Noname manuscript No.
(will be inserted by the editor)

Ignite-GPU: A GPU-Enabled In-Memory Computing Architecture on Clusters

Amir Hossein Sojoodi · Majid Salimi Beni · Farshad Khunjush

Received: date / Accepted: date

Abstract During recent years, big data explosion and the increase of main memory capacity, on the one hand, and the need for faster data processing, on the other hand, have caused the development of various in-memory processing tools to manage and analyze data. Engaging the speed of the main memory and advantaging data locality, these tools can process a large amount of data with high performance. Apache Ignite, as a distributed in-memory platform, can process massive volumes of data in parallel. Currently, this platform is CPU-based and does not utilize the GPU's processing resources. To address this concern, we introduce Ignite-GPU that uses the GPU's massively parallel processing power. Ignite-GPU handles a number of challenges in integrating GPUs into Ignite and utilizes the GPU's available resources. We have also identified and eliminated time-consuming overheads and used various GPU-specific optimization techniques to improve overall performance. Eventually, we have evaluated Ignite-GPU with the Genetic Algorithm, as a representative of data and compute-intensive algorithms, and gained more than thousands of times speedup in comparison with its CPU version.

Keywords .Apache Ignite · Parallel Processing · GPU · In-memory Computing

Amir Hossein Sojoodi
E-mail: amir.sojoodi@gmail.com

Majid Salimi Beni
E-mail: m.salimibeni@shirazu.ac.ir

Farshad Khunjush
E-mail: khunjush@shirazu.ac.ir

Department of Electrical and Computer Engineering, Shiraz University, Shiraz, Iran.

1 Introduction

With the advancement of technology in today's world, a vast amount of data is being generated at high velocities from various sources, containing several data types. These data are not valuable by themselves until they get analyzed and become useful and tangible information. Since new applications usually deal with large amounts of data and require a lot of processing power, it is not possible to process them on a single machine. Therefore, we need suitable tools that can perform these computations in a distributed manner so that they can provide more storage and processing power.

For this purpose, Apache Hadoop [1], which is an open-source version of MapReduce [2], was introduced. This platform could scale-out on any number of nodes, and works in a distributed manner. Moreover, Hadoop has its distributed disk-based file system, called the Hadoop File System (HDFS) [3], for managing and storing data on the clusters.

Considering the advent of newer applications, faster data production, and the demands to process these data, Hadoop's limitations became more prominent [4]. Some of the restrictions are as follows: supporting only batch processing, offering no support for real-time data processing, low-performance data processing due to the disk I/O bottleneck, and lacking support of some applications like machine learning. To overcome the issues mentioned, more advanced tools have been developed.

The main barrier, which limits Hadoop's performance is its disk-based distributed file system. To eliminate this barrier and enhance performance, in-memory platforms have been developed that store and process data on the main memory across the cluster. These platforms eliminated the disk IO bottleneck in Hadoop and revolutionized big data processing. Apache Spark [5] and Apache Flink [6] are two well-known in-memory platforms that were introduced to overcome Hadoop's restrictions. They can analyze streaming data in real-time and support SQL. Moreover, Spark has provided a library to facilitate distributed machine learning [7].

Although these platforms have brought many new facilities, they still possess some limitations. For example, Spark does not have a file management system itself and relies on other platforms. It also does not support many algorithms; its stream processing is not entirely real-time (it is micro-batch), and its latency is high in some situations. In addition, Flink is more known as a stream processing platform rather than a general-purpose processing engine.

Apache Ignite [8], as a recently released platform, is another distributed in-memory database for caching and processing big data. This platform provides some new features and has tried to overcome previous platforms' limitations. Ignite is suitable for data analytics, transactional, and streaming jobs. Moreover, it supports distributed machine learning and unlike Spark, it has its own file system to manage data on disk and main memory.

After the development of in-memory architecture and increasing RAM capacity, the only factor that circumscribes data analytics' performance is the processing unit. In other words, the performance bottleneck has shifted from disk I/O to computation [9]. Accordingly, performance can increase on in-memory platforms by using appropriate and faster processors.

GPUs have been introduced as an appropriate choice for data analytics and High-Performance Computing. They are massively parallel processors with high processing power and high memory bandwidth. GPUs can process data multiple times faster than CPUs. A large number of simple, small, and efficient cores enable GPUs to perform repetitive and similar operations rapidly, with high throughput. Due to their architecture, GPUs are the proper choice for running machine learning, deep learning, and real-time data analytics applications [9].

Given that in-memory platforms are currently CPU-based and use CPU to perform their computations, adding GPU-support could be an excellent approach to remove their computation bottleneck. By doing so, users on these platforms will not only benefit from the high speed of processing in memory but they also will take advantage of the processing power of GPUs. Previous studies have shown that using GPU's processing power in in-memory platforms such as Spark, Flink, and Storm [10] has considerably accelerated their execution.

Manzi et al. [11] examined the feasibility and benefits of offloading some of the spark core operations to the GPU. They ported several iterative and non-iterative applications to the GPU, whose results showed about 17X Speedup for the K-Means clustering algorithm. For other algorithms, like WordCount and RadixSort, there was a marginal speedup due to data conversion bottleneck, which is the task of converting data into GPU-friendly format.

HeteroSpark [12] was another GPU-accelerated architecture that used GPU's processing power for data and compute-intensive operations, which allowed the applications to use the GPU beside the CPU. This platform showed better performance and energy efficiency versus Spark. HeteroSpark resulted in about 18X Speedup for machine learning workloads.

Rathore et al. [13] combined Hadoop MapReduce and Spark to provide an efficient and high-performance platform suitable for streaming processes. They used the GPU as a co-processor for real-time big data processes, which resulted in higher performance than its CPU implementation.

Asai et al. [14] provided an extension for IBMSparkGPU [15], which is a Spark-based framework that executes Spark tasks on the GPU. They tried reducing the number of data exchanges between the CPU and the GPU by eliminating redundant data transfers. The result of their work on a machine learning application was about 1.3X acceleration.

Spark-GPU [16] was a CPU-GPU hybrid platform that also tried to utilize the GPU in Spark. They examined various challenges of integrating Spark with the GPU, and provided some solutions. They introduced a novel type of RDD [17] called GPU-RDD that was suitable for use in the GPU's native memory. Spark-GPU was able to accelerate machine learning algorithms about 16.13X and SQL queries 4.3X compared to Spark.

G-Storm [18] was a parallel GPU-enabled system designed to process streaming data, in which the GPU was used for its high-throughput. The platform supports various data types and applications, and its overhead for data pro-

Table 1 A summary of related work

Related Work	Base Platform(s)	Utilized Processor(s)	WorkLoad(s)	Acceleration
Manzi et al.	Apache Spark	GPU	K-means Clustering Word Count Radix Sort	17X - -
HeteroSpark	Apache Spark	Hybrid CPU-GPU	Logistic Regression K-means Clustering	18X 16X
Rathore et al.	Hadoop/Apache Spark	Hybrid CPU-GPU	Stream	More than 6X
Asai et al.	Apache Spark	GPU	Logistic Regression	1.3X
Spark-GPU	Apache Spark	Hybrid CPU-GPU	K-means Clustering Logistic Regression SQL Queries	5.71X 16.13X 4.83X
G-storm	Apache Storm	GPU	Continuous Query Matrix Multiplication	7X 2.3X
GFlink	Apache Flink	GPU	SpMV	5.1X
Lunga et al.	Apache Spark	GPU	Deep Learning Inference	400X

cessing workloads is remarkably low. Their results showed that the platform achieved more than 7X throughput improvement on continuous query and 2.3X in the matrix multiplication application.

GFlink [19] is another distributed in-memory platform that utilizes GPU's memory bandwidth as well as its computation resources. To enhance the performance of this platform, they deployed various methods and introduced an efficient communication mechanism between the JVM[1] and the GPU. They implemented an adaptive locality-aware scheduling method that resulted in higher performance versus the CPU-based implementation of Flink.

The integration of in-memory platforms and GPUs have been utilized in other areas as well. Lunga et al. [20] used Spark and GPU processing power to process several thousand terabytes of satellite images. Using GPUs, their results showed an acceleration of about 400X in deep learning inference. Table 1 summarizes the related work.

Due to the importance of this issue, Nvidia is also officially adding GPU-support to Spark 3.0 for some data analytics, machine learning, and deep learning applications. [21].

Apache Ignite, as a CPU-based platform, is not capable of utilizing the GPU yet. Adding this feature, it can benefit from the high processing power of the GPU to accelerate its computations. Thus, we have considered adding GPU-support to existing Apache Ignite implementation. For this purpose, we have designed Ignite-GPU: a GPU-enabled in-memory computing architecture on clusters. The proposed platform enables Ignite to utilize GPUs to speed up applications that are data or compute-intensive.

In this paper, we have examined various ways of integrating GPUs with Ignite, and have introduced multiple techniques for better utilization of GPUs. We have chosen the Genetic Algorithm as a workload to examine the proposed techniques' effects on performance. Experimental results show a considerable performance improvement of proposed work in the Genetic Algorithm, with the presence of GPUs.

