Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?

Johan Peltenburg(✉), Ahmad Hesam, and Zaid Al-Ars

Computer Engineering Lab, Delft University of Technology, Delft, Netherlands
{j.w.peltenburg,z.al-ars}@tudelft.nl,a.s.hesam@student.tudelft.nl

Abstract. Advancements in the field of big data have led into an increasing interest in accelerator-based computing as a solution for computationally intensive problems. However, many prevalent big data frameworks are built and run on top of the Java Virtual Machine (JVM), which does not explicitly offer support for accelerated computing with e.g. GPGPU or FPGA. One major challenge in combining JVM-based big data frameworks with accelerators is transferring data from objects that reside in JVM managed memory to the accelerator. In this paper, a rigorous analysis of possible solutions is presented to address this challenge. Furthermore, a tool is presented which generates the required code for four alternative solutions and measures the attainable data transfer speed, given a specific object graph. This can give researchers and designers a fast insight about whether the interface between JVM and accelerator can saturate the computational resources of their accelerator. The benchmarking tool was run on a POWER8 system, for which results show that depending on the size of the objects and collections size, an approach based on the Java Native Interface can achieve between 0.9 and 12 GB/s, ByteBuffers can achieve between 0.7 and 3.3 GB/s, the Unsafe library can achieve between 0.8 and 16 GB/s and finally an approach access the data directly can achieve between 3 and 67 GB/s. From our measurements, we conclude that the HotSpot VM does not yet have standardized interfaces by design that can saturate common bandwidths to accelerators seen today or in the future, although one of the approaches presented in this paper can overcome this limitation.

1 Introduction

With the advance of the big data era, many different big data processing and storage frameworks have been developed. Many of these frameworks are written in languages that use a Java Virtual Machine (JVM) [10] as the underlying platform to execute compiled programs. This allows a cluster to easily scale out, adding nodes of any type of hardware, as long as they can run a JVM. A well known example is Apache Spark [18] which is written in Scala and is generally run on the OpenJDK HotSpot virtual machine.

Although the performance of programs run on the JVM can (in very specific situations) come close to the performance of native implementations, the added layers of abstraction still impose limits [8]. Speeding up JVM applications beyond

J.M. Kunkel et al. (Eds.): ISC High Performance Workshops 2017, LNCS 10524, pp. 220–236, 2017.
https://doi.org/10.1007/978-3-319-67630-2_18

Just-In-Time (JIT) compilation can be done using native libraries to squeeze out the last bits of performance that the underlying platform has to offer, sacrificing some of the portability of the application. While still a long way to go, the big data field is slowly catching up with the performance known from the high-performance computing (HPC) domain [1]. However, as the end of multicore scaling approaches, scaling up even native CPU performance will be troublesome in the near future [6].

Thus, as big data problems become bigger, there is a need to go even beyond the performance that traditional multicore systems can offer. For this reason, the research and industrial community is looking at other paradigms, such as combining accelerators or near-memory computing with big data platforms. In this paper, the focus is on accelerators.

GPGPU computing is currently the most popular method for accelerated computing. GPUs offer superior performance for tasks with a lot of thread-level parallelism and floating-point calculations. Effort is also put in the more power-efficient FPGA accelerators, suitable for deeply pipelined datapaths and other highly parallel algorithms. In either case, there is little explicit support for accelerated computing in specifications or implementations of major JVMs at the time of writing.

One of the major challenges during integration of JVM programs with accelerators is transferring the data represented as objects in the JVM memory to the accelerator. The interface between the data stored in objects managed by the JVM and an accelerator incurs a specific amount of overhead. If this overhead is higher than the performance gained from the accelerator, there is no point in investing effort to accelerate an algorithm.

Because we look at this matter within the context of big data frameworks, we assume that there is an application that would like to perform a transformation on a parallel collection of data items, represented as JVM objects. One example is a map transformation, which is a common operation for big data applications. The goal of this paper is now to give an overview of four different yet feasible approaches in transferring the object data from the JVM to the accelerator. Furthermore we attempt to quantify the overhead of this data transfer based on the layout of the object and the approach taken. This can help future development and integration of accelerators with JVM-based big data frameworks.

The contributions of this paper are:

- We give an overview of the most feasible approaches in transferring data between JVM and accelerator.
- We present a benchmarking tool that quantifies the parameters of the model for a given platform. This allows designers and researchers to get an estimation of the performance of their accelerated implementation, when JVM objects hold the source data. It will also give an indication on which approach will suit their performance requirements. The tool can also be used as a static analysis tool on custom object layouts.
- We measure the data transfer performance of each of the approaches on the OpenJDK HotSpot VM running on a POWER8 system.

The organization of the paper is as follows. Section 2 will discuss related work. In Sect. 3 a more thorough problem definition is given. In Sect. 4 we will give an overview of four feasible approaches. Section 5 comprises the experimental setup. In Sect. 6 the results are presented. A conclusion is given in Sect. 7.

2 Related Work

When accelerators are controlled by and attached to a host system, it is assumed that the accelerator interface partially consists of a native library. Therefore, the problem of transferring object data from JVM to accelerator is initially similar to the problem of native function access to JVM memory.

