A GPU-Enabled Extension for Apache Ignite to Facilitate Running Genetic Algorithms

Majid Salimi Beni
Department of Computer Science, Engineering and IT, School of Electrical and Computer Engineering
Shiraz University
Shiraz, Iran
Email: m.salimibeni@shirazu.ac.ir

Amir Hossein Sojoodi
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
Email: amir.sojoodi@queensu.ca

Farshad Khunjush
Department of Computer Science, Engineering and IT, School of Electrical and Computer Engineering
Shiraz University
Shiraz, Iran
Email: khunjush@shirazu.ac.ir

***Abstract*—With the increasing rate of data generation in recent years, there is a need for modern tools to process these massive amounts of data. To that end, in-memory platforms are becoming increasingly popular, which can process high volumes of data at high speed and performance by utilizing Main Memory. Apache Ignite is one of the in-memory platforms that can process in parallel on multiple nodes. Although this platform provides many useful features, one of its limitations is the lack of utilizing the GPU's high processing power. Undoubtedly, using GPUs for operations that deal with heavy processing or high data volumes can be very beneficial, and significantly accelerate processing. One of the algorithms supported by Ignite is the Genetic Algorithm, which usually deals with large amounts of data, and might be very time-consuming. In this paper, we have provided an extension for Ignite in which users can utilize GPUs to run their Genetic Algorithm applications. Also, we have used various GPU-related optimization techniques to improve performance and finally evaluated our extension with three benchmarks. Our results proved the ease of use, and the high performance of the proposed work compared to Ignite.**

Keywords—Apache Ignite, In-memory Computing, GPU, Genetic Algorithm

I. INTRODUCTION

With the development of IoT technologies in recent years and the emergence of new applications, large data volumes are being generated, which traditional tools were not able to process and handle these data. For this purpose, newer tools were needed to store and process these data with high performance. One of these tools is Apache Hadoop [1], which has a processing engine called MapReduce, and a distributed file system called HDFS [2] to store data on distributed disks. Also, the platform is able to scale on any number of nodes in a cluster and act in parallel. However, with the rise of new demands, such as stream processing and machine learning, this platform was no longer able to meet the requirements. The platform's file system stores data on the disk, which also makes it a bottleneck that slows down processing [3]. Therefore, more advanced tools were needed to not only meet new requirements but also make processing faster.

To address the problems mentioned above, a couple of new platforms have been developed. The three most famous are Apache Storm [4], Apache Spark [5], and Apache Flink [6]. Their architecture is based on in-memory computing architecture, and they provide capabilities for stream processing and distributed machine learning. Although these platforms almost addressed the challenges, there remained still some limitations. Storm and Flink, for example, have focused on real-time processing, and for other application types, they need to integrate with other platforms. Moreover, Spark, as a general-purpose processing engine, supports only a few machine learning algorithms [7], does not process exactly real-time, and it does not contain a file management system on its own.

Apache Ignite [8], as a newly released platform, is a distributed in-memory database for caching and processing big amounts of data that tried to cover prior platforms' restrictions. Ignite is an ideal choice for distributed machine learning, data analytics, transactional and streaming jobs, and is suitable for data and compute-intensive operations.

Thanks to in-memory architecture on these platforms, the disk I/O bottleneck has been resolved, which was a performance limiter in Hadoop. As the Main Memories are getting faster and their capacity increases, the only factor that slows down the processing in in-memory platforms is the processing unit (CPU), which means that the bottleneck has shifted from the disk I/O to the computation [3]. Since these platforms use CPU for their processing, which has few processing cores, low bandwidth, and it is not suitable for processing big data, utilizing an appropriate processor like GPU, would be an excellent option to eliminate the computation bottleneck.

Due to the nature of Ignite applications, which are generally data-intensive and compute-intensive, utilizing GPUs as co-processors in these applications can significantly accelerate their computation rate. GPUs are parallel processors that have a large number of simple processing cores (compared to CPUs) and have a very high memory bandwidth. Their architecture makes them suitable for simple, repetitious operations, with high throughput. Hence, GPUs are an excellent choice for applications like machine learning, deep learning, and stream processing [3].

With respect to the fact that Ignite is not capable of utilizing GPUs currently and considering the nature of Ignite applications, we decided to provide GPU-support in Ignite's Genetic Algorithm. In our prior work, we implemented a prototype of the provided work and presented its implementation steps [17].

In this paper, we provide an extension relying on Ignite that enables application developers to implement their Genetic Algorithm to run on GPUs. In other words, we have provided the facility to utilize GPUs in Ignite to accelerate the Genetic Algorithm. We chose the Genetic Algorithm because it has iterative and parallel behavior; it is usually

978-1-7281-8111-0/20/$31.00 ©2020 IEEE

time-consuming and deals with large data volumes. So, it can be a good candidate to run on GPUs. This algorithm has many applications in various industries and artificial intelligence.

