HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Machine Learning Algorithms

Peilong Li, Yan Luo, Yu Cao, Ning Zhang

University of Massachusetts Lowell

Spark SUMMIT EAST

Background

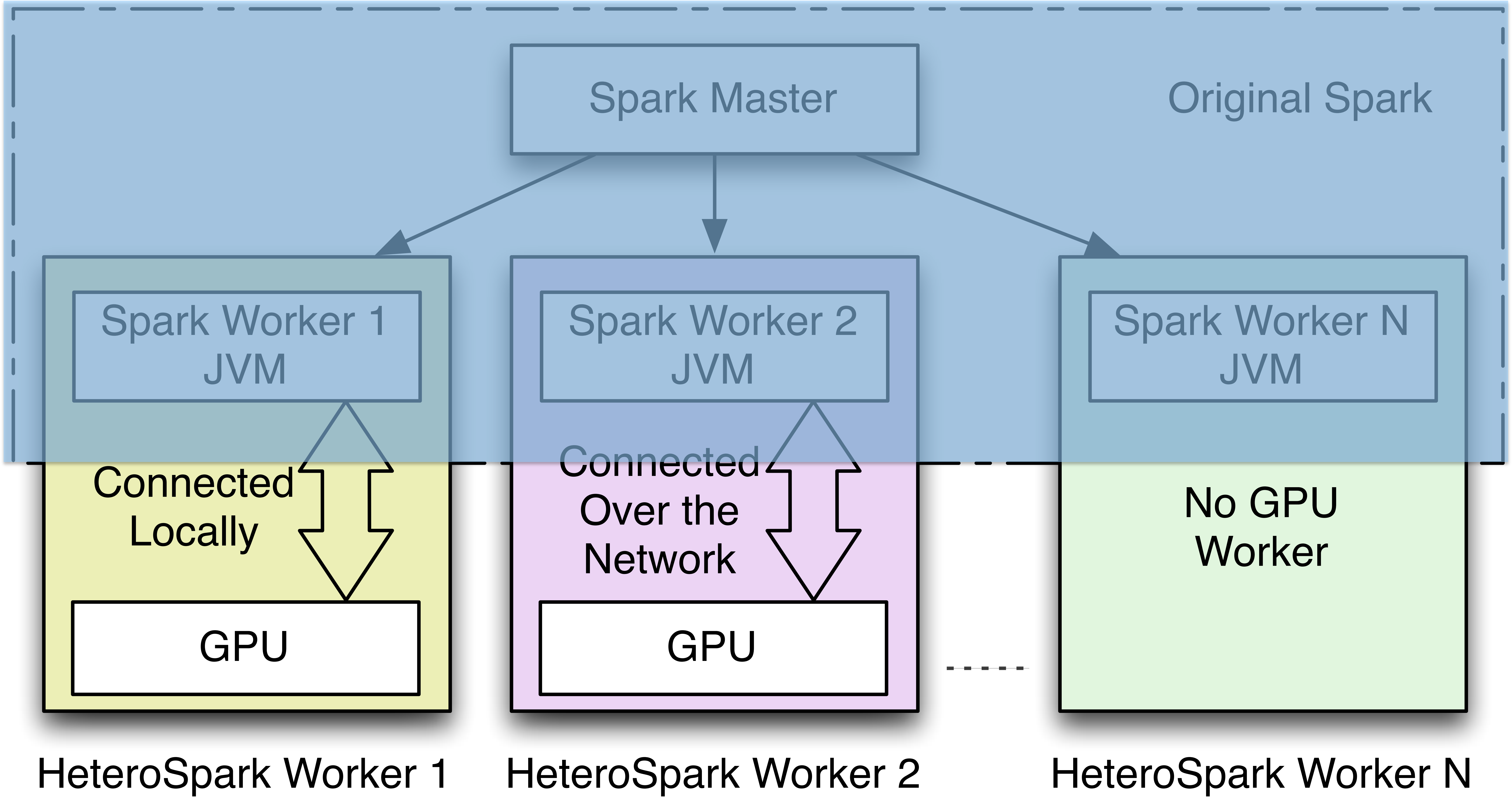
- GPU outperforms CPU in a broad area of applications:
  - Machine learning, image processing, bioinformatics, etc.
- Pros and cons of current solutions:

Current Solutions	Pros	Cons
Single GPU	Good data parallelism	Difficult to handle large scale dataset due to memory size
GPU Cluster	Good data parallelism	Complexity in data partitioning (MPI, OpenMP)
CPU Cluster	Scalability, programmability	Single node performance due to limited number of cores

Motivations

- **Acceleration**: Integrate GPU accelerators into current Spark platform to achieve further data parallelism and algorithm acceleration.
- **Plug-n-play**: “Plugin” style design – current Spark applications can choose to enable/disable GPU acceleration.
- **Portability**: Existing Spark code can be easily ported to the heterogeneous platform.

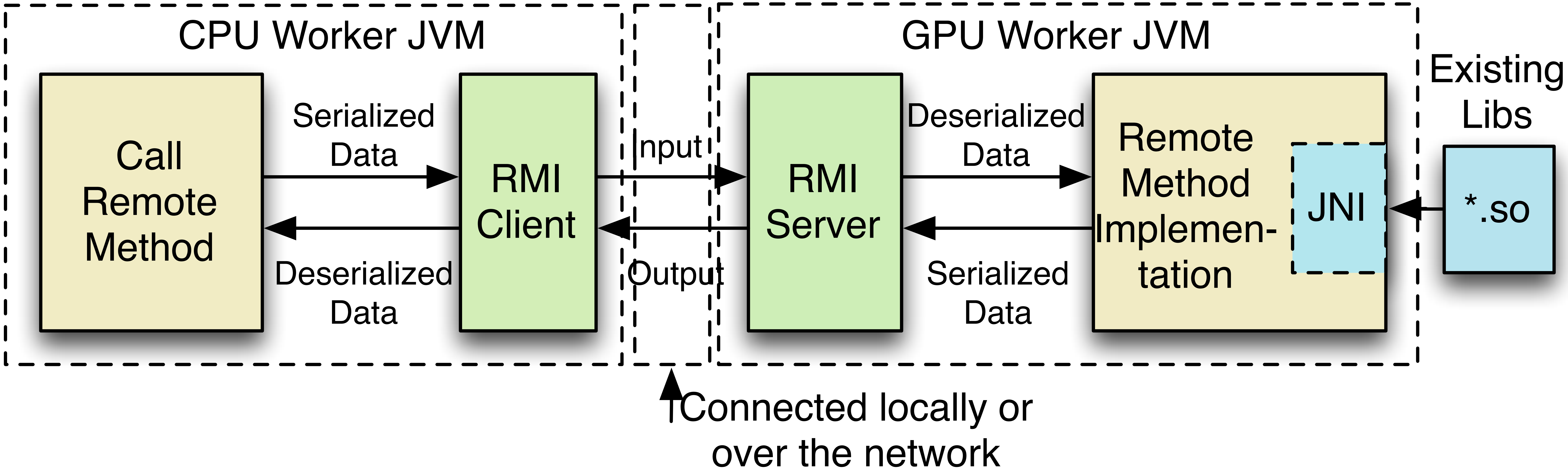
HeteroSpark Architecture



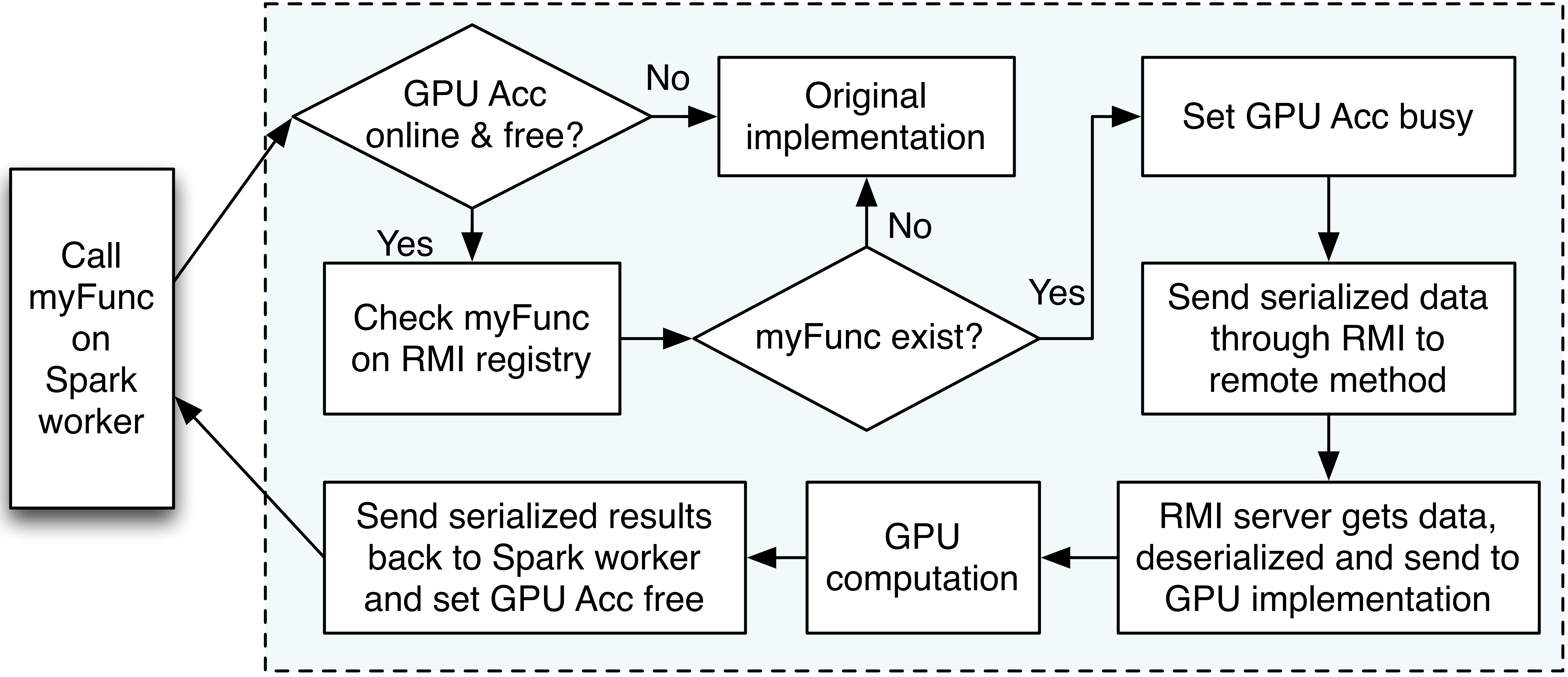
- Enable/disable GPU accelerators in configuration file
- Three configs:
  - Local GPU
  - Remote GPU
  - No GPU