For this reason, the JVM implements the Java Native Interface (JNI). Many open-source projects exist (e.g. JNA [11], JavaCPP [2]) that mainly attempt to simplify integration of native libraries with Java programs through sugaring or abstraction of the JNI (which is normally used by writing C or C++ code). We will measure the best case performance of JNI without any of the overhead introduced by the frameworks.

Several researchers that attempt to integrate accelerators with JVM based big data frameworks note that the JNI interface causes a major performance bottleneck for their applications. In the work of [3,9], the massive latency that the JNI introduces is hidden by task pipelining, effectively overlapping JNI access with accelerator execution. Also, when a succession of transformations will take place on the accelerator, intermediate data is cached in local memory and can also be broadcast to other nodes as is.

Another interesting scheme is described in [7], which uses direct ByteBuffers for which the backing array is mapped to a region accessible through direct memory access by an FPGA. The authors use the Xilinx Zynq system, in which the host CPU shares the same physical memory as the FPGA. Such hardware setup enables easy access to the data from the accelerators, although it is not yet common in today's big data clusters.

In the Apache Spark project, with the introduction of the Tungsten engine, data items can be stored in off-heap memory using DataFrames and Datasets. At the time of writing, this is currently done mainly to prevent garbage collection overhead on the large collections of data. The community has been discussing the possibility of feeding data that is stored off-heap directly to native libraries, but any design, implementation or measurements have not yet been presented [16]. Furthermore, objects are serialized into the off-heap memory using serializers that in many cases will also introduce extra overhead by e.g. compressing the data [13].

More recently, with TensorFrames [5], integration of GPGPU accelerated computing in Spark using off-heap managed data is shown. However, internally the data is first deserialized back into the JVM and then JavaCPP is used (which is JNI based) to transfer the data row-wise through a native library to a GPGPU, which is rather inefficient. In the work on Spark-GPU [17], a similar approach is seen. Data items first have to be transferred to some off-heap memory region, before passing it to a GPGPU. In a specific case with string objects, the authors show that reading back this data from a GPU-friendly projection in an off-heap structure incurs significant overhead of between 10.5× to 18.3×.

Many of these previous works focus on accelerating a specific application, in which the interface between JVM and accelerator is not the main point of thorough investigation. In this work, we aim to give an overview and quantify *in detail* the properties of this interface, since it is one of the most critical components in such a system.

3 Problem Definition

In light of the advancing interest in accelerating JVM based big data applications and frameworks, the main question that this paper aims to address is as follows. *Which approaches exist to transfer data held by objects in a JVM to an accelerator, and how efficient are they?*

To scope the question, we assume that there is an application holding a collection of objects that contain data of interest to be used in a transformation. The transformation is implemented in an accelerator. This commonly means that each object in the collection will be transformed to some new object, or it will be reduced to some final result. We also assume the collection is parallel, thus the transformation may be applied to the objects in parallel as well, i.e. the objects within a collection do not refer to each other.

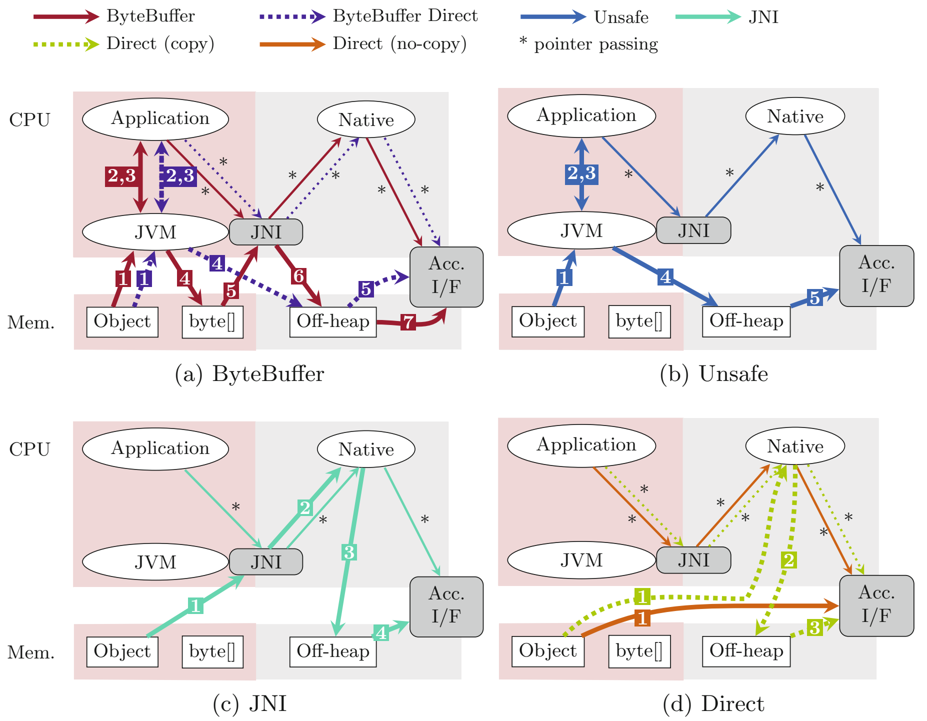
The fields of an object that represent its state and data, can be of the following types:

- A primitive (e.g. an integer, float or character).
- A reference to a child object or a child array object.

One exception is the array object type; it can hold multiple primitives or references, where each primitive or reference does not have a separate field identifier, merely an index. For the sake of simplicity we will assume that there are no loops in the reference graph of the object, i.e. all object reference graphs are trees.

