The TensorFlow Partitioning and Scheduling Problem: It's the Critical Path!

Ruben Mayer, Christian Mayer, Larissa Laich
{firstname.lastname}@ipvs.uni-stuttgart.de
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany

ABSTRACT

State-of-the-art data flow systems such as TensorFlow impose iterative calculations on large graphs that need to be partitioned on heterogeneous devices such as CPUs, GPUs, and TPUs. However, partitioning can not be viewed in isolation. Each device has to select the next graph vertex to be executed, i.e., perform local scheduling decisions. Both problems, partitioning and scheduling, are NP-complete by themselves but have to be solved in combination in order to minimize overall execution time of an iteration. In this paper, we propose several heuristic strategies to solve the partitioning and scheduling problem in TensorFlow. We simulate the performance of the proposed strategies in heterogeneous environments with communication-intensive workloads that are common to TensorFlow. Our findings indicate that the best partitioning and scheduling heuristics are those that focus on minimizing the execution time of the critical path in the graph. Those strategies provide a speed-up of up to 4 times in comparison to strategies that are agnostic to the critical path, such as hash-based partitioning and FIFO scheduling.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Computing methodologies** → Machine learning;

KEYWORDS

TensorFlow, partitioning, scheduling, critical path

ACM Reference Format:
Ruben Mayer, Christian Mayer, Larissa Laich. 2017. The TensorFlow Partitioning and Scheduling Problem: It's the Critical Path!. In *Proceedings of DIDL'17: Workshop on Distributed Infrastructures for Deep Learning (DIDL'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3154842.3154843

1 INTRODUCTION

General-purpose distributed data processing systems for graph-structured data have experienced a phenomenal growth in the last decade. Because of their superior performance and scalability, highly-optimized systems such as Google's TensorFlow [1] have created the foundation of the recent breakthroughs in the field of machine learning - focusing on deep learning applications such as image and video classification, natural language processing, and world-class Go playing machines.

Distributed machine learning systems such as TensorFlow express the computation as a directed data flow graph where graph vertices represent computational operations and edges transport data between these operations. This abstraction empowers the data scientist to harness the power of multi-core infrastructures while expressing arbitrary complex computations. The major objective is to minimize execution time of the given computational task. Deep learning applications often require the graph to be executed in an iterated fashion with many iterations of supervised learning algorithms on huge data sets. Hence, graphs can be executed thousands of times. For instance, training neural networks can take tens of hours on dozens of devices [3]. Especially the choice of partitioning the graph onto multiple devices and scheduling graph vertices for execution on the devices has huge impact on the performance of the data flow system in terms of execution time. Both problems, i.e., partitioning and scheduling, are NP-complete and, hence, are only feasible if solved in a heuristic fashion.

However, although the problem of partitioning and scheduling is a key challenge to minimize execution time of the data flow graphs, we identified significant lack of research in the following areas. First, there is no formal description of the TensorFlow partitioning and scheduling problem and, as a consequence, no proof about the NP-completeness. Second, existing partitioning heuristics focus on minimizing the amount of communication rather the execution time of the data flow graph, while existing scheduling approaches assume global scheduling decisions only rather than pushing scheduling logic to the devices.

The goal of this paper is to conduct a thorough study on a wide spectrum of partitioning and scheduling algorithms as a basis to decide which algorithms to use for distributed deep learning problems. In particular, our contributions are as follows. (1) We formulate the TensorFlow partitioning and scheduling problem (denoted as *TF*) and prove NP-hardness by reducing from the well-known NP-complete single execution time scheduling [6]. (2) We develop several partitioning and scheduling heuristics specifically tailored to *TF* that are based on intuitive ideas. (3) We evaluate the performance of the proposed heuristics on different graphs in a simulation. Results indicate that the best partitioning and scheduling heuristics are based on minimizing the execution time of the critical path, providing a speed-up of up to 4 times in comparison to strategies

DIDL'17, December 11–15, 2017, Las Vegas, NV, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5169-0/17/12…$15.00
https://doi.org/10.1145/3154842.3154843

