Parallel Computing 112 (2022) 102942

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Performance and accuracy predictions of approximation methods for shortest-path algorithms on GPUs

Busenur Aktılav, Işıl Öz [*]

Izmir Institute of Technology, Computer Engineering Department, Izmir, Turkey

ARTICLE INFO

Keywords:
Approximate computing
GPU computing
Machine learning

ABSTRACT

Approximate computing techniques, where less-than-perfect solutions are acceptable, present performance-accuracy trade-offs by performing inexact computations. Moreover, heterogeneous architectures, a combination of miscellaneous compute units, offer high performance as well as energy efficiency. Graph algorithms utilize the parallel computation units of heterogeneous GPU architectures as well as performance improvements offered by approximation methods. Since different approximations yield different speedup and accuracy loss for the target execution, it becomes impractical to test all methods with various parameters. In this work, we perform approximate computations for the three shortest-path graph algorithms and propose a machine learning framework to predict the impact of the approximations on program performance and output accuracy. We evaluate random predictions for both synthetic and real road-network graphs, and predictions of the large graph cases from small graph instances. We achieve less than 5% prediction error rates for speedup and inaccuracy values.

1. Introduction

As Dennard scaling comes to an end and Moore's law slows down, the present computing systems are reaching the fundamental limits of the energy required for fully correct computation. Both industry and research communities have shifted their focus to innovative solutions from the traditional scaling for energy-efficient computing [1]. Approximate computing, which maintains an acceptable reduction in output accuracy, has started to play a key role in the efficiency of the applications tolerating computation errors. Various approximation methods applied in different layers of the computing stack demonstrate resource/correctness tradeoffs for the systems by targeting significant energy savings [2–6].

Heterogeneous computing systems offer high performance and less energy consumption by combining a wide range of device structures and configurations. Building heterogeneous systems by bringing together general-purpose multi-core processors (CPUs) and data-parallel graphics processing units (GPUs) enables efficient computation for high performance and energy consumption in large-scale computing platforms [7].

Graph algorithms serve for solving various real-life problems by maintaining the graph data structure to represent the big data. Since processing large amounts of data takes unreasonable time in single-core systems, the parallel implementations of graph algorithms have been developed to utilize the parallel resources of multi-core CPU or many-core GPU architectures [8–11]. While some graph-based computations require an exact result, others may tolerate some errors to be accepted as a correct result. Hence, the approximate computations may improve the performance by providing energy efficiency [12,13]. Software-based approximation, where the high-level source code is modified by introducing code transformations, requires an exhaustive method and parameter selection process to find out the best technique maintaining both performance and accuracy. However, exploring all possible approximation methods and related parameters for all target algorithms is not practical. Therefore, we need to find a way to find out the effects of the approximation techniques for the particular program instance by considering multiple criteria (e.g. performance, correctness).

In this work, we investigate approximation techniques for the shortest-path graph algorithms and propose an ML-based methodology to predict the performance and accuracy impacts of the approximation techniques without executing all alternative approximations. We believe that our work can be easily adapted by other graph algorithms as well. Our main contributions are as follows:

- We define a set of software-level approximation methods for the main GPU-based shortest-path implementations for Dijkstra, Bellman–Ford, and Hybrid algorithm combining two algorithms.

* Corresponding author.
E-mail address: isiloz@iyte.edu.tr (I. Öz).

https://doi.org/10.1016/j.parco.2022.102942
Received 10 October 2021; Received in revised form 14 March 2022; Accepted 23 May 2022
Available online 30 May 2022

- We perform detailed experiments to demonstrate the effect of the approximations on the program performance and result correctness. Our executions employ both the individual approximation techniques with a set of parameters and the combination of multiple approximations.
- Based on our observations that reveal the wide range of approximation effects, we present an ML-based prediction mechanism to find out the performance and accuracy impacts of the approximation techniques and the related parameters for the target execution without running the programs.
- We evaluate both random predictions based on random train/test data split and predictions of the large graph cases from small graph instances. We utilize synthetic graphs generated by Kronecker generator [14] and real road-network graphs provided by DIMACS Implementation Challenge [15]. Our prediction model achieves less than 5% prediction error rates for speedup and inaccuracy values.

While there is existing ML-based performance prediction [16,17] and design-space exploration literature [18,19], our work presents a systematic way for approximation methods for GPU-based shortest-path implementations, generates empirical data based on a large set of executions, builds a prediction model by employing different machine learning algorithms and rigorous experimental results, and guides the developer for the approximations by predicting both performance and accuracy for target executions. We believe that our study will impact how approximation methods can be applied for graph algorithms, and how the target executions can be compared and traded-off for performance and accuracy on target GPU platforms.

The remainder of this paper is organized as follows: Section 2 presents some background on GPU architectures and shortest-path algorithms. We explain our approximation methods and prediction framework in Section 3. Then the experimental results are outlined in Section 4. Finally, in Section 5, we summarize the work with some conclusive remarks.

2. Background and motivation

2.1. GPU architecture and programming model

While introduced for real-time rendering in graphics applications, currently, GPU devices have been increasingly supporting non-graphics computing. Refined GPU architectures and programming models increase flexibility and energy efficiency. A modern GPU architecture contains many cores. Each core, located in a core cluster, is responsible for single-instruction-multiple-thread (SIMT) execution. While the cores inside the same core cluster have access to the scratchpad memory (shared memory or L1 cache), all the cores can communicate through L2 cache structure via interconnect. DRAM-based global device memory maintains larger but relatively slower data access for all threads executing in the device.

A GPU program starts its execution in a CPU, allocates memory space on the GPU, transfers data into GPU global memory, and starts a kernel function execution by creating thousands of threads. Each thread executes the same program (SIMT) by processing different parts of the given data. Threads that execute on the GPU are part of a compute kernel specified by a function. Besides data-parallel applications that can benefit from many parallel execution units of GPUs, large-scale graph computations with billions of vertices and edges, utilize the massive degree of parallelism and the high memory bandwidth provided by GPUs [20]. While graph processing includes irregular data access patterns [21] and also moving data from CPU to GPU results in substantial overhead, parallel GPU cores still offer a promising solution for high performance [10,22,23].

2.2. Shortest-path algorithms

One of the classical optimization problems in graph theory is the shortest-path problem. Specifically, in the single-source shortest path (SSSP) problem [24], the aim is to find the smallest combined weight of edges required to reach every node, for a given weighted graph and a source node. Many real-world problems, such as navigation systems, social networks, databases, and web searching [25,26], arise from finding the shortest paths from a given source to all the other nodes. There are two common algorithms to solve the SSSP problem: Dijkstra's algorithm and Bellman–Ford's algorithm. Additionally, Hybrid [27] algorithm combines these two approaches by utilizing their advantages.

2.2.1. Dijkstra's algorithm

The most well-known algorithm for solving the SSSP problem in the absence of negative weights in the graphs was proposed by Dijkstra in 1959. Dijkstra's algorithm provides an optimal sequential solution to the SSSP problem. Its time complexity is $O(ElogV)$. There have been many attempts to parallelize Dijkstra's algorithm efficiently in the literature [28].

Dijkstra's algorithm (given in Algorithm 1) has a greedy approach and finds the next best solution hoping that the final result is the best solution. In the algorithm, when a low-cost path is discovered, the cost of the visited vertex is changed. If there is no change in the cost, then the algorithm terminates. It works fast in CPU compared to the other algorithms. However, the downside of the algorithm is that it does not work on the graphs with negative-weight edges, additionally, it is hard to parallelize in the GPU.

Algorithm 1: Sequential Dijkstra's Algorithm

```
create priority queue Q ;
for vertices $v \in V(G)$ do
    $d(v) = \infty$ ;
    $prev(v) = NULL$ ;
    if $v \neq s$ then
        $insert\_to\_queue(Q, v)$ ;
    end
end
$d[s] = 0$;
while $Q \neq \emptyset$ do
    $u = extract\_min(Q)$ ;
    for each neighbor $v \in u$ do
        $alt = d[u] + weight(u, v)$ ;
        if $alt < d[v]$ then
            $d[v] = alt$ ;
            $prev[v] = u$ ;
        end
    end
end
```

2.2.2. Bellman–Ford's algorithm

Another well-known algorithm for solving the SSSP problem is Bellman–Ford's algorithm, which has time complexity, $O(VE)$. Unlike Dijkstra's algorithm, the graphs with negative-weight edges can be processed and the algorithm can be parallelized easily in the GPU.

In Bellman–Ford's algorithm (given in Algorithm 2), firstly, the length of the path from the source node to all other vertices is overestimated. Then those estimates are iteratively relaxed by finding new paths that are shorter than the previous paths. *Relax* procedure (given in Algorithm 3) checks if, starting from u, it is possible to improve the distance to v. This process is repeated V times, since in the worst-case scenario, a vertex's path length might need to be readjusted V times. Finally, it is executed one more time (i.e., $(V+1)^{th}$) to check if there is any negative cycle in the graph. If the algorithm still updates the path distances, then there is a negative cycle in the graph. However, if there is no change in the distance path then the algorithm is finished, and there is no negative cycle in this graph.