Using this extension, Genetic Algorithms that might take days to execute would run much faster. The proposed work demonstrates a significant acceleration and better support of lager populations (lager input sizes) for the Genetic Algorithm. Our main contributions are as follows:

- We have implemented user-friendly APIs for the Genetic Algorithm to utilize GPU through Apache Ignite.
- We have used various GPU-specific optimization techniques to improve performance.
- We have provided great flexibility for application developers to be able to adjust different GPU-related parameters based on their type of problem or their GPU specifications.
- Finally, We have evaluated the performance of provided extension with problems such as String Matching (HELLO WORLD), TSP and, Knapsack 0-1.

The remainder of this paper is organized as follows. Section II provides some related work. Section III presents the background needed to read this paper. Section IV describes the work presented, and Section V reports our experimental results. Eventually, Section VI concludes the paper.

II. Related Work

Exploiting GPUs in in-memory platforms has shown to be beneficial and fruitful. During recent years, many researchers have investigated using GPUs as a co-processor on these platforms for various types of algorithms and problems.

Manzi et al. [9] investigated the practicability and advantages of offloading some of the Apache Spark basic operations to the GPU. They ported several iterative and non-iterative workloads to the GPU, and their results showed about 17X Speedup for the K-Means clustering algorithm. For other applications, like RadixSort and WordCount, there was a marginal acceleration due to the overheads of converting data into a GPU-friendly format.

HeteroSpark [10] was another GPU-accelerated architecture that used GPU's processing power for performing data and compute-intensive operations alongside the CPU. This platform was not only easy to use but also showed better performance and energy efficiency in comparison to Spark. Their results showed about 18X speedup for machine learning workloads.

Rathore et al. [11] combined the Hadoop MapReduce and Spark to provide an effective and high-performance stream processing platform. They utilized GPUs for the real-time processes that dealt with big data volumes. The result of their work was a higher performance for real-time applications.

Spark-GPU [12] was a CPU-GPU platform that was able to utilize both CPU and GPU. In this work, various challenges of integrating Spark with GPUs were examined. In addition, they proposed a GPU-friendly type of Spark RDD [13] called GPU-RDD suitable for use in the GPU memory hierarchy. Spark-GPU resulted in about 16.13X speedup for machine learning applications, and 4.3X for SQL queries workload.

G-Storm [14] was an Apache-Storm-based platform designed for processing streaming data by utilizing GPUs. G-Storm can support various data types and is able to run applications on the GPU with very low overhead. This platform demonstrated more than 7X throughput improvement on continuous query workload, and 2.3X speedup in matrix multiplication.

GFlink [15] is another GPU-capable in-memory platform that developed on the top of Apache Flink, which is able to utilize GPU's high memory bandwidth and its computation power. Moreover, various methods were applied to improve its performance that resulted in an efficient communication mechanism between the JVM and the GPU. Also, an adaptive locality-aware scheduling method was implemented, that resulted in a better performance.

III. Background

In this section, we discuss the theoretical foundations needed to read this paper. First, we introduce Apache Ignite, then briefly explain the architecture of the GPU and the Genetic Algorithm.

A. *Apache Ignite*

Apache Ignite is an in-memory distributed open-source platform that is suitable for transactional, analytical, and streaming workloads. This platform can deliver in-memory performance at the petabyte scale. Ignite was accepted as an Apache Incubator program in 2015.

Ignite has provided many features that cover most of the flaws of previous platforms. Ignite can be used as an in-memory cache that supports various APIs for key-value and SQL. It can also be utilized as an in-memory Data Grid, which can be deployed to accelerate existing databases like RDBMS, NoSQL, and Hadoop. Furthermore, Ignite can be used as an in-memory database, and scale up and out on any number of nodes. Moreover, not only can it operate in standalone mode, but it also can be used alongside other platforms, and is able to be deployed in the cloud, containerized, and provisioning environments.

Ignite primary features include:

- Memory-Centric Storage: Ability to store and process data in memory (as well as on disk).
- Distributed SQL: It is a memory-centric distributed database.
- Distributed key-value: Read, write, and transact using key-value pairs with high-speed in cache and data grid.
- ACID Transactions: ACID (Atomicity, Consistency, Isolation, and Durability) compliance across all distributed datasets.
- Collocated Processing: By sending computation to nodes, it prevents noise in the data.
- Machine Learning: Ignite supports various distributed machine learning models and algorithms.

These features have been provided in various built-in APIs so that application developers can use them to develop their desired applications. It should be noted that when running an application on Ignite, a node that operates as the master, divides, and sends desired data along with an executable operation (called Compute Task) to other nodes so that each node has its share of computation to run on its CPU. After the end of each node's process, the result of the computation returns to the master node.

