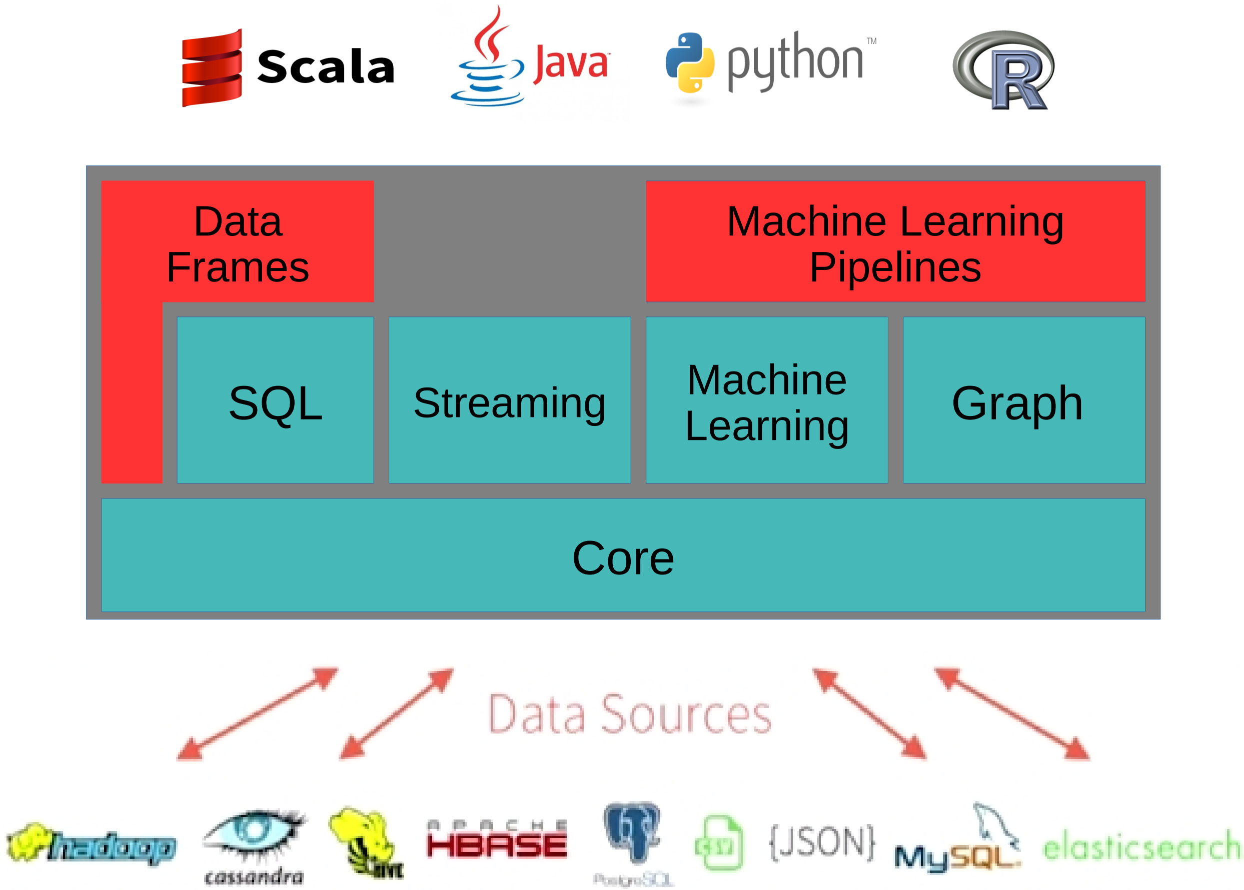
A Java Implementer's Guide to Better Apache Spark Performance

Tim Ellison
IBM Runtimes Team, Hursley, UK

Jfokus

tellison
@tpellison

Apache Spark is a fast, general purpose cluster computing platform


Scala
Java
python
R
Data Frames
Machine Learning Pipelines
SQL
Streaming
Machine Learning
Graph
Core
Data Sources
hadoop
cassandra
HIVE
APACHE HBASE
PostgreSQL
{JSON}
MySQL
elasticsearch.

Apache Spark APIs

- Spark Core
  - Provides APIs for working with raw data collections
  - Map / reduce functions to transform and evaluate the data
  - Filter, aggregation, grouping, joins, sorting

```
text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

- Spark SQL
  - APIs for working with structured and semi-structured data
  - Loads data from a variety of sources (DB2, JSON, Parquet, etc)
  - Provides SQL interface to external tools (JDBC/ODBC)

```
context.jsonFile("s3n://...")
  .registerTempTable("json")
results = context.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

- Spark Streaming
  - Discretized streams of data arriving over time
  - Fault tolerant and long running tasks
  - Integrates with batch processing of data

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

- Machine Learning (MLlib)
  - Efficient, iterative algorithms across distributed datasets
  - Focus on parallel algorithms that run well on clusters
  - Relatively low-level (e.g. K-means, alternating least squares)

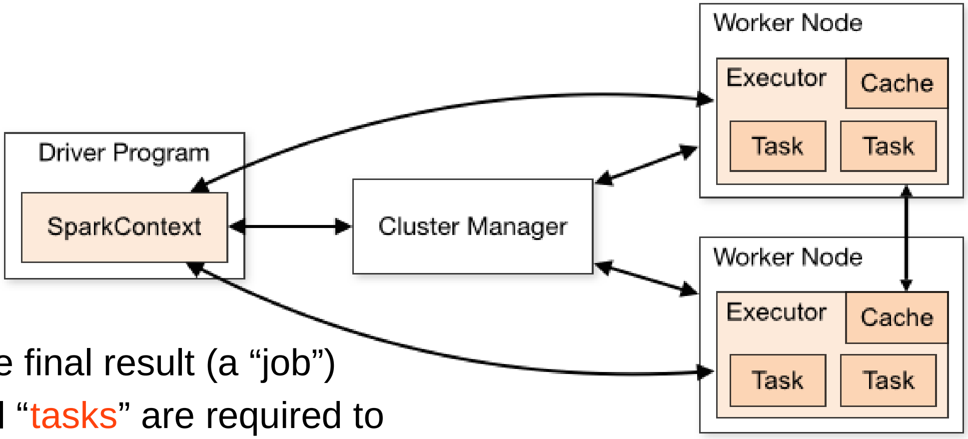
```
points = spark.textFile("hdfs://...")
              .map(parsePoint)

model = KMeans.train(points, k=10)
```

- Graph Computation (GraphX)
  - View the same data as graph or collection-based
  - Transform and join graphs to manipulate data sets
  - PageRank, Label propagation, strongly connected, triangle count, ...