Algorithm 2: Sequential Bellman–Ford's Algorithm

```
for vertices $u \in V(G)$ do
    $d(u) = \infty$ ;
end
$d(s) = 0$;
for edges $(u, v) \in E(G)$ do
    $Relax(u, v, w)$ ;
end
```

Algorithm 3: Relax Procedure

```
Function Relax($u, v, w$):
    if $d(u) + w < d(v)$ then
        $d(v) = d(u) + w$;
    end
return
```

2.2.3. Hybrid algorithm

Hybrid algorithm (given in Algorithm 4) combines the first two algorithms, i.e., Bellman–Ford's and Dijkstra's algorithms, and calculates the distances from the source to all other nodes [27]. It is proposed to show that Dijkstra's algorithm can actually work with the graphs with negative-weight edges and it realizes its purpose by running Dijkstra's algorithm several times.

Algorithm 4: Sequential Hybrid Algorithm

```
$i \leftarrow 0$ ;
while no change in distance or $i = |V| - 1$ do
    i++;
    $Dijkstra\_scan()$;
end
if no change in distance then
    return
else
    There exist negative cycle
end
```

2.3. Approximate computing

It is essential to improve the energy efficiency for applications that require high workloads to deal with the massive information that needs to be processed. A promising solution, known as approximate computing, introduces *acceptable errors* into the computing process and promises significant energy-efficiency gains.

There are different ways to achieve approximate computing [29]. The approximation of the computations can be performed either in the software with code modifications [30,31] or in the hardware by approximating the circuits [5,32]. In this work, we perform software-level approximate computations for the shortest-path algorithms requiring high performance with some tolerable error rate. While graph algorithms solve various real-life problems by representing big data, processing a huge amount of data takes too much time. Even though the parallel implementation of the graph algorithms maintains lower execution times, performance gain may not be sufficient for time-critical applications. On the other hand, those applications could tolerate some errors by providing faster executions. Therefore, applying approximation techniques by exchanging with some error rate is the best way to satisfy the performance requirements.

There are many approximation methods performed for graph computations in the literature. Singh and Nasre [12] present four approximation methods including Reduced Execution, Partial Graph Processing, Approximate Graph Representation, and Approximate Attribute Values, where they utilize the following approaches to achieve approximation: cutting-short the execution, processing only some part of the graph, running the algorithm on an approximate graph, and transforming SSSP algorithm into BFS algorithm, respectively.

Singh and Nasre, in their other work, [33], propose GPU-specific approximations. They focus on the GPU-specific aspects affecting the performance and address memory coalescing, memory latency, and thread divergence problems by presenting three techniques to boost performance. In this work, we do not focus on GPU-specific techniques.

Slim Graph [30] presents a practical lossy graph compression framework and programming model for approximate computing in graphs. It accelerates many graph algorithms, reduces storage use, and provides high accuracy of the resulting graphs.

3. Methodology

In this work, we firstly implement the parallel versions of Bellman–Ford's, Dijkstra's, and Hybrid algorithms, and apply the approximation techniques to increase the performance. Then, we gather the data from various graphs with various approximation techniques and use machine learning models to predict the speedup and the inaccuracy rates for the given graphs. Specifically, we predict how much the execution time is reduced and how inaccurate the expected result is computed (error rates as differences between distance values). In this section, we first explain the CUDA implementations of our SSSP algorithms. Secondly, we introduce our approximation techniques applied in the target codes. Finally, we present our prediction model built to estimate the speed-up and the inaccuracy rates of the approximate versions.

3.1. SSSP algorithms implementation details

The shortest path algorithms aim to find the shortest paths from a single source node to all other nodes in a given graph. In this work, we implement Bellman–Ford's, Dijkstra's, and Hybrid algorithms in CUDA programming model.

For the parallel Bellman–Ford's algorithm (given as a template in Algorithm 5), we utilize the implementation and the optimization techniques performed by Busato et al. [34]. Firstly, as a preprocessing step, we eliminate the self-loops from the graphs since they cannot change the tentative distance of u. Secondly, we add the source vertex to the queue by adding its neighbors to the queue in the CPU. Thus, we increase the parallelism and avoid unnecessary kernel launch and data copy operations. Therefore, we do not have to implement the source edge class technique, which suggests the direct update of its neighbors since they are never visited before, in the kernel code. Thirdly, we implement the out-degree edge class, which specifies that there is no need for an update if the vertices with out-degree are equal to zero, and they are ignored during the algorithm iterations. Finally, we apply the duplicate removal with 64-bit atomic instructions. In the parallel Bellman–Ford implementation, the duplicate vertices are generated because more threads concurrently access the same vertex for the relax operation. This causes a vertex to be added to the next-frontier more than once. To avoid duplicate vertices, Busato et al. [34] propose a technique that involves adding extra information to each vertex (in addition to the distance value). The distance of each vertex is coupled with the number of the current algorithm iteration. They are stored into a 64-bit int2 CUDA data type. The distance value is stored in the 32 most significant bits while the iteration number is stored in the 32 least significant bits. Algorithm 6 and Algorithm 7 present the high-level and low-level implementations of the atomic relax operations, respectively.

Algorithm 5: Parallel Bellman–Ford's Algorithm

```
for vertices $u \in V(G)$ do
    $d(u) = \infty$ ;
end
$d(s) = 0$;
$F_1 \leftarrow \{s\}$;
$F_2 \leftarrow \emptyset$;
while $F_1 \neq \emptyset$ do
    parallel for vertices $u \in F_1$ do
        $u \leftarrow DEQUEUE(F_1)$;
        parallel for vertices $v \in adj[u]$ do
            if $d(u) + w < d(v)$ then
                $d(v) = d(u) + w$;
                $ENQUEUE(F_2, v)$;
            end
        end
    end
    $SWAP(F_1, F_2)$;
end
```

Algorithm 6: Atomic Relax Pseudocode

```
Function Relax_Atom($u$, $v$, $w$):
    if $d(u) + w < d(v)$ then
        $d(v) = d(u) + w$;
        if $IterationNum[v] \neq currentIterationNum$ then
            ENQUEUE(V) ;
        end
    end
return
```

Algorithm 7: Atomic64 Relax Implementation

```
Function Relax_Atom($u$, $v$, $w$):
    u_info = MERGE(d(u), currentIteration);
    old_info = ATOMIC_MIN(&vertexInfo[v], u_info);
    if $old\_info.iteration \neq currentIteration$ then
        ENQUEUE(v);
    end
return
```

Algorithm 8: Parallel Dijkstra's Algorithm

```
for vertices $u \in V(G)$ do
    $d(u) = \infty$ ;
    $ud(u) = \infty$ ;
    $visited(u) = 0$ ;
end
$d(s) = 0$;
$visited(s) = 1$;
while $visited \neq \emptyset$ do
    parallel for vertices $u \in visited$ do
        $Dijkstra\_Kernel1()$;
        $Dijkstra\_Kernel2()$;
    end
end
```

Algorithm 9: Dijkstra Kernel1

```
$tid \leftarrow getThreadID$;
if $visited(tid)$ then
    $visited(tid) \leftarrow false$ ;
    for all neighbors nid of tid do
        if $ud(nid) > d(tid) + w(nid)$ then
            $ud(tid) \leftarrow d(tid) + w(nid)$;
        end
    end
end
```

Algorithm 10: Dijkstra Kernel2

```
tid ← getThreadID ;
if $visited(tid) > ud(tid)$ then
    $d(tid) \leftarrow ud(tid)$;
    $visited(tid) \leftarrow true$;
end
$ud(tid) \leftarrow d(tid)$
```

For the parallel Dijkstra's algorithm (given as a template in Algorithm 8), we utilize the implementation proposed by Harish and Narayanan [28]. We use a distance array $d(u)$, an updating distance array $ud(u)$, and a boolean visited array. In our first kernel function, ($Dijkstra\_Kernel1$ given in Algorithm 9), in each iteration, each vertex checks whether it is visited before. The distance of each of its neighbors is updated, if it is visited and the distance is larger than the summation of the distance of the current vertex and the edge weight to that neighbor. However, the new distances are not reflected in the distance array. They are updated in the updating distance array. At the end of the first kernel execution, we launch a second kernel ($Dijkstra\_Kernel2$ given in Algorithm 10). It simply compares the distance and updating distance arrays, and updates the distance array if the distance in the updating array is smaller. The second kernel for updating the distance array is necessary because there is no synchronization mechanism between the CUDA streaming multiprocessors.

