# Generalizable Resource Allocation in Stream Processing via Deep Reinforcement Learning

**Xiang Ni,**♣* **Jing Li,**♡ **Mo Yu,**♠ **Wang Zhou,**♠ **Kun-Lung Wu**♠

♣Citadel, ♠IBM Research, ♡New Jersey Institute of Technology

xiang.ni@citadel.com, jingli@njit.edu, {yum, klwu}@us.ibm.com, wang.zhou@ibm.com

## Abstract

This paper considers the problem of resource allocation in stream processing, where continuous data flows must be processed in real time in a large distributed system. To maximize system throughput, the resource allocation strategy that partitions the computation tasks of a stream processing graph onto computing devices must simultaneously balance workload distribution and minimize communication. Since this problem of graph partitioning is known to be NP-complete yet crucial to practical streaming systems, many heuristic-based algorithms have been developed to find reasonably good solutions. In this paper, we present a graph-aware encoder-decoder framework to learn a *generalizable* resource allocation strategy that can properly distribute computation tasks of stream processing graphs unobserved from training data. We, for the first time, propose to leverage graph embedding to learn the structural information of the stream processing graphs. Jointly trained with the graph-aware decoder using deep reinforcement learning, our approach can effectively find optimized solutions for unseen graphs. Our experiments show that the proposed model outperforms both METIS, a state-of-the-art graph partitioning algorithm, and an LSTM-based encoder-decoder model, in about 70% of the test cases.

## Introduction

In various industrial domains, such as aviation, medicine, transportation, telecommunication, and banking, online stream processing has been widely used to analyze high-rate data with high throughput and generate live results in a timely manner (IBM Streams 2019; Flink 2019). The computation in stream processing is driven by incoming data flowing through a Directed Acyclic Graph (DAG)[1]. A **stream graph** is comprised of **operators**, which conduct computation on the incoming tuples, and directed **edges**, each of which connects two operators and transmits tuples between them. **Tuples** are structured data items with strongly-typed attributes. Operators are event-driven and execute only when there is a tuple received. Figure 1 shows an example of a stream graph.
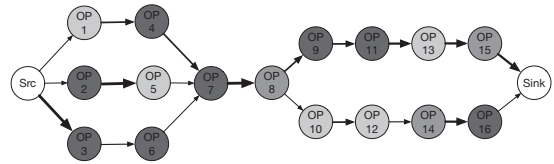


Figure 1: An example stream processing graph. The nodes of the graph represent operators labeled with the amount of computation work. The darker color of an operator illustrates higher CPU utilization. The directed edges depict data flowing from the source to destination operators. The edge width represents the amount of data flowing through that connection.

To exploit the abundant parallelism in stream graphs, operators are distributed to computing devices (resources, e.g., CPUs or GPUs) for execution. One objective in a stream processing system is to maximize the **throughput** — the number of tuples processed per second. If two connected operators are distributed to the same device, they can communicate via a simple function call (IBM Streams 2019). If they are distributed to different devices, however, they need to communicate over a network. As function calls are more efficient than network communications, it is important to properly distribute operators to computing devices. A good resource allocation strategy should well balance the trade-offs between distributing computation evenly across devices and minimizing communication cost between devices.

The distribution of operators to resources can be formulated as a two-step **graph partitioning** problem: first determining the right number of partitions $k$ and then performing $k$-way partitioning to divide operators into $k$ sets[2]. Finding the optimal graph partition is known to be NP-complete. In practice, only approximate solutions are possible with the adoption of heuristic rules (Karypis and Kumar 1998). Generally, existing graph partitioning libraries share a common drawback: they usually fail to give a good approximation of the minimum number of partitions $k$ and may not consistently generate high-quality partitions, since $k$ is usually

---

[1]Some systems, e.g., (IBM Streams 2019), allow cycles.

---

[2]This is referred to as the *fusion* of operators into Processing Elements (PEs) in (IBM Streams 2019), where each set is a PE.

graph-dependent but the structure of graphs can change from application to application. Analytical performance modeling and prediction have also been proposed (Li, Tang, and Xu 2016). However, existing models either have strong assumptions of data arrival rate and node connectivity, or fail to fully capture the complex factors affecting the data processing throughput in practical stream processing systems.

This paper focuses on learning a ***generalizable resource allocation strategy*** that can intelligently partition stream graphs with *different* structures while capturing performance-relevant factors of practical stream processing systems. Our proposed graph-aware encoder-decoder framework is able to produce optimized solutions to graphs that are not observed in the training data. In particular, we make the following contributions.

**DRL as a solution for graph partitioning**  We conduct the first study of using deep reinforcement learning (DRL) to train a good graph partitioning strategy that is generalizable to different stream graphs. Since graph partitioning is NP-complete, a general approximation approach is to reformulate it as a search problem for partitioned graphs. DRL has been proven to be a good mechanism for improving the search strategy in structured prediction problems, e.g., syntactic parsing (Chang et al. 2015) and neural architecture search (Zoph and Le 2016; Liu et al. 2017b). Inspired by these successes, we apply DRL to graph partitioning on stream processing systems. Our DRL solution is formulated as a sequence of resource allocation predictions. Each step of the prediction makes use of the state representations on relations between the current assignment and the global property of the stream graph, in order to approximate the global search rewards. Therefore, our approach can benefit from the capability of deep networks to efficiently learn the state representations of stream graphs and partial allocations, as well as their relations in a search policy.