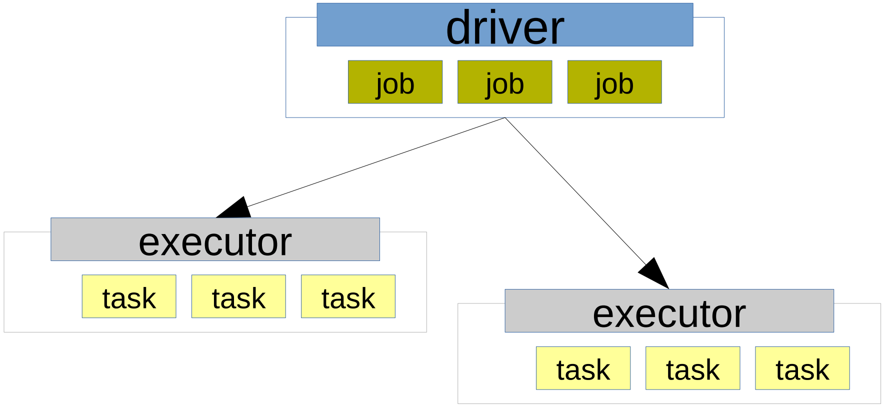
```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

Cluster Computing Platform



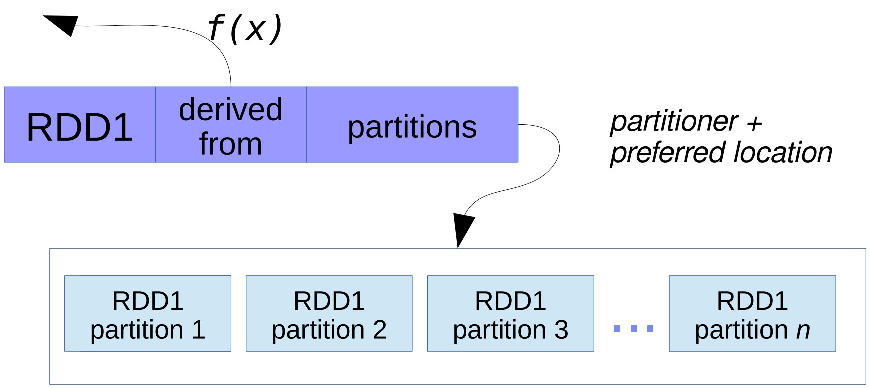
- Master Node "the driver"
  Evaluates user operations
  - Creates a physical execution plan to obtain the final result (a "job")
  - Works backwards to determine what individual "tasks" are required to produce the answer
  - Optimizes the required tasks using pipelining for parallelizable tasks, reusing intermediate results, including persisting temporary states, etc ("stages of the job")
  - Distributes work out to worker nodes
  - Tracks the location of data and tasks
  - Deals with errant workers

- Worker Nodes "the executors" in a cluster
  Executes tasks
  - Receives a copy of the application code
  - Receives data, or the location of data partitions
  - Performs the required operation
  - Writes output to another input, or storage



Resilient Distributed Dataset

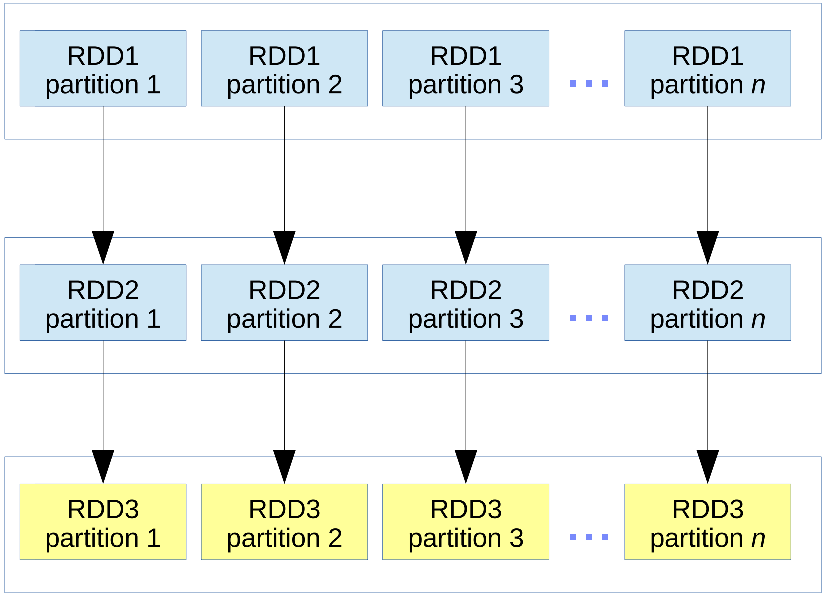
- The Resilient Distributed Dataset (RDD) is the target of program operations
- Conceptually, one large collection of all your data elements – can be huge!
- Can be the original input data, or intermediate results from other operations
- In the Spark implementation, RDDs are:
  - Further decomposed into partitions
  - Persisted in memory or on disk
  - Fault tolerant
  - Lazily evaluated
  - Have a concept of location optimization



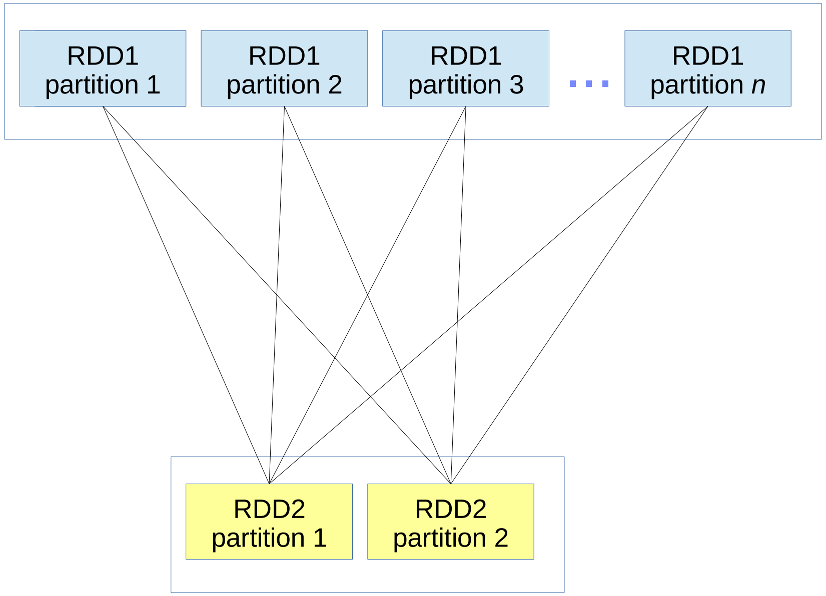
Performance of the Apache Spark Runtime Core

- Moving data blocks
  - How quickly can a worker get the data needed for this task?
  - How quickly can a worker persist the results if required?
- Executing tasks
  - How quickly can a worker sort, compute, transform, … the data in this partition?
  - Can a fast worker work-steal or run speculative tasks?

"Narrow" RDD dependencies e.g. map()
pipeline-able



"Wide" RDD dependencies e.g. reduce()
shuffles



A few things we can do with the JVM to enhance the performance of Apache Spark!

1) JIT compiler enhancements, and writing JIT-friendly code

2) Improving the object serializer

3) Faster IO – networking and storage

4) Offloading tasks to graphics co-processors (GPUs)

JIT compiler enhancements, and writing JIT-friendly code

JNI calls are not free!

```
JNIEXPORT void JNICALL Java_org_xerial_snappy_SnappyNative_arrayCopy
  (JNIEnv * env, jobject self, jobject input, jint offset, jint length, jobject output, jint output_offset)
{
        char* src = (char*) env->GetPrimitiveArrayCritical((jarray) input, 0);
        char* dest = (char*) env->GetPrimitiveArrayCritical((jarray) output, 0);
        if(src == 0 || dest == 0) {
                // out of memory
                if(src != 0) {
                        env->ReleasePrimitiveArrayCritical((jarray) input, src, 0);
                }
                if(dest != 0) {
                        env->ReleasePrimitiveArrayCritical((jarray) output, dest, 0);
                }
                throw_exception(env, self, 4);
                return;
        }

        memcpy(dest+output_offset, src+offset, (size_t) length);

        env->ReleasePrimitiveArrayCritical((jarray) input, src, 0);
        env->ReleasePrimitiveArrayCritical((jarray) output, dest, 0);
}
```

https://github.com/xerial/snappy-java/blob/develop/src/main/java/org/xerial/snappy/SnappyNative.cpp

Style: Using JNI has an impact...

- The cost of calling from Java code to natives and from natives to Java code is significantly higher (maybe 5x longer) than a normal Java method call.
  - The JIT can't in-line native methods.
  - The JIT can't do data flow analysis into JNI calls
    - e.g. it has to assume that all parameters are always used.
  - The JIT has to set up the call stack and parameters for C calling convention,
    - i.e. maybe rearranging items on the stack.

- JNI can introduce additional data copying costs
  - There's no guarantee that you will get a direct pointer to the array / string with `Get<type>ArrayElements()`, even when using the `GetPrimitiveArrayCritical` versions.
  - The IBM JVM will always return a copy (to allow GC to continue).

- *Tip:*
  - *JNI natives are more expensive than plain Java calls.*
  - e.g. create an unsafe based Snappy-like package written in Java code so that JNI cost is eliminated.

Style: Use JIT optimizations to reduce overhead of logging checks

- Spark's logging calls are gated on the checks of a static boolean value

```
// Log methods that take only a String
protected def logInfo(msg: => String) {
  if (log.isInfoEnabled) log.info(msg)
}
```

trait Logging

Spark

```
} else {
  logInfo(
    s"TID $taskAttemptId waiting for at least 1/2N of shuffle memory pool to be free")
  memoryManager.wait()
}
```

- *Tip: Check for the non-null value of a static field ref to instance of a logging class singleton*
  – e.g.