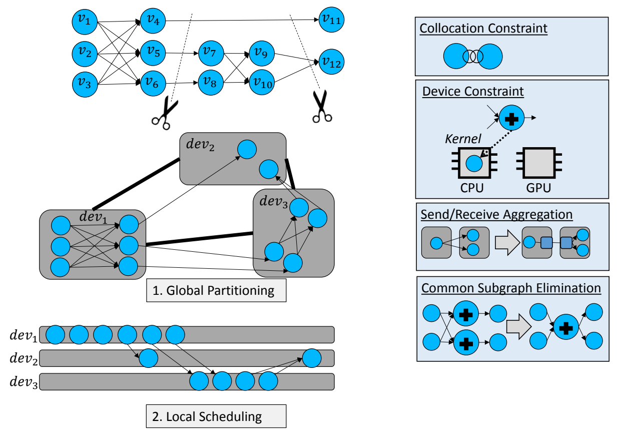

Figure 1: Problem Formulation.

that are agnostic to the critical path such as hash-based partitioning and FIFO scheduling.

2 PROBLEM FORMULATION

In this section, we formulate the TensorFlow partitioning and scheduling problem (abbreviated *TF*).

Let graph $G = (V, E)$ be the directed, acyclic data flow graph with vertices $V = \{v_1, ..., v_n\}$ and edges $E = \{e_1, ..., e_m\} \in V \times V$. Edges transport data from *source to target vertices*, i.e., the output of the source vertex serves as input to the target vertex. Associated to each edge $e_i \in E$ is the amount of communication $t_i \in \mathbb{R}$. For instance, in TensorFlow an edge is denoted as tensor, i.e., an array of (primitive) data values. Vertices are denoted as *schedulable* if data on all incoming edges is available. Associated to each vertex $v_i \in V$ is its computational complexity $c_i \in \mathbb{R}$.

Let D be set of devices $D = \{dev_1, ..., dev_k\}$. Each device $dev_i \in D$ has computational speed $s_i \in \mathbb{R}$. For example if vertex v_1 with computational complexity $c_1 = 10$ (operations) is executed on device $dev_1 \in D$ with computational speed $s_1 = 10$ (operations per second), then the execution takes $\frac{c_1}{s_1} = 1$ second. Furthermore, device $dev_1 \in D$ has maximal memory capacity $C_i \in \mathbb{R}$. For instance, data leaving the source vertex of an edge consumes memory of the device – that can have more or less severe memory restrictions. Each two devices are connected via a (physical or virtual) network link. The bandwidth is given by the bandwidth matrix $B \in \mathbb{R}^{k \times k}$. For instance, devices dev_1 and dev_2 communicate with bandwidth $B_{1,2}$ bytes per second.

The set of *collocation constraints* $\mathbb{C} \in V \times V$ encodes the symmetric relation of vertices that have to be placed on the same device. For instance, stateful operations need to be placed on the same device as their states [1]. Moreover, real-world data flow applications come with implicit or explicit *placement constraints* of computational operations denoted as device constraints $\mathbb{D} \in V \times D$. Examples are restrictions of the kernel, i.e., the concrete implementation of an operation, or of the hardware, e.g. GPU preferences.

In Figure 1, we give an example. The directed acyclic graph (DAG) consists of twelve vertices that are partitioned onto three devices. This partitioning is global, i.e., a logically centralized algorithm selects vertices to be assigned to devices. After the partitioning, each device contains a set of vertices to be executed. There may be several schedulable vertices at each point in time, e.g., device dev_1 could schedule vertices v_1, v_2, or v_3.

The partitioning function $p : V \longrightarrow D$ assigns vertices to devices and the scheduling function $f : V \longrightarrow \mathbb{N}$ assigns vertices to time slots in which the vertices are executed. The goal is to minimize the execution time of the global schedule:

$$min_f(max_{v \in V} f(v)), \tag{1}$$

Note that the function f returns the *starting time* of a vertex execution while we are interested in minimizing the maximal *finishing time*. But we can overcome this problem easily by connecting all vertices in the DAG with out-degree zero (i.e., sink vertices) via a zero cost graph edge to an artificial final sink vertex with computation complexity zero. The starting time of the artificial sink vertex relates to the finishing time of overall computation.