CPU-GPU Communication

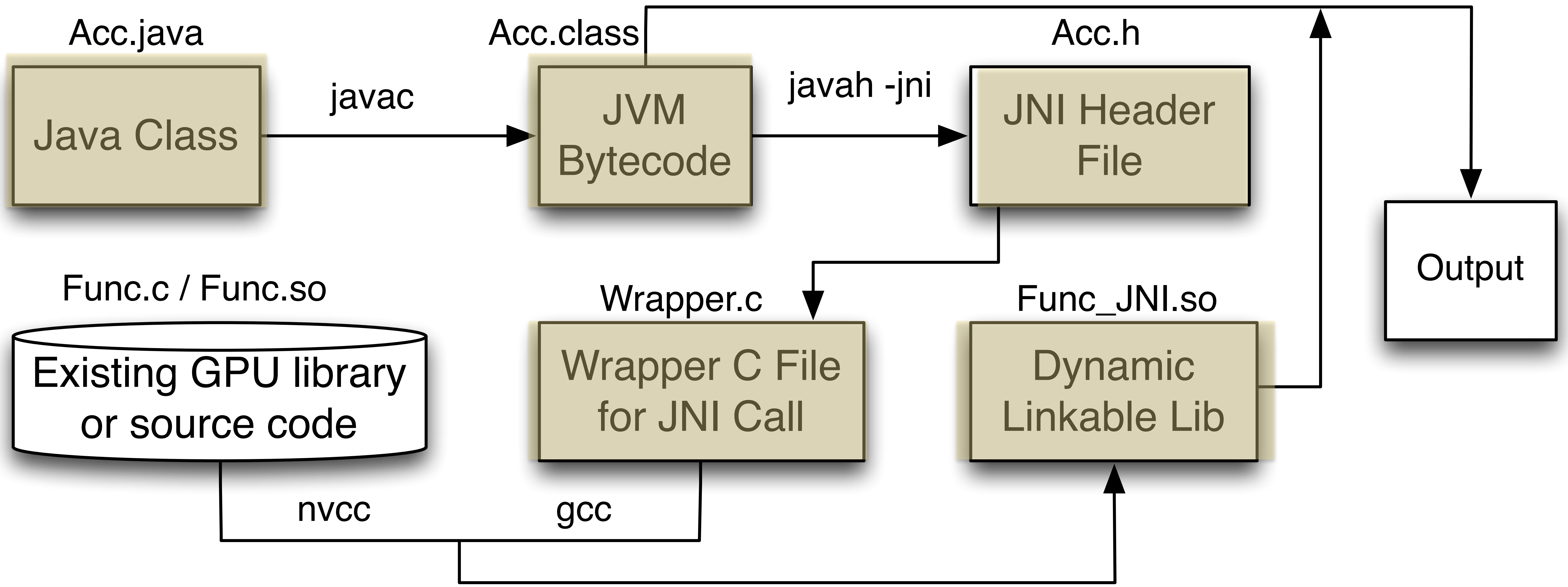
- Java Remote Method Invocation: methods of remote Java objects can be invoked from other Java virtual machines (on different hosts)
- RMI uses object serialization to marshal and unmarshal parameters



HeteroSpark Glue Logic


Call myFunc on Spark worker
GPU Acc online & free?
No
Original implementation
Set GPU Acc busy
Yes
Check myFunc on RMI registry
No
myFunc exist?
Yes
Send serialized data through RMI to remote method
Send serialized results back to Spark worker and set GPU Acc free
GPU computation
RMI server gets data, deserialized and send to GPU implementation

GPU Accelerator Development



- Accelerate your application:
  - ➢ Use HeteroSpark existing libraries.
  - ➢ Develop HeteroSpark accelerator libs.

