

Computation Offloading in JVM-based Dataflow Engines

Haralampos Gavriilidis¹

Abstract: State-of-the-art dataflow engines, such as Apache Spark and Apache Flink scale out on large clusters for a variety of data-processing tasks, including machine learning and data mining algorithms. However, being based on the JVM, they are unable to apply optimizations supported by modern CPUs. On the contrary, specialized data processing frameworks scale up by exploiting modern CPU characteristics. The goal of this thesis is to find the sweet spot between scale-out and scale-up systems by offloading computation from dataflow engines to specialized systems. We propose two computation offloading methods, reason about their applicability, and implement a prototype based on Apache Spark. Our evaluation shows that for compute-intensive tasks, computation offloading leads to performance improvements of up to a factor of 2.5x. For certain UDF scenarios, computation offloading performs worse by up to a factor of 3x: our microbenchmarks show that 80% of the time is spent on serialization operations. By employing data exchange without serialization, computation offloading achieves performance improvements by up to 10x.

Keywords: dataflow engines; computation offloading; data exchange; native execution

1 Introduction

Dataflow engines are a popular tool for large-scale data processing, due to their user-friendly programming interfaces and their ability to scale-out on clusters. Through their numerous application programming interfaces (APIs), dataflow engines provide domain-specific abstractions for a variety of data processing tasks, such as relational processing, graph processing, and machine learning. Dataflow programs consist of chained data processing operators, while custom functionality is added by user-defined functions (UDFs). State-of-the-art dataflow engines, such as Apache Spark [Za12] and Apache Flink [Al14] provide their standard APIs in JVM-based programming languages, e.g. Scala and Java, since they are built on the JVM. JVM-based languages are a convenient choice for users, because of their higher-level abstractions, but at the same time are incapable of applying lower-level optimizations supported by modern CPUs [Cr15; Es18]. Additionally, in the case of big data applications, the garbage collection of a large number of objects causes execution stalls [Ng16]. On the contrary, specialized processing frameworks, such as database systems [Ne11] and numerical libraries are built using lower-level programming languages, and therefore provide better support for CPU-specific optimizations. To enhance dataflow

¹ Technische Universität Berlin, Database Systems and Information Management Group, Einsteinufer 17, 10587 Berlin harry_g@mailbox.tu-berlin.de

engines with native performance, we propose to offload computations from the JVM to native runtimes. We describe the applicability of computation offloading in current systems, and discuss the challenges of such an integration. More specifically, we make the following contributions:

- We propose two methods for computation offloading in state-of-the-art dataflow engines and describe a prototype implementation on Apache Spark.
- We evaluate our approaches with microbenchmarks and end-to-end benchmarks using several types of UDFs.

2 Background

In this section we give a brief overview of dataflow engines and the JVM memory model.

2.1 Dataflow Engines

In the following we describe the main characteristics of state-of-the-art dataflow engines, such as Apache Spark [Za12] and Apache Flink [Al14].

Programming Model. Dataflow engines employ distributed collection processing, introduced by the MapReduce paradigm. Users implement dataflow programs by chaining second-order functions, such as `map` and `reduce`. Custom functionality is added by UDFs. Higher-level APIs offer domain-specific operators, e.g. for relational processing, graph processing and machine learning. Each operator describes a transformation to the dataset. The composed operator tree forms a directed graph, used for optimization and execution.

Architecture and Execution Model. Dataflow engines use the master/worker model in a shared-nothing environment. Each worker node runs a JVM instance, which is used to execute processing tasks. Workers receive processing instructions and execute them independently. Intermediate computation results are exchanged between workers through their network interface. During execution, workers store and process data in memory, and move it to the disk only when it exceeds the memory size. The discussed dataflow engines implement the iterator model, where every physical operator is executed element-wise for all data records. For failed computations, dataflow engines use recovery mechanisms, such as lineage graphs and snapshots.