We have to ensure that the memory constraint is fulfilled (cf. Equation 2), i.e., in each point in time $l \in \mathbb{N}$ and for each device $dev_j \in D$ the total memory usage for keeping data on all input edges of not yet scheduled vertices does not exceed the maximal capacity C_j. We denote the set of active edges on device dev_j at time l as $E_{active}(l, j)$.

$$\forall dev_j \in D, l \in \mathbb{N} : \sum_{e_i \in E_{active}(l,j)} t_i < C_j \tag{2}$$

Additionally, we require the collocation constraints to hold, i.e.,

$$\forall v_i, v_j \in V : (v_i, v_j) \in \mathbb{C} \longrightarrow p(v_i) = p(v_j). \tag{3}$$

Finally, the device constraints have to hold, i.e.,

$$\forall v_i \in V, dev_j \in D : (v_i, dev_j) \in \mathbb{D} \longrightarrow p(v_i) = dev_j. \tag{4}$$

2.1 NP-completeness

THEOREM 2.1. *TF is NP-complete*

PROOF. First, we reduce the NP-complete single execution time scheduling [6], denoted as *Scheduling*, to Decision-*TF*, i.e., $max_{v \in V}(f(v)) < t_{max}$ (Lemma 2.2). This is the associated decision problem to *TF* and therefore is in the same complexity class [6]. Second, we show that Decision-*TF* is in NP (Lemma 2.3).

LEMMA 2.2. *Scheduling can be reduced to Decision-TF in polynomial time.*

PROOF. The NP-complete single execution time scheduling is the following problem (cf. [6]): Given a set S of jobs that take unit time on any device, a partial order $\prec$ on the set of jobs, k' processors, and a time limit t_{max}. Is there a scheduling function $g : S \longrightarrow \{0, ..., t_{max} - 1\}$ such that the following three properties hold? (i) The scheduling function respects the ordering relation, i.e., $v \in S \prec v' \in S \longrightarrow g(v) < g(v')$. (ii) The time limit is not exceeded, i.e., $\forall v \in S : g(v) < t_{max}$. (iii) There are at most k' active jobs at each point in time, i.e., $\forall i \in \{0, ..., t_{max} - 1\} : |\{v \in S | g(v) = i\}| \leq k'$.

We reduce *Scheduling* to Decision-*TF* using the following method. The graph $G = (V, E)$ is given by the set of vertices $V = S$ and the set of edges E, where there is an edge (v_1, v_2) iff the ordering relation defines $v_1 \prec v_2$. We set the number of devices $k = k'$. Furthermore, we set $\forall i, j \in \{0, ..., |V|\}$ to $t_i = B_{i,j} = 0$, i.e., we ignore network communication and data to be transported over edges. We set the

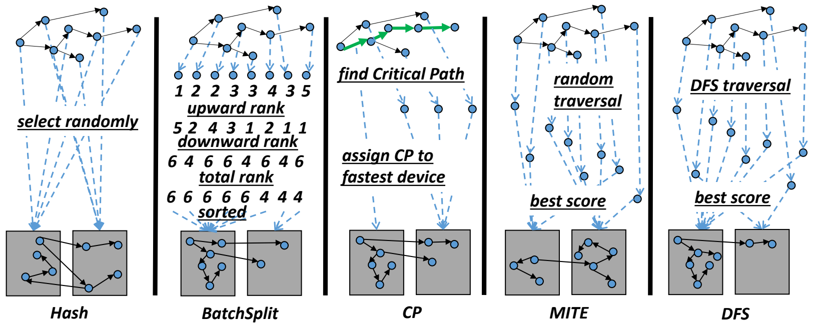

Figure 2: Overview: partitioning strategies.

computational complexity $\forall i, j \in \{0, ..., |V|\}$ to $c_i = 1$ and the device speeds $\forall i \in \{0, ..., k\}$ to $s_i = 1$, such that processing each vertex on each device takes unit time. There are no collocation and device constraints, i.e., $\mathbb{C} = \emptyset$ and $\mathbb{D} = \emptyset$. Clearly, setting these parameters to constants is a polynomial time reduction.