This paper is the first attempt that utilizes GPU's processing power in Apache Ignite. Also, some of the techniques and optimizations used in this

[1] Java Virtual Machine

paper have not been adopted in any of the related work, and the presented techniques may be applicable to other platforms as well. It should be noted that it is not possible to compare the proposed work with the related work, because they are not open-source; in addition, none of them officially support the Genetic Algorithm.

Our main contributions are as follows:

- We have examined the use of GPUs in Apache Ignite and identified challenges ahead.
- We have come up with innovative solutions for integrating Apache Ignite, or other distributed in-memory platforms, with GPUs.
- Based on Ignite, Ignite-GPU has been designed and implemented, which addresses all existing challenges and utilizes the GPU through Apache Ignite efficiently.
- Various APIs[2] were provided for utilizing and easy use of GPUs on Ignite.
- Multiple optimization techniques have been applied to enhance the performance of Ignite-GPU.
- Finally, We have compared the performance of proposed work with Ignite and have evaluated the results of each of the optimizations.

The rest of the paper is organized as follows. Section 2 provides the necessary background to read the paper and summarizes the architecture of the GPU and Ignite. Section 3 describes Ignite-GPU, which first outlines the goals and challenges ahead, and ultimately describes the solutions and optimizations. Experimental results are presented in Section 4, and Section 5 is the conclusion and future work of the paper.

2 BACKGROUND

In this section, we briefly describe the key concepts required to read this paper. To begin with, we discuss the Ignite's execution model, then describe the GPU architecture and the CUDA[3] [22] programming model.

2.1 Apache Ignite

Apache Ignite is a newly released open-source platform used for storing and processing large amounts of data, that can be distributed across the nodes in a cluster. GridGain Systems open-sourced Ignite in late 2014, then it was accepted as an Apache Incubator program.

The Apache Ignite architecture is based on storing data in RAM—in-memory computing, which causes a dramatic increase in processing speed, compared to disk-based processing engines. Ignite provides an easy-to-use interface for developers to process a vast amount of data in real-time. Data in

[2] Application Program Interface
[3] Compute Unified Device Architecture

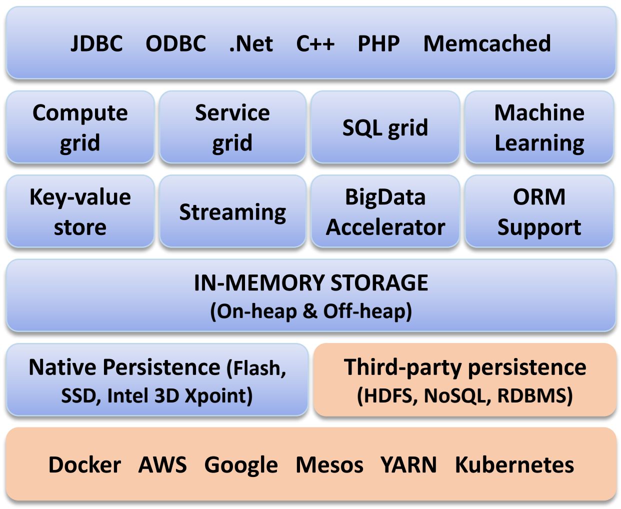

JDBC ODBC .Net C++ PHP Memcached
Compute grid
Service grid
SQL grid
Machine Learning
Key-value store
Streaming
BigData Accelerator
ORM Support
IN-MEMORY STORAGE (On-heap & Off-heap)
Native Persistence (Flash, SSD, Intel 3D Xpoint)
Third-party persistence (HDFS, NoSQL, RDBMS)
Docker AWS Google Mesos YARN Kubernetes

Fig. 1 Ignite main features. Source: Adapted from: [24]

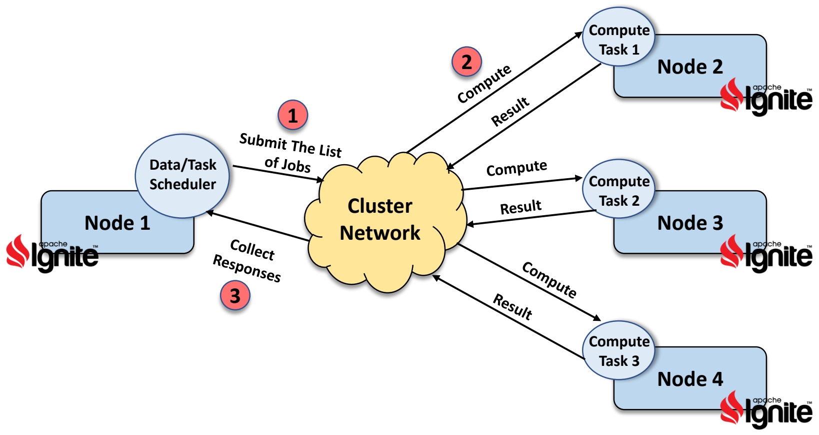


Fig. 2 Ignite Compute Task procedure. Source: Adapted from: [25]

Ignite is stored as key-value pairs on distributed caches, and each node of the cluster can have its own partition of the data. Furthermore, Ignite automatically rebalances the data while adding or removing a node from the cluster. All of these transactions in Ignite are ACID [23]. Not only can it be run in standalone mode, but it also has the ability to be deployed in the cloud, containerized, and provisioning environments.

As it is illustrated in Fig. 1, the main features of Ignite are Data Grid, Compute Grid, Service Grid, SQL Grid, Bigdata Accelerator, Streaming Grid, Machine Learning, Third-party persistence store, and ORM support. All features are provided for application developers in a variety of APIs, enabling

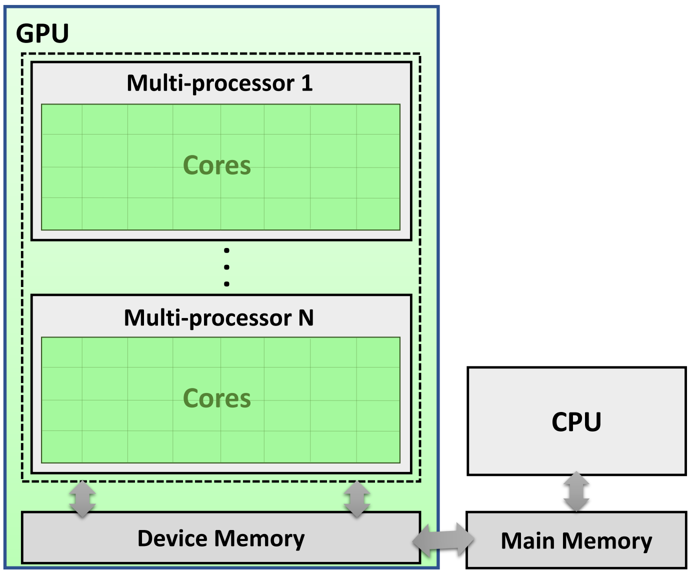


Fig. 3 The GPU architecture model

them to produce their desired applications. It should be mentioned that all of these APIs are running on the CPU.

Fig. 2 shows the procedure of a computing task in Ignite. When an application developer assigns a task to a node, the node acts as the master (in this figure, node 1 is master), partitions the input data, and delivers the data alongside an executable task to each node through the network (Step 1 and 2). This executable job and its corresponding data are called a Compute Task in Ignite's environment. Then, while each node completes its share of computation, the results are returned to the master (Step 3).

It should be noted that the data is distributed among nodes through Ignite's in-memory caches. Application developers can set these caches to be shared among all (or some) nodes or be exclusive for each node, based on their application and its requirements. Ignite manages the accesses to these caches automatically, and users are not involved in these issues.

2.2 GPU, CUDA, and JCUDA

GPUs are used as powerful co-processors alongside CPUs and are utilized in general-purpose computing due to their architecture, programming model, high performance, and energy efficiency. One of the applications of GPUs is high-performance data analytics [9] that they are used to address the insatiable desire for faster data processing and computation. Because of their massively parallel processing capabilities, GPUs are able to process data with much more throughput than the CPU, making them a good candidate for compute-intensive operations and iterative algorithms. Due to their high processing speed, the GPUs are also being used for real-time processing [9, 18].

As Fig. 3 demonstrates the GPU architecture, they are consist of a large number of cores grouped as multi-processors. Based on a scheduling policy, all of the cores of a multi-processor execute a similar operation on different data. This architecture is called Single Instruction Multiple Data (SIMD). All of these multi-processors fetch their corresponding data from the device memory, directly connected to the main memory. Not only the device memory has very high bandwidth, but it also exhibits low latency.

Threads in the GPU programming model are categorized as thread grids, each of which contains multiple thread blocks (thread groups). While each device is the unit of resource allocation to thread grids, multi-processors are the unit of resource allocation to thread blocks. These thread blocks contain smaller units, named warps, that are dynamically scheduled on multi-processors by warp schedulers. Processing resources like registers and shared memory are shared between threads in a thread block. It should be remarked that threads reside on the GPU (named device threads) and on the CPU (named host threads) are different. Device threads are more lightweight, and their creation and context-switch costs are considerably lower than those of the host threads.

GPU occupancy is one of the most critical factors, having a significant effect on GPU's performance, defines as the ratio of active scheduling units to the maximum number of available ones. Depending on the GPU's compute capability, its active scheduling units should be maximized, and its processing power should be fully utilized to gain more remarkable performances.