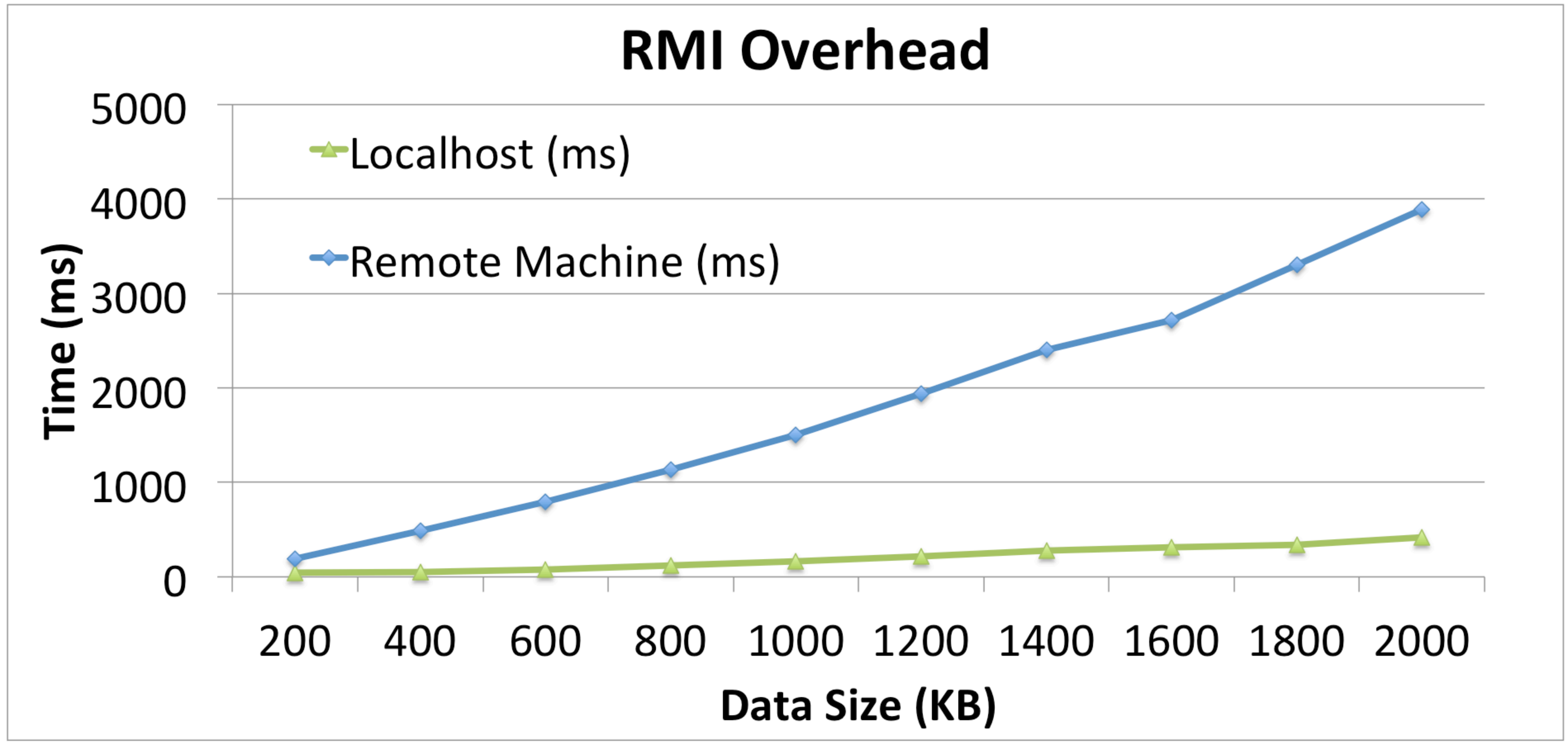
```
File sharedLibrary = new File("Func_JNI.so");
System.load(sharedLibrary.getAbsolutePath());
…
public native int func(float[] para1, int para2 …);
…
int retArray = obj.func(para1, para2…);
```

```
public native int func(float[] para1, int para2 …);
…
int retArray = obj.func(para1, para2);
```

```
// Read input parameters
jdouble *para1 = (*env)->GetFloatArrayElements(env, array1, 0);
func(para1, para2); // Call the native function which is implemented on GPU
// Return data back to Java env
…
(*env)->ReleaseFloatArrayElements(env, array1, para1, 0);
```

Why RMI?

Solutions	Pros	Cons
RMI	Secure, faster, lightweight	Java-specific
COBRA	Language independent	No GC supported
SOAP	XML-based web service	Heavy and slow
Spring+JMS	Message queuing; simple programming interface	Spring framework learning curve, extra dependency