Next, we show: Decision-*TF* == *true* $\leftrightarrow$ *Scheduling* == *true*.

$\rightarrow$ Let Decision-*TF* == *true*. This directly implies that there exists a scheduling function f that satisfies the maximal execution time, i.e., $min_f(max_{v \in V} f(v)) < t_{max}$. The schedule given by the function f satisfies the three conditions of the *Scheduling* problem: (i) the ordering is respected (the DAG directly encodes the required partial ordering $\prec$ as constructed by the reduction), (ii) the time limit t_{max} is respected as $max_{v \in V} f(v) < t_{max} \rightarrow \forall v \in S : f(v) < t_{max}$, and (iii) there are at most k active jobs (there are only k devices). Hence, *Scheduling*== *true*.

$\leftarrow$ Let *Scheduling* == *true*, i.e., the properties (i), (ii), and (iii) are fulfilled for a scheduling function g. Because of (ii) the time limit is kept, i.e., $\forall v \in S : g(v) < t_{max}$, and therefore $min_f(max_{v \in V} f(v)) < t_{max}$. Hence, Decision-*TF*== *true*.

□

Lemma 2.3. *Decision-TF is in NP.*

Proof. In order to show that a problem is in NP, we have to find an algorithm that decides whether a given problem instance solves Decision-*TF*. This is achieved by the following two steps. First, we calculate the maximal execution time $max_{v \in V}(f(v))$ and check whether it is smaller than t_{max}. Second, we determine whether the memory, collocation, and device constraints are fulfilled. Clearly, these validation methods can be performed in polynomial time. □

As Decision-*TF* is in NP and the NP-complete *Scheduling* problem can be reduced to Decision-*TF*, Decision-*TF* and hence, *TF*, are NP-complete. □

2.2 Challenges

Several specific challenges have to be addressed for TensorFlow Partitioning and Scheduling:

Scalability: Existing scheduling algorithms globally select the processor and the start time for each vertex [5]. We argue that this might be feasible for task scheduling, where the tasks are relatively coarse-grained computational units. However, for large numbers of fine-grained graph vertices the time needed to calculate the global schedule might be too high. Additionally, the potential for stale schedules increases dramatically if millions of small errors of estimating computational complexity accumulate. Therefore, we propose to first partition the graph using scalable partitioning heuristics and then solve the scheduling problem locally on the machines. This is the de facto standard for TensorFlow partitioning.

Heterogeneity: In TensorFlow, there are heterogeneities on every level: the computational complexity of vertices, the communication volume of edges, the memory capacity of devices, the computational speed of devices, and the bandwidth between devices. All of these values can be highly heterogeneous in real-world deployments as shown in [1].

3 PARTITIONING APPROACHES

In this section, we describe strategies to partition the graph such that the local scheduling algorithms running on the devices can exploit the locality and idle time is minimal. An overview of the proposed strategies is depicted in Figure 2.

3.1 Hashing

The simplest strategy is to randomly assign vertices to devices proportionally to the capacity of the devices by using a hash function. Collocation constraints are considered by iteratively merging collocated vertices into a collocation group and assigning this collocation group randomly. Device constraints are fulfilled by restricting the potential set of devices for each vertex to those devices to which assignment is allowed and choosing a random device out of those. Memory requirements are fulfilled in a similar manner by excluding devices with insufficient spare capacity. Random assignment is very simple and fast and requires minimal space and time complexity leading to good scalability. However, hashing ignores locality of the graph vertices, leading to high communication overhead and slow execution of the DAG.

