Accelerating Java Streams With A Data Analytics Hardware Accelerator

Karthik Ganesan
karthik.ganesan@oracle.com

Ahmed Khawaja
ahmed.khawaja@oracle.com

Luyang Wang
luyang.wang@oracle.com

Shrinivas Joshi
shrinivas.joshi@oracle.com

Michelle Szucs
michelle.szucs@oracle.com

Melina Demertzi
melina.demertzi@oracle.com

Yao-Min Chen
yaomin.chen@oracle.com

Oracle Corp., USA

ABSTRACT

In this paper, we demonstrate how new technology from Oracle can be utilized to provide big data analytics acceleration in a streamlined fashion. Specifically, our approach leverages the acceleration capabilities of the Data Analytics Accelerator (DAX) unit provided by Oracle's T7/M7/S7 SPARC processors and the Java Stream API to seamlessly accelerate Java applications by up to 20X, while using drastically fewer resources.

Keywords

Java; hardware accelerator; data analytics; big data

1. INTRODUCTION

With the ever increasing importance of data analytics, we need approaches that allow us to process large amounts of data efficiently and in a programmer-friendly manner. The Java Virtual Machine (JVM) and the Java language have been foundational in the execution of many big data analytics applications and frameworks. One of the features in the Java language which allows the grouping and fast processing of large data is the Java collections framework (JCF) and the Java Stream API [3].

In this paper, we develop a standalone Java library that offloads the Java Stream API to the DAX on Oracle's T7/M7/S7 SPARC systems to achieve high-performance data analytics while using drastically fewer compute resources.

2. JAVA 8 STREAM API

The Stream API [3] was introduced in Java 8. One of its key capabilities is to process data in collections, such as an SQL query, to simplify data processing. Using this API, we can write very abstract query-like code without going into the details of iterating over the collection elements.

For instance, consider a Java method (Figure 1) which iterates through an array of daily high temperature data and counts the number of days that temperature exceeded 90 degrees Fahrenheit (F). This traditional implementation would typically be compiled into serial code ignoring the large amount of data parallelism [1] in this logic and result in very long execution times. An experienced developer could parallelize this code manually resulting in better performance at much increased programmer effort.

```
private static int Hotdays_count_nonStreamWay (int[]
      TemperatureArray) {
         int count = 0;
         for (int val : TemperatureArray) {
                 if (val > 90) count++;
         }
         return count;
}
```

Figure 1: Java method to count no. of days temperature > 90F

The same logic could be implemented using Java 8 Stream, as illustrated in Figure 2, in a less error-prone way and in fewer lines of code. With Stream, we can explicitly specify whether an operation is data-parallel by using the construct called *parallel*(). With this hint, the Stream library automatically creates multiple threads to process the data in parallel, which results in better performance on a multi-threaded or multicore system.

```
private static int Hotdays_count_StreamWay (int[]
      TemperatureArray) {
         return DaxIntStream.of(TemperatureArray).parallel().
              filter(t->t>90).count();
}
```

Figure 2: Stream API to count no. of days temperature > 90F

While a thread-parallel implementation brings many performance benefits, it still suffers from some overhead in terms of thread creation and resource utilization. A hardware accelerator targeted to speed up stream-style code can be more performant than a thread-parallel implementation and get the work done with dramatically fewer resources on the chip.

3. DAX FOR STREAM ACCELERATION

The DAX co-processor, seen in Figure 3 on Oracle's T7/M7/S7 systems provides hardware acceleration for query-like operations and it is a great match for accelerating Stream functions. DAX can perform specialized functions (including Scan, Select, Extract, Fill, and Translate) at blindingly fast speeds. Therefore, if DAX is exposed to Java through the Stream API, we can achieve significant acceleration of multiple data analytics frameworks written in Java and other JVM languages, such as Scala, with minimal effort from the developer side. For instance, we can call the Stream API from a program running on Apache Spark and get all the

ICPE'17 April 22-26, 2017, L'Aquila, Italy
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4404-3/17/04.
DOI: http://dx.doi.org/10.1145/3030207.3053664

benefits that DAX provides without the need of re-writing the application from scratch for a specific platform or an accelerator. The use of Stream API in the code is seamless, but for best performance, needs to be written adhering to certain patterns. Currently, the API supports offloading of IntStream. Acceleration for other streams and object-based streams is possible and left for future work. Best practice documents are available online [2].