Fig. 1 presents the normalized execution times of our parallel executions for our synthetic graphs (The graph details are given in Table 2). Since Bellman–Ford's algorithm is more suitable for parallelization, our parallel implementation outperforms two other algorithms. However, we apply our approximation methods to all three algorithms by

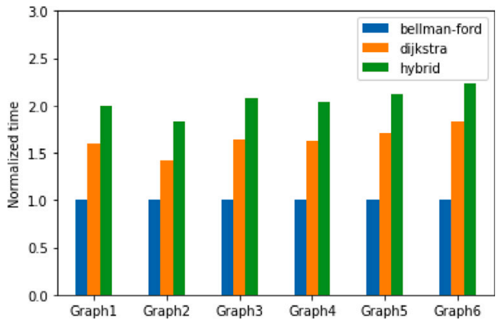


Fig. 1. Normalized execution times for shortest-path implementations.

considering their possible diverse characteristics. While Bellman–Ford potentially performs better, the approximation methods may impact both performance and accuracy in different ways.

3.2. Approximation methods

For the approximation methods, we utilize the techniques proposed by Singh and Nasre [12]. Moreover, we propose additional approximation methods specific to our target implementations. We note that the approximation methods are generic approximations rather than GPU-specific, so that they can applied to either sequential or CPU-based parallel implementations of the algorithms.

We perform the same methods for all the algorithms unless stated otherwise and provide the specific implementation for Bellman–Ford's and Dijkstra's algorithms. In our evaluation phase, we collect both speedup (i.e. performance improvement) and inaccuracy (i.e. error in the result) values for our synthetic graphs. We calculate both speedup and inaccuracy values by considering the execution time and resultant outputs of the exact versions. Since both metrics represent normalized values, instead of having the values for each graph separately, we demonstrate average inaccuracy and speedup values collected from all six graphs for four approximation methods (Figs. 2, 4, 6, 8). On the other hand, for easier comparison of our target algorithm, we include the results only for the largest graph, namely, *Graph6*, in Figs. 3, 5, 7, 9.

3.2.1. Method 1: Reduced execution

In the reduced execution approximation method, we interrupt the execution by halting the outermost loop early. Namely, we execute the loop for fewer iterations, and examine the performance improvement and inaccuracy values.

Algorithm 11: Reduced Execution for Parallel Bellman–Ford's Algorithm

```
1  while $round < \underline{iterationNumber}$ do
2      parallel for vertices $u \in F_1$ do
3          $u \leftarrow DEQUEUE(F_1)$;
4          parallel for vertices $v \in adj[u]$ do
5              if $d(u) + w < d(v)$ then
6                  $d(v) = d(u) + w$;
7                  $ENQUEUE(F_2, v)$;
8              end
9          end
10     end
11     $SWAP(F_1, F_2)$;
12 end
```

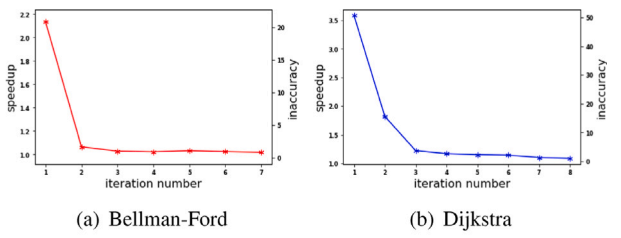
Algorithm 12: Reduced Execution for Parallel Dijkstra's Algorithm

```
1  while $visited \neq \emptyset$ && $round < \underline{iterationNumber}$ do
2      parallel for vertices $u \in visited$ do
3          $Dijkstra\_Kernel1()$;
4          $Dijkstra\_Kernel2()$;
5      end
6  end
```

As seen in the first line of both Algorithm 11 and Algorithm 12, if the current *round* number is smaller than the desired *iteration number*, the algorithm continues to work, otherwise it halts early. By eliminating the number of iterations in the original execution, we expect higher performance with lower total execution time.

We examine each possible iteration number for the given graphs and the effects on both performance and inaccuracy percentage. By reducing the number of iterations, we achieve performance gains from several steps like launching the kernel, copying back and forth the edges to be processed, the work done in the kernel itself, and reducing the number of atomic operations. We mostly observe small changes in accuracy.

Fig. 2 presents how speedup and inaccuracy values change by executing the code with different iteration counts. We observe that cutting off the last two iterations does not lead to an error but improves the execution time. We examine these last two iterations, and find out that very little work (2–10 edges added to the queue) is done and mostly they do not affect the outcome of the algorithms (but not guaranteed). We also observe that the algorithms complete their work in 7–8 iterations.



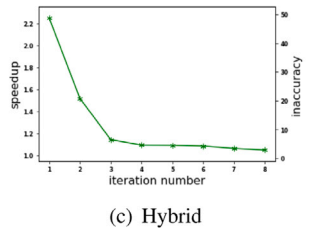


Fig. 2. Speedup-Inaccuracy variation with various iteration counts (average values of all synthetic graphs).

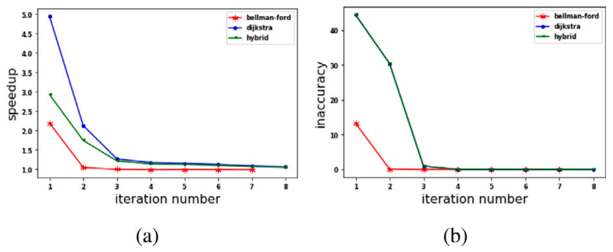


Fig. 3. Comparison of the algorithms' speedup and inaccuracy variation with iteration count (values for *Graph6* as a representative graph).

In Bellman–Ford's algorithm, most of the work is done in the first two iterations. If we cut off the rest of the iterations, the inaccuracy rate becomes less than 2%. In Dijkstra's algorithm, most of the work is done in the first three iterations. If we cut off the rest of the iterations, the inaccuracy rate becomes less than 1%. In Hybrid algorithm, we execute Dijkstra's algorithm several times so that it could also work with the graphs having negative weights. If there is no negative weight, then the algorithm runs Dijkstra twice. When the iteration number of the algorithm is reduced, it still runs Dijkstra twice, however, at each run it stops the algorithm prematurely. If we cut off the last five iterations, the error rate becomes less than 1%.

Fig. 3 demonstrates speedup and inaccuracy change separately for the comparison of our target algorithms. The inaccuracy values of Dijkstra and Hybrid algorithms are the same since they both run the same algorithm. Dijkstra seems to reach to higher speedup (~3.5 times) compared to the others while it results in very high inaccuracy (~50%) with those higher speedup levels.

3.2.2. Method 2: Minimum edge number selection

In the minimum edge number selection approximation method, we do not process the nodes that have less than the specified edge number. Thus, we reduce the number of atomic operations inside the kernel. With fewer atomic operations, which are serial executions, the method gives us a great advantage to speed up the execution with mostly a minor impact on accuracy. As seen at lines 4 of Algorithm 13 and Algorithm 14, we apply this technique by checking the number of edges of the current vertex and continue processing if it has more than the

Algorithm 13: Minimum Edge Number Selection for Parallel Bellman–Ford's Algorithm

```
1  while $F_1 \neq \emptyset$ do
2      parallel for vertices $u \in F_1$ do
3          $u \leftarrow DEQUEUE(F_1)$;
4          if $u$ has at least minEdgeNumber edges then
5              parallel for vertices $v \in adj[u]$ do
6                  if $d(u) + w < d(v)$ then
7                      $d(v) = d(u) + w$;
8                      $ENQUEUE(F_2, v)$;
9                  end
10             end
11         end
12     end
13     $SWAP(F_1, F_2)$;
14 end
```

Algorithm 14: Minimum Edge Number Selection for Dijkstra Kernel1

```
1  $tid \leftarrow getThreadID$;
2  if $visited(tid)$ then
3      $visited(tid) \leftarrow false$ ;
4      if $edgeNumber\ of\ tid > minEdgeNumber$ then
5          for all neighbors nid of tid do
6              if $ud(nid) > d(tid) + w(nid)$ then
7                  $ud(tid) \leftarrow d(tid) + w(nid)$;
8              end
9          end
10     end
11 end
```

specified number of edge. It guarantees that the nodes to be processed have more edges than the specified number. To find out the minimum number of edges to process, the edge distribution on the nodes is preprocessed. Firstly, the number of edges of each node is sorted in non-decreasing order by degree. Then, the nodes that do not have any edges are removed. The rest of the array is divided into 10 chunks, and the last chunk is divided into 10 chunks as well. Since the nodes with fewer edges are much more than the nodes with many edges, we divide the array in this way. This preprocessing, which is applied before the execution of the main process, has a small overhead but it achieves good improvement on the execution time with a small error rate. On average, 25% speed-up is achieved with less than 6% error rate by processing the top 1% of the nodes (see Figs. 4 and 5). We must note that these are average speed-ups. We observe that when the graph size increases, the speedup increases, and the inaccuracy rate decreases.