GPU programs are typically written in CUDA or OpenCL [26]. The NVCC[4] [27] translates the CUDA code into two parts: one for running on the CPU, and one for running on the GPU. The kernel is referred to that part of the code that runs on the GPU. Then, the NVCC compiler converts the GPU part to PTX[5] [28]—which is a pseudo-assembly language—and eventually, the graphics driver turns it into binary code, which runs on the GPU cores.

To exploit the GPUs' resources like CUDA kernels in Java applications, we need a communication bridge that can establish a connection between CUDA and Java. Two of the most famous options are JCUDA [29] and JNI[6] [30]. Although JCUDA has a higher development complexity, it has a better performance than JNI [31]. Containing a binding to CUDA, JCUDA enables us to load and execute CUDA kernels in Java programs. In addition, it provides facilities for allocating memory on the host and the device and transferring data between them.

Typically, to run a task on the GPU, one needs doing the following three steps in consequence:

1. Copying the kernel required data from the host to the device memory.

[4] Nvidia CUDA Compiler
[5] Parallel Thread Execution
[6] Java Native Interface

2. Launching the kernel.
3. Returning the result data from the device to the host memory.

Transferring data back and forth between the host and the device memory is costly and can result in performance degradation. Gregg et al. [32] showed that the required time to perform a particular task on the GPU, considering the data transfer time can be 2 to 50X greater than there is no data transfer. Hence, one should try to eliminate unnecessary communications at design time, to gain more performance.

3 DESIGN AND ARCHITECTURE

Firstly, this section discusses the design goals and challenges ahead in the integration of GPUs with Apache Ignite. Then, the design that addresses the problems comes afterward.

3.1 Design Goals

In this design, we have tried to pursue the following goals:

- Feasibility: One of our most significant goals is to show that Apache Ignite can be integrated with GPUs successfully. Therefore, Ignite can use GPUs alongside CPUs, as co-processors, to accelerate its processes.
- Performance: One of our other primary goals is to achieve higher performance compared to Ignite's CPU implementation to accelerate the processing of user applications.
- Flexibility: This platform should allow developers to implement their custom applications on the GPU, and adjust the parameters of their applications, according to their requirements.
- Ease of Use: The provided platform needs to be easy to use so that the application developers can use it effortlessly. For this purpose, we need to implement suitable APIs to provide the desired functionalities in working with GPUs.
- Portability: Application developers should be able to deploy this platform on any machine, equipped with a General-purpose GPU (GPGPU), without any particular changes.
- Scalability: Because Ignite is a distributed platform, and can scale on thousands of nodes, its GPU version must also be able to distribute on any number of nodes.

3.2 Challenges

Ignite applications—which are primarily data analytics, exhibit iterative behavior, and are suitable for running on GPUs. However, there are challenging obstacles in integrating GPUs into Ignite. Addressing these challenges efficiently will result in dramatic performance improvement and better utilization of existing processing resources. We explore these challenges in the following.

- *Data Conversion Overhead:* Data in Ignite is stored as JVM objects, which are unsupported by GPUs. Before sending it to the GPU, the data should be converted to a GPU-friendly format. This conversion consumes much time, which may even cause speed-down [2].
- *Data Type Support:* Supporting all available data types on GPUs requires excessive programming efforts, including design, development, and test. In fact, for each data type, a new GPU kernel and data transfers are required.
- *Data Transfer:* Frequent data transfers between the host and the device are time-consuming and cause performance degradation. To cope with this challenge, we should omit redundant data transfers and keep the working set on the device memory as long as possible. Minimizing these unnecessary data exchanges, dramatically increases performance [14].
- *Insufficient device memory:* According to user applications, the data volume may be larger than the size of the device memory, which leads to program crashes or data losses. Therefore, because of the limited capacity of the device memory, a couple of dynamic pre-checking tests should be performed.
- *Garbage Collection:* In some iterative algorithms, a particular kernel might be launched repeatedly with various input data. These recurring launches of the kernel associated with frequent memory allocations and deallocations for the data transfers. Our studies have shown that these allocations and deallocations are also time-consuming.
- *Utilizing the GPU:* GPUs possess high processing power, and they would be advantageous if only their resources are adequately utilized. It is essential for the platform to use the maximum available processing potential of GPUs.
- *Memory Coalescing:* Due to the GPU architecture, one of the key factors affecting performance is how threads access the memory. When this access is coalesced—consecutive GPU threads access consecutive memory units (sequential access), performance will be better.

3.3 Ignite-GPU Overview

Ignite-GPU is an in-memory architecture, enabled to operate in a heterogeneous cluster of CPU-GPU. This framework delivers all benefits of standard Ignite; moreover, it can handle some Ignite applications that do not respond in a reasonable time. In our design, we have tried to overcome all the mentioned challenges and achieve the desired goals. On this platform, GPUs have been used to execute data-intensive and compute-intensive applications. So, Ignite application developers can utilize the GPU alongside the CPU. It should be noted that the provided platform is designed with the assumption that there is only one GPU on each node.

Ignite includes multiple possible execution scenarios. One of which is as follows: first, the master node delivers required initial data alongside a process to the workers. Next, each worker receives its data, processes it, and returns results to the master. Ultimately, every node waits for the other nodes to synchronize. If the application is an iterative one, this procedure is repeated. Similarly, Ignite-GPU follows this proceeding with some differences: When the data reach the workers, they transmit the data to the GPU instead of processing it on the CPU. On each node, the GPU performs the corresponding processing and pushes the results data to the Ignite associated caches to be sent to the master for synchronization.

Since Ignite is developed with Java, to fulfill the need for establishing a connection between Java and CUDA, we used JCUDA. JCUDA is a programming interface that can be used in Java programs to invoke CUDA functions, including user-defined kernels. Using this interface, programmers can directly call CUDA kernels and transfer data between host and device in their Java codes without being worried about the technical details of bridging between Java runtime and CUDA runtime [33]. In the following, we describe the solutions proposed for each challenge.

The first problem was the *Data Conversion Overhead*, which was the cost of converting JVM objects to a GPU-friendly format. In the cases that we need to access the data itself on the GPU, we should pay off the cost of converting the objects to GPU-friendly data types. In this situation, first, each node converts the received data, before forwarding it to the GPU, into the array format—or any appropriate form. Then, the required space should be allocated on the device (which JCUDA provides this possibility). Finally, the data should be sent to the GPU for launching the kernel.

Apart from converting data to the appropriate format, we need to match the kernel input data type to the data sent from the host. One naïve solution is to implement a separate kernel for each data type which requires a lot of programming efforts and reduces code reusability. To overcome this problem, called *Data Type Support*, we proposed a data transfer mechanism. The data, regardless of its type, on the host is converted into a byte array and sent to the GPU. On the device side, we have implemented a union called `data_unit` that extracts the data with the desired type from this byte array. This data can represent the defined types of `data_unit` union like character, string, integer, long, and double. As demonstrated in the following, this union is defined in the *kernels* file and supports the mentioned types of data.

```
typedef union {
    char c;
    int i;
    double d;
    long l;
    char s [SIZE];
 } data_unit;
```

By retrieving the desired data type from the union, the compiler does data conversion automatically and fetches that type of data. Using this mechanism, there is no need to implement a new kernel to support each data type, and regardless of the data type of the application, we can support it with the same kernel.

The subsequent challenge ahead is transferring the data between the host and the device, which Ignite-GPU handles it efficiently. Without the involvement of application developers, Ignite-GPU converts the data to the appropriate format, sends it to the device, and returns the results to the host after kernel execution. Efficient data transfer and reducing unnecessary data copies can have a significant impact on the performance; therefore, we have tried to omit redundant copies and keep the in-use data on the device memory as long as possible. We have employed a technique named *bottom-up integration* to eliminate redundant data transfers—that will be described further in the current section.

The next issue is insufficient device memory. In this platform, each node computes the size of the data before sending it to the GPU. If the data size is bigger than the device memory capacity, gives a warning to the user and asks whether to divide data into smaller chunks. If the user accepts, the data will be broken down into a fitting power of 2, prior to sending it to the GPU.

Due to the inherent characteristic of Ignite applications—that are mostly iterative, they may cause *Garbage Collection overheads* for their kernel calls. Therefore, we have managed device and host memory allocations/deallocations within the whole program. In Ignite-GPU, each cluster node has the duty of allocation and deallocation for its data on the host and device. Considering that most of Ignite applications have iterative behavior, a couple of allocation-deallocation is needed for each kernel call in every iteration. In Ignite-GPU, all of the required data spaces are allocated at the beginning of the program. They are reused during the iterations, and all of them are released before the program's termination. It reduces the overheads of frequent memory operations for kernel calls in iterative algorithms and improves the performance.

One of the other important challenges is *Utilizing the GPU*. Ignite-GPU provides this flexibility to application developers to adjust the program's parameters based on their needs and their GPU architecture. Application developers can set all of the GPU associated parameters like *blockSize*. By setting these parameters due to the GPU hardware specifications, the GPU resources can be more utilized.