2.2 JVM Heap and Off-Heap

The JVM memory is organized around objects. New objects are created on the Heap, which is periodically scanned by the garbage collector. The garbage collector organizes objects based on their age and cleans unreferenced objects. Next to the Heap, JVM applications have access to the Off-Heap. Binary data residing on the Off-Heap is accessed as in lower-level

languages, using memory pointers. Since the Off-Heap is not managed by the JVM, it is not subject to garbage collection. However, it is necessary to implement manual memory management within applications to avoid JVM crashes.

3 Computation Offloading in Dataflow Engines

In this section, we describe our computation offloading approach by the example of UDF processing. First, we analyze the current UDF processing method in dataflow engines and discuss potential bottlenecks. Second, we discuss two mechanisms for data exchange between a worker JVM process and a native process. Finally, we describe the implementation of our computation offloading prototype in Apache Spark.

3.1 Current UDF Processing

UDFs are important building blocks of dataflow programs. Depending on the semantics of the surrounding operator, UDFs are either applied on single elements or on groups of elements. In JVM-based dataflow engine APIs, UDFs are implemented in Java or Scala. During distributed execution, workers deserialize binary data received from their network interface before processing the UDF, as illustrated in Fig. 1. Incoming data is deserialized from the Off-Heap memory to the Heap, before processing the UDF. Subsequently, the UDF result is serialized from the Heap to the Off-Heap memory, and sent over the network interface. We observe three performance bottlenecks in the current UDF processing method.

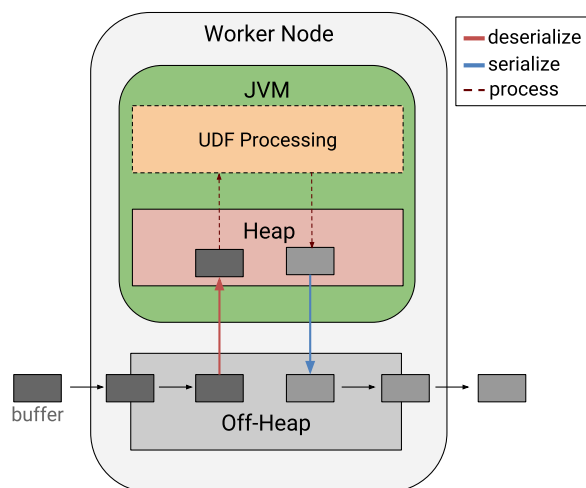


Fig. 1: Current UDF processing method.

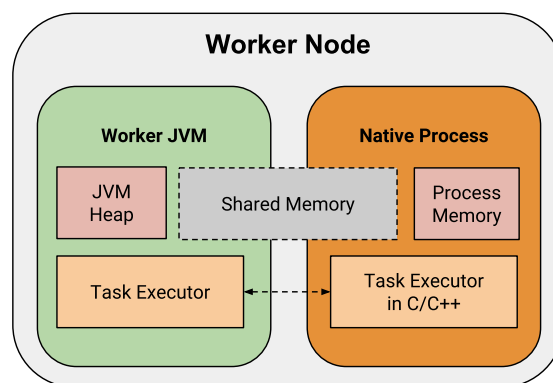


Fig. 2: UDF computation offloading.

First, instead of directly processing the data, workers must deserialize it and transform it into objects². Second, because of the dataflow engine execution model, data is serialized

² Memory management in Flink: <https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html>

row-wise. This means that for every row on the Off-Heap, the respective object will be created on the Heap. Garbage collection of a large number of objects leads to execution stalls [Ng16]. Third, the JVM runtime hinders lower-level optimizations, e.g. SIMD instructions, supported by modern CPUs [Cr15; Es18].

3.2 UDF Offloading Architecture

In the following, we describe an architecture for moving UDF computations from the JVM to native environments. Executing the UDF in a native environment avoids the deserialization operations, the garbage collection overhead, and allows for optimized processing using CPU-specific instructions. The architecture of our approach is depicted in Fig. 2. Using a shared memory area, the worker is able to exchange data between the the JVM and a native process. The UDF is offloaded from the JVM task executor to a native task executor. We propose two methods to exchange data between the JVM and the native runtime.