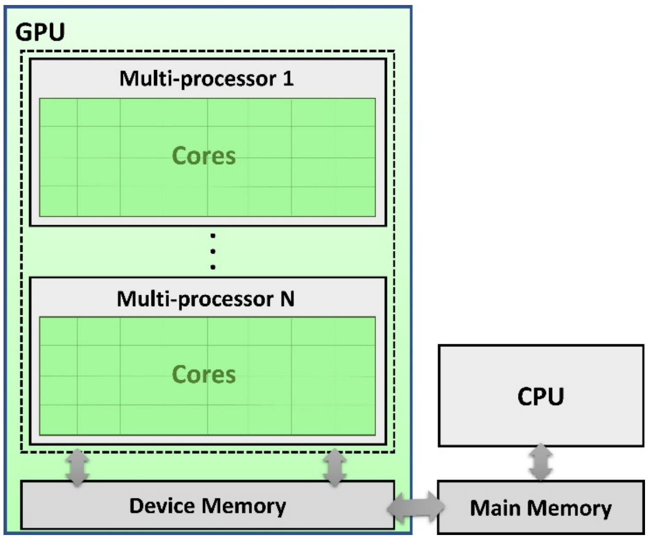

Figure 1. The GPU architecture model

B. *GPU and its Programming Model*

GPUs are parallel processors that maintain very high bandwidth and energy efficiency. They have a much larger (and, of course, simpler) number of processing cores compared to CPU, and their architecture makes them appropriate for data analytics and data processing workloads. In recent years GPUs are commonly utilized because of their high-performance and high-throughput to handle compute-intensive operations and real-time data processing [3], [14].

Figure 1 represents the overall scheme of GPU architecture. The GPU consists of a large number of cores grouped as multi-processors. All of the cores of a multi-processor execute a similar operation on different data, which 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 (host memory). The device memory has very high bandwidth and low latency.

To gain remarkable performances using GPUs, their processing resources should be fully utilized, and it is essential that the programmer use suitable optimization techniques based on the GPU's architecture and type of application. GPU programs are typically written in CUDA, PyCUDA, or OpenCL. The NVCC is responsible for translating the CUDA code into chunks for running on the CPU, and GPU. The "kernel" is referred to that part of the code that runs on the GPU.

To utilize GPU's resources in Java applications, we need a communication bridge to establish a connection between CUDA and Java. Accordingly, JCUDA and JNI are two of the most popular choices. Although JCUDA has a higher development complexity, it has a better performance than JNI [16]. JCUDA facilitates programmers to load and execute CUDA kernels in Java programs, and it has special functionalities for allocating/deallocating and exchanging data between the host and the device memory.

Commonly, to run a program on GPUs, first, the kernel's required data should be copied from the host to the device memory. Then, the kernel should be launched, and finally, after the kernel execution is finished, the execution's results should be returned from the device memory to the host. Since transferring data between host and device is expensive and can result in performance degradation, the programmer should try to eliminate unnecessary data exchanges.

C. *Genetic Algorithm*

Genetic Algorithm is a simulation of the biological evolutionary process, and it is mostly used to find an optimal solution in large and complex datasets. This algorithm is being employed in a lot of real-world applications like computer gaming, automotive design, robotics, traffic/shipment, investments, and routing. As shown in figure 2, this algorithm consists of four main steps: Selection, Fitness calculation, Crossover, and Mutation. Initially, the algorithm produces an initial population of possible solutions (chromosomes)—each chromosome consists of genes. Then, in each iteration, it selects a subset of best solutions based on an evaluation criterion. Next, it performs Mutation and Crossover operations on this subset to produce more optimal chromosomes for the next generation. This procedure is repeated until the algorithm meets a termination criterion.

In Ignite, the Genetic Algorithm is a subset of Machine Learning APIs. Genetic Algorithm operations such as Fitness calculation, Crossover, and Mutation are modeled as an Ignite Compute Task to distribute among nodes, and each node operates on its share of data.