3.2.3. Method 3: Maximum edge number selection

We perform approximation not only by restricting the lower bound of the edge numbers of the nodes but also by restricting the upper bound of the edge number of the nodes. We simply check the number of the edges for the current vertex, and continue the execution if the vertex has less than the specified edge value. With this upper limit and potentially with fewer vertices processed simultaneously, we reduce the number of atomic operations. Additionally, since the work done by each thread is limited, each thread works on a similar amount of data; thus, the technique achieves load balance. Hence, we expect to have performance improvements for the target execution. Similar to the minimum edge number selection method, we apply this technique by modifying the *if* statement at line 4 in Algorithm 13 and Algorithm 14 by setting a maximum number instead of a minimum. It guarantees that

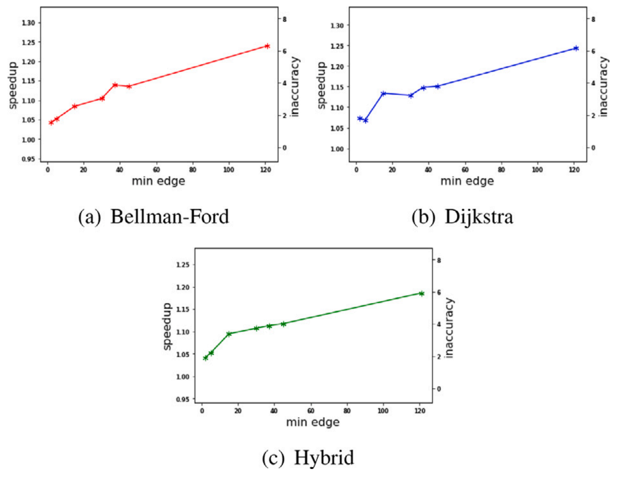


Fig. 4. Speedup-Inaccuracy variation with minimum edge processing (average values of all synthetic graphs).

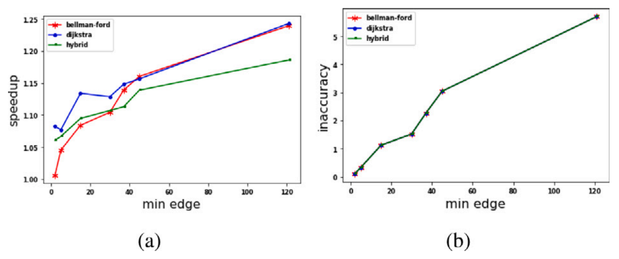


Fig. 5. Comparison of the algorithms' speedup and inaccuracy variation with minimum edge processing (values for *Graph6* as a representative graph).

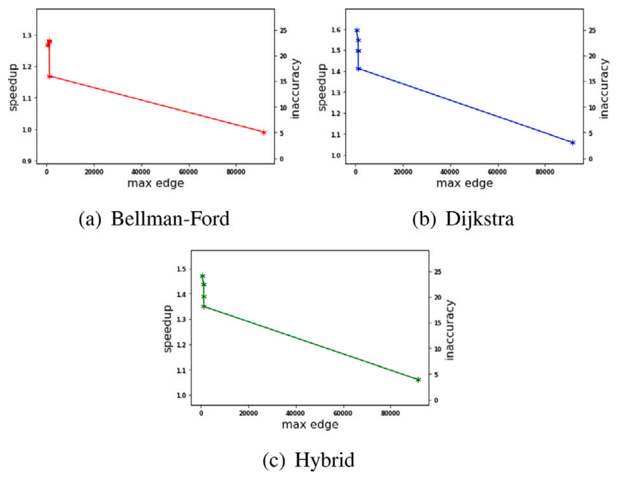
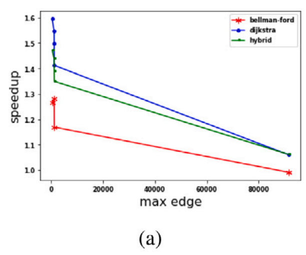


Fig. 6. Speedup-Inaccuracy variation with maximum edge processing (average values of all synthetic graphs).

the nodes to be processed have fewer edges than the specified number. We perform similar preprocessing to find out the maximum number of edges to process. Firstly, the number of edges of each node is sorted in non-decreasing order by degree. Then, the nodes that do not have any edges are removed. The rest of the array is divided into 10 chunks, and the last chunk is divided into 10 chunks as well. Again, the last chunk is divided into 10 chunks, and we take the last 10 chunks, which is equivalent to the top 0.1% of the nodes. On average, 50% speed-up is achieved with less than 25% error rate (see Figs. 6 and 7).



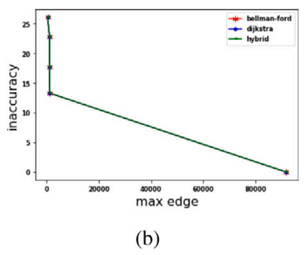


Fig. 7. Comparison of the algorithms' speedup and inaccuracy variation with maximum edge processing (values for *Graph6* as a representative graph).

3.2.4. Method 4: Partial queue processing

In this approximation method, in each iteration, we take some percentage of the nodes randomly to send it to the kernel to be processed. We expect to see performance improvements due to the smaller number of vertex processing with this method. However, queue reconstruction takes additional time as the cost of the technique and can affect the execution time. Our experimental results show that the change in the percentage rate to be processed in the queue does not affect the error rate significantly. However, processing the small percentage of the queue (nearly 25%) reduces the execution time by almost 30% with less than 30% error rate in Bellman–Ford algorithm (see Fig. 8(a)). However, the other algorithms do not perform similarly (see Fig. 9). In contrast to Bellman–Ford, Dijkstra and Hybrid algorithms get slow down by the partial queue processing approximation. The reason is that we process the nodes to be added to the queue in the GPU kernel in our Bellman–Ford implementation (Line 1 in Algorithm 15). However, in the Dijkstra code, we keep track of the visited nodes and have to preprocess them in the CPU (Line 1 in Algorithm 16). Due to the additional CPU computation, the execution time for Dijkstra and Hybrid algorithms increases.

Algorithm 15: Partial Queue Processing for Parallel Bellman–Ford's Algorithm

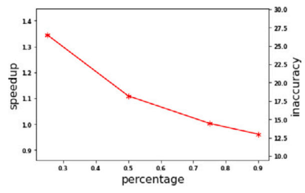
```
1  $F_1 = F_1 - \underline{nodesEliminated}$;
2  parallel for vertices $u \in F_1$ do
3      $u \leftarrow DEQUEUE(F_1)$;
4      parallel for vertices $v \in adj[u]$ do
5          if $d(u) + w < d(v)$ then
6              $d(v) = d(u) + w$;
7              $ENQUEUE(F_2, v)$;
8          end
9      end
10 end
11 $SWAP(F_1, F_2)$;
```

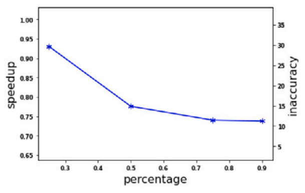
Algorithm 16: Partial Queue Processing for Parallel Dijkstra's Algorithm

```
1  $visited = visited - \underline{nodesEliminated}$;
2  while $visited \neq \emptyset$ do
3      parallel for vertices $u \in visited$ do
4          $Dijkstra\_Kernel1()$;
5          $Dijkstra\_Kernel2()$;
6      end
7  end
```

3.2.5. Combinations of approximation methods

In the previous sections, we see that even though the individual approximation methods help us achieve some speedups with tolerable





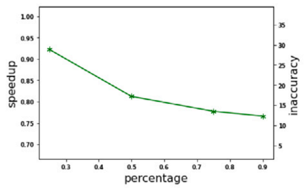
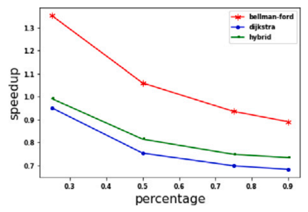


Fig. 8. Speedup-Inaccuracy variation with partial queue processing (average values of all synthetic graphs).



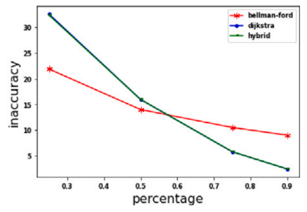


Fig. 9. Comparison of the algorithms' speedup and inaccuracy variation with partial queue processing (values for *Graph6* as a representative graph).

inaccuracy percentages, they do not provide high speedups. Therefore, we combine those approximation techniques with each other and achieve higher speedups with less inaccuracy percentage. Table 1 presents some of the combinations, and their speedup and inaccuracy percentage. We perform much more approximations by combining the techniques with different parameters.

3.3. Prediction model

As demonstrated in the previous section, the parallel Bellman–Ford's algorithm performs the best among our target algorithms and its approximate versions also employ larger performance improvements with lower errors in the results. Therefore, we focus on Bellman–Ford's algorithm in our prediction study and include its executions in our dataset. We perform the approximation techniques with different parameters and obtain a wide range of speedup and inaccuracy values. While one technique maintains larger speedup values, it can miscalculate the result value substantially. Similarly, the parameters of the approximation methods can greatly impact the speedup and inaccuracy results. Since it is impractical to apply all the possible techniques with their various parameters, we build regression models to predict the impact of the approximation methods without executing the codes. Especially, it is important to obtain the effects for large graph processing scenarios. We aim to predict both speedup and inaccuracy percentage values for the target large-size graphs by executing the approximate codes for relatively small graphs. Additionally, we execute our program versions for real road-network graphs and include them in our evaluation in order to investigate the efficiency of our prediction model by considering realistic scenarios.