The main problem within this scope is due to the fact that a programmer running applications on a JVM has no explicit control over the location or the layout of the objects in memory. In a system where one has control over both layout and location of objects, one may choose to lay out the data in such a way that it is convenient for an accelerator interface to access. This usually means that the data at least resides in a contiguous memory region.

Thus, to perform the transformation on one object of the collection, all primitives that reside in the object tree must first be obtained. This involves traversing the object tree, accessing all the primitives, whether they are in fields or in arrays. When this data is collected and saved in a contiguous memory region, this process is also known as *serialization.* Serialization is used to store the object to disk or to transfer it over a network, hence the object must usually be placed in a contiguous memory region so that it may fit in a file or a message. Later on,



(a) ByteBuffer

(b) Unsafe

(c) JNI

(d) Direct

Fig. 1. Four different approaches of transferring the object data to the native environment. The thick lines represent the data path, and pointer/reference passing between different programs is shown with an asterisk. Label numbers indicate data flow order.

the serialized object is deserialized into the memory of another JVM. However, for a transformation to take place in an accelerator, the primitives must merely be transported to the accelerator's local memory; not necessarily reconstructing it in such a way that a JVM program can access it again.

Accelerators are often controlled by a host CPU. In most cases this CPU will also run the JVM. Controlling the accelerator from the application will therefore involve calling at least one, but possibly multiple native functions. In major JVM implementations, this can be done using the Java Native Interface (JNI). Therefore, there are two ways of where object traversal could take place; either by bytecode running on the JVM itself or by a native function invoked through the JNI.

To address the question posed at the beginning of this section, the next section will give an overview of four feasible approaches to obtain the primitives and transfer them to an accelerator.

Table 1. Summary of characteristics of different approaches

Approach	Traversal	Serialized	Copies	Portability	Support	Seen in
ByteBuffer	bytecode	yes	1-2	high	high	[13]
ByteBuffer (off-heap)	bytecode	yes	1	high	high	[7]
Unsafe	bytecode	yes	1	medium	low	[5][17]
JNI	native	yes	1-2	medium	high	[11][2][4]
Direct (copy)	native	yes	1	low	low	—
Direct (no-copy)	native	no	0	low	low	—

4 Overview of Data Transfer Approaches

This section first discusses the approaches of accessing a single JVM object using a single thread, and then how multiple objects can be accessed using multiple threads.

4.1 Single-Object, Single-Thread Approaches

There are four basic approaches by which the data of an object could be transferred to an accelerator (also shown in Fig. 1), namely:

(a) *ByteBuffer* approach — Using the JVM to traverse the object tree and write it into a byte array using the `java.nio.ByteBuffer` or its derivatives like `IntBuffer` or `FloatBuffer`. The byte array is then passed to the accelerator interface through the JNI.
(b) *Unsafe* approach — Using the JVM to traverse the object tree and write it directly to off-heap memory using the `sun.misc.Unsafe` library. The address of off-heap memory location of the object is then passed to the accelerator interface.
(c) JNI approach — The object reference is passed as an argument through the JNI to a native function. Then, the native function uses JNI functions such as `Get<Primitive>Field` or `Get<Primitive>ArrayElements` to obtain the primitives.
(d) *Direct* approach—Traversing the object tree directly while it resides inside the JVM memory. The accelerator interface may directly load the data or it may first be serialized in some off-heap memory location.

The following subsections will discuss each approach in more detail. A summary can be found in Table 1.

ByteBuffer approach: For this approach, the object tree is traversed using JVM bytecode. Primitives and primitive arrays are copied to a `ByteBuffer` using its `put` and `get` methods. ByteBuffers are objects that wrap around a byte array (called the *backing array*). They allow an easy interface to load and store primitives from and to the byte array. The reference to this byte array can be passed through the JNI to a native function that interfaces with the accelerator.

To obtain the actual array, the JNI function `Get<Primitive>ArrayElements` or `GetPrimitiveArrayCritical` can be used. This may[1] cause another copy (although less likely in the latter case) of the data, but in either case it makes the array accessible to the native function. A variant to this approach is where the ByteBuffers can also wrap around an off-heap byte array, if the byte array is allocated using the `allocateDirect` method. In this case, the data only has to be copied from VM memory to the byte array once. The address of the off-heap byte array can be obtained in the native code by using the JNI function `GetDirectBufferAddress`.

Unsafe approach: The Unsafe approach uses the `sun.misc.Unsafe` library. This library allows C-like memory operations such as allocation and freeing of off-heap memory. Within the Java programming paradigm it is considered 'unsafe' because usually memory management is not done explicitly by the programmer. The library is tightly coupled with the HotSpot VM, but its interface is not officially supported or standardized. Traversal of the tree is done using JVM bytecode. Primitives and primitive arrays are copied to an off-heap allocated memory location using `put` and `get` methods. This makes the Unsafe approach quite similar to the ByteBuffer Direct approach. To access the data, the memory address of the off-heap structure can be simply passed to a native function interfacing with the accelerator using the JNI.