```
// Log methods that take only a String
protected def logInfo(msg: => String) {
  if (infoLogger != null) infoLogger.log(msg)
}
```

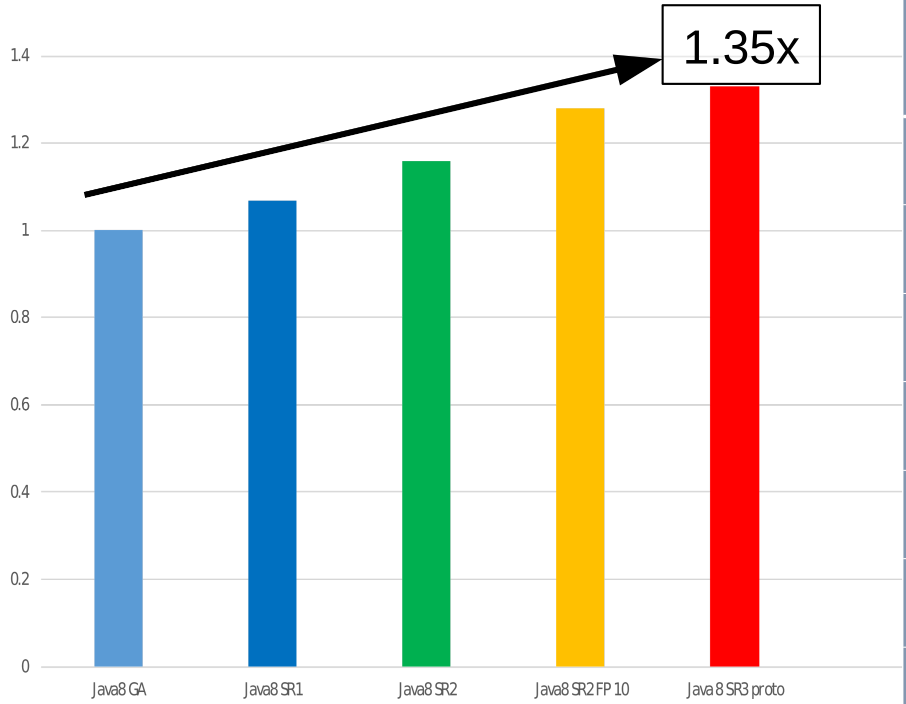
  – Uses the JIT's speculative optimization to avoid the explicit test for logging being enabled; instead it ...
    1) Generates an internal JIT runtime assumption (e.g. `InfoLogger.class` is undefined),
    2) NOPs the test for trace enablement
    3) Uses a class initialization hook for the `InfoLogger.class` (already necessary for instantiating the class)
    4) The JIT will regenerate the test code if the class event is fired

Style: Judicious use of polymorphism

- Spark has a number of highly polymorphic interface call sites and high fan-in (several calling contexts invoking the same callee method) in map, reduce, filter, flatMap, ...
  - e.g. `ExternalSorter.insertAll` is very hot (drains an iterator using `hasNext/next` calls)
- Pattern #1:
  - `InterruptibleIterator` → Scala's `mapIterator` → Scala's `filterIterator` → …
- Pattern #2:
  - `InterruptibleIterator` → Scala's `filterIterator` → Scala's `mapIterator` → …
- The JIT can only choose one pattern to in-line!
  - Makes JIT devirtualization and speculation more risky; using profiling information from a different context could lead to incorrect devirtualization.
  - More conservative speculation, or good phase change detection and recovery are needed in the JIT compiler to avoid getting it wrong.
- Lambdas and functions as arguments, by definition, introduce different code flow targets