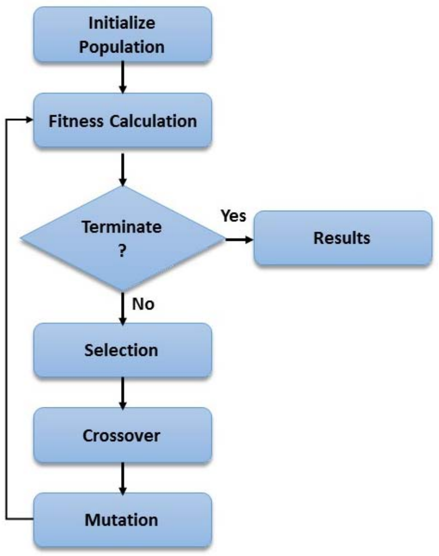

Figure 2. Genetic Algorithm steps

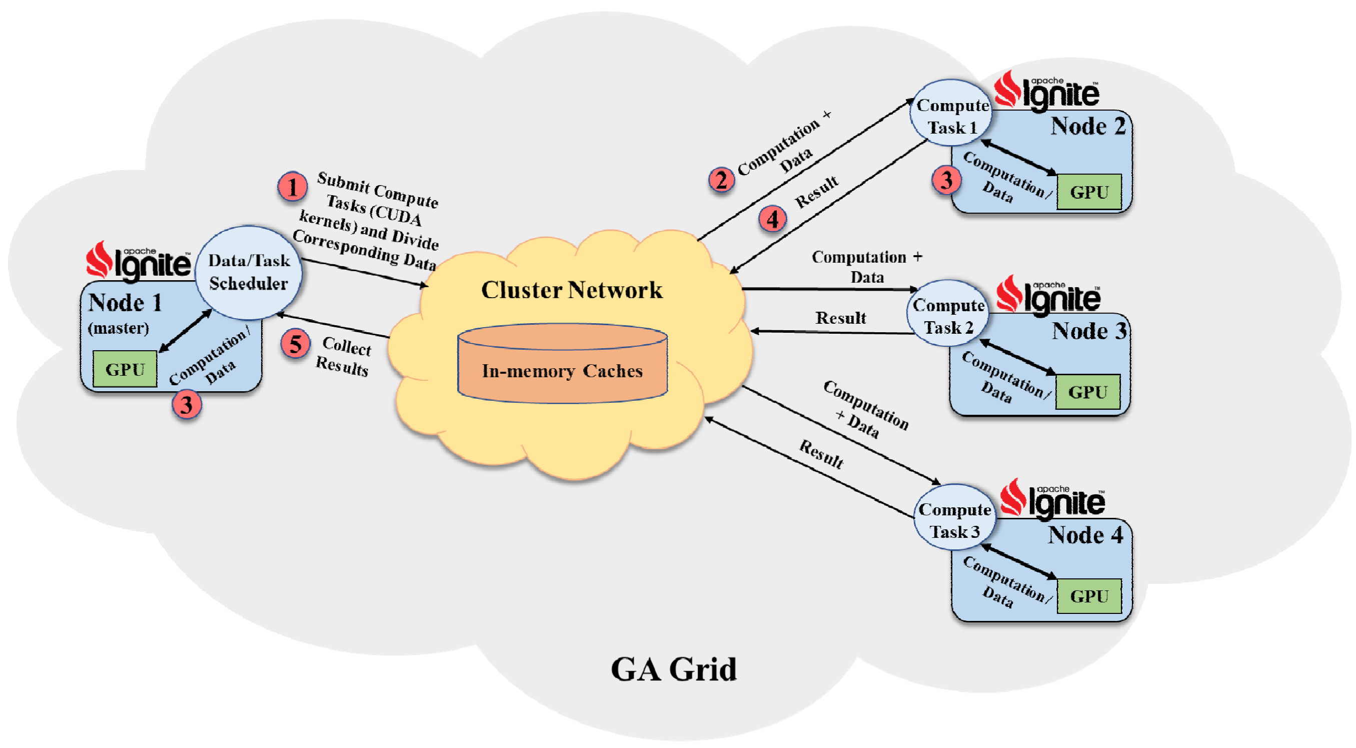

Figure 3. System Architecture. Initially, node 1 acts as master and distributes Compute Tasks (data and computation) to run on GPUs.

IV. Proposed System

In this section, we discuss the presented work and its design details.

A. *System Overview*

The provided extension is entirely based on Apache Ignite's in-memory architecture. In this paper, we provide GPU usability as a co-processor alongside the CPU for the Genetic Algorithm through Apache Ignite. Thus, by utilizing GPU, application developers not only can benefit from the speed of in-memory computing but also can utilize GPUs to improve the performance.

Ignite considers each operation of the Genetic Algorithm as a Compute Task. Accordingly, Ignite creates a Compute Task for each functionality (such as Mutation) and sends the computation along with the corresponding data to nodes. After the current Compute Task's execution is completed, the results are returned to the master node, and a new Compute Task will be created for another operation (such as Crossover). The same steps are repeated for all functionalities of the algorithm. Consequently, for one particular chromosome, the Mutation operation may be performed on one node and the Crossover operation on another node.

Given that the Genetic Algorithm has iterative behavior, building four Compute Tasks for Selection, Mutation, Crossover, and Fitness in every iteration is expensive. On the other hand, on GPU, four kernels are needed to perform these operations, each of which requires data transfers between host and device. Hence, we decided to perform all of these four Compute Tasks as a single one on the GPU, so there will be just one Compute Task and one kernel call in each iteration of the algorithm. It not only prevents creating multiple Compute Tasks, but it also prevents calling multiple kernels and removes redundant data transfers between host and device.

In Figure 3, the proposed extension architecture for the Genetic Algorithm is illustrated. As shown in the figure, at the beginning of each iteration of the algorithm, the master node distributes computations and their corresponding data to cluster nodes (Step 1). Then, the data is delivered to nodes through the corresponding in-memory caches (Step 2). As soon as worker nodes receive the data, copy the data to their GPUs’ device memory, and start processing by the GPU (Step 3). After the processing phase is completed, the results are copied to their corresponding cache and return to the master node (Step 4 and 5).