Table 1
The examples for the combination of the approximation techniques.

	iterationNum	maxEdgeDeg	maxProcessEdge	minProcessEdge	Graph%	Inaccuracy	Speedup
1	2	25054	1340	0	100	9.88	2.27
2	8	25054	1340	3	100	9.35	2.00
3	8	25054	1340	6	100	9.70	2.07
4	4	25054	1340	3	100	9.35	2.02
5	8	25054	465	21	100	19.83	3.02
6	2	25054	1283	42	100	19.32	3.11
7	7	39530	1996	92	100	19.81	3.26
8	2	39530	694	29	100	18.40	3.11
9	2	39530	1996	92	100	19.83	3.31
10	2	25054	465	50	100	24.89	4.46
11	2	25054	454	50	100	27.22	4.77
12	5	39530	676	92	100	28.13	4.36
13	6	39530	676	92	100	28.13	4.48

Table 2
Graphs used in our experiments.

Graph name	Type	Nodes	Edges
Graph1	Synthetic	524288	7968035
Graph2	Synthetic	1048576	16084739
Graph3	Synthetic	2097152	32417950
Graph4	Synthetic	4194304	65243481
Graph5	Synthetic	8388608	131155371
Graph6	Synthetic	16777216	263434146
BAY	road-network	321270	800172
CAL	road-network	1890815	4657742
COL	road-network	435666	1057066
CTR	road-network	14081816	34292496
E	road-network	3598623	8778114
FLA	road-network	1070376	2712798
LKS	road-network	2758119	6885658
NE	road-network	1524453	3897636
NW	road-network	1207945	2840208
NY	road-network	264346	733846
USA	road-network	23947347	58333344
W	road-network	6262104	15248146

Table 3
The features and their ranges for our prediction model.

Feature	Range	Data type
graphSize	1–32	Discrete
iterationNum	0–1	Continuous
minProcessEdge	0–1	Continuous
maxProcessEdge	0–1	Continuous
percentage	0–1	Continuous
sOriginalDistance	0 or 1	Categorical
sMinEdgeToProcess	0 or 1	Categorical
sMaxEdgeToProcess	0 or 1	Categorical
sPartialGraphProcess	0 or 1	Categorical
sReduceExecution	0 or 1	Categorical
sAtomicBlock	0 or 1	Categorical
error	0–1	Continuous
speedup	0–1	Continuous

3.3.1. Data format and size

We apply individual or combined approximation methods, run the code in our target architecture, calculate the shortest path, and save the related data as a *csv* file format. Specifically, we record the following data at each time a graph is processed and the shortest path is calculated: (1) the total number of vertices in the graph, (2) the total number of edges in the graph, (3) the possible maximum number of edges for a single vertex in the graph, (4) the total number of iterations in the main loop of the algorithm, (5) the number of minimum processing edge in the kernel (for each vertex that contains higher number of the specified edges), (6) the number of maximum processing edge in the kernel (for each vertex that contains lower number of the specified edges), (7) the percentage number that identifies how many of the vertices in the queue should be processed in the next iteration, (8) the signals of the approximation techniques (it is 1 if the specified approximation technique is applied in the calculation of the shortest path, otherwise 0), (9) the inaccuracy rate of the final calculation of the shortest path, (10) the time elapsed for the execution of the algorithm.

For the execution time measurement, we only consider the kernel execution time, and do not take into consideration the preprocessing steps, since we want to clearly see the effects of the approximation techniques. We first calculate the shortest path for the given graph without applying any approximation methods and save the computed path as the accurately calculated distance. We utilize this path data to compare the outcome of the executions, when we perform the approximation techniques. In order to make this comparison, we need to start at the same vertex for each case. Therefore, we choose a standard starting vertex, which has the highest number of edge degrees. We perform the approximation techniques including *Reduced Execution*, *Minimum Edge Number Selection*, *Maximum Edge Number Selection*, and *Partial Queue Processing*, individually, and also combine these approximation techniques (two, three, and four of them) to see if we can achieve higher speedups with fewer errors in our computations. We consider multiple test scenarios for each synthetic graph in different sizes. These graphs are generated by the Kronecker generator [14], which creates an edge list according to the Graph500 parameters [35]. The edge list is returned in an array with three rows, where $StartVertex$ is the first column, $EndVertex$ is the second column, and $Weight$ is the third column. We use the scales 19, 20, 21, 22, 23, 24, and the edge factor 16 for our generated graphs. Additionally, in our prediction part, we consider 12 real road networks from the 9th DIMACS Implementation Challenge [15]. Table 2 presents the number of nodes and edges for our target graphs. Since synthetic graphs and real graphs own diverse characteristics, we build separate regression models for them.

We run each test case 10 times, discard the values with minimum and maximum execution times, take the average of the remaining times, and record the average as the execution time for each case. For some of the approximation techniques, where the outcome of the calculated shortest path is not stable and changeable (taking the random vertices in the queue to process in the next cycle), we take the average of the errors in the calculations.

3.3.2. Preprocessing

We use data normalization techniques to improve our prediction accuracy. Data normalization is one of the most important preprocessing steps in machine learning since it is common to have data with different ranges [36]. The machine learning algorithms perform better if data is normalized to the same range. If all the data has different ranges, it increases the difficulty of the problem that is being modeled. For instance, large input values, such as the number of vertices in the graph, may result in a model that learns large weight values. This makes the model unstable and it suffers from poor performance issues during learning. $graphSize$ is the projection of the number of vertices. Normally, the vertex numbers of graphs are large values (e.g., power of 2 values between 2^{19} and 2^{24} for synthetic graphs), but we fit these numbers between 1 and 32 by assigning values between 2^0 and 2^5. The number of iterations, $iterationNum$ takes values in a varying range but it is normalized between 0 and 1 as stated in Table 3. $minProcessEdge$ represents the vertices that have the specified number of edges or more, while $maxProcessEdge$ represents the vertices that have the specified number of edges or less. Their value depends on the graph size and the graph characteristics, since they are chosen by specific rules (as explained earlier in Section 3). For example, $minProcessEdge$ varies between 2 and 120 and $maxProcessEdge$ varies between 359 and 238592 for our synthetic graphs. In the preprocessing step, they are normalized to the values in the range between 0 and 1.

The normalization formula used in the preprocessing phase is as follows:

$$X_{normalized} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{1}$$

After performing preprocessing, we have the features given in Table 3 for our prediction model. Since the graph size is one of the crucial parts to estimate the execution time, we use the number of vertices in the graph ($graphSize$) as our feature in the prediction model. We use the iteration number of the main loop ($iterationNum$), as the iteration number increases the execution time increases as well while inaccuracy decreases. The partial graph processing percentage ($percentage$) determines how many vertices will be processed in the queue. The number of minimum edges ($minProcessEdge$) and the maximum edges ($maxProcessEdge$) to process are used as our features because their value has an effect on the speedup and inaccuracy as well. Additionally, the signal values starting with s in Table 3 are included as our feature because they contain the information of which approximation technique is applied. Finally, for the sake of multiple-output prediction, we normalize the inaccuracy and speedup values to the same range between 0 and 1 so that the prediction results of each output would be more accurate. As a result, the aforementioned features and their represented values play an important role in the prediction step.

3.3.3. Machine learning algorithms

In our prediction model, we utilize the following five regression algorithms.

- *Linear Regression (LR)* is a supervised learning algorithm. It is a linear model and assumes a linear relationship between input and output variables and predicts the dependent variable based on a given independent variable.
- *K-Nearest Neighbors (KNN) Regression* depends on the nearest neighbors of each point for learning. The output is predicted by local interpolation of the targets associated with the nearest neighbors in the training set.
- *Random Forest (RF) Regression* constructs a very large number of classifying decision trees on various sub-samples of the training dataset. The prediction is the average prediction across the decision trees.
- *Decision Tree (DT) Regression* trains a model in the structure of a tree and predicts the value of a target variable by learning simple decision rules. It fits a sine curve and learns local linear regressions approximating the sine curve with a set of if-then-else decision rules.
- *Gradient Boosting (GB) Regression* builds the model in a stage-wise fashion and allows for the optimization of arbitrary differentiable loss functions.

Firstly, we predict the single output values (i.e., inaccuracy and execution time separately), then the multiple outputs (i.e., inaccuracy and execution time jointly) with those regression algorithms. We evaluate the performance of the predictors with different evaluation metrics given as follows:

- *Mean Absolute Error (MAE)* measures the average magnitude of the errors. It is the average over the test samples of the absolute differences between the predicted and the observed values, where all individual differences have equal weight.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - x_i| \tag{2}$$

- *Mean Squared Error (MSE)* is simply the average of the squares of the errors.