3.2 Path-Based Heuristics

Instead of randomly assigning vertices to devices, it might be better to place the vertices that are on the *critical path* all on the fastest device(s). The critical path is the path in the dataflow graph that has the longest computation time from source to sink vertex. Speeding up the processing of the critical path, hence, would speed up the overall computation time of the dataflow graph.

We propose two different strategies that try to optimize the assignment of the critical path. The first strategy, *Batch Split*, estimates the critical path by means of calculating the *rank* of each vertex and then assigns batches of vertices that have the highest ranks to the fastest devices. The second strategy, *Critical Path*, is also based on calculating the vertex ranks, but tries to assign the complete critical path to the fastest device.

In particular, we define two ranks for each vertex, the *upward rank* and the *downward rank*. The upward rank of vertex v_i is the summed computational complexity of vertices along the longest path from v_i to any sink vertex and defined as:

$$upRank(v_i) = \max_{v_j \in succ(v_i)}(upRank(v_j) + c_i) \quad (5)$$

The downward rank of v_i is the summed computational complexity of predecessor vertices along the longest path from any source vertex to v_i and defined as:

$$downRank(v_i) = \max_{v_j \in pred(v_i)}(downRank(v_j) + c_i) \quad (6)$$

3.2.1 Batch Split. The idea of the Batch Split strategy is to avoid expensive critical path computation but still prioritize optimal placement of vertices that are located *on long paths*.

First, we calculate the *total rank* for each vertex v that is the sum of the upward rank and the downward rank. The total rank can be calculated efficiently by traversing the whole graph similar to Dijkstra's algorithm and calculate upward and downward ranks *on the way*. Then, we sort vertices by total rank in descending order and assign them independently to the fastest feasible device. Due to the sorting of vertices, the complexity of the Batch Split strategy is in $O(log(n) \times n)$ (the calculation of total rank can be performed in linear runtime).

3.2.2 Critical Path. The Critical Path (CP) strategy tries to achieve minimal execution time by assigning the complete critical path to the fastest device. With this strategy, no additional communication latency is added to the computation time of the critical path.

Hence, we first compute the critical path based on downward ranks, which works as follows. (1) The algorithm starts at the source vertices and computes for each vertex of the complete graph the downward rank. (2) Choose the sink vertex with the maximum downward rank. This vertex is on the critical path. (3) Follow the predecessor relation from the chosen sink vertex to the source, adding the visited vertices to the critical path. In case a vertex has multiple predecessors, follow all such paths. (4) When reaching a source vertex, a path is completed. Choose the longest path between the chosen sink vertex and any of the connected sink vertices in the predecessor relation of the sink vertex. This is the critical path.

The critical path is assigned to the fastest feasible device. If no device can hold the complete critical path, it is divided among the fastest devices. A vertex v_k that is not on the critical path is assigned to the device dev_i with minimal sum of execution times of already assigned vertices plus the execution time of v_k on dev_i (cf. Equation 7).

$$\min_{dev_i \in D}\left(\left(\sum_{v_j \in V: p(v_j)=dev_i} \frac{c_j}{s_i}\right) + \frac{c_k}{s_i}\right) \quad (7)$$

3.3 Multi-Objective Heuristics

Instead of only taking into account the critical path, we identify four objectives in assigning vertex v to device dev: (i) introduce minimal communication overhead, (ii) prefer fast devices, (iii) prefer devices with high memory capacity, and (iv) prefer vertices on critical paths to be assigned first. We propose two different heuristics that take into account that broader set of optimization objectives.

3.3.1 MITE. The idea of the MITE (Memory, Importance, Traffic, and Execution time) strategy is to consider all four optimization objectives and use a heuristic function to assign the vertices to devices. A vertex v_i is assigned to the device dev_l such that the following score is *minimized*:

$$\begin{aligned} mite(v_i, dev_l) = & mem(dev_l) \times \\ & imp(v_i, dev_l) \times \\ & traffic(v_i, dev_l) \times \\ & execTime(v_i, dev_l) \end{aligned} \quad (8)$$

The memory score is the percentage of memory utilization of dev_l. Based on how many vertices are already assigned to dev_l by the partitioning algorithm, the memory utilization will grow and less loaded devices will be favored by the partitioning algorithm.