With regard to GPUs' models or CUDA versions may vary on different nodes, to work in a distributed environment, we copy the kernel file to all nodes using Ignite's file system at the beginning of the program. Thereby, all nodes will have the same kernel file, and each node will generate a PTX file based on its CUDA runtime version for running on its GPU.

Figure 4 shows a detailed pseudo-code of what is happening in Ignite's master node in our presented extension. As it is provided in the second and third line of the pseudo-code, users should specify the Genetic Algorithm's data structure and details in the master node. Then, in lines 5 to 9, users can alter default execution configurations. In line 9, users should implement and pass a method that gets a chromosome as input and returns a value as the fitness score of the input chromosome. After that, lines 11-28 contain common operations done by the master node, which was designed and implemented for general use cases of Genetic Algorithms.

```
//GA data structure and details:
$ChromosomeLength \leftarrow L$
$GenesPopulation \leftarrow$ {List Of Objects}
//Execution configurations (e.g.):
$MaxGeneration \leftarrow 100$
$PopulationSize \leftarrow 1000$
$GPUStreams \leftarrow 8$
$GPUBlockSize \leftarrow 32$
$FitnessFunction \leftarrow$ A Callable Method
// Master node:
Configure data caches of the cluster
Configure Ignite File System (IGFS)
Initialize data cache with chromosome population
Share fitness function and other configurations on IGFS
$Count \leftarrow 0$
while $Count \leq MaxGeneration$ do
    Distribute Chromosomes to the workers
    Wait for the results
    Gather new Chromosomes from workers
    $BestScore \leftarrow head(sort(population))$
    if $BestScore \geq margin$ then
        Log the chromosome with the $BestScore$
        break
    end
    $Count \leftarrow Count + 1$
end
Send $end$ signal to workers
Clean up data caches and IGFS
```

Figure 4. Pseudo-code of the Genetic Algorithm in the provided extension's master node.

In figure 5, the process of a worker node in this platform is depicted. It can be seen that the Workers' code does not change across various Genetic applications. The only difference here is the fitness function, which is explicitly created by users for a particular application. In line 6, the fitness function is obtained from the master node.

```
1  Starts with the submitted configurations
2  Gets GPU source file from IGFS
3  Gets other configurations from Ignite cache
4  Using JCuda, attach the GPU code to the running JVM
5  Warm-up the GPU and allocate required data and streams
6  $FitnessFunction \leftarrow$ Function provided by the master
7  $Task \leftarrow$ Wait and get a task from the master
8  while $Task == GAIteration$ do
9      Copy population partition from Ignite cache to the GPU streams
10     Perform Mutation and Cross Over on the GPU
11     Evaluate chromosomes fitness by $FitnessFunction$ with JCuda
12     Copy back the results to the Ignite cache
13     Inform master of the task completion
14     $Task \leftarrow$ Wait and get a task from the master
15     if $Task == WrapUp$ then
16         Deallocate memories of the host and the device
17         Clean up configurations and meta data
18         break
19     end
20 end
21 Wait for the master
```

Figure 5. Pseudo-code of the Genetic Algorithm in the provided extension's worker node.

We used JCUDA to communicate between Java and CUDA. In addition, for the convenience of application developers, we have provided several APIs for allocating memory on host and device, transferring data from host to device and vice versa, launching the kernel, and configuring the associated parameters. We have also provided other APIs for initializing GPUs and setting their parameters like Block Size so that users can adjust these parameters depending on their GPU model and their compute-capability.

One challenge here is converting chromosomes—which are JVM objects in Ignite, into a GPU-friendly format. To do so, we copy the objects into a byte array before transferring them to device memory and send this byte array to GPU. By defining a *Union* in the kernel, we can access the data in the desired data type in GPU.

In the provided extension, we efficiently support the operations of initializing GPU, transferring data between host and device, and launching the kernel in Ignite. We also provide the ability to transfer different types of data between host and device to cover various Genetic Algorithm problems with different data types. Application developers should only write (or modify) their kernel code in the *UserKernels.cu* file and specify their kernel's name in their application.

For implementing the Genetic Algorithm in this GPU-enabled extension, we consider one CUDA kernel that contains Selection, Mutation, Crossover, and Fitness together. In this kernel, we have assigned a GPU thread to each chromosome to do Genetic operations on it. In addition, given that the Genetic Algorithm deals with high data volumes, and it is time-consuming, we also used various optimization techniques. One of these techniques is CUDA streams in which the GPU can perform computation while transferring data by overlapping computation and communication. This not only increases performance but also enables GPU to handle higher data volumes in device memory. Users can specify the number of required Streams based on their size of the input at the beginning of their program.