JNI approach: The JNI approach is less straightforward, since traversing the object tree is done through JNI calls in the native code. First, the references to the classes of the objects in a tree must be obtained using `FindClass`. Then, the field IDs of the classes must be obtained using `GetFieldID`. Object references can be traversed using `GetObjectField`. Finally, with `Get<Primitive>Field`, primitive fields can be obtained. The functions `Get<Primitive>ArrayElements` or `GetPrimitiveArrayCritical` can be used to obtain array values, where both functions potentially copy the values into a newly allocated region that must be released afterwards. Because an accelerator cannot call JNI functions directly, it is assumed that when the JNI approach is used, the primitives are stored in some memory allocated by the native code, and thus the object is serialized. The serialized object is then passed to the accelerator interface.

Direct approach: The Direct approach involves traversing the object tree and obtaining the values from the JVM memory itself. Traversing the object tree is done through calculating pointers to the objects from JVM references directly. Fields are taken from offsets on the object pointers. This approach has low portability since the way in which references are represented and translated to virtual memory addresses is not standardized across implementations. For example, in the HotSpot VM, this depends on VM parameters and platform address size. References (called *ordinary object pointers* or OOPs) can be 32- or 64-bits, where the 64-bit representation is an actual native pointer, but the 32-bits representation might be a compressed OOP [12]. Also, the offsets or

[1] this depends on whether the representation of the array in the VM is the same as the native representation, and if the VM garbage collector supports "pinning".

implementation of field storage is not specified. Therefore, this approach is not straightforward and is extremely platform-dependent.

If the accelerator has an interface that allows to initiate loads/stores from/to the host application memory that is running the JVM (e.g. CAPI [15], or with CPU + FPGA SoCs where the acceleration fabric shares the data bus of the CPU [7], or with techniques such as NVIDIA's Unified Memory in CUDA), it is not required for the object to be serialized. Instead, the actual object traversal may take place on the accelerator itself. Therefore, the Direct approach allows serialized (copy) but also unserialized (no-copy) access to the object, which is unique to this approach.

We have not found an accelerator interface leveraging this technique published in literature. Note that for server-grade systems this technique seems yet unfeasible, since the latency of contemporary accelerator interfaces is still in the order of microseconds. When reference are traversed in large data structures with small objects, this will result in poor performance, because the ratio of requests to data is high.

It might appear that proper functioning of this approach can be endangered by the JVM garbage collection mechanism. However, when starting the Direct approach through a single JNI invocation that uses an object reference as a parameter, this reference is made a local reference. This means that as long as the JNI function has not yet returned, the garbage collector will not move the object of this reference, or its children. The reference could even be made global such that the object will not be garbage collected at all and can be passed between different JNI invocations or threads.

4.2 Parallel Access of Collections of Objects

When there is a collection of objects to be processed by an accelerator, and the objects of the collection do not refer to other objects in the collection, the collection may also be accessed in parallel. This can help to increase the throughput on multicore systems. In case of the Direct approach with load/store capable accelerators, loads for data of multiple objects could be pipelined.

To support parallel access for the ByteBuffer approaches, each thread gets its own ByteBuffer object with backing array in order to prevent race conditions. The backing arrays are obtained through the JNI and could be merged in the native code. They could also be sent to the accelerator interface in sequence, thus from the accelerator point of view, it will no longer be a single collection per se. This might introduce some overhead or require a slightly more complex control structure for the ByteBuffer approach.

For the Unsafe approach, the memory is allocated once, then each thread gets its own instance of `sun.misc.Unsafe`, again to prevent race conditions. Then, each thread operates on a different offset of the destination memory.

For the JNI approach, parallel access is more complicated. Because references to objects are only valid in the corresponding thread that obtained them through the JNI, the reference to the array holding the collection must first be made global before it is passed to each thread. New threads must also register with the JVM using the JNI function AttachCurrentThread before they may call other JNI functions. After accessing the values, the threads must detach and the global reference must be released to allow garbage collection to take place on the objects.

For the Direct approach, parallel access is straightforward. Multiple threads may have multiple outstanding accesses of JVM memory simultaneously, and store them on different offsets of off-heap memory.

5 Experimental Setup

To measure the access times of the different approaches, a benchmarking tool was implemented (see also Fig. 2). As an input to the tool, a layout of an object tree is first specified. From this specification, Scala sources and ultimately JVM bytecode is generated, containing the required object classes and an Instantiator class which contains methods to instantiate the object tree and fill it with random data. Furthermore, for each approach, in the top-level class, methods are generated to serialize the object to a byte array or to off-heap memory corresponding to the description of the ByteBuffer and Unsafe approach, respectively. These classes are then compiled to JVM bytecode. For the JNI and Direct approach, functions callable through the JNI are generated in C. They are then also compiled to a shared library, together with functions that access the data for the ByteBuffer and Unsafe approach. Finally, a second program (benchmark runner) can take the class files and library as an input and run measurements of object access time. The benchmarking tool is available as an open-source project [14]. Using this tool, it is possible to measure the access times of different types and sizes of object trees. It is possible to generate two types of object layouts. One layout where the root object does not contain any references, except to primitive arrays (a *leaf* object). A leaf object contains only a variable amount of primitives and arrays of variable size. Another layout where the object has a specified width W and depth D, such that at the root level it contains W references to the second level, where each object contains $max(W-1,1)$ references, until the level equals D. Only at the last level, the primitives and arrays are instantiated. This layout allows us to also make a linear object tree, by setting $W = 1$ and

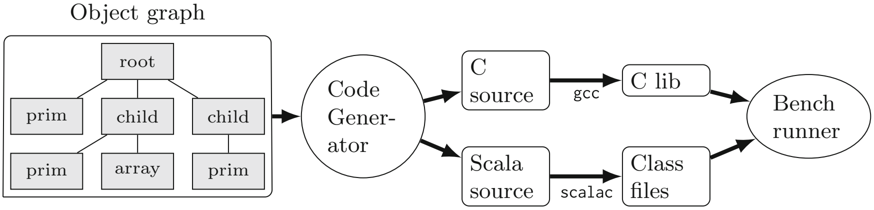


Fig. 2. General flow of the benchmarking tool

$D > 1$. It is also possible to supply a custom class layout or object tree to the tool, such that it may be used as a static analysis tool for existing applications.