The final challenge is *Memory Coalescing*, which is related to how data is accessed on the GPU. In this platform, all memory accesses are coalesced, which means all GPU threads have serial access to the memory, and no memory access conflict occurs.

Considering the requirements of using the GPU, we have provided various APIs, including initializing the driver, creating context, and module loading (which loads the PTX file) on each cluster node. Also, there are specific APIs for launching a kernel and transferring data from host to device and vice versa.

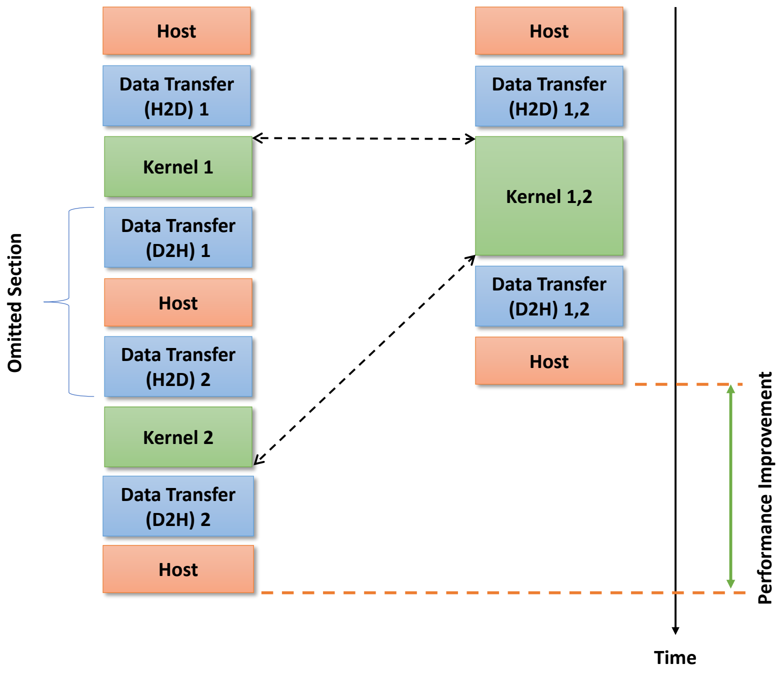


Fig. 4 Bottom-up consecutive kernels' integration procedure

To provide more flexibility for application developers, they can specify their desired existing kernel name to access it through their program. Moreover, they can personalize the existing kernels or implement new ones in the *Kernels.cu* file and call them.

To port an application to the GPU, initially, it is necessary to analyze the application carefully and identify its time-consuming parts. To that end, in the first phase, we profile the application, identify its time-consuming modules, and try running those modules separately on the GPU. Calling each kernel needs data transfers between the host and device, and takes much time and leads to performance degradation. In the second phase, we tried to integrate these separate kernels and maintain the desired data on the device, instead of consecutive copies between the host and the device. By implementing this feature, all the separated kernels are performed seamlessly on the GPU, and the output of every kernel will be the input of the next one. Consequently, the data is once copied onto the device and once it is copied back to host.

Fig. 4 demonstrates two execution timelines and their integration procedure. As can be seen, there are two kernels on the left side that each of which needs two data transfers (One for the host to device and one for vice versa). The data will not remain on the device between two consecutive kernel executions, and it causes two redundant data transfers between them. After execution of the first kernel, the data is copied to the host and then copied back to the device memory to run the second kernel. In contrast, on the right timeline, two redundant data transfers are omitted, and the data remains on the GPU between two kernel executions. The first kernel passes its data directly to the second kernel, and both kernels are executing in succession. By doing so, many redundant data transfers are omitted which causes a remarkable performance improvement.

Similar to Apache Ignite, Ignite-GPU is scalable and can operate on any number of nodes on a cluster. In this platform, each node is responsible for initializing and managing its GPU. At the beginning of the program, the master node broadcasts *Kernels.cu* file to all nodes using IGFS (Ignite File System). This will make the application developer needless to copy the kernels file on each node manually. Even, by modifying the kernels file in the master, all nodes will have the same kernels. After receiving the file from the master, each node generates a PTX file based on its CUDA runtime version to run it on its device(s). If the CUDA versions or GPUs models vary in other nodes, this method will be more reliable, ensuring there is no problem in execution. Besides, the generated PTX will be optimal for running on the underlying hardware.

3.4 Optimizations

We have implemented various techniques to improve performance and measured the impact of each of these methods, which will be described in the following.

- Using CUDA Streams: Due to dealing with large amounts of data in this platform, copying data between host and device is extremely time-consuming, and GPU cores will be idle during this copy operation. To address this problem, we use CUDA streams, which is a technique to launch multiple operations asynchronously on GPU. In this way, data-transfer and computation phases are overlapped. Commonly used for large data volumes, this method results in more efficient use of the GPU and increases performance. For more optimization, we have only created streams once at the first iteration of the program, and in the next kernel calls, we use the previous streams. Memory is pinned here to prevent the Operating System from swapping out the memory pages as a part of the Virtual Memory management system. Ignite-GPU handles the creation and management of the streams automatically.
  Also, we have provided this flexibility for the application developers to use the streams optionally. By setting a Boolean flag in the configurations of the program, they can specify whether they want to use streams or not, and they can also determine the number of streams. The number of streams can be set according to the data size and problem type.
- Using Global/Shared Memory: Global memory is available for all GPU threads, but shared memory is only shared between the threads inside a thread block. Shared memory is commonly used for applications that have

more data reuse. It has a smaller size and faster access rate than the global memory. We allow application developers to use shared or global memory on demand by setting its corresponding configurations.

- Using Constant Memory: Constant memory is an excellent candidate to be used in the applications that we need to store static and constant data on the GPU, and frequently access it. This memory is read-only by the GPU threads and has a small size. Application developers can utilize constant memory in Ignite-GPU based on their application's requirements to gain more performance.
- Copying Data indexes: Ignite stores the data in the key-value format in its cache. Depending on the application's type, in some cases, we can copy the data indexes to the GPU instead of transmitting data itself. While each data is mapped to a unique key in the cache, there is chance of working on data indexes or reproducing the data from its indexes on the GPU side. Not only does it cause a substantial reduction in the amount of the sent data to the GPU—resulting in faster data transfer, but it also makes the data independent of its type. That is, regardless of the data type, we can perform our operation on the GPU, and the Data Conversion is somehow evaded.

4 EXPERIMENTAL RESULTS

In this section, Ignite-GPU's performance is evaluated. We initially describe the experimental environment, then present the observed results.

Every experiment is performed ten times, and the reported results represent the average of these ten experiments. Besides, all the reported results are the average execution time of one generation of the Genetic Algorithm, which is equivalent to the execution time of an iteration of the algorithm.

4.1 Experimental Environment

We performed our experiments on a cluster with 4 GPU-equipped nodes. Each node is a XenServer VM and has 16 dedicated CPU cores with 2.0 GHz Intel Xeon E5-2620 and 30 GB of memory. Moreover, each node has an NVIDIA GeForce GTX 680 GPU with 1536 CUDA cores and 1.12 GHz clock rate, which has 2 GB of memory with 3.0 GHz memory clock and 173 GB/s of memory bandwidth. The CUDA version is 8.0, and the installed operating system on all nodes is Ubuntu 16.04. Our architecture is based on Apache Ignite 2.7.0.

4.2 Workloads

The Genetic Algorithm, which is officially supported by Ignite, can be considered as a good candidate to be evaluated in this platform and run on GPU through Ignite. The computational jobs in most Ignite applications are modelled as Compute Tasks and they are designed based on this concept. Similarly, Ignite's Genetic Algorithm is based on this concept, so it can be a good representative of many other Ignite applications. Moreover, due to its iterative behaviour, it is similar to other Ignit's machine learning algorithms in this respect, so the provided solutions for this algorithm would be applicable to other machine learning algorithms. On the other hand, the Genetic Algorithm is compute and data-intensive and its time complexity grows with the increase of population size, that these types of applications are suitable to run on GPUs.

In the following, we first describe the Genetic Algorithm and then explain how this algorithm executes on Ignite-GPU.

4.2.1 Genetic Algorithm

Genetic Algorithm represents a subset of Ignite machine learning APIs and is suited to find the optimal solution in large and complex datasets. This algorithm is deployed in many real-world applications like automotive design, computer gaming, robotics, investments, traffic/shipment routing, etc. Genetic Algorithms consist of four main stages: selection, fitness calculation, crossover, and mutation. This algorithm initially produces an initial population that is a large set of possible solutions (chromosomes)—and each chromosome is made up of genes. In each iteration, it selects a set of best solutions based on an evaluation criterion and performs mutation and crossover operations on this subset to produce better-fitted chromosomes for the next generation. This procedure is iterated until it approaches the optimal solution [24]. The Genetic Algorithm here uses Roulette Wheel for the selection, as well as Single Point crossover and Swap mutation.

In Ignite, most of the computational operations are considered as a Compute Task and will be sent to the nodes to run. The initial population generated randomly is placed in the *population cache* and partitioned between nodes. *Gene Cache* holds all possible genes and gives each node a copy. The overall scheme of this algorithm in a distributed environment on Ignite is shown in Fig. 5. As shown, Each node operates and performs Genetic operations on its partition of data and at the end of each iteration, returns its results to the master node.