The importance score favors important vertices to be placed on fast devices. An important vertex is a vertex with a high total rank (total rank is defined in Section 3.2 as the sum of upward and downward rank of a vertex). In the score computation, the total rank of a vertex is normalized by the highest total rank among all vertices, and the speed of a device is normalized by the speed of the fastest device. Formally, the importance score is defined as:

$$imp(v_i, dev_l) = 1 - \left(\frac{totalRank(v_i)}{\max_{v_j \in V}\{totalRank(v_j)\}} \times \frac{s_l}{\max_{dev_k \in D}\{s_k\}}\right) \quad (9)$$

The traffic score is computed as follows. If a vertex v is placed on device dev and a neighboring vertex $v' \neq v$ is already assigned to another device $dev' \neq dev$, additional traffic is introduced. We define the traffic score as follows:

$$traffic(v_i, dev_l) = \sum_{v_j \in V, (v_j, v_i) \in E} \frac{t_i}{B_{p(v_j),l}} \quad (10)$$

The execution time score is the predicted execution time of v_i on dev_l as defined in Section 2, normalized by the maximum execution time of v_i on any device.

3.3.2 Depth First Search. The idea of Depth First Search (DFS) partitioning is to traverse the graph with a DFS algorithm and assign visited vertices using a multi-objective heuristic function. In particular, the DFS algorithm starts at the source vertex with the highest total rank (as defined in Section 3.2). Whenever a vertex is visited, it is assigned to the device that minimizes the function

$$dfsScore(v_i, dev_l) = traffic(v_i, dev_l) \times execTime(v_i, dev_l) \quad (11)$$

with traffic and execution time score as defined in the MITE heuristics in Section 3.3.1.

4 SCHEDULING

In this section, we describe the scheduling algorithm that runs on each device (device is a broad term that can also refer to, e.g., a CPU core). We assume that a partitioning algorithm has already performed the partitioning decision, such that to device, a part of the graph is assigned. Now, the scheduling algorithm decides the execution order of the graph vertices on that device.

In particular, the function $f : V \rightarrow \mathbb{N}$ assigns the device's vertices to numbers which define the order of execution. The following criteria are set by the scheduling problem: (1) Each vertex is executed exactly once in a scheduling cycle. (2) A device can execute at most one vertex simultaneously. (3) Scheduling of a vertex is non-preemptive, i.e., the running execution of a vertex on a device cannot be interrupted. (4) A vertex v can only be executed when all tensors of the predecessors in the graph are computed and transferred to the device of v. When all ingoing tensors are available, the vertex is denoted *executable*. (5) A device can only schedule vertices that are assigned to it by the partitioning algorithm. A vertex cannot be assigned to multiple devices. (6) A device has to remain idle when there are no executable vertices assigned to it.

Besides the classical non-preemptive scheduling algorithms such as FIFO, we have devised two scheduling algorithms tailored to the TensorFlow problem.

Highest Path Computation Time first (PCT) scheduling prefers the execution of the vertex whose longest path of direct and indirect successors takes most computation time. The rationale behind this strategy is to minimize the execution time of the *critical path* on the graph.

Maximum Successor Rank first (MSR) scheduling tries to minimize idleness across all devices, by taking into account how many downstream vertices *on the global graph* depend on a vertex (i.e., the rank) and rewards vertices whose execution activate downstream devices that are currently idle. The rationale behind this strategy is to maximize the *resource utilization* of the devices.