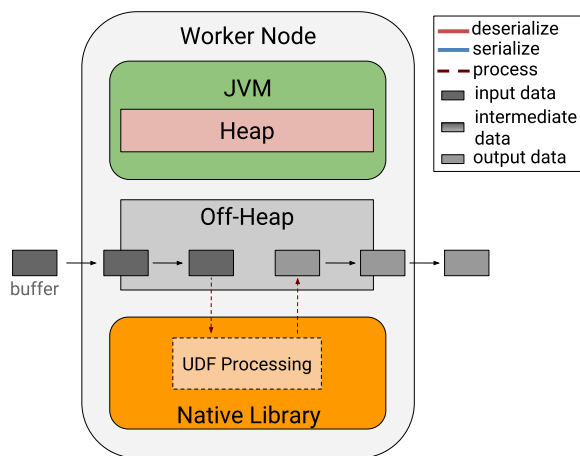


Fig. 3: Off-Heap data exchange.

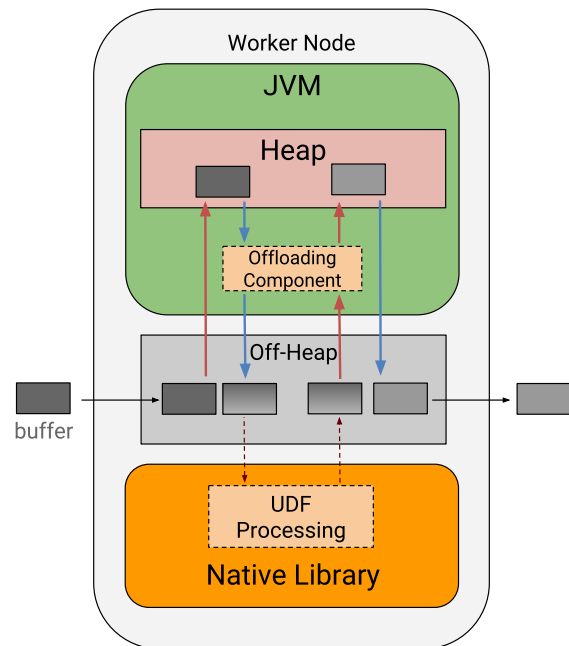


Fig. 4: On-Heap data exchange.

Off-Heap Offloading. During the *off-heap* data exchange method illustrated in Fig. 3, the Off-Heap memory is used as a shared memory region between the worker JVM and the native process. Since languages such as C and C++ process binary data, it is not necessary to transform the data before UDF execution in the native process. By exchanging a memory pointer, the native process has access to the data on the Off-Heap. A disadvantage of this method, is that native UDF execution is carried out directly on network buffers. This means that if the data is serialized row-wise in the network buffers, no optimizations for columnar processing can be applied. Moreover, network buffers contain metadata, such as event logs, next to the data records, which hinders efficient memory access during UDF processing.

On-Heap Offloading. To address the limitations of the *off-heap* method regarding data formats, we introduce the *on-heap* method. As illustrated in Fig. 4, after receiving and deserializing the data to the JVM Heap, the objects are serialized to the JVM Off-Heap, through the offloading component. The result of the native UDF is written to the JVM Off-Heap. Subsequently, the offloading component deserializes the result binary data to the Heap, which is then serialized again by the worker to send it over the network interface. An advantage of the *on-heap* approach is that its implementation is possible without altering the dataflow engine core. Moreover, since a custom serializer is provided in the offloading component, it is possible to choose the format for UDF processing, instead of being bound to the format in the network buffer. A disadvantage of the discussed approach is the overhead of the mediating computation offloading component. On the one hand, it allows to choose between data formats for improving the UDF performance. On the other hand, the intermediate serialization and deserialization operations lead to significant performance overhead, as we discuss in the next section.

The main advantage of the *off-heap* method is that serialization operations are completely avoided. However, it is not possible to implement it within current systems, without changing system components. On the contrary, the *on-heap* method is applicable without any modifications. The main disadvantage of the *on-heap* method, is the overhead of the intermediate serialization operations, which allow to change the data format before processing. Converting the data to a columnar format, is beneficial for certain operations, e.g. aggregations that reference few columns of a dataset. Additionally, aggregation UDFs have a small result size, a property that minimizes the *on-heap*'s intermediate serialization step overhead. Furthermore, using the *on-heap* method, efficient memory access is guaranteed in the native UDF, since the offloading component serializes only necessary record data.