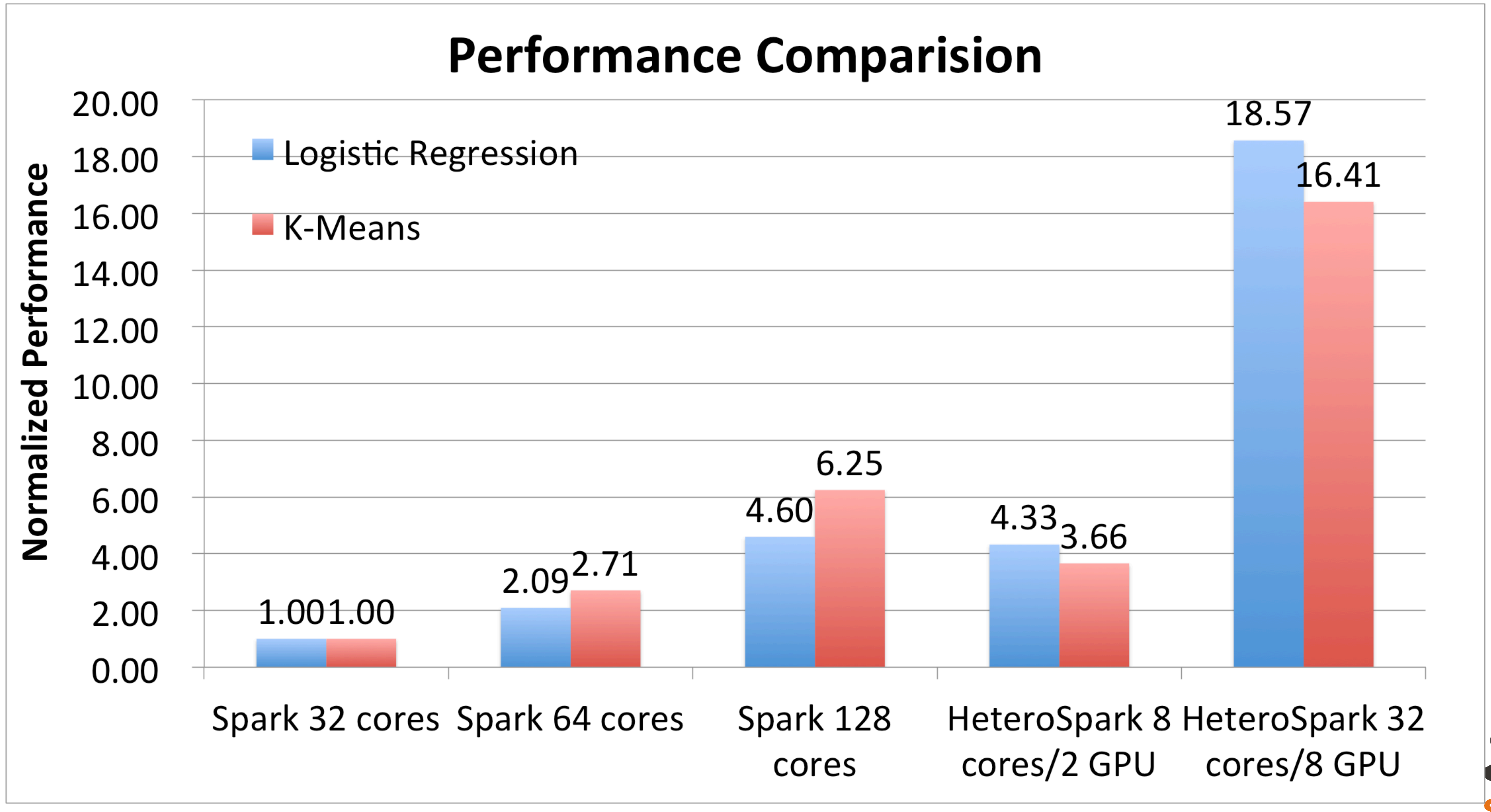


- RMI overhead is < 500 ms with 2MB of data if connected locally.
- In most cases, we will use local GPU which is connected via PICe.

Performance Evaluation

- Benchmark applications:
  - ➢ Logistic regression
    - ➢ Dataset: Criteo (click prediction task), training 11.15 GB , test 1.46 GB
  - ➢ K-Means
    - ➢ Dataset: MNIST-8M, (handwritten digits), 8.1 M data, 24.8 GB

- System setup:
  - ➢ CPU: EC2 m3.xlarge nodes
  - ➢ GPU: EC2 g2.2xlarge



Conclusion

- **Acceleration**: HeteroSpark enhances Spark by accelerating machine learning algorithms and reducing CPU resources.
- **Plug-n-play**: zero interference with original application if choose to “mute” acceleration.
- **Portability**: non-tedious work to port existing Spark application into HeteroSpark (if using the maintained libraries).

Future Work

- **Serialization Overhead**: Utilize faster serialization technique for communication.
- **Simplified Interface**: Use Spring framework to simplify remote method innovation interface.
- **Spawning the Library**: Integrating more machine learning libraries into HeteroSpark, esp. deep learning algorithms.

Thank you!