  - Passing in widely implemented interfaces produce many different bytecode sequences
  - When we in-line we have to put runtime checks ahead of in-lined method bodies to make sure we are going to run the right method!
  - Often specialized classes are used only in a very limited number of places, but the majority of the code does not use these classes and pays a heavy penalty
  - e.g. Scala's attempt to specialize Tuple2 Int argument does more harm than good!
- *Tip: Use polymorphism sparingly, use the same order / patterns for nested & wrappered code, and keep call sites homogeneous.*

Effect of Adjusting JIT heuristics for Apache Spark

Improvements in successive IBM Java 8 releases



1/Geometric mean of HiBench time on zLinux 32 cores, 25G heap

Performance compared with OpenJDK 8

	IBM JDK8 SR3 (tuned)	IBM JDK8 SR3 (out of the box)
PageRank	160%	148%
Sleep	101%	113%
Sort	103%	147%
WordCount	130%	146%
Bayes	100%	91%
Terasort	157%	131%
Geometric mean	**121%**	**116%**

HiBench huge, Spark 1.5.2, Linux Power8 12 core * 8-way SMT

Replacing the object serializer

Writing a Spark-friendly object serializer

- Spark has a plug-in architecture for flattening objects to storage
  - Typically uses general purpose serializers, e.g. Java serializer, or Kryo, etc.
- Can we optimize for Spark usage?
  - Goal: Reduce time time to flatten objects
  - Goal: Reduce size of flattened objects
- Expanding the list of specialist serialized form
  - Having custom write/read object methods allows for reduced time in reflection and smaller on-wire payloads.
  - Types such as `Tuple` and `Some` given special treatment in the serializer