3.3 Prototype on Spark

We implement our *on-heap* offloading prototype on top of Apache Spark. We use the interface of the `mapPartitions` operator to implement the computation offloading component, which serializes and deserializes all intermediate input and output data. For the serialization operations we use the `ByteBuffer` library³, which allows to create `ByteBuffer` objects on the Off-Heap (`DirectBuffer` objects). We use custom row and column serialization for data exchange. For the row-oriented serialization, we write all input element attributes consequently to one `DirectBuffer`, while for the column-oriented serialization we use as many `DirectBuffer` objects as input columns. For our prototype, we implement UDFs in C, and compile them to shared libraries using `gcc`. We use the Java Native Interface⁴ (JNI), a framework for embedding native applications within the JVM runtime, to register the native functions in our prototype. Data exchange is established by serializing data to `DirectBuffer` objects and passing pointers and the input data size as the native function parameters.

³ <https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html>

⁴ <https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>

Our prototype uses pre-compiled compute kernels implemented in C. To build optimized kernels for the native task executor, lower-level libraries that make use of specialized hardware and SIMD instructions can be used, e.g. Intel MKL⁵, BLAS⁶. For arbitrary UDF support, it is necessary to extend the prototype with a compiler component, which translates UDFs to lower-level representations. Furthermore, to implement the *off-heap* approach in current systems, UDF interfaces which provide access to serialized buffers for the native task executor are necessary⁷, for both row and columnar formats.

4 Evaluation

In this section, we evaluate the performance of our proposed computation offloading approach. We first present a set of Java microbenchmarks that measure the performance of standard UDF processing compared to the *on-heap* and *off-heap* offloading methods. Furthermore, we evaluate our prototype on Apache Spark in several UDF scenarios.

4.1 Microbenchmarks

We conduct a set of Java microbenchmarks to get insights about the performance of the standard approach and our introduced offloading methods.

Setup. We execute our benchmarks on a machine with an Intel(R) Xeon(R) E5530 CPU (16 cores) with a clock rate of 2.40GHz and 20 GB of main memory, running Ubuntu 14.04. For the benchmarks we use Java version 1.8 and Scala version 2.11 with the Hotspot VM 25.181-b13. We initialize the JVM with 10 GB memory. We use GCC to compile C code to native libraries and provide the compilation flags `-O3 -march=native -mtune=native`, which enable vectorization among other optimizations. We perform the microbenchmarks using JMH⁸, a benchmarking library for Java. We fork each experiment three times, perform ten warmup iterations and then execute the experiment five times. We report the average execution time of five repetitions.

Offloading Approaches. To evaluate our offloading approaches, we implement UDFs of two types, a euclidean distance function that is applied to two fields of every row, and an aggregation function that sums all elements of all rows. For a fixed data size of 500MB containing integers, we scale the number of rows and columns. For example, the 2 column dataset has 62,500,000 rows, while the 10 column dataset has 12,500,000 rows.

We present the results of the two UDF microbenchmarks in Fig. 5. The standard UDF processing method involves deserialization, processing and serialization on the JVM. The *off-heap* UDF method involves only processing in C, while the *on-heap* method involves the

⁵ <https://software.intel.com/en-us/mkl>

⁶ <http://www.netlib.org/blas/>

⁷ Current Spark and Flink both support relational operations on serialized data, but not for UDFs