**Generalizable model with graph-aware state representations**  A good graph partitioning strategy should be generalizable to different input graph structures. Therefore, to learn a search policy for unseen graphs, the DRL solution must be able to capture the desired properties of graph topology into the state representations and the state representation space must be transferable among different graphs. In stream graphs, such properties include the global information of the entire graph, such as each operator's position in the graph, the critical-path of the graph, the relation between operators' CPU utilization and communication cost, and the relation between devices' assignment predictions. To this end, we make the state representations be aware of (1) the graph topology, by leveraging the graph embedding generated by the recent advance in graph neural networks (Kipf and Welling 2016; Hamilton, Ying, and Leskovec 2017; Xu et al. 2018); and (2) the predicted assignment history, by enhancing the prediction of each device with its relevant graph neighbors' predicted assignments.

**New benchmark and evaluation**  To evaluate the proposed approach, we construct a benchmark containing $3,150$ stream graphs that are representative of different computation and communication requirements. In addition, we conduct the evaluation in a realistic setting where the graphs in testing are unobserved from training[3]. We compare our proposed graph-based DRL method against two well-regarded solutions: METIS (Karypis and Kumar 1998), a state-of-the-art graph partitioning library, and an LSTM-based encoder-decoder model designed to optimize the device placement for one fixed graph (Mirhoseini et al. 2017). Our experiments show that our proposed approach achieves superior performance by improving resource allocations for $76\%$ of graphs and increasing average throughput by $12\%$.

## Problem Definition

This section formally defines the resource allocation problem in stream processing and formulates it as a search problem. Our training and testing data consists of different stream graphs $\{G_x\}$. The computation of a stream processing task forms a graph $G_x = (V, E)$, as illustrated in the left graph of Figure 2. Each node $v \in V$ is an operator characterized by its ***CPU utilization*** (number of instructions required per second) and ***payload*** (total size of tuples produced by the operator). Each directed edge $e = (v_{in}, v_{out}) \in E$ represents the connection between operators $v_{in}$ and $v_{out}$, via which $v_{in}$ transmits its output tuples to $v_{out}$ as input. Each edge is labeled with its communication cost.
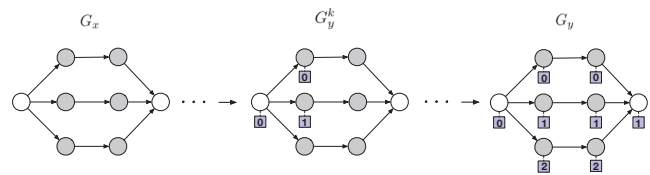


Figure 2: Graph-to-graph generation process. Square nodes indicate that the connected operators have been allocated to that device.

The goal of the resource allocation given an input graph $G_x$ and a set of devices (e.g., CPUs) $D$ is to predict a ***device placement graph*** $G_y$ where each operator $v \in G_x$ is assigned to a device $d_v \in D$. We aim to train a "meta" model on a set of stream graphs that is able to make good resource allocation predictions for ***unseen stream graphs***: graphs with different topology or different distribution of operator CPU utilization and payload compared to graphs in the training data set. In this work, we focus on homogeneous devices where the device ids can be interchangeable and leave the case of heterogeneous devices as future work. The right graph in Figure 2 illustrates the target graph $G_y$ where each node in $G_x$ is appended with a new ***device id node*** (the square nodes in the figure) depicting its allocation.

Our task of predicting $G_y$ is challenging because of the intertwined dependencies between device allocations. Intuitively, the placement $d_{v_i}$ of a node $v_i$ depends not only on *the topology of the input graph* $G_x$, but also on *the placements of the other nodes*. Formally, the prediction follows $P(d_{v_i}|G_x, d_{v_{i-1}}, d_{v_{i-2}}, d_{v_1})$, which is difficult to be decomposed. Hence, our task requires joint inference of the whole

---

[3]Our code and data set are released at https://github.com/ xiangni/DREAM.

graph $G_y$. We formulate this joint inference problem as a search problem that appends one device id node to each operator node in $G_x$ at each step, and finally outputs the graph $G_y$. Because of the high dependencies among the previously predicted $d_v$s, our task is much harder compared to classification over single nodes, and it is closer to a special case of graph-to-graph generation, where the output $G_y$ contains one additional device id node for each operator node in $G_x$.

Moreover, the difficulty of our setting also lies in the diverse input graphs with complex topologies in training and testing, while in previous work (Mirhoseini et al. 2017), training and testing are done on one single graph so a model can memorize the graph topology. Thus, a generalizable way to represent the many different graphs is necessary. In this work, we design and train our model to capture the meta-information of graph topologies so that it can directly predict good placements for various graphs, without requiring to individually train a different model for each of the different graphs as in (Mirhoseini et al. 2017).