- Sharing object representation within the serialized stream to reduce payload
  - But may be defeated if `supportsRelocationOfSerializedObjects` required
- Reduce the payload size further using variable length encoding of primitive types.
  - All objects are eventually decomposed into primitives

Writing a Spark-friendly object serializer

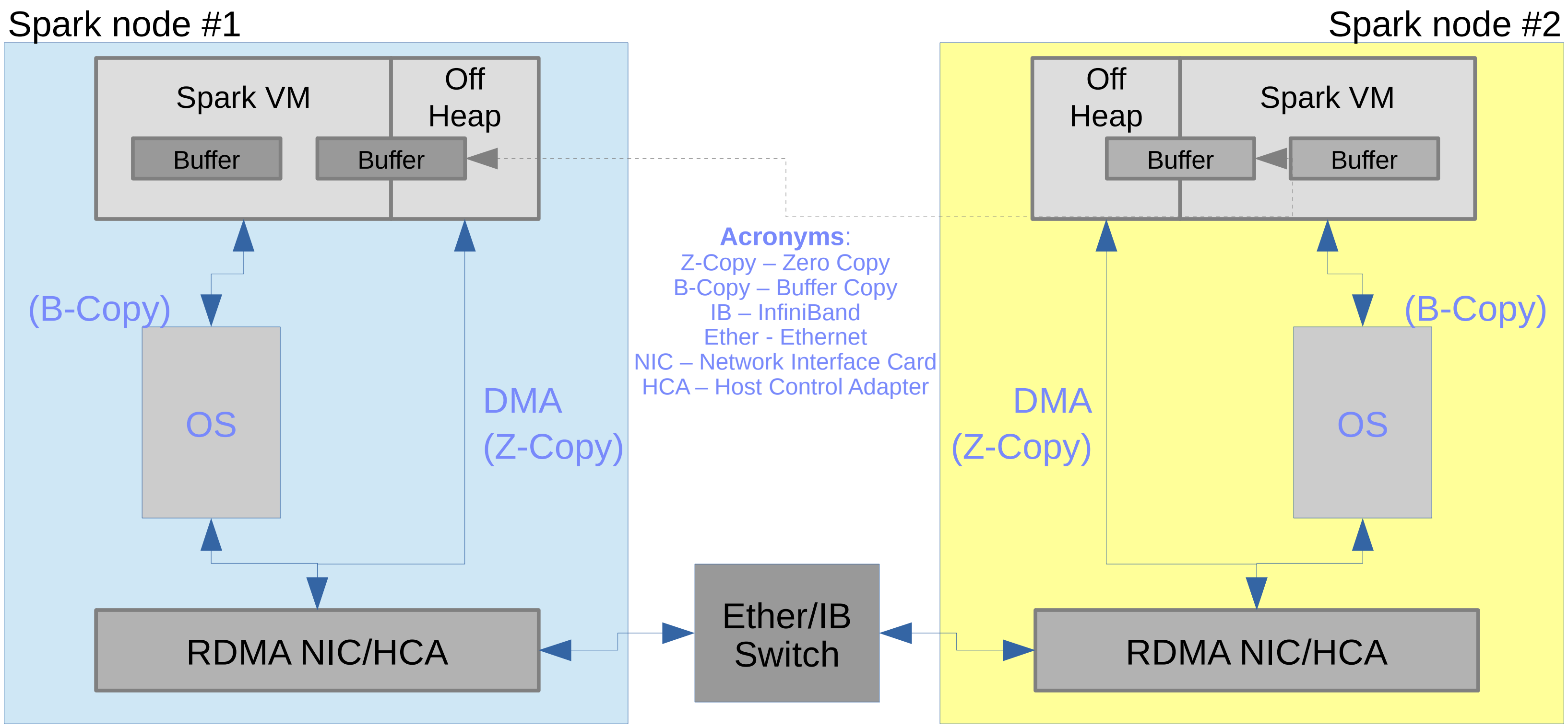
- Adaptive stack-based recursive serialization vs. state machine serialization
  - Use the stack to track state wherever possible, but fall back to state machine for deeply nested objects (e.g. big RDDs)

- Special replacement of deserialization calls to avoid stack-walking to find class loader context
  - Optimization in JIT to circumvent some regular calls to more efficient versions

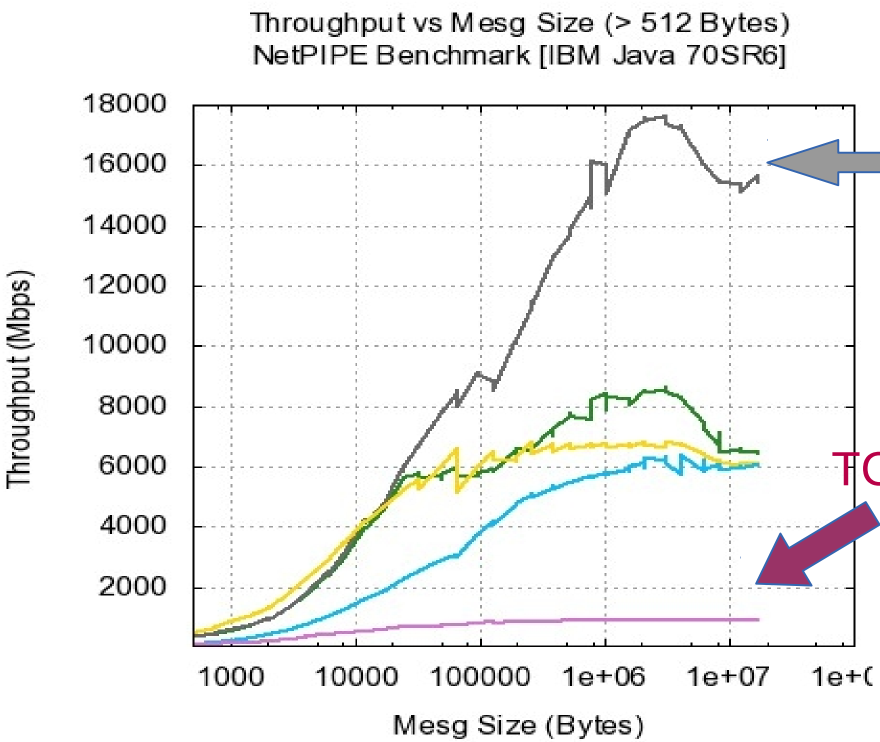
- *Tip: These are opaque to the application, no special patterns required.*
- *Results: Variable, small numbers of percentages at best*

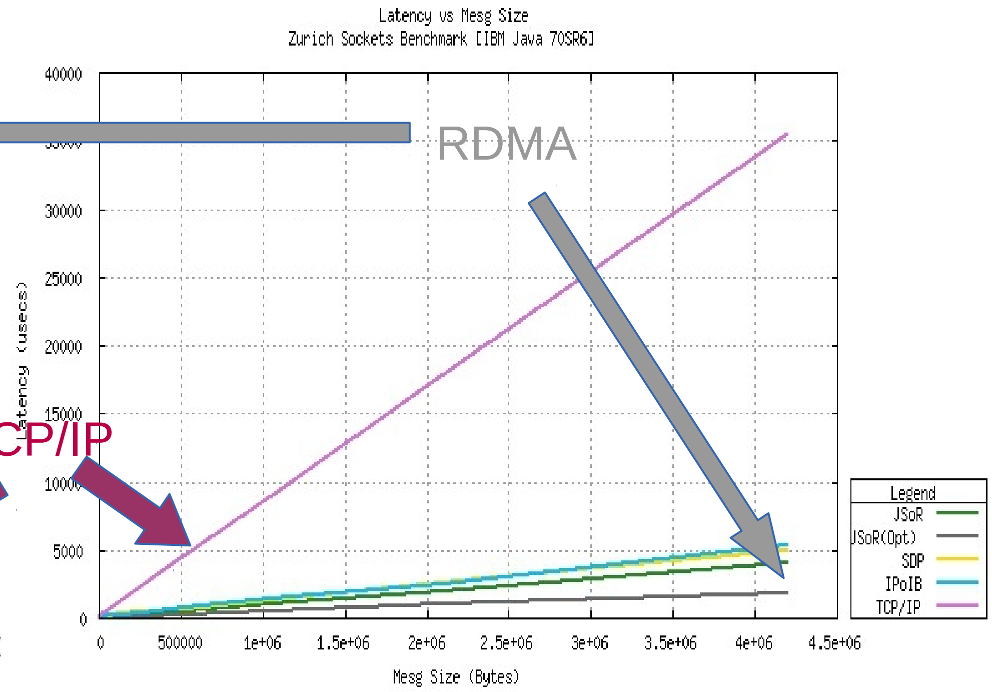
Faster IO – networking and storage

Remote Direct Memory Access (RDMA) Networking



- Low-latency, high-throughput networking
  - Direct 'application to application' memory pointer exchange between remote hosts
  - Off-load network processing to RDMA NIC/HCA – OS/Kernel Bypass (zero-copy)
  - Introduces new IO characteristics that can influence the Spark transfer plan





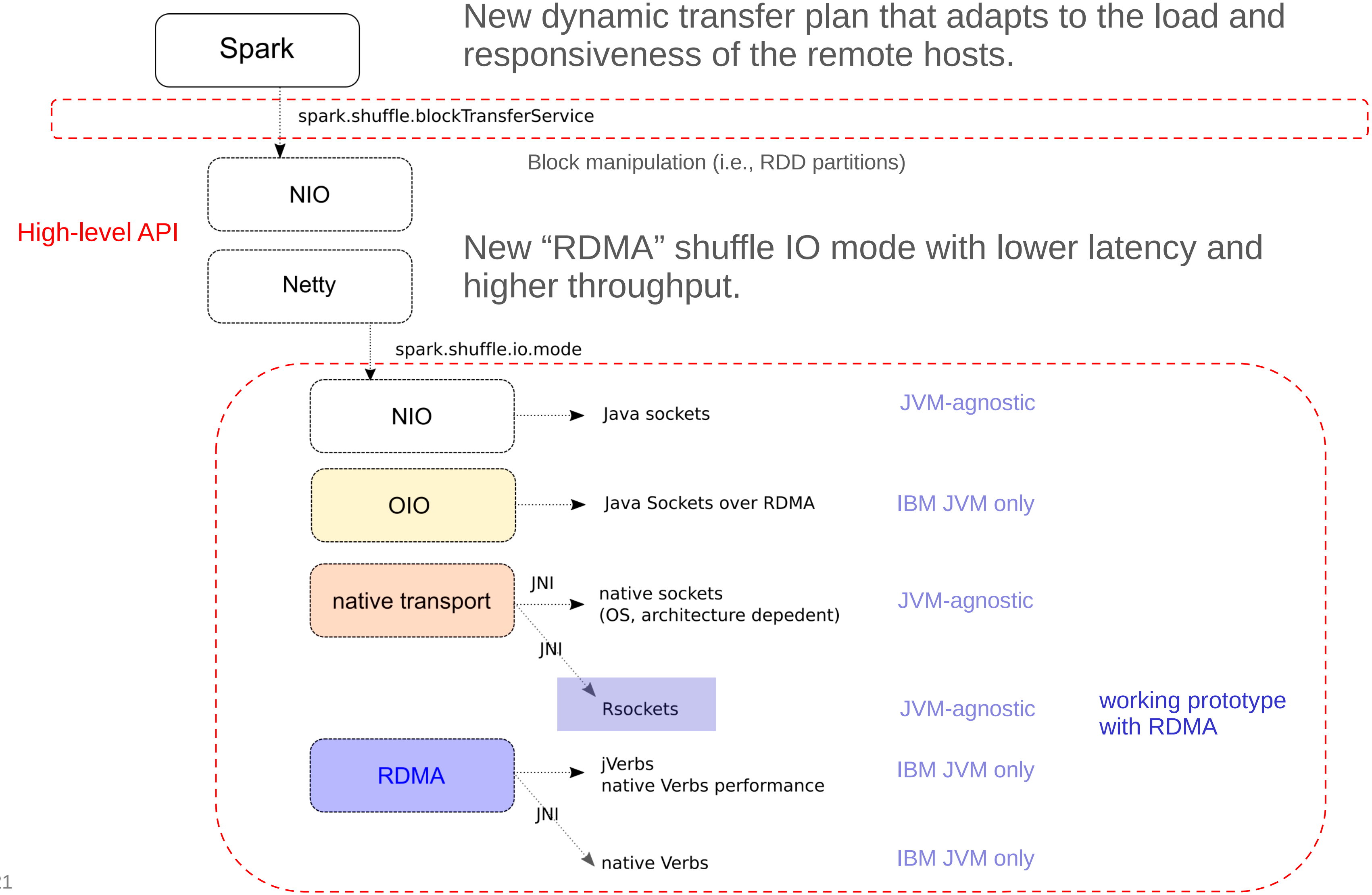
RDMA exhibits improved throughput and reduced latency.

Available over java.net.Socket APIs or explicit jVerbs calls

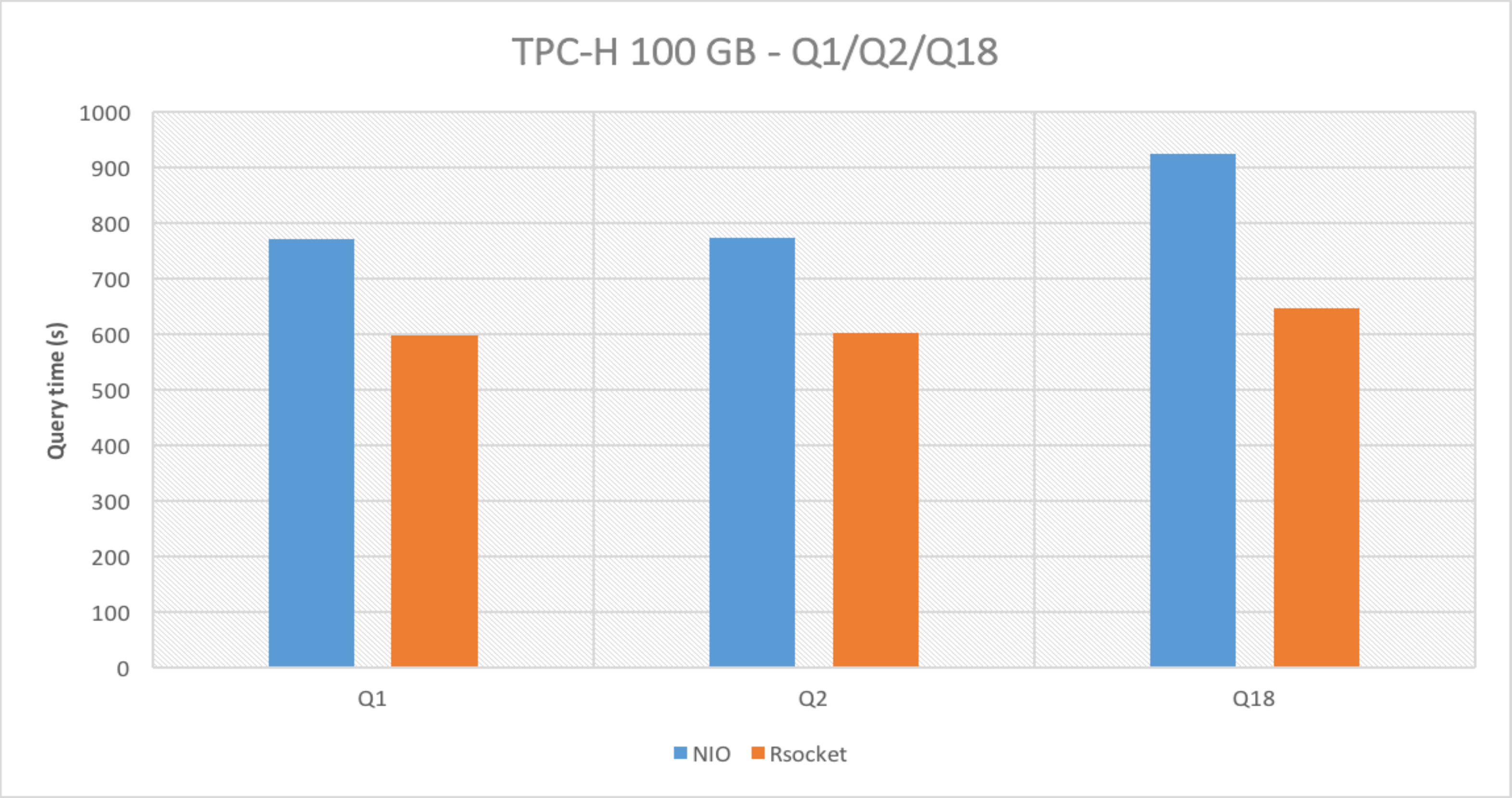