By varying the parameters of the aforementioned object tree layouts, it is possible to obtain the parameters of the model for a specific platform and JVM implementation. By varying the number of primitives in a leaf object, the average time to copy a primitive can be measured. By increasing the array size of a leaf object, the average array copy time per element can be measured. By varying the depth in a linear object tree, the average time to traverse a reference can be measured.

Parallel collection serialization and parallel access performance is also measured to get the peak performance for the platform. From the hardware point of view, when more cores and hardware threads run in parallel, we may assume that the internal memory infrastructure is at some point saturated, resulting in peak performance for that platform. Thus, besides specifying the object tree layout for a benchmark, it is also possible to generate a collection of N of these objects and specify with how many threads to access the objects. Furthermore, because run-time measurements on the JVM are very noisy, the experiments are repeated R times and averaged. These numbers are reported per experiment in Sect. 6.

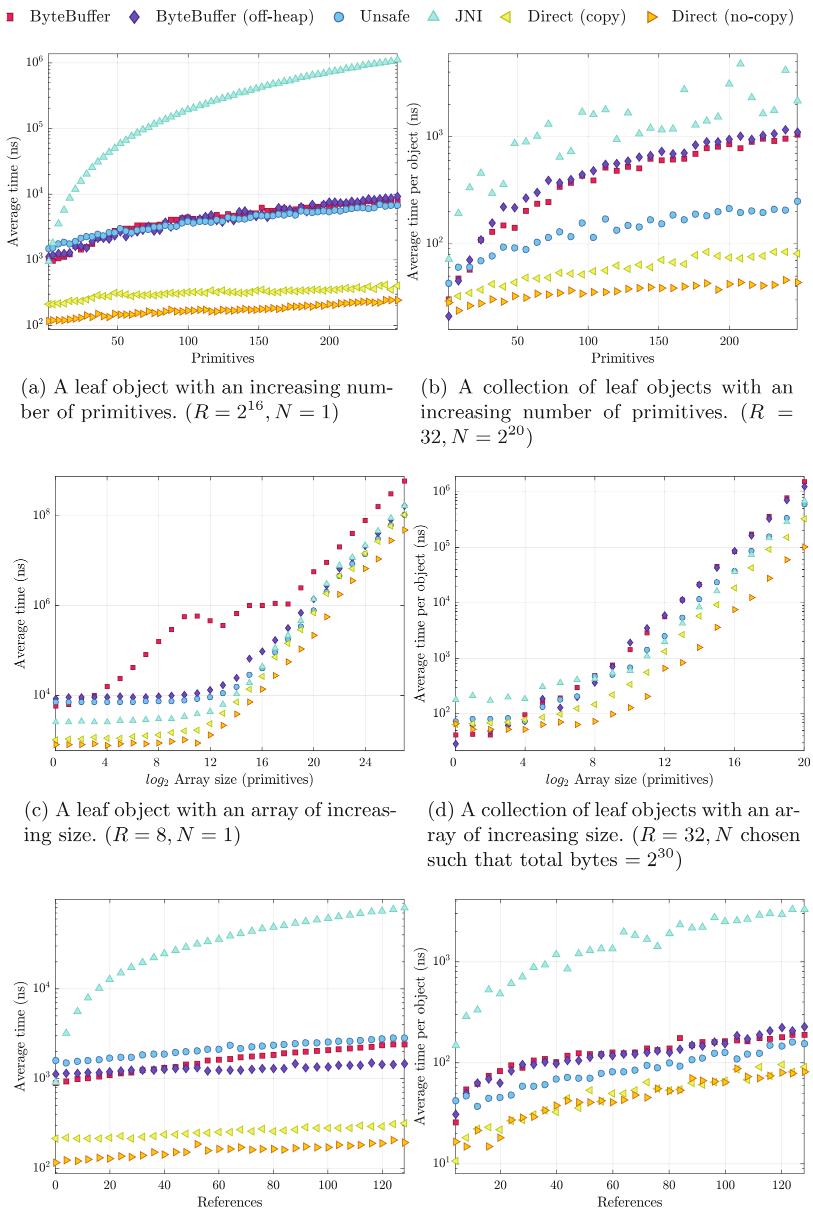
The benchmarking system consists of two POWER8 CPUs running at 3.42 GHz on an IBM Power System S824L (8247-42L) with 256 GiB of total RAM. We confine our measurements to one of the two CPUs only. Primitives used are 32-bit integers. Attaching an actual accelerator is outside the scope of this paper, but by accumulating all serialized primitives into a single value on the CPUs we can validate the correctness of each approach and we can make a fair comparison for the Direct (no-copy) case. This is in theory the fastest approach, because it does not copy at all. Without any operations on the data it would otherwise only have to resolve all references without accessing the data itself. Native threads are controlled through OpenMP and JVM threads are statically managed.