4.1 PCT Scheduling

PCT schedules always the executable vertex whose longest path of direct and indirect successors takes most computation time. The upwards path computation time (PCT) is calculated from the sink vertices as shown in Equation 12. The computation time of a sink vertex v is the execution time of v on its assigned device. If v is not a sink vertex, the maximal transfer time of a tensor to any of the successor vertices as well as the path computation time of that successor vertex are added. The transfer time is considered 0 if both vertices are placed on the same device. Else, it is the edge weight (size of the tensor) divided by the the transfer rate between the two devices.

$$PCT(v_i) = \max_{v_j \in succ(v_i)} \Big(PCT(v_j) + trans(v_j, v_i) \Big) + \frac{c_i}{s_{p(v_i)}} \qquad (12)$$

PCT is calculated once for each vertex, after the partitioning is decided. The calculated schedule is used for every iteration of the TensorFlow program.

4.2 MSR Scheduling

The drawback of PCT scheduling is that the interdependency of vertices on different devices are not taken into account. However, it could be beneficial to schedule vertices on one device a first that are necessary to activate vertices on another device b in order to avoid b's idleness.

Our Maximum Successor Rank First (MSR) scheduling algorithm takes into account such dependencies. Each vertex v on a device is scored not only based on the number of successor vertices, but also on the number of successor *devices* (i.e., on how many devices the successor vertices of v are distributed), and on the number of successor devices that are idle and get *activated* when the vertex has been executed. The idea is to minimize idleness over all devices, such that throughput will be maximized.

Table 1: Properties of the evaluated neural networks.

Graph	#nodes	#edges	average node degree	#colocated nodes
convolutional_network	347	531	1.53	104
recurrent_network	3,069	5,533	1.8	533
dynamic_rnn	5,271	9,214	1.75	1,356

The MSR scoring function is specified as follows:

$$\begin{aligned} MSR(v_i) = \sum_{v_j \in succ(v_i)} \Big(& \alpha * 1 \\ & + \beta * \big(p(v_j) \neq p(v_i)\big) \\ & + \gamma * (|pred(v_j)| = 1) \\ & + \delta * \big(idle(p(v_j)) \wedge (|pred(v_j)| = 1)\big) \Big) \end{aligned} \qquad (13)$$

The weights α, β, γ and δ can be used to balance the scoring such that specific goals are emphasized. E.g., a relatively high value of δ favors the scheduling of vertices that lead to the activation of other idle devices.

5 EVALUATIONS

To evaluate the proposed partitioning and scheduling strategies, we employ an event-based simulation. This allows us to evaluate large real-world networks on a large simulated number of devices.

5.1 Experimental setup

Simulation Parameters. In the evaluations, we simulated 50 devices. The simulated devices had assigned a speed in a random range of 10 to 100 operations per time unit. The transfer rate between the devices is set to a random number in the range between 10 and 60 bytes per time unit. Tensor sizes were again randomly set between 1 and 100 bytes. The number of operations to compute a vertex function was randomly set between 1 and 100 operations. The numbers are based on real tensor sizes and on devices available from Amazon AWS.

TensorFlow Networks. Three real TensorFlow networks were extracted from the TensorFlow examples on Github[1]. The properties of these networks are listed in Table 1. The co-location constraints were directly taken from the TensorFlow networks.

Baselines. The Heterogeneous Earliest Finish Time (HEFT) algorithm [5] solves the related *task scheduling problem* but can not be used directly for the TensorFlow partitioning and scheduling problem. We modified the HEFT algorithm slightly in the processor selection phase in order to ensure satisfaction of the TensorFlow constraints. First, we consider only *feasible devices* for a vertex v, i.e., devices to which collocated vertices are already assigned and that satisfy the device constraints of vertex v. In this phase the earliest finish time of a vertex for each **feasible** device is computed

[1]https://github.com/aymericdamien/TensorFlow-Examples/tree/master/examples/3_NeuralNetworks