Faster network IO with RDMA-enabled Spark

New dynamic transfer plan that adapts to the load and responsiveness of the remote hosts.

New “RDMA” shuffle IO mode with lower latency and higher throughput.

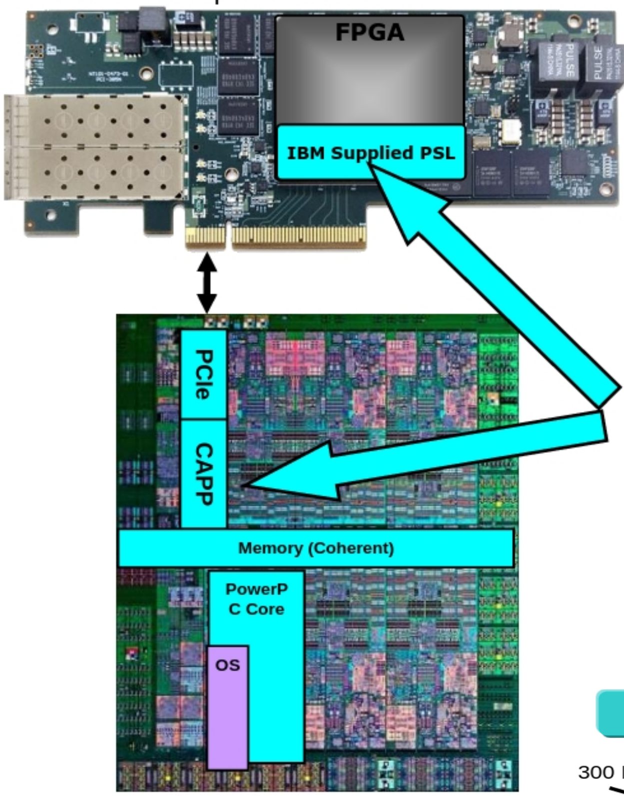


Shuffling data shows 30% better response time and lower CPU utilization



Faster storage with POWER CAPI/Flash

CAPI Developer Kit Card

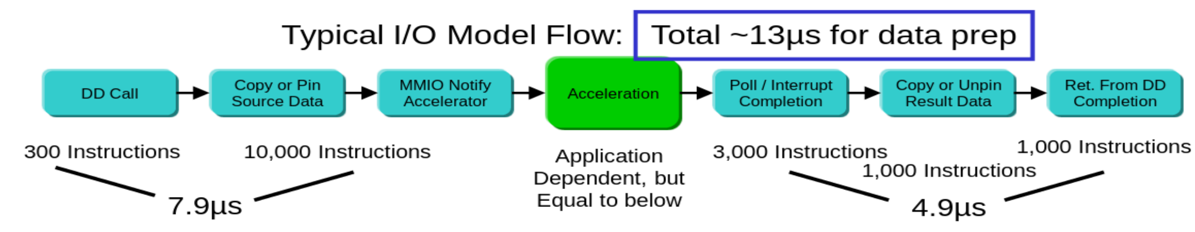


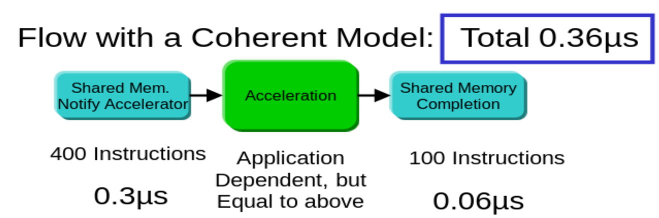
POWER8 Processor

- POWER8 architecture offers a 40Tb Flash drive attached via Coherent Accelerator Processor Interface (CAPI)
  – Provides simple coherent block IO APIs
  – No file system overhead

- **Power Service Layer (PSL)**
  – Performs Address Translations
  – Maintains Cache
  – Simple, but powerful interface to the Accelerator unit

- **Coherent Accelerator Processor Proxy (CAPP)**
  – Maintains directory of cache lines held by Accelerator
  – Snoops PowerBus on behalf of Accelerator





Proprietary Hardware to enable CAPI

Operating System Enablement
Ubuntu LE Kernel Extensions
libcxl function calls

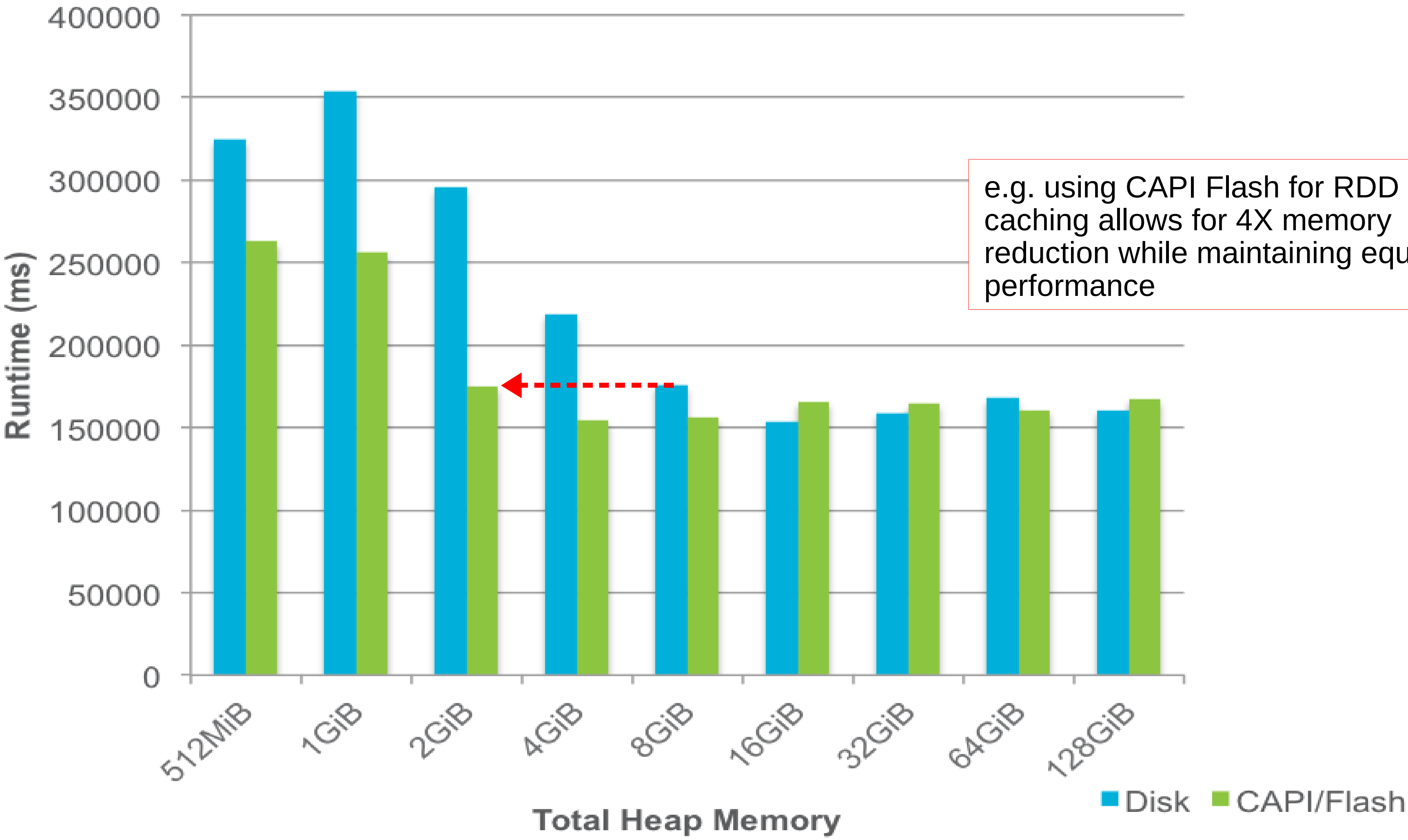
Faster disk IO with CAPI/Flash-enabled Spark

- When under memory pressure, Spark spills RDDs to disk.
  - Happens in `ExternalAppendOnlyMap` and `ExternalSorter`
- We have modified Spark to spill to the high-bandwidth, coherently-attached Flash device instead.
  - Replacement for `DiskBlockManager`
  - New `FlashBlockManager` handles spill to/from flash
- Making this pluggable requires some further abstraction in Spark:
  - Spill code assumes using disks, and depends on `DiskBlockManger`
  - We are spilling without using a file system layer
- Dramatically improves performance of executors under memory pressure.
- Allows to reach similar performance with much less memory (denser deployments).

Power8 + CAPI

IBM Flash System 840

x Degrees of Separation on Spark

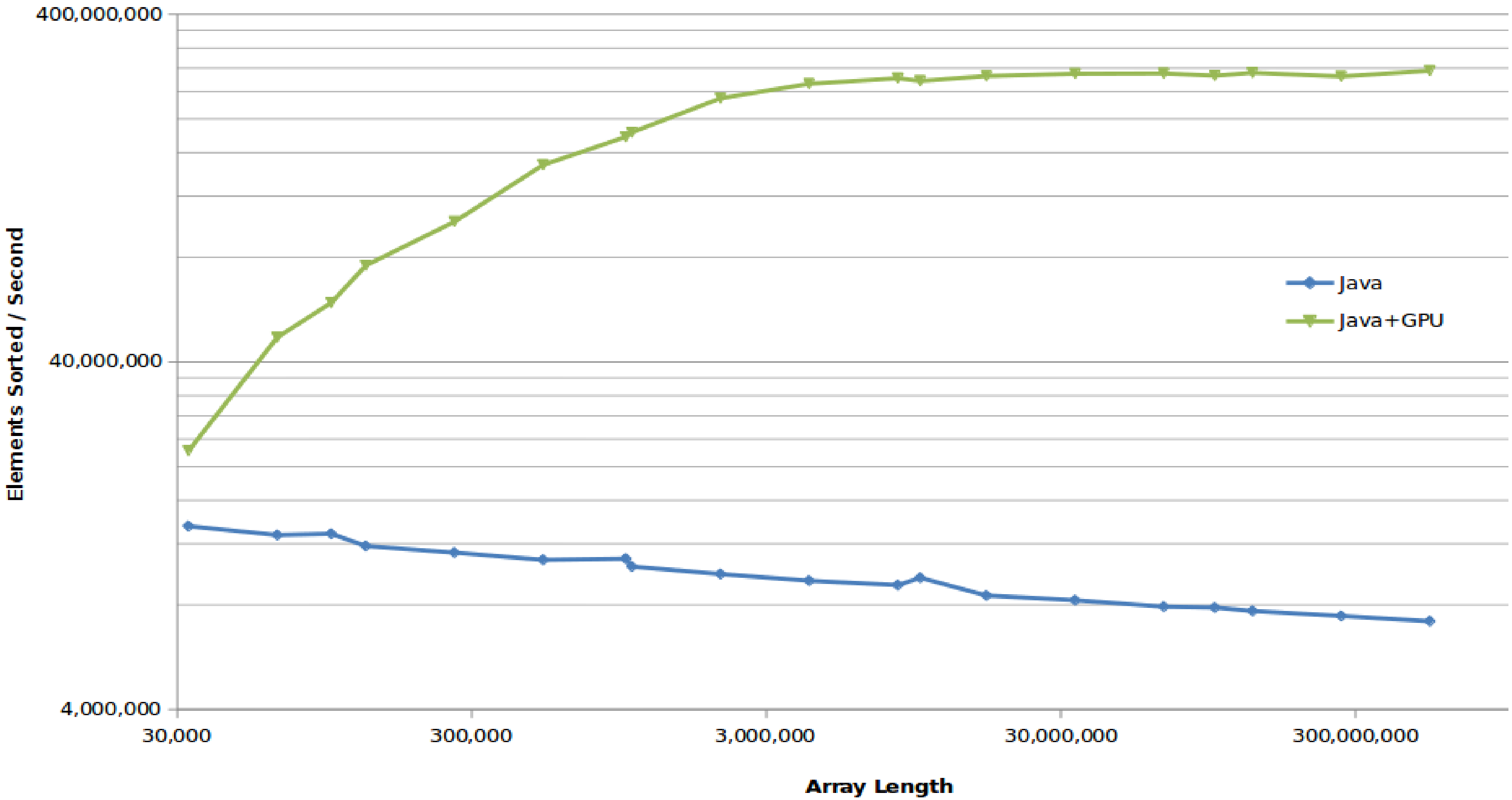


e.g. using CAPI Flash for RDD caching allows for 4X memory reduction while maintaining equal performance

Offloading tasks to graphics co-processors

GPU-enabled array sort method

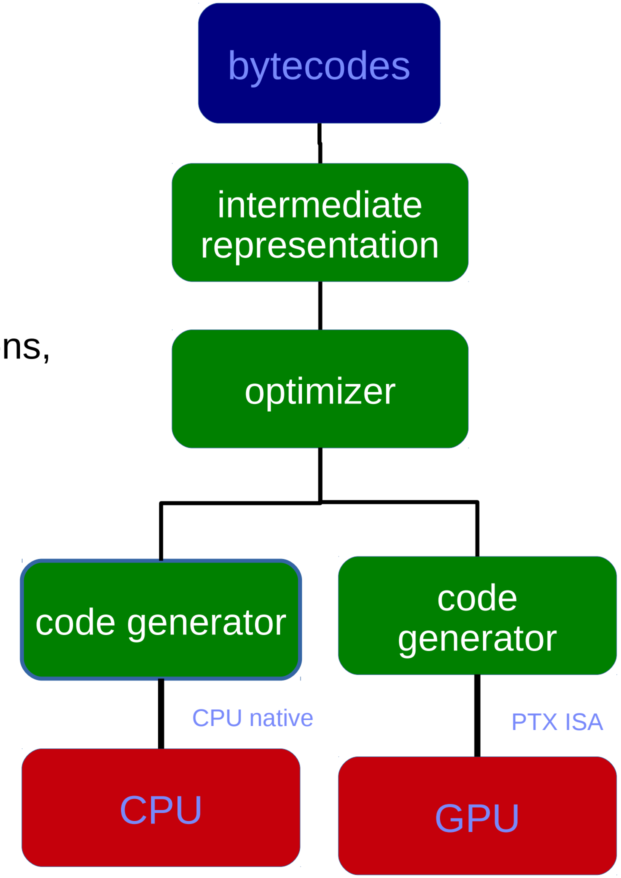
- Some `Arrays.sort()` methods will offload work to GPUs today
  - e.g. sorting large arrays of `ints`



IBM Power 8 with Nvidia K40m GPU

JIT optimized GPU acceleration

- As the JIT compiles a stream expression we can identify candidates for GPU off-loading
  - Arrays copied to and from the device implicitly
  - Java operations mapped to GPU kernel operations
  - Preserves the standard Java syntax and semantics



- Comes with caveats
  - Recognize a limited set of operations within the lambda expressions,
    - notably no object references maintained on GPU
  - Default grid dimensions and operating parameters for the GPU workload
  - Redundant/pessimistic data transfer between host and device
    - Not using GPU shared memory
  - Limited heuristics about when to invoke the GPU and when to generate CPU instructions

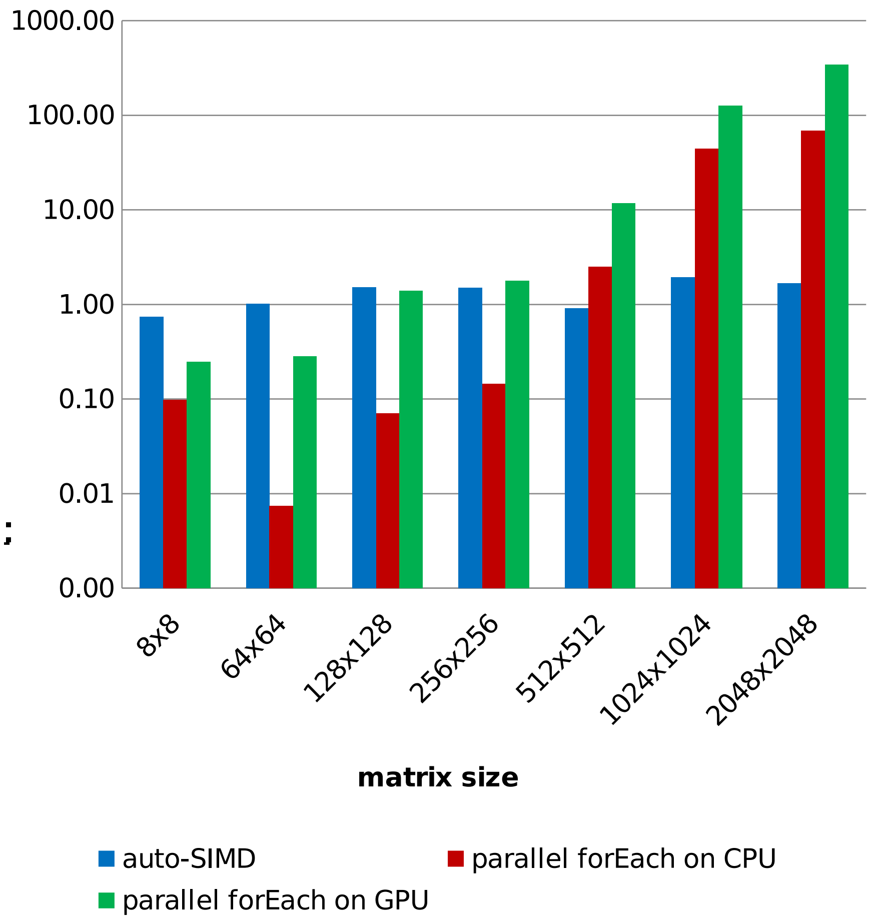
GPU optimization of Lambda expressions

The JIT can recognize parallel stream code, and automatically compile down to the GPU.