Another optimization that has been done is reducing the number of time-consuming memory allocations/deallocations on the host and device. In general, when a program utilizes the GPU(s), a set of allocations/deallocations and data copy operations are mandatory. However, if this program has a iterative behavior and runs in consecutive independent similar steps, its required memory can be allocated only once before the first step of the program and deallocated only once at the end of the program execution cycle. In this way, the required data can be passed to the pre-allocated memory locations before each iteration's calculation phase. As Genetic Algorithms ideally possess these characteristics, we can improve their performance by removing redundant memory operations. To implement this idea, we designed and developed a wrapper for memory operations in which every node of cluster keeps track of memory allocations, including its type, size, and validity. In this way, we reduced the number of device memory allocation/deallocation calls across the multiple passes of the Genetic Algorithm execution span.

Moreover, to support multiple streams, data on the host should be pinned to memory before initiating a copy operation between host and device. This pinning action has to be done for every data movement to let the copy operation run without interruption by operating system page management system. As this pinning action also has to be done similarly for every Genetic Algorithm step, the pinned memory location can be allocated and deallocated only once before the first step and after the last step of Genetic Algorithm respectively.

V. EXPERIMENTAL RESULTS

In this section, the proposed work's performance is evaluated. We first describe the experimental environment and then present the results. It should be noted that all experiments were repeated ten times, and the reported results are the average of these ten experiments. Also, all the reported numbers are the execution time of one iteration of the genetic algorithm.

A. Experimental Environment

We performed our tests on a cluster with 2 GPU-equipped nodes. Each node is a XenServer VM and has 16 dedicated CPU cores with 2.0 GHz Intel Xeon E5-2620 and 25 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 developed based on Apache Ignite 2.7.0.

B. Benchmarks

To evaluate the proposed extension, we selected three benchmarks to solve with the Genetic Algorithm, including HELLO WORLD, TSP, and Knapsack 0-1. The Genetic Algorithm here uses Roulette Wheel for the Selection, as well as Single-point Crossover and Swap Mutation. We evaluated these three benchmarks by comparing Ignite and the presented extension with different population sizes on one and two nodes. Also, we set other parameters according to our GPUs, including Block Size (32) and the number of Streams (8).

1) HELLO WORLD

This problem is to reach the “HELLO WORLD” string from alphabet letters using the Genetic Algorithm. In this case, the optimal solution is “HELLO WORLD,” and the fitness criteria is the similarity of each chromosome to this string. In this problem, the chromosome length is 11.

Figure 6 compares the performance of the HELLO WORLD problem on Ignite and our extension, on a single node and a 2-node cluster. As shown, performance has improved with the increase of population size (which is the size of input of the problem). On a single node, for population size 1K, the speedup is about 3X, for 10K is 8X, for 100K is 35X, and for 1M is 111X. On 2-node cluster, speedup for 1K, 10K, 100K, and 1M inputs are 0.7X, 7X, 34X, and 116X, respectively. Ignite takes too much time for population sizes larger than 1 million and cannot scale for large inputs on both single and two nodes. Although for all input sizes, our extension has higher performance than Ignite, for 1K on two nodes, the proposed work shows a little bit more execution time.

On top of that, for 1K, the performance on a single node is better than two nodes. The reason is that the smaller the data size, the less computation is required. Therefore, utilizing GPUs or using multiple nodes, in this case, might be unbeneficial due to the overheads of distributing data between nodes and data transfers between host and device.

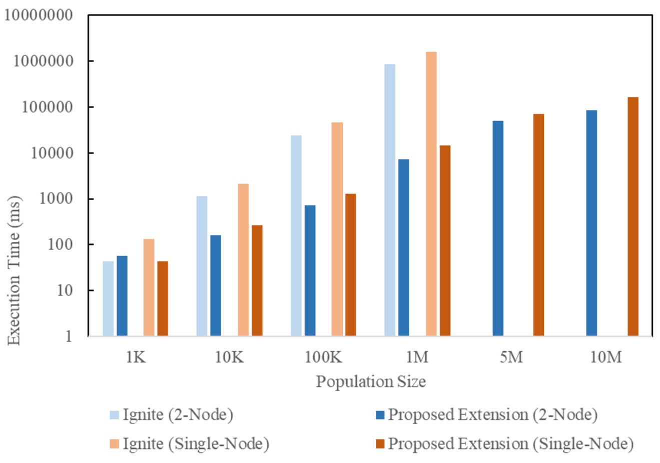

Figure 6. Performance of HELLO WORLD on Ignite and the proposed extension, on a single node, and 2-node.

2) TSP

Traveling Salesman Person (TSP) is an NP-hard problem. In TSP, given two-dimensional coordinates of a number of cities, we want to start with a city, visit all cities one time, and return to the origin city. In this problem, the goal is to find the shortest path using the Genetic Algorithm. The distance between every two cities is calculated using their Euclidean distance. The number of cities in the under-study problem is 500, and the coordinates of each city include two double numbers in x and y dimensions. Hence, each chromosome contains an array of 500 cities, with no duplicate, and the fitness criterion is to minimize the length of the route.