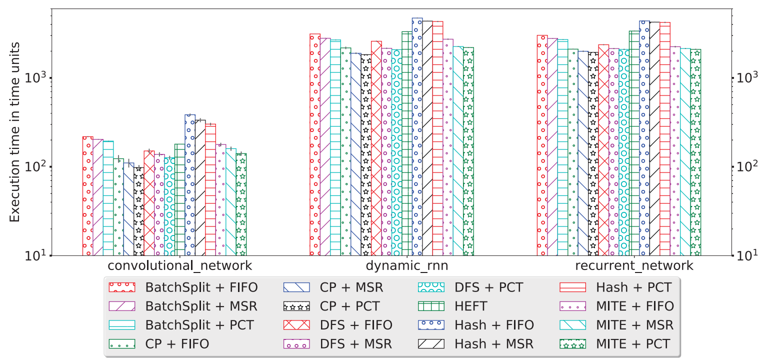

Figure 3: Execution time of 1 iteration at different partitioning and scheduling strategies on 50 devices.

and assigned to the device with the lowest earliest finish time. If the vertex is collocated, all collocated vertices are assigned but as their predecessor vertices might be not assigned yet the earliest finish time can not be computed. So their computing time slot is only computed and added to the devices time schedule when it is their turn due to the task prioritization list.

For scheduling, a second baseline we consider is FIFO scheduling, i.e., the vertices are scheduled according to the time they become executable. If two vertices become executable at the same time, the FIFO scheduler chooses randomly which of them to schedule.

5.2 Experiments and Results

Six different partitioning strategies were executed in combination with three different scheduling strategies on the three networks on 50 simulated devices. For MSR scheduling, the weights were set to $\alpha = 1, \beta = 1, \gamma = 1, \delta = 5$. Some of the strategies are non-deterministic as, e.g., the order of vertices being assigned to devices might differ.

The results of the simulation runs are depicted in Figure 3. The visualized execution time is the average of 10 executions and the standard mean deviation is shown as a gray line on each bar. The results show that both partitioning and scheduling have a significant impact on the TensorFlow performance. The best strategy (CP partitioning + PCT scheduling) is up to 4 times as fast as the worst strategy (i.e., Hash partitioning + FIFO scheduling). This was the case throughout all tested TensorFlow networks.

The reason for the bad performance of Hash partitioning and FIFO scheduling is that those strategies are agnostic to the characteristics of the *TF* problem. Hash partitioning leads to good load balancing, but this is not most important in reducing the data flow computation time from source to sink. FIFO scheduling does not favor the fast execution of the critical path either. It becomes clear that the focus of both partitioning and scheduling strategies on reducing the computation time of the critical path is of immense importance.

6 RELATED WORK

Graph partitioning has drawn a lot of attention as a preprocessing step in large-scale graph processing [4, 7]. However, graph partitioning strategies for graph processing focus on minimizing the traffic between devices and keeping the load balanced. In contrast to that, the *TF* problem deals with a data flow from source vertices to sink vertices. Hence, partitioning strategies that are successful for graph processing problems may not generalize to the *TF* problem.

The HEFT [5] scheduling algorithm evaluated in this paper has shown good performance in comparison to 20 other heuristic scheduling strategies for data flow execution [2]. However, such scheduling algorithms place vertices (or tasks) on devices (or processors) dynamically at run-time. We do not consider a relocation of vertices at run-time from one device to another as a promising way to tackle the *TF* problem. Instead, the more intuitive approach is to partition the graph first and then perform the scheduling locally on the devices.

7 CONCLUSION

In this paper, we have formally defined the TensorFlow partitioning and scheduling problem, proven the NP-completeness of the problem, and proposed a number of heuristics that solve the problem. The evaluation results based on a simulation indicate that those partitioning and scheduling strategies yield the best results that focus on the minimization of the execution time of the critical path, outperforming other strategies by up to a factor of 4.

In the future work, we plan to implement the proposed strategies in the TensorFlow framework to verify the simulation results. In particular, we aim to extend the tested deep learning networks to other types and larger scales, to see whether the findings generalize.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and others. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA.*
[2] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. 2008. *Comparative Evaluation Of The Robustness Of DAG Scheduling Heuristics.* Springer US, Boston, MA, 73–84.
[3] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, and others. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems.* 1223–1231.
[4] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel. 2016. GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS).* 118–128.
[5] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274.
[6] J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384 – 393.
[7] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. 2017. An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 493–504.