The problem we focused on to solve by the Genetic Algorithm, is a character matching problem that tries to reach the "HELLO WORLD" string from alphabet letters. In this case, *Gene Cache* is filled with A to Z and Space characters, and the optimal solution is "HELLO WORLD." Besides, a fitness score is used to measure the optimality of each solution. During each iteration, the algorithm evaluates newly generated chromosomes—each of which is a string with a length of eleven, by calculating their fitness. The Fitness criteria is the similarity of each chromosome to the "HELLO WORLD" string. In this problem, the chromosome length is 11 and is equal to the length of the "HELLO WORLD" string.

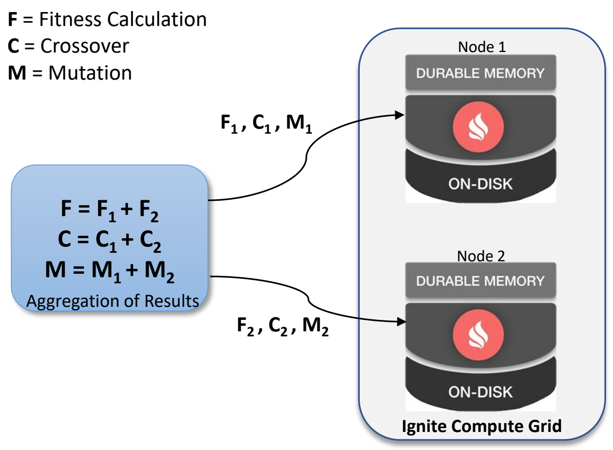


Fig. 5 Genetic Algorithm execution model in Ignite. Source: Adapted from [24]

4.3 Our Methodology

To implement the Genetic Algorithm on Ignite-GPU, in the initial implementation phase, we first analyze the algorithm. Profiling results demonstrate that the most time-consuming parts of the algorithm are the main functionalities of the Genetic Algorithm—selection, mutation, crossover, and fitness evaluation. Next, we try performing each of these time-consuming functionalities independently on the GPU (i.e., four GPU kernels). To implement any program on the GPU, it is essential that one sends data in batch to the device memory. Ignite sends chromosomes one by one to all nodes for processing, so we have to transform it to batch-like data partitioning. Making data partitioning policy appropriate to the GPU execution model reduces data partitioning overheads and the number of Ignite's cache accesses, and also eliminates the overheads of creating a Compute Task for each chromosome.

Fig. 6 demonstrates our naïve implementation of Ignite-GPU's execution time versus Ignite. This initial implementation shows about 1.3X speedup for population size 1K, 16X for 10K, 296X for 100K, and 2859X for 1 million in comparison to the standard version of Ignite. In other words, speedup increases with an increase in population size. For larger input data, our naïve implementation showed better performance in comparison with another state-of-the-art CUDA implementation of the under-study problem that gained about 1500X speedup for 1 million chromosomes [34]. Furthermore, the current Ignite implementation of the Genetic Algorithm has a time complexity that grows too much for larger population sizes. As demonstrated, it does not progress with population sizes larger than 1M, so it is not reasonable for one to use Ignite's implementation of such applications.

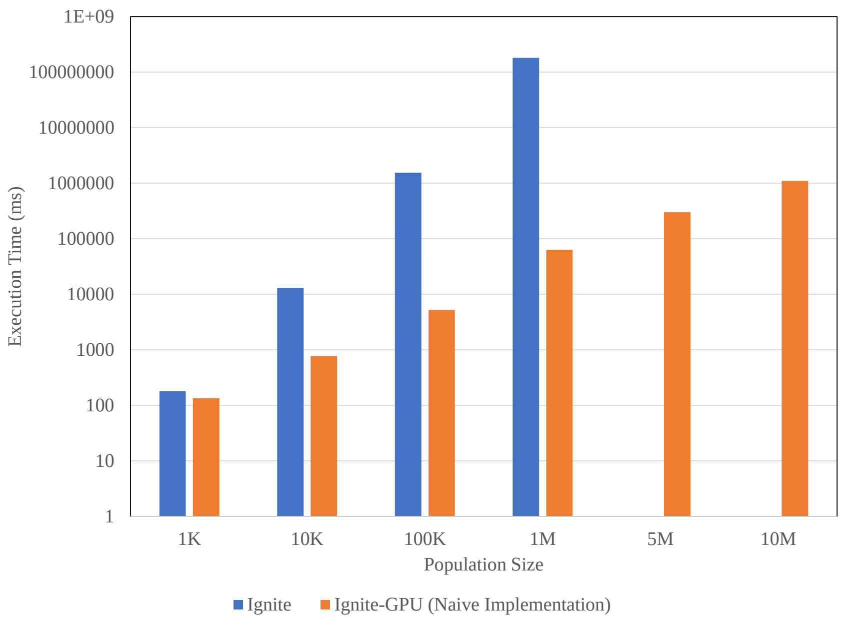


Fig. 6 Performance of Genetic Algorithm on Ignite and the naïve implementation of Ignite-GPU on a single node. (Block size: 32)

While performing the four operations of the Genetic Algorithm separately on GPU, the data is frequently transferred between host and device. The main bottleneck that restricts gaining more performance is the overhead of these frequent data transfers. To resolve this problem, we integrate these four operations—selection, crossover, mutation, and fitness evaluation, using our bottom-up integration technique and implement all of them as one kernel. This technique eliminates seven data transfers between host and device. That is, the data is copied into the device memory and returned back to the host only once. Fig. 7 illustrates the scheme of this integration procedure for the Genetic Algorithm, and Fig. 8 represents the effectiveness of this integration. As can be seen in this figure, for all population sizes, the bottom-up kernels' integration caused about 3X acceleration in comparison to the naïve implementation.

Each stored chromosome in *Population Cache* is considered as a JVM object, and we need to convert it to a suitable format and data type to be used in the GPU. For this purpose, each node copies its *Population Cache*'s data into an array in sequence, then passes the array to the GPU. The array can have the pre-determined data types of `data_unit` union.

The nature of the distributed Genetic Algorithm is such that each node must send its data to the master node at specific time intervals (at the end of each iteration) for synchronization. So, the master node receives the results of all nodes at the end of each iteration. These results include the chromosomes in which crossover and mutation operations are performed on them,

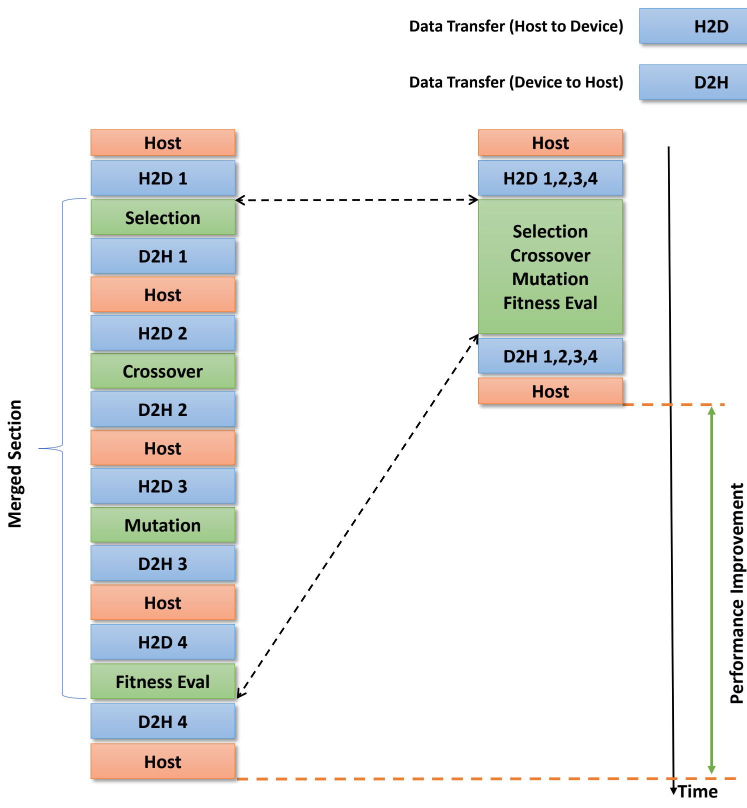


Fig. 7 The bottom-up integration scheme for the Genetic Algorithm

and their fitness is calculated. Master sorts the results according to their fitness and merges them. If the algorithm does not reach the optimal solution, the subsequent iteration begins.

To run all Ignite-GPU instances with the same arbitrary configuration, we have created another shared Ignite cache, in which we broadcast all the required settings needed for utilizing the GPU, such as using streams and their number, using shared memory, and so on. Application developers can specify them at the beginning of the program. They can also modify the *Evaluation* device function, which is responsible for fitness assessment based on their type of problem, and write their fitness assessment criterion. We have delivered the capability, which allows users to use shared memory based on their problem type. However, our results showed that using shared memory in the HELLO WORLD problem is unbeneficial.