Figure 7 shows the execution time of Ignite and our proposed extension for the TSP problem on one and two nodes. On one node, the speedups for 1K, 10K, and 100K are about 2.5X, 6.5X, and 27X, respectively. For inputs greater than 100K, neither Ignite nor the provided extension could respond. The reason is that chromosome sizes are too large, and numbers are in double type, so device memory and RAM cannot accommodate huge populations. If we had GPUs with larger device memory, and more RAM capacity, we would be able to process larger input sizes.

On multiple nodes, due to the division of input data between two nodes, which provides more data storage potential, the proposed work is able to respond to the population size 1 million. The speedups for 1K, 10K, and 100K populations are about 2X, 6.5X, and 28X, respectively. All in all, for the TSP problem, the proposed work shows higher performance in comparison with Ignite on one and two nodes.

3) Knapsack 0-1

The Knapsack 0-1 problem is defined as having some items; each has a weight (double) and a value (integer). We want to select a number of items that maximize the value of the chosen items while their weight does not exceed a threshold. Genetic Algorithm can also solve this problem as an optimization problem. Here, we have a total of 30 items, and we want to select 10 out of 30. The chromosome length, in this case, is 10, and we are not allowed to pick duplicate items.

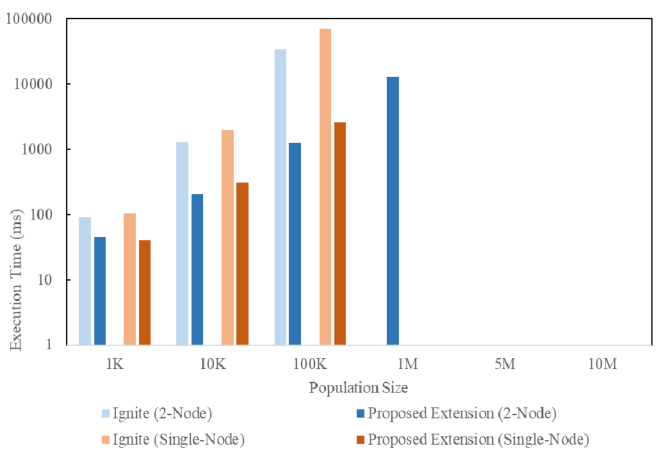

Figure 7. Performance of TSP on Ignite and the proposed extension, on a single node, and 2-node.

Figure 8 presents the execution time of the Knapsack 0-1 problem on Ignite and the extension provided. As depicted, Ignite does not respond to inputs larger than 100K. Similarly, like the two previous problems, performance improves with the rise of population size. The speedup here on one node is about 2.5X, 8.8X, and 38X for 1K, 10K, and 100K inputs, respectively. On multi-node, for 1K, 10K, and 100K inputs, the speedup is about 2X, 7.5X, and 38X, respectively.

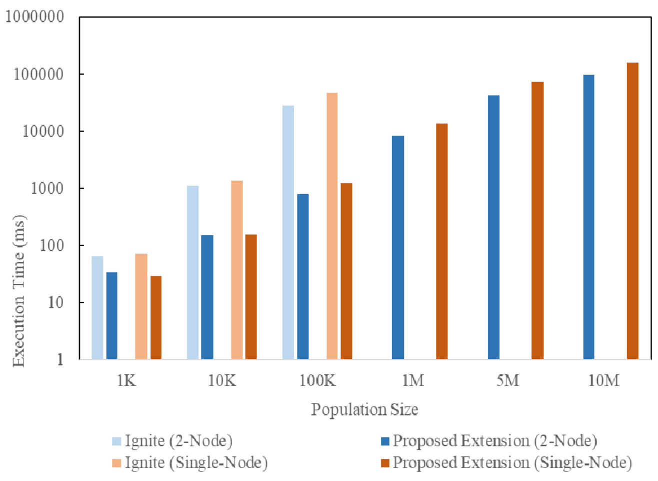

Figure 8. Performance of Knapsack 0-1 on Ignite and the proposed extension, on a single node, and 2-node.

C. Discussion

As observed in the three studied benchmarks, our GPU-enabled extension shows better performance for the Genetic Algorithm in comparison with Apache Ignite. In three studied problems, the performance improved with the increase of input data size, indicating better support of larger input sizes.

As illustrated, the acceleration obtained in the TSP is slightly less than the other two workloads, which is mainly due to the nature of this problem and the length of the chromosome. Since each GPU thread has the responsibility of working on one chromosome, in this problem, each thread is responsible for running the Mutation, Crossover, and calculating and evaluating the Fitness of each chromosome. Therefore, the longer the chromosome is, the more job will be assigned to each thread.