⁸ <http://openjdk.java.net/projects/code-tools/jmh/>

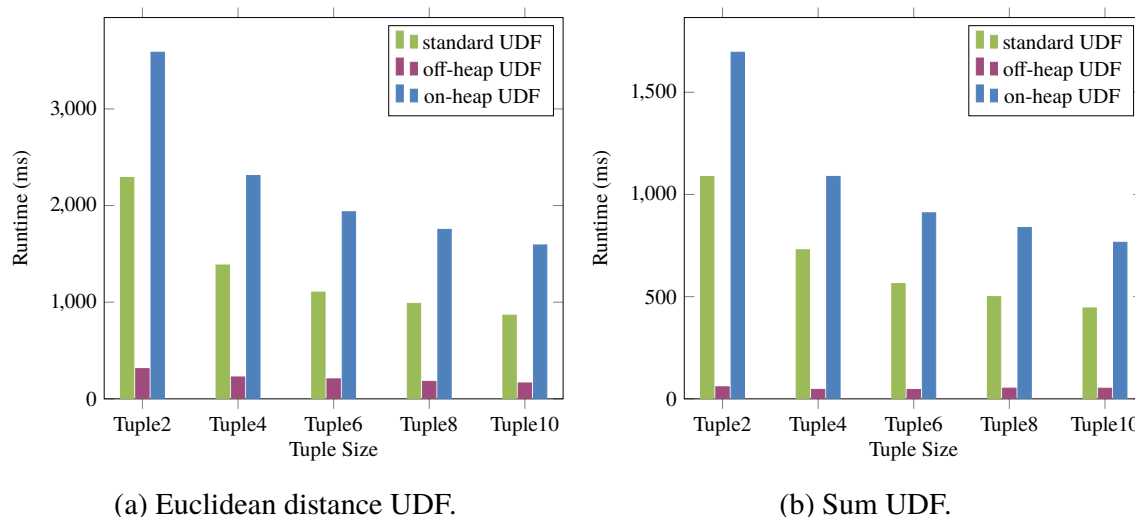


Fig. 5: Standard UDF processing compared to *Off-heap* and *On-Heap* computation offloading.

standard serialization and deserializations on the JVM, processing in C and an additional serialization step for the intermediate results, as described in Sect. 3. For the distance UDF, we observe that the *off-heap* method outperforms the standard approach by up to a factor of 10x, for a dataset of two columns. The performance gap drops while increasing the column size, since a larger column size means less created JVM objects, and hence less serialization overhead. The *on-heap* method performs almost 1.5x worse than the standard method, since it involves additional serialization operations for the intermediate result. We observe the same results for the sum UDF. The performance gap between the *off-heap* and the standard method is larger, a result possibly explained by faster aggregation processing in C.

Standard UDF Analysis. In order to explain the performance of the standard UDF processing method, we microbenchmark all three involved operations in the distance function: deserialization of the input, processing, and serialization of the output. The results are shown in Fig. 6. We observe that for a dataset size of 500MB the performance for processing a two-column dataset is 2x worse than the performance of processing a ten-column dataset. The detailed results show that when the column number is increased, the serialization and deserialization overhead is minimized, and the processing performance is increased. When the number of columns is increased, the number of rows is decreased, hence less objects are created and less serialization and deserialization overhead is created. Nevertheless, for ten-column datasets, the execution time of serialization and deserialization operations attribute to 80% of the overall UDF execution time.

4.2 Prototype Evaluation

We implement two UDFs using our *on-heap* offloading approach in Apache Spark, as discussed in Sect. 3. We use two synthetic datasets that contain elements of type double. For a fixed size of 5GB, the first dataset has 2 columns, while the second dataset has 10