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (y_i - x_i)^2 \tag{3}$$

- *Root Mean Squared Error (RMSE)* is a quadratic scoring rule that measures the average magnitude of the errors. It is the standard deviation of the residuals (the prediction errors). The residuals are a measure of how far from the regression line the data points are. RMSE is a measure of how to spread out these residuals are so it tells you how concentrated the data is around the line of the best fit.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (y_i - x_i)^2}{n}} \tag{4}$$

- *R2 score* represents the proportion of the variance that has been explained by the independent variables in the model. It provides an indication of the goodness or the badness of the fit. It is a measure of how likely unseen samples are to be predicted by the model through the proportion of the explained variance. The best possible score is 1.0 and it can be negative as well if the model is arbitrarily worse. Since R2 is adopted in various research disciplines, there is no standard guideline to determine the level of predictive acceptance. However, Henseler et al. [37] propose a rule of thumb, which describes R2 values with 0.75, 0.50, and 0.25 as substantial, moderate, and weak, respectively.

$$R^2 = 1 - \frac{RSS}{TSS} \tag{5}$$

$$Residual\ Sum\ of\ Squares\ (RSS) = \sum_{i=1}^{n} (y_i - x_i)^2 \tag{6}$$

$$Total\ Sum\ of\ Squares\ (TSS) = \sum_{i=1}^{n} (y_i - \overline{y}) \tag{7}$$

where n represents the total number of predictions, y and x represent the predicted and the observed values, respectively.

4. Experimental study

4.1. Experimental setup

We compile our programs with CUDA 9.0 and run the approximation experiments in an Intel Xeon-based workstation with 2x Xeon Silver 4114 processors, 32 GB main memory and an NVIDIA Quadro P4000 GPU device. For our prediction model, we utilize the algorithms implemented in the scikit-learn library [38].

4.2. Experimental results

In this section, we present the results of different machine learning algorithms in our prediction models.

Table 4 presents the prediction results of our models in terms of *Mean Absolute Error (MAE)*, *Root Mean Squared Error (RMSE)*, and *R2 score*. While we report three metrics to demonstrate our results, we focus on*R2 score* to evaluate the prediction models. We consider three different prediction scenarios including the prediction of only inaccuracy rate (*Inaccuracy*), only speedup (*Speedup*), and both inaccuracy rate and speedup (*Multiout*). Essentially, *Multiout* model aims

Table 4
Prediction results of ML algorithms with different metrics.

		MAE			RMSE			R2		
		Multiout	Inaccuracy	Speedup	Multiout	Inaccuracy	Speedup	Multiout	Inaccuracy	Speedup
LR	rnd[a]	0.085	0.054	0.117	0.122	0.070	0.158	0.661	0.792	0.531
	large[b]	0.111	0.085	0.137	0.158	0.102	0.198	0.352	0.456	0.249
	real[c]	0.093	0.020	0.165	0.157	0.024	0.221	0.587	0.591	0.583
KNN	rnd	0.025	0.032	0.018	0.038	0.043	0.031	0.951	0.921	0.982
	large	0.042	0.056	0.028	0.063	0.069	0.056	0.843	0.748	0.939
	real	0.032	0.008	0.056	0.070	0.012	0.099	0.905	0.893	0.916
RF	rnd	0.002	0.001	0.003	0.005	0.002	0.006	0.999	0.999	0.999
	large	0.020	0.023	0.018	0.029	0.031	0.028	0.969	0.951	0.985
	real	0.030	0.008	0.054	0.069	0.012	0.097	0.908	0.896	0.920
DT	rnd	0.002	0.001	0.004	0.007	0.003	0.009	0.999	0.999	0.999
	large	0.020	0.023	0.018	0.029	0.031	0.029	0.969	0.951	0.984
	real	0.031	0.008	0.054	0.072	0.012	0.101	0.900	0.893	0.913
GB	rnd	0.014	0.012	0.015	0.020	0.017	0.022	0.989	0.987	0.991
	large	0.026	0.024	0.027	0.036	0.032	0.040	0.957	0.945	0.970
	real	0.049	0.008	0.089	0.091	0.011	0.127	0.885	0.909	0.860

[a]Random selection of train and test data.
[b]Prediction of the largest graph results from small graphs.
[c]Prediction of the real graph results.

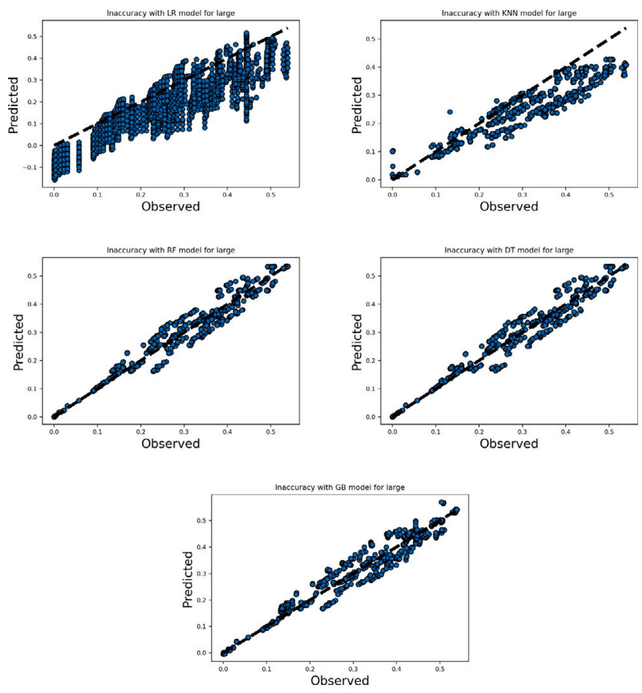


Fig. 10. Observed and predicted **inaccuracy** values with **single output** prediction model for **large** graphs.

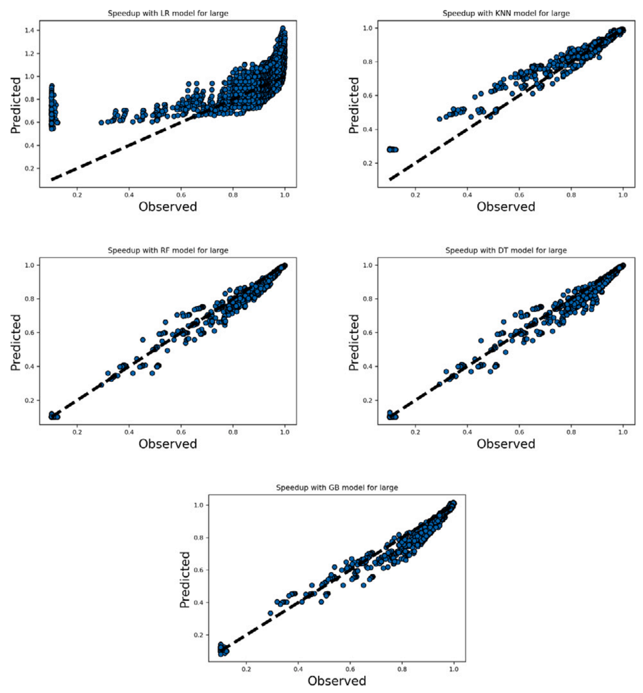


Fig. 11. Observed and predicted **speedup** values with **single output** prediction model for **large** graphs.

to predict both inaccuracy and speedup at the same time, and its metrics, namely *MAE*, *RMSE* and *R2*, are calculated by taking the average of both outcomes. We report *Multiout* model results to better understand the prediction success on both *inaccuracy* and *speedup* at the same time. Moreover, we build prediction models by utilizing different training/test data points. Firstly, given as rnd in Table 4, we randomly split the synthetic graph data points as training and test data, 80% and 20%, respectively, and apply our ML algorithms to predict the test data points from the training. Secondly, given as $large$ in Table 4, we build our prediction model such that we train the model with the data points (obtained by applying all approximation methods) belonging to the 5 smallest synthetic graphs, then we make predictions for the data points (approximation methods) belonging to the largest graph (*Graph6*). We specifically choose this training and test data to see if we can predict the approximation results of the large graphs from the small graphs, without executing the approximate versions for large graph data that requires unreasonable times. Additionally, $real$ rows in Table 4 present the prediction results for our regression models built from 12 real road-network graphs. Specifically, we split the data points as training and test data, 80% and 20%, respectively, and apply our ML algorithms to predict the real data points from the training.