On top of that, In TSP, the Fitness calculation includes calculating the Euclidean distance between two coordinate points, which is a time-consuming operation. On the other hand, in Mutation and Crossover, it should be checked that in each route of cities, after performing these operations, there will be no duplicate city, and this makes the work of each thread heavier. As a result, the obtained speedup for TSP is slightly less than the other two benchmarks.

In addition, since GPUs are mostly suitable for processing data and compute-intensive jobs, utilizing them for smaller population sizes (such as 1K) may not be fruitful. In this case, employing GPUs adds overheads of data transfer between host and device, data conversion to the appropriate format, and memory allocations/deallocations for each data transfer. Besides, building Compute Tasks and distributing them among nodes is costly, so for small population sizes, running on a single node can be a better choice. All in all, it may not always be a good idea to scale-out computation or use GPUs to gain more performance, so depending on the size of data and the amount of computation, it can be determined.

Although the use of CUDA streams allows us to handle larger data sizes with GPU, in the TSP problem, we were unable to process huge data sizes due to the limited capacity of the device memory. Ignite also has difficulties with very large population sizes of the Genetic Algorithm. In some cases, the processing fails due to lack of processing resources, and in some cases, lack of main memory capacity causes processing failure.

Moreover, as you have seen in the investigated benchmarks, our extension can handle different types of data, including character, double, and integer.

It is worth mentioning that the accelerations reported in this paper are purely related to utilizing GPUs. By doing some Ignite-specific tunings and optimizing the way of data and task distribution on Ignite by enlarging the data batches on Compute Tasks, the cumulative speedup obtained by these changes and utilizing GPUs will be up to thousands of times. However, we have decided just to report the real effect of employing GPUs in Apache Ignite.

VI. CONCLUSION

In this paper, we provided an extension to Apache Ignite to facilitate running Genetic Algorithms using GPUs, so that application developers can utilize GPU's processing power through Ignite. For ease of use, we implemented some APIs and provided GPU usability through these APIs. Moreover, we performed some optimizations to improve performance and then evaluated the proposed work with three different types of problems: HELLO WORLD, TSP, and Knapsack 0-1. Our experimental results show higher performance and better support of larger inputs for the investigated problems. It can be concluded that using GPUs in Ignite can be very useful for compute-intensive or data-intensive applications.

Accordingly, users can benefit from GPU's processing power in addition to in-memory processing speed in Ignite.

References

[1] *Apache Hadoop.* http://hadoop.apache.org/. [Accessed: 01-Nov-2019].

[2] Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In *IEEE 26th symposium on mass storage systems and technologies (MSST)* (pp. 1-10). IEEE.

[3] Mizell, E. and Biery, R. (2017). Introduction to GPUs for data analytics. 1st ed. O'Reilly Media.

[4] *Apache Storm*. http://storm.apache.org/. [Accessed: 01-Nov-2019].

[5] *Apache Spark™ - Unified Analytics Engine for Big Data*. http://spark.apache.org/. [Accessed: 01-Nov-2019].

[6] *Apache Flink*. http://flink.apache.org/. [Accessed: 01-Nov-2019].

[7] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D.B., Amde, M., Owen, S. and Xin, D. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, *17*(1), 1235-1241.

[8] *Apache Ignite™*. http://ignite.apache.org/. [Accessed: 01-Nov-2019].

[9] Manzi, D., & Tompkins, D. (2016, April). Exploring GPU acceleration of apache spark. In *IEEE International Conference on Cloud Engineering (IC2E)* (pp. 222-223). IEEE.

[10] Li, P., Luo, Y., Zhang, N., & Cao, Y. (2015, August). Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *IEEE International Conference on Networking, Architecture and Storage (NAS)* (pp. 347-348). IEEE.

[11] Rathore, M. M., Son, H., Ahmad, A., Paul, A., & Jeon, G. (2018). Real-time big data stream processing using GPU with spark over hadoop ecosystem. *International Journal of Parallel Programming*, *46*(3), 630-646.

[12] Yuan, Y., Salmi, M. F., Huai, Y., Wang, K., Lee, R., & Zhang, X. (2016, December). Spark-GPU: An accelerated in-memory data processing engine on clusters. In *IEEE International Conference on Big Data (Big Data)* (pp. 273-283). IEEE.

[13] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S. and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (pp. 15-28).

[14] Chen, Z., Xu, J., Tang, J., Kwiat, K., & Kamhoua, C. (2015, October). G-Storm: GPU-enabled high-throughput online data processing in Storm. In *IEEE International Conference on Big Data (Big Data)* (pp. 307-312). IEEE.

[15] Chen, C., Li, K., Ouyang, A., Zeng, Z., & Li, K. (2018). 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.

[16] Bordelon, C. (1999). A Reasonable C++ Wrappered Java Native Interface. *arXiv preprint cs/9907019*.

[17] Sojoodi, A. H., Salimi Beni, M., & Khunjush, F. (2020). Ignite-GPU: a GPU-enabled in-memory computing architecture on clusters. *The Journal of Supercomputing*, 1-28.