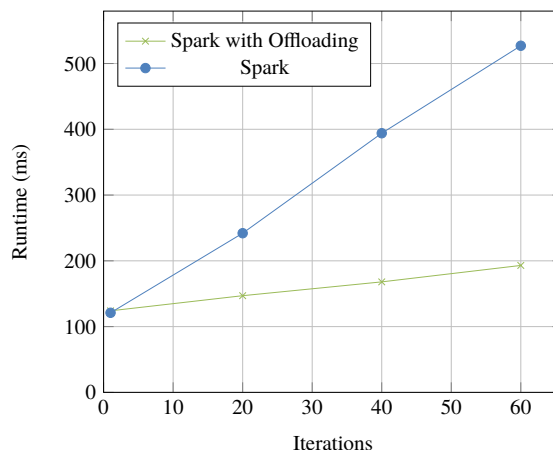
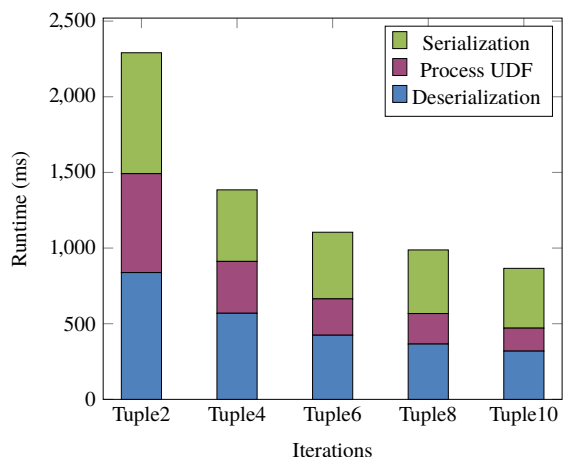


Fig. 6: Detailed standard UDF performance. Fig. 7: Compute-heavy UDF in Spark

columns. In the prototype, we use a columnar format for data serialization.

Distance UDF. The first UDF, is the euclidean distance function described earlier. For the two-column dataset, we apply the distance function on both columns, while for the ten-column dataset we apply the distance function on two columns. Furthermore, for the ten-column dataset, we apply the distance function on all ten fields, calculating the distance between two five-dimensional points (for each row). For each input row, the UDF produces an output row with an additional column, the result of the distance function. The results in Fig. 8a show that the *on-heap* computation offloading approach (Spark with offloading) has similar performance to the standard Spark UDF processing (Spark) only in the case of a two-column dataset. For the ten-column dataset, the offloading approach performs up to 3x worse than the standard method. The performance behavior is explained by the intermediate serialization steps needed for the *on-heap* approach.

MinMax UDF. The second UDF, is an aggregation that finds the minimum and maximum elements of a column. As in the previous experiments, we apply the aggregation UDF to both columns of a two-column dataset, on two columns of a ten-column dataset, and on ten columns of a ten-column dataset. The results are shown in Fig. 8b. The offloading approach (Spark with offloading) outperforms the standard Spark UDF processing (Spark) by a factor of 1.5x for the two column dataset, and by a factor of 1.3x for the ten column dataset when the UDF references only two columns out of ten. When the UDF references all ten columns of the ten-column dataset, the performance of the offloading approach is 4.5x worse than the standard UDF processing. The result of the first two experiments shows that the offloading is beneficial, despite the intermediate serialization overhead. For the experiment with the ten-column dataset, we serialize only the two needed columns, this is why the two methods have similar performance. Nevertheless, when all columns are referenced inside the UDF, the serialization overhead is too high, and the offloading is not beneficial.

Compute-intensive UDF. In our last experiment, we evaluate our prototype for compute-intensive UDFs, by applying a distance function on two columns of the two-column dataset, within an iteration. To increase the UDF workload, we increase the number of iterations.

We include a factor that changes in each iteration, to avoid potential compiler optimizations. The results in Fig. 7 show that for a small workload, the performance between the two approaches is the same, similarly to the previous experiments. However, when the workload is increased, the performance of Spark with offloading gets a speedup of 2.5x.