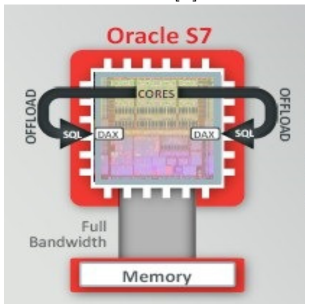

Figure 3: High-level view of a SPARC S7

4. STANDALONE LIBRARY

We create a standalone library available on Oracle Software in Silicon Developer Cloud [4] with the same interface as that of the standard Stream API with offloads to DAX. Table 1 shows the Stream operations offloaded to DAX and their correspondence to DAX primitives. The filter function returns a stream consisting of the elements matching the given predicate passed to this function. The map(ternary) is a special usage of map function that returns an integer array of zeroes and ones after applying a given predicate specified as a boolean ternary operator. Based on this mapping to DAX primitives, an existing code that uses Stream API can take advantage of the library and experience significant performance improvement without any source code changes, except for a minor change of adding an import statement and using DaxIntStream instead of IntStream.

Stream Operation	DAX Primitive
DaxIntStream.filter	Scan and Select
DaxIntStream.allMatch	Scan
DaxIntStream.anyMatch	Scan
DaxIntStream.noneMatch	Scan
DaxIntStream.filter.count	Scan
DaxIntStream.filter.toArray	Scan and Select
DaxIntStream.map(ternary).toArray	Scan, Select, and Extract

Table 1: Mapping of Stream operations to DAX primitives

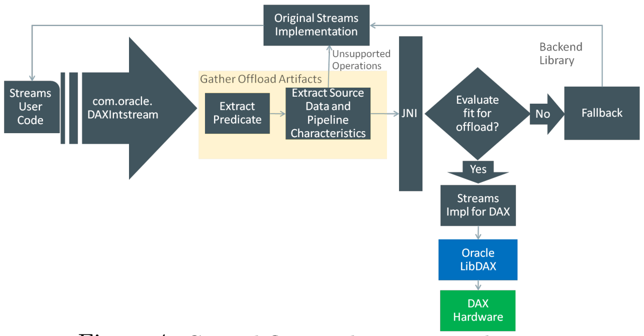

Figure 4: Control flow in the system architecure

The flowchart in Figure 4 shows the control flow in the system architecture. First, the offload artifacts are gathered in terms of predicate, pipeline characteristics, and the source

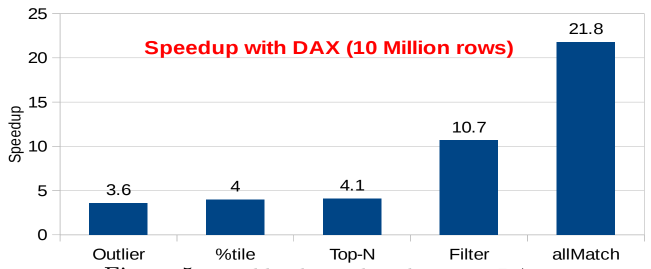

Figure 5: Workloads sped up by using DAX

data. The offload artifacts are checked against a few quick rules to determine if offloading is possible. If the offload artifacts are not conducive to offloading, the execution falls back to the traditional stream implementation. If the offload artifacts are conducive to offloading, the JNI gateway is invoked with the offload artifacts. Runtime decisions are made regarding whether to run an operation on the DAX coprocessor or on the core based on detailed heuristics that determine profitability in the back-end library. Only the streams marked as parallel are offloaded to DAX.

5. USE CASES

The analytics use cases that involve SQL-style Java, such as weather analysis, top-N integers, cube building, outlier detection, percentile calculators, and the K-Nearest Neighbor (KNN) algorithm benefit the most from our approach.

The standalone library for offloading stream functions to DAX achieves up to 22 times faster execution of Java-based analytics applications at significantly lower resource utilization. In addition to the performance benefits seen by acceleration on DAX, we free resources on the cores to be used for other operations. This results in less resource consumption and cooler datacenters. Furthermore, because the DAX hardware is exposed through an existing Java API, we can write platform-agnostic Java code that gets automatically offloaded to DAX "under the hood" on a SPARC platform.

The chart in Figure 5 shows the potential of this technology in speeding up some of the workloads we have experimented with. As part of a demo, we will showcase live demonstration of the speedups from using this technology to do fraud detection for an online retailer.

6. CONCLUSION

In this paper, we demonstrate how a new technology from Oracle can be used to accelerate big data analytics. We develop a standalone library that uses the Java Stream API and the DAX provided by Oracle's T7/M7/S7 SPARC processors to achieve high performance (up to 22X faster than Java-based analytics applications) while using drastically fewer compute resources.

7. REFERENCES

[1] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, Sept. 1972.
[2] K. Ganesan. Accelerating Java Streams with the SPARC Data Analytics Accelerator. https://community.oracle.com/docs/DOC-1006352.
[3] Java 8. Interface Stream. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html, 2016.
[4] Oracle. Software in Silicon Developer Cloud. http://swisdev.oracle.com, 2016.