In a real application, more dimensions to the problem are relevant, such as how to lay out the serialized objects in the memory such that its structure is convenient to process in a specific accelerator, how to retain references within the serialized format, how to deal with cyclic object graphs, how to deal with static fields of classes, and more detailed problems which are commonly seen in serialization. However, because we are primarily interested in the feasibility and best-case performance, these dimensions are also outside the scope of this work.

6 Results

6.1 Single Object

In Fig. 3a, the access time of a leaf object with an increasing number of primitive fields is shown. We found that the ByteBuffer, Unsafe and Direct approaches show a mainly linear increase in access time, while the JNI approach shows a significant quadratic increase when the number of fields increases in



(a) A leaf object with an increasing number of primitives. ($R = 2^{16}, N = 1$)

(b) A collection of leaf objects with an increasing number of primitives. ($R = 32, N = 2^{20}$)

(c) A leaf object with an array of increasing size. ($R = 8, N = 1$)

(d) A collection of leaf objects with an array of increasing size. ($R = 32, N$ chosen such that total bytes $= 2^{30}$)

(e) An object with a linear object tree with increasing depth. ($R = 2^{16}, N = 1$)

(f) A collection of objects with a linear object tree of increasing depth. ($R = 4, N = 2^{21}$)

Fig. 3. Average latency of accessing JVM objects.

a leaf object. This is due to the use of the JNI function `GetFieldID` for this particular experiment. This function looks up the field identifier string in the HotSpot symbol table. Suppose the number of primitives is p. We must search p symbols p times to access all primitives. Thus the time complexity to access a field by using `GetFieldID` is $O(p^2)$.

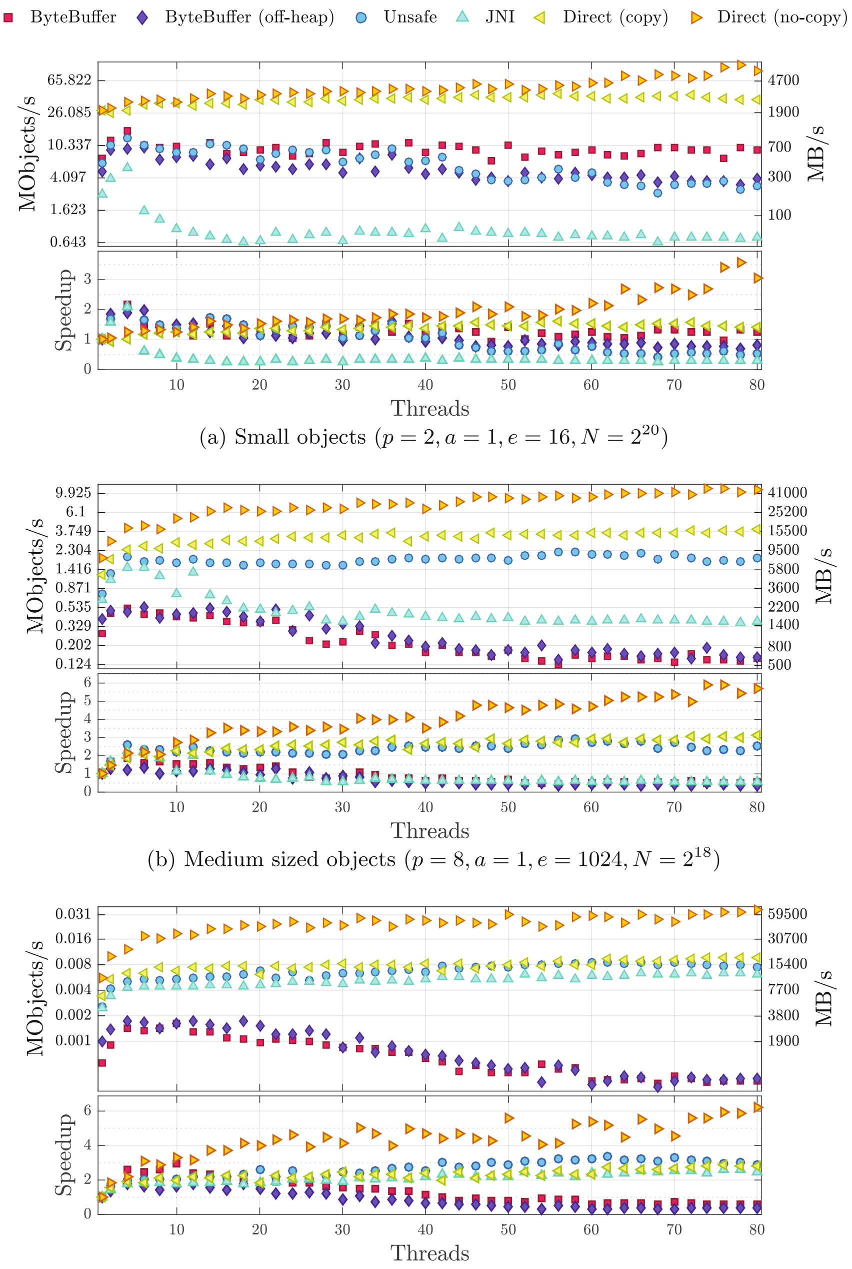
In Fig. 3c the access time of a leaf object containing only an array is shown. Accessing arrays of different sizes clearly shows the effect of the CPU cache hierarchy. The derivative of the access time with respect to the array size is small in regions where the array still fits in the cache. It becomes larger for array sizes that do not fit in the cache. Any initial overhead of the copies becomes relatively small. For each approach that has the same number of copies (see Table 1), for large arrays, their access times converge, because internally array copies are usually performed by highly optimized `memcpy` calls (although variants depending on the native platform exist, e.g. for the Unsafe and ByteBuffer approaches).