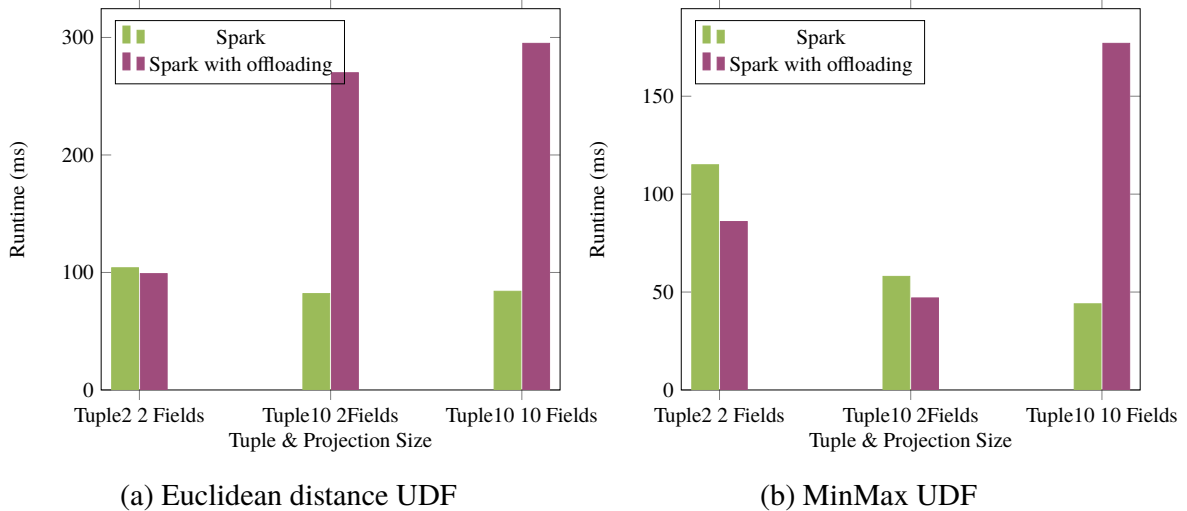


Fig. 8: Computation Offloading prototype evaluation.

4.3 Discussion

The results of our microbenchmarks show that serialization overhead is not negligible, since it takes up to 80% of the execution time. *Off-heap* computation offloading is the most performant approach, compared to the standard and the *on-heap* offloading approach, since it does not involve any serialization operations. However, as discussed in Sect. 3 it can not be implemented in current dataflow engines without altering system core elements. The benchmarks of our prototype show that it is possible to achieve better performance than the standard processing method, despite the serialization overhead of the *on-heap* approach. This performance is achieved by serializing only necessary fields in the UDF and using a columnar format for processing. We observe that the overhead of intermediate serialization operations caused by the *on-heap* approach is negligible for compute-intensive UDFs, since in that case computation offloading achieves a speedup of 2.5x.

5 Related Work

To enable native performance in dataflow engines, Essertel et al. propose Flare [Es18], a code generation approach. Flare accelerates SparkSQL programs, by carrying out execution and data readers in its own runtime, completely cutting off the Spark execution engine. Rosenfeld et al. propose a similar approach, where SparkSQL programs are transformed to database query plans and executed within a database engine [Ro17]. On the contrary, our

approach does not completely cut off the execution from the dataflow engine runtime, but it is used to offload specific parts of dataflow programs, e.g. UDFs.

Project Tungsten⁹ is an effort of Spark developers to enhance the platform with native performance, by introducing binary processing, explicit memory management and code generation. However, Project Tungsten is built on the JVM and does not use a separate native runtime, as we propose in this paper.

6 Conclusion

We proposed an approach to offload computations from a JVM-based dataflow engine to a native environment. To that end, we described two data exchange mechanisms for computation offloading, which allow to process UDFs outside the JVM Heap, and to exploit modern CPU characteristics. Our evaluation showed that computation offloading is beneficial for compute-intensive UDFs and in certain cases for aggregation UDFs.

References

- [Al14] Alexandrov, A. e. a.: The stratosphere platform for big data analytics. *The VLDB Journal* 23/6, pp. 939–964, 2014.
- [Cr15] Crotty, A. e. a.: Tupleware: "Big" Data, Big Analytics, Small Clusters. In: *CIDR*. 2015.
- [Es18] Essertel, G. e. a.: Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In: *13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, pp. 799–815, 2018.
- [Ne11] Neumann, T.: Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4/9, pp. 539–550, 2011.
- [Ng16] Nguyen, K. e. a.: Yak: A High-Performance Big-Data-Friendly Garbage Collector. In: *OSDI*. Pp. 349–365, 2016.
- [Ro17] Rosenfeld, V.; Mueller, R.; Tözün, P.; Özcan, F.: Processing Java UDFs in a C++ environment. In: *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, pp. 419–431, 2017.
- [Za12] Zaharia, M. e. a.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX, pp. 15–28, 2012.

⁹ <https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html>