```
public void multiply() {
  IntStream.range(0, COLS*COLS).parallel().forEach(
    id -> {
      int i = id / COLS;
      int j = id % COLS;
      double sum = 0;

      for (int k = 0; k < COLS; k++) {
        sum += input1[i*COLS + k] * input2[k*COLS + j];
      }
      output[id] = sum;
    });
}
```

Speed-up factor when run on a GPU enabled host



IBM Power 8 with Nvidia K40m GPU

Moving high-level algorithms onto the GPU

UniProt

DrugBank
Open Data Drug & Drug Target Database

BJC British Journal of Cancer

Unified Medical Language System®

Chemical Similarity

Drug1	Drug2	Sim
Salsalate	Aspirin	.9
Dicoumarol	Warfarin	.76

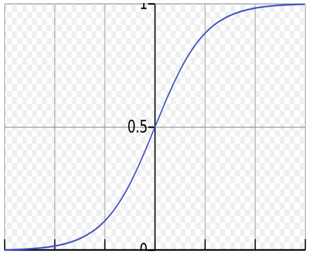
Drug1	Drug2	Sim
Salsalate	Aspirin	.7
Dicoumarol	Warfarin	.6

Interactions

Drug1	Drug2
Aspirin	Gliclazide
Aspirin	Dicoumarol

Drug1	Drug2
Aspirin	Probenecid
Aspirin	Azilsartan

Logistic Regression Model

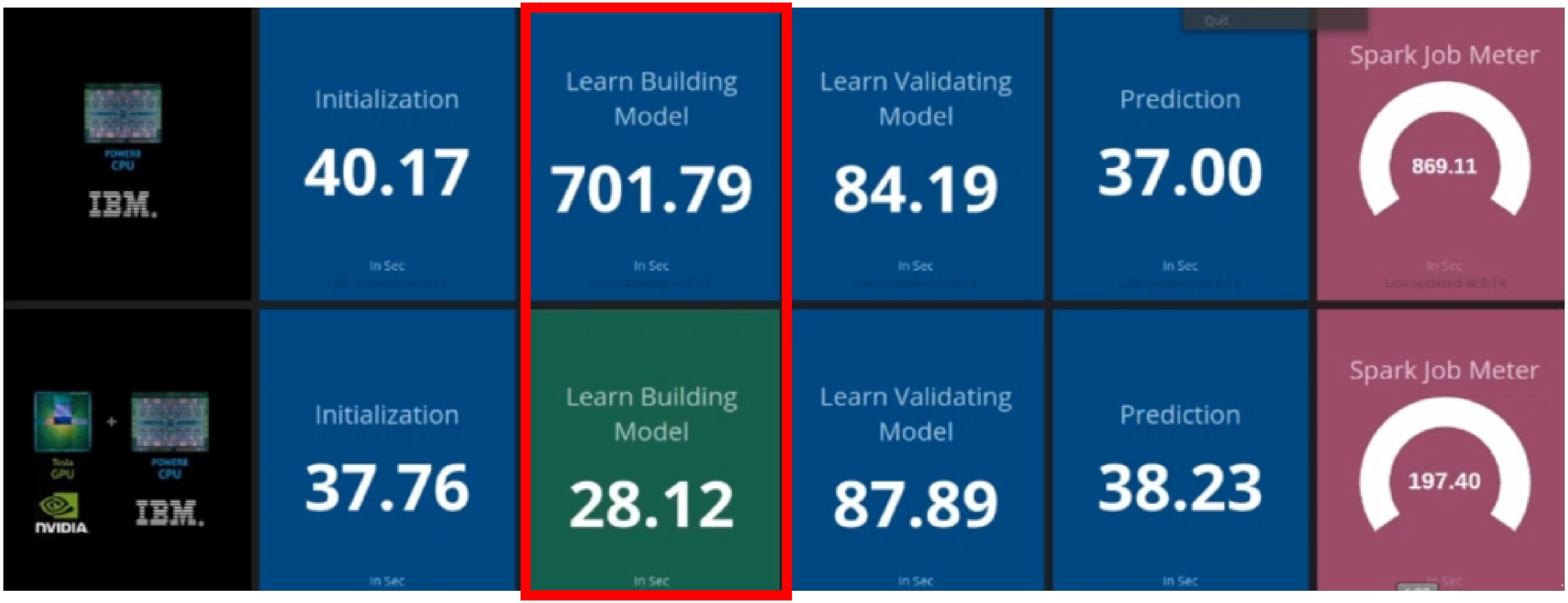


Drug1	Drug2	Best Sim1*Sim1	Best SimN*SimN
Salsalate	Gliclazide	.9*1	.7*1
Salsalate	Warfarin	.9*.76	.7*.6

Interactions Prediction

Drug1	Drug2	Prediction
Salsalate	Gliclazide	0.85
Salsalate	Warfarin	0.7

Drug1	Drug2	Prediction
Salsalate	Gliclazide	0.53
Salsalate	Warfarin	0.32



- ***25X Speed up*** for Building Model stage (replacing Spark Mllib Logistic Regression)
- Transparent to the Spark application, but requires changes to Spark itself

Summary

- We are focused on Core runtime performance to get a multiplier up the Spark stack.
  - More efficient code, more efficient memory usage/spilling, more efficient serialization & networking, etc.

- There are hardware and software technologies we can bring to the party.
  - We can tune the stack from hardware to high level structures for running Spark.

- Spark and Scala developers can help themselves by their style of coding.

- All the changes are being made in the Java runtime or being pushed out to the Spark community.

Lightning-Fast Cluster Computing

- There is lots more stuff I don't have time to talk about, like GC optimizations, object layout, monitoring VM/Spark events, hardware compression, security, etc. etc.
  - mailto: tellison@apache.org

http://ibm.biz/spark-kit