As seen in Table 4, for all three prediction scenarios (i.e., *Inaccuracy*, *Speedup*, and *Multiout* predictions), the prediction models, where we split data randomly, have much lower prediction errors than the models, where we utilize small graphs to predict the large graph outcomes. Since we have larger speedup values and more diverge behavior for the large graphs, the model that does not have any data point in its

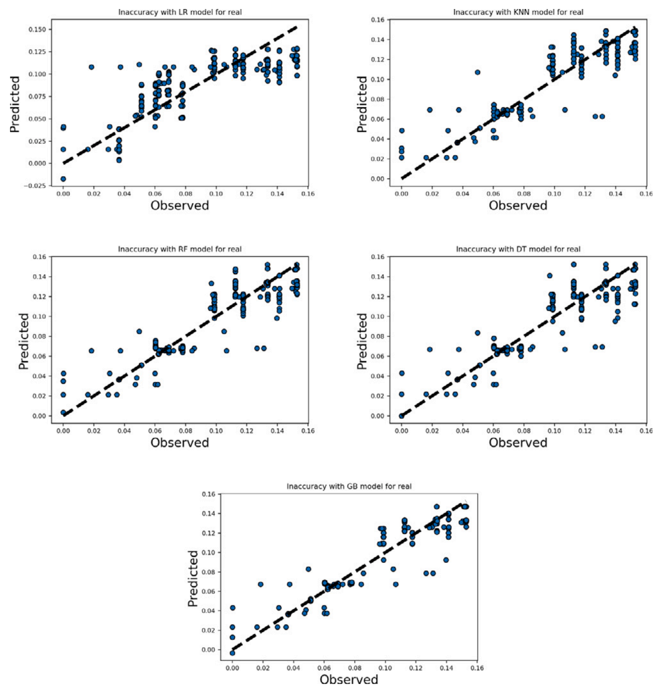


Fig. 12. Observed and predicted **inaccuracy** values with **single output** prediction model for **real** graphs.

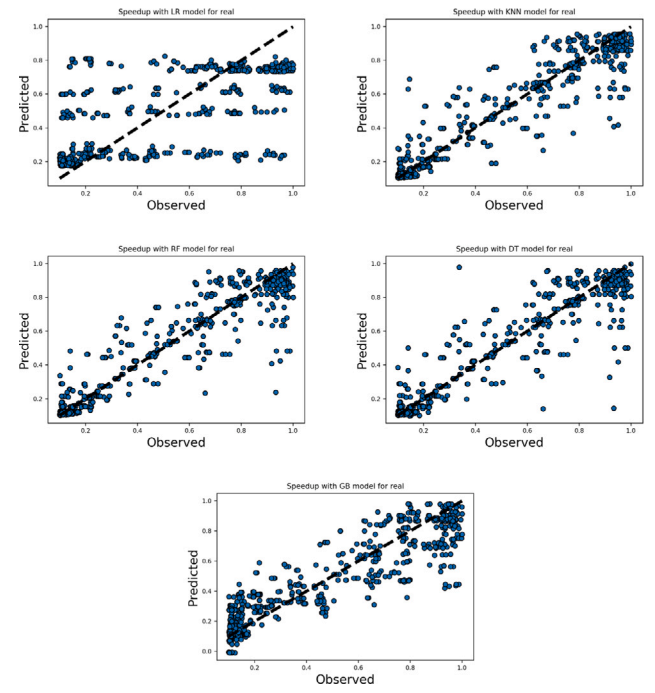


Fig. 13. Observed and predicted **speedup** values with **single output** prediction model for **real** graphs.

training set for large graph executions fails to make correct estimations. Additionally, the prediction models based on real-graph data employ higher prediction errors. Especially, for very large real graph instances (e.g., *USA*), the speedup values with approximate versions are substantial and difficult to predict with relatively smaller graph data.

As mentioned in Section 3.3.3, we use five different machine learning algorithms to fit our data. These algorithms are *Linear Regression (LR)*, *K-Nearest Neighbors (KNN)*, *Random Forest (RF)*, *Decision Tree (DT)*, and *Gradient Boosting (GB)*. While we see from Table 4, *RF*, *DT*, and *GB* achieve low prediction errors, which means their predictions are very close to the actual values, *LR* suffers from poor prediction results for all the scenarios. Since our model is too complex and not linear, *LR* algorithm does not perform well. When we look at *KNN* results, we can see that *KNN* performs well when data is split randomly (rnd). However, it produces larger prediction errors when data is split into small graphs and large graphs ($large$) as well as for real graph scenarios ($real$). Since *KNN* does not model the nearest points of the graphs, which it never sees (the training data does not include the largest graph), it suffers from poor predictions.

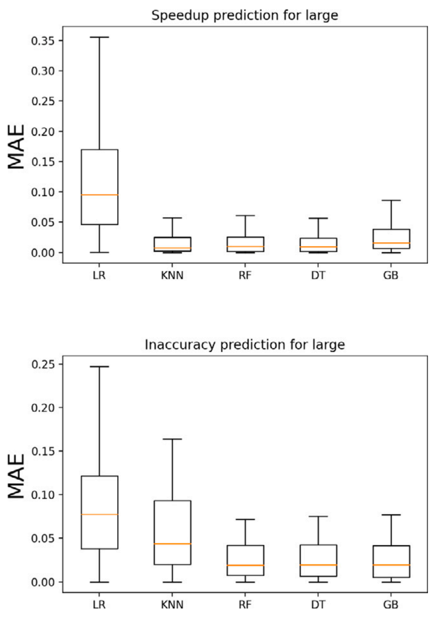


Fig. 14. **Speedup** and **inaccuracy** prediction errors for **single output** models with **large** graphs.

We can also note that the prediction results of the three prediction scenarios (i.e., *Inaccuracy*, *Speedup*, and *Multiout*) are very close to each other. Predicting the impact of the approximation methods on inaccuracy and speedup, either separately or together, yields similar results due to our normalization procedure. Both inaccuracy and speedup values contribute the prediction outcomes in a similar way.

Figs. 10 and 11 present observed and predicted inaccuracy and speedup values with the models that the outcomes are predicted separately (*single output*) for large graph prediction scenarios. We use the model, where training data includes smaller graphs and test data consists of the largest graph data points. *LR* fails in the cases that test values do not lie in the specific lines. For both speedup and inaccuracy values, *LR* tends to fit the data to the local lines, however, test data behaves non-linearly, consequently, *LR* does not catch that behavior. Moreover, the difference between predicted and test values for *KNN* is very large, especially for inaccuracy values. Since *KNN* works with the common intuition, in which data points with similar features tend to be similar, it does not successfully predict the values for data points that it has never seen before. *Inaccuracy* values for the large graphs are relatively larger than the small graphs, which are in training data set, *KNN* stucks at small values that exist in the training set (for small graphs) and fails to predict the larger values in the test set (for the largest graph). On the other hand, as more complex algorithms, *RF*, *DT*, and *GB* perform better to fit test data. While they do not estimate the exact values, they predict the trend, and find more accurate (with less difference) results for all the cases. Although test data points spread over a wide space, those three algorithms can model the trend better and perform well for the prediction of both *Inaccuracy* and *Speedup* values.

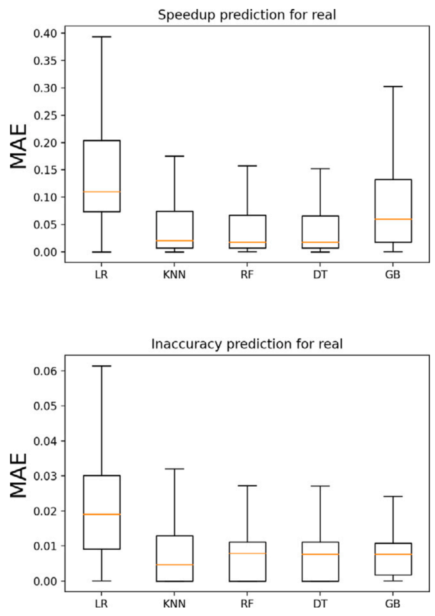


Fig. 15. **Speedup** and **inaccuracy** prediction errors for **single output** models with **real** graphs.

Figs. 12 and 13 present the observed and predicted inaccuracy and speedup values for *real* graph prediction scenarios. Our observation for the large-graph prediction case is also valid for these graph instances. While *LR* tends to fit the data points linearly, the other ML algorithms can model the diverse behavior more successfully by employing small differences between observed and predicted values. On the other hand, the success rates of the algorithms are not as high as *rnd* and *large* instances due to the real graphs' diverse behavior (not uniform as our synthetic graphs).

Figs. 14 and 15 present the percentage error rates for different single output regression models. As discussed earlier in this section, *LR* and *KNN* prediction error rates are large for the models predicting *large* graph instances. Additionally, the variance in the error rates is larger due to the failure in the model predictions. Since both LR and *KNN* (especially for *Inaccuracy* values) are not able to predict the pattern, the prediction success rates are also not stable. We can say that they perform randomly for some cases other than making intelligent predictions. For *real* graph predictions, *LR* performs the worst, however the other algorithms employ similar prediction error rates, which are higher than the *large* graph predictions.

5. Conclusion

We present a prediction methodology for the approximation methods applied in GPU-based shortest-path graph algorithms. Based on our implementations for three shortest-path algorithms, we perform approximations for higher performance by sacrificing some accuracy in the results. Our prediction approach estimates inaccuracy and speedup values for the specific approximations targeting the specific algorithm. By utilizing our approach, one can find out the effects of the approximation methods on both performance and correctness of the target execution without executing the approximate shortest-path program. Especially, for large graphs, it is practical to understand the approximation effects and choose the suitable approximation technique based on the requirements of the program.