In Fig. 3e, the access time of a linear object tree is shown. We found that the access times increases in a mainly linear manner with respect to the number of references traversed.

6.2 Parallel Performance

In the case of a parallel collection, we first attempt to find a suitable number of threads. For each approach we measuring three cases; 1. small objects (2 primitives and an array with 16 primitives) 2. medium objects (8 primitives and an array of 1024 primitives) and 3. large objects (64 primitives and an array of 800 × 600 primitives. Reference traversal performance is included in these measurements since a collection consists of many references to all its objects. The collections are of such a size that their serialized representation is over several hundred MiB, to make sure each thread has enough work to justify the overhead from spawning it. The results of these three measurements are seen in Fig. 4.

From these measurements, we found that for all approaches except the Direct approaches, the scalability is rather poor. This is most likely due to race conditions on specific resources of the VM. These approaches scale very badly when the ratio of reference to data is high (small objects case). In the case of a high ratio of reference to data, the maximum number of threads even gives the best performance for the Direct approaches. In the medium and large object cases, the speedup increases all the way to the maximum number of threads only for the Direct (no-copy) case. This approach only performs a single load and accumulate on each data item. The gains from multi-threading are therefore more significant than in the case of the Direct (copy) approach, because the computation to communication ratio is three times higher. The Direct (copy) approach loads, stores and then loads again and accumulates the data. The Unsafe approach also scales rather well in the medium and large measurement, compared to the JNI approaches, which scales only well for the large case. The ByteBuffer approach does not scale very well beyond four threads. From these measurements we set a suitable number of threads for each approach as shown in Table 2.



(a) Small objects ($p = 2, a = 1, e = 16, N = 2^{20}$)

(b) Medium sized objects ($p = 8, a = 1, e = 1024, N = 2^{18}$)

(c) Large objects ($p = 64, a = 1, e = 800 \times 600, N = 2^{10}$).

Fig. 4. Throughput and speedup of accessing collections of JVM objects.

Table 2. Maximum throughput and corresponding number of threads for the small, medium and large benchmark.

Approach	Small (threads)	MB/s	Medium (threads)	MB/s	Large (threads)	MB/s	Threads used for Fig. 3
ByteBuffer	4	1142	4	2159	10	3121	4
ByteBuffer (off-heap)	6	706	6	2231	4	3346	4
Unsafe	4	914	58	9080	62	16604	10
JNI	4	389	6	6094	78	12332	4
Direct (copy)	56	3232	80	16273	76	18788	80
Direct (no-copy)	78	7366	76	47064	80	67381	80

6.3 Collection

In the case of a parallel collection, the same types of measurements are performed as in the case for a single object, although on a collection of N of these objects. These measurements are shown in Fig. 3.

From measurements shown in Fig. 3b, we found that the quadratic increase in access time for the JNI approach is now relatively insignificant, because the field identifiers only have to be looked up once for the whole collection. However, the JNI approach is still slow, because for each field primitive, the function call `Get<Primitive>Field` JNI must still be made.

Figure 3d shows that after different initialization times, approaches with the same number of copies converge towards the same access latency as the arrays get larger, as was the case for the single-thread single-object measurements.

Lastly, for the measurement of reference traversal in a linear object graph (Fig. 3f), the direct approaches show similar access time, followed by the Unsafe and ByteBuffer approaches. Theoretically, there should not be much difference between the Unsafe and ByteBuffer approaches, because reference traversal is done in the JVM in the same way for both approaches. The difference in average time for the ByteBuffer approach is due to the overhead induced by spawning the threads and ByteBuffer objects. For the Unsafe approach, this overhead is much lower. Again, the JNI approach shows an order of magnitude worse performance.

6.4 Discussion

Contemporary commercially available accelerator cards are often connected via PCI-e GEN3 with peak bandwidths of almost 8 GB/s or 16 GB/s, depending on the configuration. One example includes the POWER8 system where the CAPI interface can be used over such a link. Even newer interfaces such as NVIDIA NVLink are expected to achieve up to 80 GB/s.

From the measurements presented in this section, it can be seen that the ByteBuffer approaches are generally unfavorable, because they cannot achieve near the bandwidths of the PCIe range easily. They are only faster than using the JNI approach in the case of accessing a collection of small objects. The JNI approach can be a feasible solution, but only when the ratio of references-to-data is low (e.g. when there are few but large arrays in the objects). At the same time, the Unsafe approach performs better in most cases and it is much easier to program, because traversal of the object graph can be written or generated in the JVM based source language. If it is necessary to saturate the link, with small objects (a high reference/data ratio) the only feasible solution is to take the Direct approach.