Assigning fair work to the GPU threads can improve performance by better utilization of GPU. Accordingly, we assign one chromosome to each thread to process and operate on it. By doing so, we eliminate the need for some

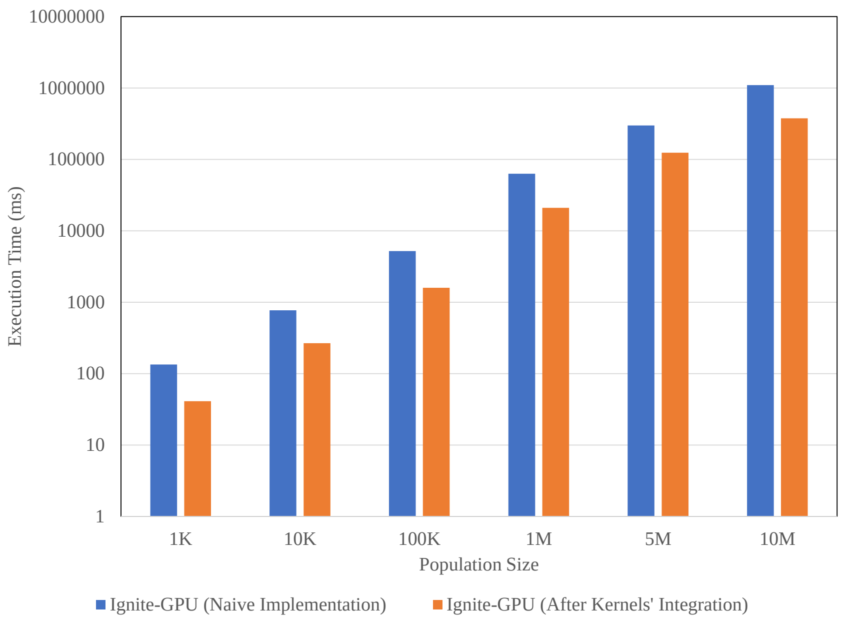
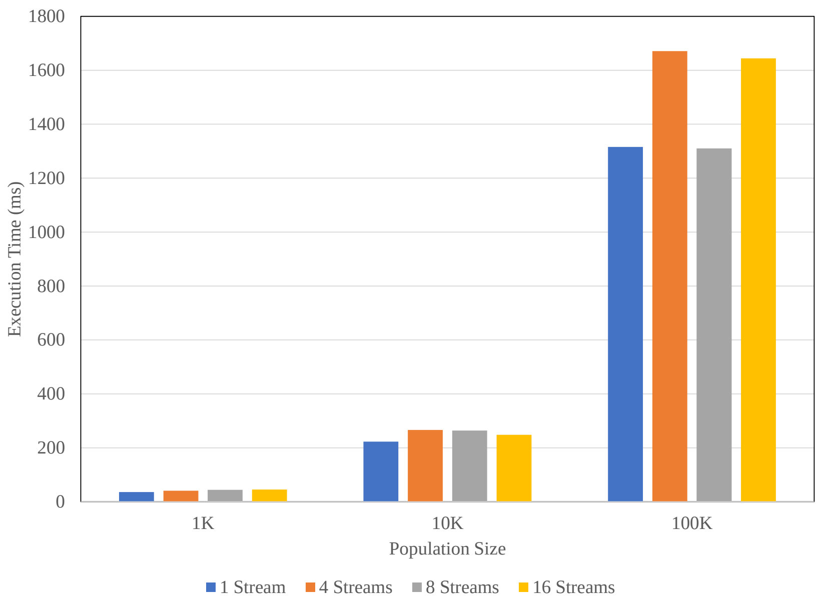


Fig. 8 Performance of the Genetic Algorithm on Ignite-GPU, before and after bottom-up kernels' integration procedure on a single node. (Block size: 32)

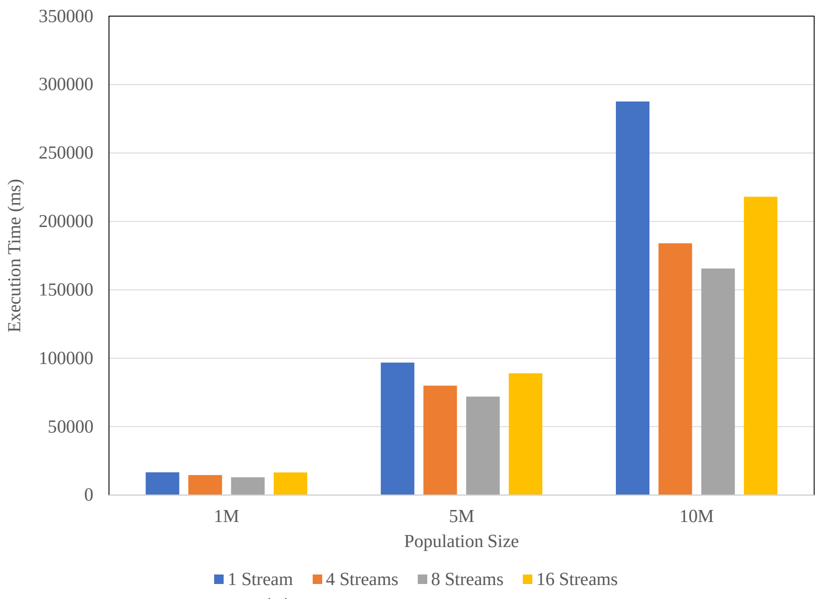
syncthreads(), which is a performance bottleneck, and resolve the memory coalescing challenge.

One of the other optimizations applied to this application is the use of CUDA streams, which enables users to determine the number of streams based on the size of data. By using streams, performance improves about 1.8X. Naturally, the number of streams can vary slightly depending on the size of the input data and application type. In some situations, data is not divisible by the number of streams or block size. To overcome this problem, we add padding to the end of the last data partition. This padding contains a copy of the most valuable chromosomes that helps the Genetic Algorithm to converge in fewer iterations.

Fig. 9a and Fig. 9b evaluate the effect of using CUDA streams on performance for different numbers of streams and population sizes. As illustrated, for small population sizes, using streams is not beneficial, and using one stream, which is equal to do not using streams, shows a better performance. In other words, utilizing streams for smaller data sizes is unbeneficial because the creation, destruction, and division of data between streams cause some overhead. However, for large populations, performance improves by using streams. As illustrated, the execution time by using four, eight, and sixteen streams is fewer than no-stream (one stream) mode for large population sizes. Moreover, utilizing sixteen streams shows more execution time compared to eight and four streams due to the overhead. Thus, there should be a tradeoff between



(a) For small population sizes.



(b) For large population sizes.

Fig. 9 Performance of Ignite-GPU with streams and without streams on a single node (Block size: 32)

the number of streams and the size of population, and it is on the user to choose the appropriate number of streams based on their application.

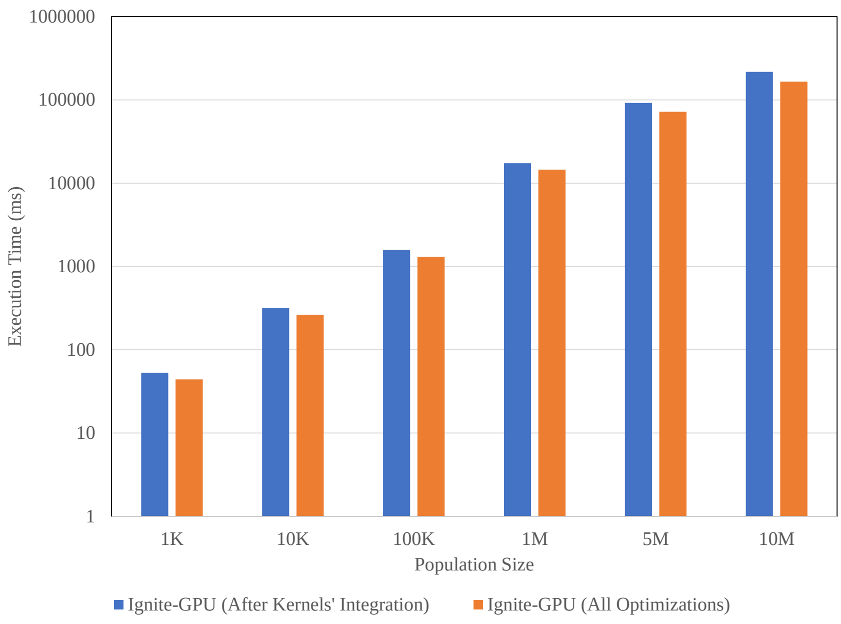


Fig. 10 Effect of using constant memory and removing Garbage Collection overheads on performance on a single node (Streams: 8, Block Size: 32)

Finally, by applying other optimizations, such as using constant memory and removing garbage collection overheads of device/host pointers, the results show about 31% performance improvement for 10M population size over previous optimizations. Fig. 10 compares the cumulative performance of using constant memory, removing garbage collection overheads, utilizing 8 streams, and applying bottom-up integration with the time there is garbage collection overheads and no constant memory. Although the performance improvement of using constant memory and removing garbage collection overheads is marginal, they indeed affect performance, and it is better to apply these optimizations.