CRediT authorship contribution statement

Busenur Aktılav: Data curation, Methodology, Software, Writing – original draft. **Işıl Öz:** Conceptualization, Validation, Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK), Grant No: 119E011.

References

[1] S. Mittal, J.S. Vetter, A survey of methods for analyzing and improving GPU energy efficiency, ACM Comput. Surv. 47 (2) (2014) http://dx.doi.org/10.1145/2636342.

[2] S. Mittal, A survey of techniques for approximate computing, ACM Comput. Surv. 48 (4) (2016).

[3] M. Samadi, J. Lee, D.A. Jamshidi, A. Hormati, S. Mahlke, SAGE: Self-tuning approximation for graphics engines, in: 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2013, pp. 13–24.

[4] D. Maier, B. Cosenza, B. Juurlink, Local memory-aware kernel perforation, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, in: CGO 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 278–287, http://dx.doi.org/10.1145/3168814.

[5] D. Peroni, M. Imani, H. Nejatollahi, N. Dutt, T. Rosing, ARGA: Approximate reuse for GPGPU acceleration, in: 2019 56th ACM/IEEE Design Automation Conference, DAC, 2019, pp. 1–6.

[6] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti, A. Daglis, N.E. Jerger, B. Falsafi, S. Misailovic, A. Sampson, D. Zufferey, Exploiting errors for efficiency: A survey from circuits to applications, ACM Comput. Surv. 53 (3) (2020) http://dx.doi.org/10.1145/3394898.

[7] T.M. Aamodt, W.W.L. Fung, T.G. Rogers, M. Martonosi, General-Purpose Graphics Processor Architecture, 2018.

[8] P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: S. Aluru, M. Parashar, R. Badrinath, V.K. Prasanna (Eds.), High Performance Computing – HiPC 2007, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 197–208.

[9] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU graph traversal, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, in: PPoPP '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 117–128, http://dx.doi.org/10.1145/2145816.2145832.

[10] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J.D. Owens, Gunrock: A high-performance graph processing library on the GPU, SIGPLAN Not. 51 (8) (2016) http://dx.doi.org/10.1145/3016078.2851145.

[11] S. Hong, T. Oguntebi, K. Olukotun, Efficient parallel graph exploration on multi-core CPU and GPU, in: 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 78–88, http://dx.doi.org/10.1109/PACT.2011.14.

[12] S. Singh, R. Nasre, Scalable and performant graph processing on GPUs using approximate computing, IEEE Trans. Multi Scale Comput. Syst. 4 (3) (2018) 190–203.

[13] A. Panyala, O. Subasi, M. Halappanavar, A. Kalyanaraman, D. Chavarria-Miranda, S. Krishnamoorthy, Approximate computing techniques for iterative graph algorithms, in: 2017 IEEE 24th International Conference on High Performance Computing, HiPC, 2017, pp. 23–32, http://dx.doi.org/10.1109/HiPC.2017.00013.

[14] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Z. Ghahramani, Kronecker graphs: An approach to modeling networks, J. Mach. Learn. Res. 11 (2010) 985–1042.

[15] 9th DIMACS Implementation Challenge, http://www.dis.uniroma1.it/challenge9/download.shtml.

[16] G. Wu, J.L. Greathouse, A. Lyashevsky, N. Jayasena, D. Chiou, GPGPU performance and power estimation using machine learning, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA, 2015, pp. 564–576, http://dx.doi.org/10.1109/HPCA.2015.7056063.

[17] I. Öz, M.K. Bhatti, K. Popov, M. Brorsson, Regression-based prediction for task-based program performance, J. Circuits Syst. Comput. 28 (4) (2019) 1950060:1–1950060:30, http://dx.doi.org/10.1142/S0218126619500609.

[18] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, S. Amarasinghe, Opentuner: An extensible framework for program autotuning, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, in: PACT, vol. 14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 303–316, http://dx.doi.org/10.1145/2628071.2628092.

[19] A. Panyala, D. Chavarría-Miranda, J.B. Manzano, A. Tumeo, M. Halappanavar, Exploring performance and energy tradeoffs for irregular applications: A case study on the Tilera many-core architecture, J. Parallel Distrib. Comput. 104 (2017) 234–251, http://dx.doi.org/10.1016/j.jpdc.2016.06.006.
[20] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, Q.-S. Hua, Graph processing on GPUs: A survey, ACM Comput. Surv. 50 (6) (2018) http://dx.doi.org/10.1145/3128571.
[21] S. Che, B.M. Beckmann, S.K. Reinhardt, K. Skadron, Pannotia: Understanding irregular GPGPU graph applications, in: 2013 IEEE International Symposium on Workload Characterization, IISWC, 2013, pp. 185–195, http://dx.doi.org/10.1109/IISWC.2013.6704684.
[22] D. Michail, J. Kinable, B. Naveh, J.V. Sichi, Jgrapht—A java library for graph data structures and algorithms, ACM Trans. Math. Software 46 (2) (2020) http://dx.doi.org/10.1145/3381449.
[23] C. Yang, A. Buluc, J.D. Owens, GraphBLAST: A high-performance linear algebra-based graph framework on the GPU, 2021, http://arxiv.org/abs/1908.01407 [arXiv:1908.01407].
[24] D.B. West, et al., Introduction to Graph Theory, Vol. 2, Prentice hall Upper Saddle River, 2001.
[25] N. Ganganath, C.-T. Cheng, T. Fernando, H.H.C. Iu, C.K. Tse, Shortest path planning for energy-constrained mobile platforms navigating on uneven terrains, IEEE Trans. Ind. Inf. 14 (9) (2018) 4264–4272, http://dx.doi.org/10.1109/TII.2018.2844370.
[26] Q. Lin, H. Song, X. Gui, X. Wang, S. Su, A shortest path routing algorithm for unmanned aerial systems based on grid position, J. Netw. Comput. Appl. 103 (2018) 215–224, http://dx.doi.org/10.1016/j.jnca.2017.08.008.
[27] Y. Dinitz, R. Itzhak, Hybrid Bellman–Ford–Dijkstra algorithm, J. Discrete Algorithms 42 (2017) 35–44, http://dx.doi.org/10.1016/j.jda.2017.01.001.
[28] P. Harish, P. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: High Performance Computing-HiPC 2007, Vol. 4873, 2007, pp. 197–208.
[29] Q. Xu, T. Mytkowicz, N.S. Kim, Approximate computing: A survey, IEEE Des. Test 33 (1) (2016) 8–22, http://dx.doi.org/10.1109/MDAT.2015.2505723.
[30] M. Besta, S. Weber, L. Gianinazzi, R. Gerstenberger, A. Ivanov, Y. Oltchik, T. Hoefler, Slim graph: Practical lossy graph compression for approximate graph processing, storage, and analytics, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, in: SC, vol. 19, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3295500.3356182.
[31] W. Tan, S. Chang, L. Fong, C. Li, Z. Wang, L. Cao, Matrix factorization on GPUs with memory optimization and approximate computing, in: Proceedings of the 47th International Conference on Parallel Processing, in: ICPP 2018, Association for Computing Machinery, New York, NY, USA, 2018, http://dx.doi.org/10.1145/3225058.3225096.
[32] D. Peroni, M. Imani, H. Nejatollahi, N. Dutt, T. Rosing, Data reuse for accelerated approximate warps, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (12) (2020) 4623–4634, http://dx.doi.org/10.1109/TCAD.2020.2986128.
[33] S. Singh, R. Nasre, Graffix: Efficient graph processing with a tinge of GPU-specific approximations, in: 49th International Conference on Parallel Processing, ICPP, in: ICPP, vol. 20 (2020) http://dx.doi.org/10.1145/3404397.3404406.
[34] F. Busato, N. Bombieri, An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures, IEEE Trans. Parallel Distrib. Syst. 27 (8) (2016) 2222–2233, http://dx.doi.org/10.1109/TPDS.2015.2485994.
[35] Graph500, https://graph500.org/?page_id=12#sec-3_3.
[36] D. Singh, B. Singh, Investigating the impact of data normalization on classification performance, Appl. Soft Comput. 97 (2020) 105524, http://dx.doi.org/10.1016/j.asoc.2019.105524.
[37] J. Henseler, C. Ringle, R. Sinkovics, The use of partial least squares path modeling in international marketing, Adv. Int. Mark. 20 (2009) 277–319, http://dx.doi.org/10.1108/S1474-7979(2009)0000020014, Volume title: New Challenges to International Marketing.
[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

Busenur Aktilav received the B.Sc. degree in computer engineering from Izmir Institute of Technology, Izmir, Turkey, in 2021. Her research interests include heterogeneous systems and machine learning.

Isil Oz received the B.Sc. and M.Sc. degrees in computer engineering from Marmara University, Istanbul, Turkey, in 2004 and 2008, respectively. She received the Ph.D. degree in computer engineering at Bogazici University, Istanbul, Turkey, in 2013. She is an assistant professor in the Computer Engineering Department in Izmir Institute of Technology. Her research interests include computer architecture, multicore systems, heterogeneous systems, and fault tolerant computing.