A major drawback of this approach is that it is highly platform dependent and it could even be considered more 'unsafe‘ than the Unsafe approach, since it accesses VM managed memory without some sort of interface that was designed into the VM. To mitigate this drawback, the VM could *by design* include some functionality to support fast object graph traversal, serialization and data transfer to accelerator interfaces, implemented with native code and tightly coupled with the VM as in the case of the Unsafe library.

7 Conclusion

In this paper, an overview is given for four different approaches for accelerator interfaces to obtain data from JVM managed objects; using ByteBuffers, the `sun.misc.Unsafe` library, the Java Native Interface (JNI) and to directly obtain the data from JVM managed memory (Direct).

A benchmarking tool was implemented that generates code to serialize the object or a collection of objects for use in an accelerator, using these four different approaches (where two approaches have two variants). By measuring the access times of single objects by a single thread, and access times of a parallel collection of objects by multiple threads, the performance of a POWER8 system with the HotSpot VM was measured. Furthermore, the throughput of a collection of small, medium and large objects was measurement with respect to the number of threads.

From the measurements we may conclude that the ByteBuffer approach does not perform well in most cases (it can achieve between 0.7 and 3.3 GB/s of throughput). Also, it does not scale well with the number of threads. The JNI approach can perform well in situations where the ratio of references to data is low, but also scales poorly with the number of threads (it can achieve between 0.9 and 12 GB/s of throughput). The Unsafe approach scales slightly better, up to the number of physical cores of CPU, and is also able to provide enough bandwidth to saturate common accelerator interfaces (it can achieve between 0.8 and 16 GB/s of throughput). The best approach in terms of performance is the Direct approach. It scales well and offers more than enough bandwidth for common accelerator interfaces, but its portability and ease of use is poor (it can achieve between 3 and 67 GB/s).

The measurements of the benchmarking tool can effectively be used to predict the interface speed of accelerators attached to a JVM. This may help researchers and developers to obtain a good estimation of the maximum speedup they may get by combining accelerators with JVM-based applications.

As new accelerator interfaces with higher bandwidths are introduced, the need for a faster interface that is integrated into the HotSpot VM *by design* is high. This is especially the case if users of big data frameworks based on the JVM want to make use of the computational power of accelerators.

Acknowledgment. The authors would like to thank Erik Vermij for his help using the POWER8 system and the Texas Advanced Computing Center and their partners for access to the hardware. This work was supported by the European Commission in the context of the ARTEMIS project ALMARVI (project #621439).

References

1. Anderson, M., Smith, S., Sundaram, N., Capota, M., Zhao, Z., Dulloor, S., Satish, N., Willke, T.L.: Bridging the gap between HPC and big data frameworks. Proc. VLDB Endow. **10**(8) (2017)
2. Bytedeco: JavaCPP, April 2017, https://github.com/bytedeco/javacpp
3. Chen, Y.T., Cong, J., Fang, Z., Lei, J., Wei, P.: When apache spark meets FPGAs: a case study for next-generation DNA sequencing acceleration. In: The 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2016) (2016)
4. Chen, Z.N., Chen, K., Jiang, J.L., Zhang, L.F., Wu, S., Qi, Z.W., Hu, C.M., Wu, Y.W., Sun, Y.Z., Tang, H., et al.: Evolution of cloud operating system: from technology to ecosystem. J. Comput. Sci. Technol. **32**(2), 224–241 (2017)
5. Databricks: TensorFrames: Experimental tensorflow binding for Scala and Apache Spark, April 2017, https://github.com/databricks/tensorframes
6. Esmaeilzadeh, H., Blem, E., St Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: ACM SIGARCH Computer Architecture News, vol. 39, pp. 365–376. ACM (2011)
7. Ghasemi, E., Chow, P.: Accelerating apache spark big data analysis with FPGAs. In: 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), p. 94, May 2016
8. Gouy, I.: The computer language benchmarks game, 20 March (2017), http://benchmarksgame.alioth.debian.org/
9. Huang, M., Wu, D., Yu, C.H., Fang, Z., Interlandi, M., Condie, T., Cong, J.: Programming and runtime support to Blaze FPGA accelerator deployment at datacenter scale. In: Proceedings of the Seventh ACM Symposium on Cloud Computing, pp. 456–469. ACM (2016)
10. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification, Java SE, 8th edn. Oracle (2015)
11. Open-source project: Java Native Access, April 2017, https://github.com/java-native-access/jna
12. Oracle: Java HotSpot virtual machine performance enhancements, April 2017, http://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html
13. Oracle: Object serialization stream protocol, April 2017, https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
14. Peltenburg, J.: JVM-to-Accelerator Benchmark Tool, https://github.com/johanpel/jvm2accbench

15. Stuecheli, J., Blaner, B., Johns, C., Siegel, M.: CAPI: a coherent accelerator processor interface. IBM J. Res. Dev. **59**(1), 1–7 (2015)
16. Weiss, P.: Off heap memory access for non-jvm libraries, March 2017, https://issues.apache.org/jira/browse/SPARK-10399
17. Yuan, Y., Salmi, M.F., Huai, Y., Wang, K., Lee, R., Zhang, X.: Spark-GPU: an accelerated in-memory data processing engine on clusters. In: 2016 IEEE International Conference on Big Data (Big Data), pp. 273–283, December 2016
18. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: 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, p. 2. USENIX Association (2012)