On top of that, data in the Ignite's cache is stored as key-value pairs and is distributed within the cluster. For example, in the HELLO WORLD problem, each character (gene) is stored as a couple of (long, object) pairs in the cache. A long key can be used to retrieve an object from the cache and to convert it to the desired data type. There are some cases that instead of dealing with the JVM objects on the GPU, we can operate only on its long indexes if there is a one-to-one relationship between key-value pairs. In these types of problems, keys can be used instead of the actual data to increase the data access rate. This also results in reducing the data size, which diminishes data transfer overhead and occupies less volume of device memory. Moreover, the kernel is independent of the data type, and there is no need to use `data_unit` union anymore.

Although this idea can be applied to the crossover, mutation, and selection phases of the Genetic Algorithm, it may not be applicable to other phases (e.g., fitness evaluation) that need access to the actual data. In these situations, data can be retrieved from its key on the device, using a pre-constructed hash table on the GPU. This hash table has to be created at the start of the program on the device memory of each client; then, it can be used for retrieving data from their key.

Algorithm 1: Pseudo-code of Ignite-GPU for Genetic Algorithms on the master node.

```
1  //Genetic Algorithm data structure and details:
2  $ChromosomeLength \leftarrow L$
3  $GenesPopulation \leftarrow$ {List Of Genes}
4  //Execution configurations (e.g.):
5  $MaxIterations \leftarrow 100$
6  $PopulationSize \leftarrow 1000$
7  $CUDAStreams \leftarrow 8$
8  $GPUBlockSize \leftarrow 32$
9  $ComputeTask \leftarrow$ A Callable Method //To execute on workers
10 Configure in-memory data caches of the cluster
11 Configure Ignite File System (IGFS)
12 Initialize cache with initial population
13 Copy CUDA source file and other configurations to IGFS
14 $Iteration \leftarrow 0$
15 while $Iteration \leq MaxIterations$ do
16     Partition and distribute current population among the workers
17     Send $ComputeTask$ to all nodes
18     Wait for the results
19     Gather results from workers
20     Sort(results)
21     if $TerminationCriteria$ is met then
22         Log the optimal solution
23         break
24     end
25     $Iteration \leftarrow Iteration + 1$
26 end
27 Send $Terminate$ signal to workers
28 Clean up caches and IGFS
```

To sum up, Algorithm 1 and Algorithm 2 illustrate the process of running Genetic Algorithms on Ignite-GPU on both master and worker nodes. As it is presented in Algorithm 1, the application developer should specify their Genetic Algorithm's data structure, configurations, and other problem-specific details in lines 2 and 3. Next, in lines 5 to 8, users can adjust execution configurations like the maximum number of iterations, initial population size, and also some GPU-related parameters like the number of CUDA streams and block size. Line 9 is the process of creating a Compute Task, which is the desired computation (process), which will run on the worker nodes. Then in lines 10 and 11, required Ignite in-memory caches are configured, and in line 12, they are being initialized with a randomly-generated initial population.

In the 13'th line of the algorithm, the CUDA source code (Integrated kernels) is shared between nodes to run on their GPUs. Later in the While loop, the operation of data and computation distribution, and gathering the results is performed until it reaches the maximum number of iterations or finds the optimal solution. Finally, the master node sends the termination signal to all workers and cleans up the allocated memories and configurations.

Algorithm 2: Pseudo-code of Ignite-GPU for Genetic Algorithms on worker nodes.

```
1  Wait for a job from master
2  Receive the submitted configurations
3  Get CUDA source file from IGFS
4  Using JCuda, attach the CUDA binary code to the running JVM
5  Initialize the GPU and allocate required memory on host and device
6  if UsingCUDAStreams==TRUE then
7      Create and manage streams and corresponding events
8  end
9  while ComputeTask do
10     if Terminate signal is received then
11         Deallocate memories of the host and the device
12         Clean up configurations
13         break
14     end
15     Get the population from Ignite cache
16     Convert the population to ByteArray
17     Transfer ByteArrays to GPU using CUDA streams
18     Launch the GPU Kernel to perform:
       Selection, Mutation, CrossOver, and FitnessEvaluation
19     Transfer results from GPU to host using CUDA streams
20     Copy back the results to Ignite cache
21     Send task completion signal to master
22 end
```

Algorithm 2 presents the worker's code, which is the same for all workers. As can be seen, first, workers wait for a job from the master. As soon as they receive a job (line 1), they will receive its configuration, and the CUDA source code (lines 2 and 3). Then, workers adjust the settings for the GPU code (line 4), allocate the required memory (line 5), and create demanded streams (lines 5 and 6). Line 9 checks whether there is a Compute Task, and in lines 10 to 14, if the termination signal is not received, they start the main processing phases. They pick the required data from memory, convert it to Byte Array, copy the data on the GPU using streams, and launch the GPU kernel. Lines 19 and 20 are related to returning the results, which are the chromosomes that genetic operations are done on them, and push them into the cache. It should be noted that the master node is also a worker, and it is not idle during workers computation. It also waits for a job and performs the worker code.

To summarize this section, first, a naïve implementation of Ignite-GPU was developed in which each kernel was running separately on the GPU. Next, the kernels were integrated into one GPU kernel. Then, CUDA streams were used, and finally, some other optimizations were applied. The next section compares the performance of Ignite and the fully-optimized version of Ignite-GPU.

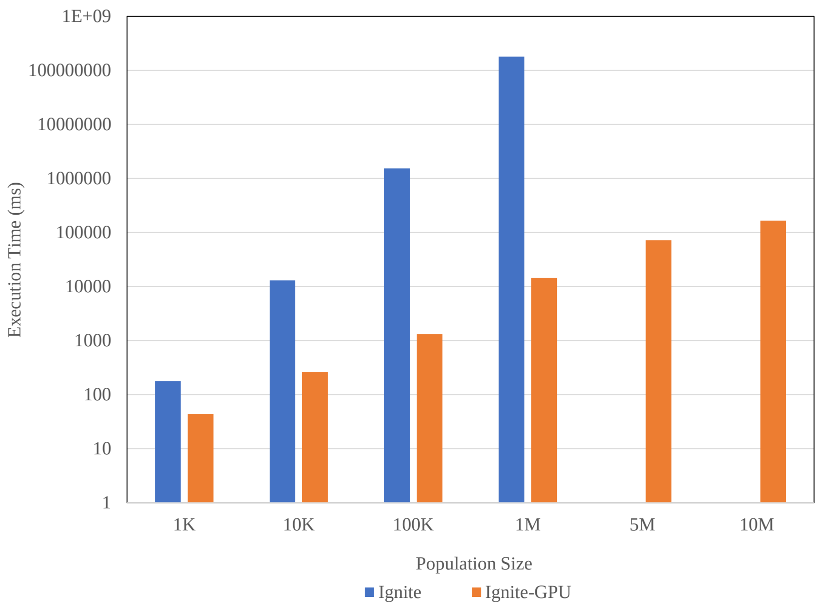
4.4 Performance Analysis

In this section, the performance of the provided platform is compared to the standard version of Ignite on one and four nodes. Ignite-GPU, in this section, is the fully optimized implementation of our platform.

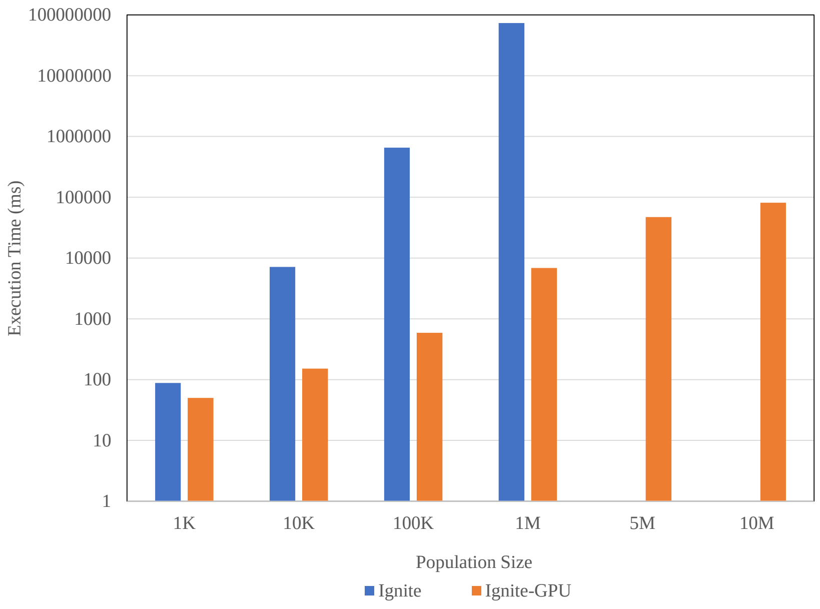
Initially, we run our experiments on a single node that the results are shown in Fig. 11a. As represented, speedup improves with the increase of the size of data (population size), and the speedup for 1K, 10K, 100K, and 1M population sizes are about 4X, 49X, 1177X, and 12391X, respectively. Although Ignite-GPU proceeds for populations larger than 1M, Ignite does not scale for large population sizes in a reasonable time, and it takes too much time to progress, so we cannot present its results for very large inputs.

In another setup, the same experiment has been done on a cluster with four nodes, and the results are shown in Fig. 11b. As can be observed, the speedup for 1K, 10K, 100K, and 1M populations reaches about 1.7X, 46X, 1110X, and 10698X, respectively. Similarly, Ignite does not progress for populations larger than 1M.

As described, the speedup on the cluster is slightly less than the speedup on a single node, which is due to the overheads of data conversion operations. In Ignite-GPU's cluster, each node should perform a data conversion operation before placing the data into the cache before forwarding it to the master, and before transmitting data on the GPU. However, as shown in Fig 12, for larger populations, the performance of Ignite-GPU improves while running on the cluster. This line graph compares Ignite-GPU's performance on single and multi-node with different population sizes. Although for 1K, 10K, and 100K, running on the cluster is not advantageous, for larger than 100K, the performance of the 4-node cluster is higher than a single node. The reason is that larger populations require more memory space and computation, so running on the cluster with more computational resources would be a better choice. All in all, based on the size of the input of the problem, it can be decided by the user whether to run on a single node or the cluster.



(a) On a single node.



(b) On the 4-node cluster.

Fig. 11 Performance of Ignite and Ignite-GPU. (Streams: 8, Block Size: 32)

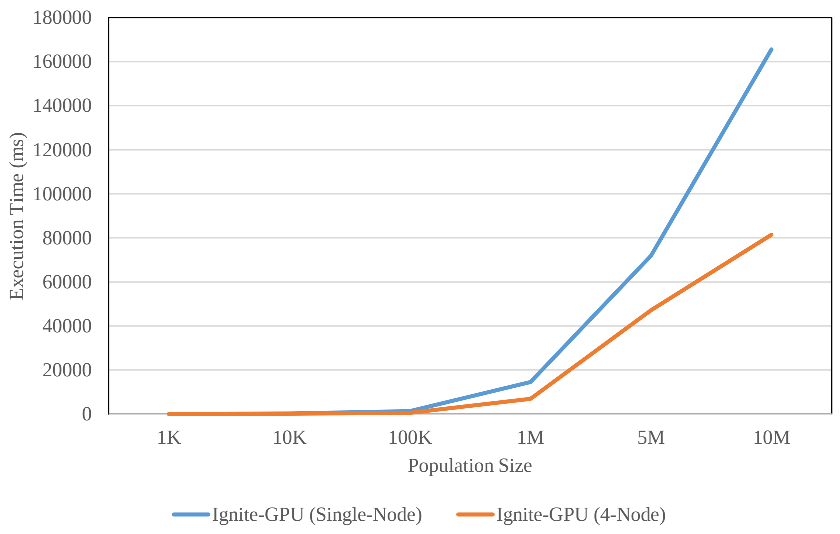


Fig. 12 Performance of Ignite-GPU on a single node and 4-node. (Streams: 8, Block Size: 32)

5 CONCLUSION

In this paper, we explored the use of GPUs as coprocessors in one of the distributed in-memory computing platforms—Apache Ignite. We tried to utilize the GPU's processing resources to gain more performance.

On the one hand, utilizing GPUs in Apache Ignite might be very useful, and by responding correctly to challenges ahead and using appropriate optimization techniques according to the application, a remarkable performance improvement can be expected.

On the other hand, there are some considerations that should be taken into account. First of all, utilizing GPUs in every context adds some costs, such as buying and maintenance of hardware. Second, writing code on the proposed platform requires knowledge about GPU and its programming model (CUDA), and also needs more programming effort. Besides, the presented work focuses on the Genetic Algorithm and provides the software infrastructure to run it on the GPU. Hence, running other algorithms on this platform requires much more programming effort.

To summarize, in this work, our goal is to add GPU-support to Ignite to accelerate the execution of time-consuming programs dealing with large volumes of data. We have investigated all the challenges facing the integration of in-memory platforms with GPUs and provided several solutions according to the under-study problem. Then we applied the proposed solutions to Ignite and introduced Ignite-GPU as a GPU-enabled distributed in-memory platform. Ignite-GPU showed about 12391X speedup on the Genetic Algorithm as a workload.

Future work we are interested in performing to improve our framework include the following:

- Ignite-GPU currently supports Genetic Algorithms, and we are studying larger and more time-consuming problems with various data types to implement on the provided platform in order to demonstrate its versatility.
- In addition to Genetic Algorithms, we plan to run other categories of Ignite applications on the GPU—such as streaming applications, employing the techniques presented in this paper.
- Being able to use the GPU as a co-processor in Ignite, we plan to divide the workload between CPU and GPU and utilize both of them simultaneously.

References

1. *Apache Hadoop.* `http://hadoop.apache.org/` [Accessed: 01-Nov-2019].
2. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
3. Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.
4. Vasiliki Kalavri and Vladimir Vlassov. Mapreduce: Limitations, optimizations and open issues. In *2013 12th IEEE international conference on trust, security and privacy in computing and communications*, pages 1031–1038. IEEE, 2013.
5. *Apache Spark^{TM} - Unified Analytics Engine for Big Data.* `http://spark.apache.org/` [Accessed: 01-Nov-2019].
6. *Stateful Computations over Data Streams," Apache Flink.* `http://flink.apache.org/` [Accessed: 01-Nov-2019].
7. Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.
8. *Open Source In-Memory Computing Platform - Apache Ignite^{TM}.* `http://ignite.apache.org/` [Accessed: 04-Jun-2020].
9. Eric Mizell and Roger Biery. *Introduction to GPUs for Data Analytics.* O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472., 1 edition, 8 2017.
10. *Apache Storm.* `http://storm.apache.org/` [Accessed: 01-Nov-2019].
11. Dieudonne Manzi and David Tompkins. Exploring gpu acceleration of apache spark. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 222–223. IEEE, 2016.
12. Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 347–348. IEEE, 2015.

13. M Mazhar Rathore, Hojae Son, Awais Ahmad, Anand Paul, and Gwang-gil Jeon. Real-time big data stream processing using gpu with spark over hadoop ecosystem. *International Journal of Parallel Programming*, 46(3):630–646, 2018.
14. Ryo Asai, Masao Okita, Fumihiko Ino, and Kenichi Hagihara. Transparent avoidance of redundant data transfer on gpu-enabled apache spark. In *Proceedings of the 11th Workshop on General Purpose GPUs*, pages 22–30. ACM, 2018.
15. *IBMSparkGPU, GitHub.* `http://github.com/IBMSparkGPU/` [Accessed: 01-Nov-2019].
16. Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. Spark-gpu: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 273–283. IEEE, 2016.
17. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 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*, pages 2–2. USENIX Association, 2012.
18. Zhenhua Chen, Jielong Xu, Jian Tang, Kevin Kwiat, and Charles Kamhoua. G-storm: Gpu-enabled high-throughput online data processing in storm. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 307–312. IEEE, 2015.
19. Cen Chen, Kenli Li, Aijia Ouyang, Zeng Zeng, and Keqin Li. Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1275–1288, 2018.
20. Dalton Lunga, Jonathan Gerrand, Lexie Yang, Christopher Layton, and Robert Stewart. Apache spark accelerated deep learning inference for large scale satellite image analytics. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 13:271–283, 2020.
21. Carol McDonald. *ACCELERATING APACHE SPARK 3.X Leveraging NVIDIA GPUs to Power the Next Era of Analytics and AI*, volume 01 of *1*. NVIDIA Corporation, 2788 San Tomas Expressway, Santa Clara, CA 95051, 1 edition, 5 2020.
22. *CUDA Zone — NVIDIA Developer.* `https://developer.nvidia.com/cuda-zone` [Accessed: 06-Jun-2020].
23. *ACID Transactions.* `https://ignite.apache.org/features/transactions/` [Accessed: 01-Nov-2019].
24. *Apache Ignite Documentation.* `https://apacheignite.readme.io/docs` [Accessed: 06-Jun-2020].
25. *Spring Boot With Apache Ignite: Fail-Fast Distributed MapReduce Closures.* `https://dzone.com/articles/spring-boot-with-apache-ignite-fail-fast-distribut` [Accessed: 06-Jun-2020].

26. *OpenCL — NVIDIA Developer.* `https://developer.nvidia.com/opencl` [Accessed: 06-Jun-2020].
27. *NVCC: CUDA Toolkit Documentation.* `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html` [Accessed: 06-Jun-2020].
28. *PTX ISA: CUDA Toolkit Documentation.* `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html` [Accessed: 06-Jun-2020].
29. Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *European Conference on Parallel Processing*, pages 887–899. Springer, 2009.
30. Craig Bordelon. A reasonable c++ wrappered java native interface. *arXiv preprint cs/9907019*, 1999.
31. Jie Zhu, Juanjuan Li, Erikson Hardesty, Hai Jiang, and Kuan-Ching Li. Gpu-in-hadoop: Enabling mapreduce across distributed heterogeneous platforms. In *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, pages 321–326. IEEE, 2014.
32. Ben Van Werkhoven, Jason Maassen, Frank J Seinstra, and Henri E Bal. Performance models for cpu-gpu data transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 11–20. IEEE, 2014.
33. *JCuda Documentation.* `http://www.jcuda.org/documentation/Documentation.html` [Accessed: 29-Jun-2020].
34. Rajvi Shah, PJ Narayanan, and Kishore Kothapalli. Gpu-accelerated genetic algorithms. *cvit. iiit. ac. in*, 2010.