## Graph-Aware Encoder-Decoder Model

Figure 3 shows the overall architecture of our proposed approach. Our model first encodes the input stream graph with a graph encoder. The resulted graph embedding is then used for a graph-aware decoder to generate device assignments in a sequential way. Finally, we describe how to optimize the entire network with DRL and how to build a simulator to speed up the reward calculation.

### Stream Graph Encoding

Our model first embeds the input graph $G_x$ to an embedding space. Each node has an embedding encoding its contextual information in the graph that is informative to its placement prediction. We achieve such encoding using the Graph Convolution Network (GCN) (Kipf and Welling 2016).

GCN iteratively updates a node's embedding (hidden states) with its neighbors' embeddings. Specifically, at the $k$th step, for each operator node $v$, we define its embedding as $\mathbf{h}_v^k$. When $k=0$, $\mathbf{h}_v^k$ is defined as its node feature vector $\mathbf{f}_v$, which contains the CPU utilization and payload of the tuples emitted from this node. Because $G_x$ is a directed graph, according to the directions of $v$'s edge connections, we categorize its neighbors into two sets, the upstream neighbors $\mathcal{N}_{\vdash}(v)$ and downstream neighbors $\mathcal{N}_{\dashv}(v)$. The node embedding thus can be categorized as two vectors $\mathbf{h}_{v_{\vdash}}^k$ and $\mathbf{h}_{v_{\dashv}}^k$, each with $m$ dimensions.

Based on the above definitions, following (Hamilton, Ying, and Leskovec 2017; Xu et al. 2018), the GCN updates $v$'s embedding as below:
- First, we aggregate the information from $v$'s upstream and downstream neighbors separately. Taking the upstream neighbors for example, for each $u \in \mathcal{N}_{\vdash}(v)$, we take its current stage representation $\mathbf{h}_{u_{\vdash}}^k$, feed it to a non-linear transformation $\mathbf{h}_{u_{\vdash}}^{(\mathrm{in})} = \tanh(\mathbf{W}_1^{(\mathbf{up})}\mathbf{h}_{u_{\vdash}}^k)$,[4] where $\mathbf{W}_1^{(\mathbf{up})} \in \mathbb{R}^{m \times 2m}$.

---

[4]Ideally, we have $\mathbf{h}_{u_{\vdash}}^{(\mathrm{in})} = \tanh(\mathbf{W}_1^{(\mathbf{up})}[\mathbf{h}_{u_{\vdash}}^k : \mathbf{f}_e(u,v)])$, where we derive edge features $\mathbf{f}_e(u,v)]$ for prediction. However, empirically we have payload features on nodes that correlates with edge

- Second, we get all the $\mathbf{h}_u^{(\mathrm{in})}, \forall u \in \mathcal{N}_{\vdash}(v)$, take the mean-pooling of the vectors and update the upstream-view embedding of $v$ as ($[\cdot : \cdot]$ refers to vector concatenation):

$$\mathbf{h}_{v_{\vdash}}^{k+1} = \tanh(\mathbf{W}_2^{(\mathbf{up})}\left[\mathbf{h}_{v_{\vdash}}^k : \frac{\sum_{u \in \mathcal{N}_{\vdash}(v)} \mathbf{h}_u^{(\mathrm{in})}}{|\mathcal{N}_{\vdash}(v)|}\right]). \quad (1)$$

Similar update is applied to the **downstream** representation of $v$ to get $\mathbf{h}_{v_{\dashv}}^{k+1}$, which operates on the $\mathcal{N}_{\dashv}(v)$ neighbors with transformation parameters $\mathbf{W}_1^{(\mathbf{down})}$ and $\mathbf{W}_2^{(\mathbf{down})}$. In our experiments, we use shared parameters for upstream and downstream updates.

The above steps are repeated $K$ times over all nodes in the graph. Finally for each $v$ we concatenate its upstream and downstream hidden states $\mathbf{h}_{v_{\vdash}}^K$ and $\mathbf{h}_{v_{\dashv}}^K$ as its final node representation. We denote the vector as $\mathbf{h}_v$ for short in the following sections.

We further compute the graph encoding to convey the entire graph information. The embedding of each node $\mathbf{h}_v$ is fed to a fully connected neural network layer, followed by an element-wise max-pooling layer. The output vector is thus graph encoding, used as input to the graph-aware decoder, as illustrated in Figure 3.

### Graph-Aware Decoding of Device Allocation

The prediction of the resource allocation is to assign each operator node $v$ in $G_x$ to a device $d_v$, conditioning on the graph property and the assignments of other nodes. Given an arbitrary order of nodes $\{v_1, v_2, ..., v_{|V|}\}$ from $G_x$, the problem can be formulated as:

$$P(G_y = \{d_{v_1}, d_{v_2}, \cdots, d_{v_{|V|}}\}|G_x) \quad (2)$$
$$= \prod_t P(d_{v_t}|d_{v_{t-1}}, d_{v_{t-2}}, \cdots, d_{v_1}, G_x).$$

The joint probability cannot be trivially decomposed, as the dependency between the new assignment $d_{v_t}$ and all previous assignments highly depend on the property of graph $G_x$.

In the decoding stage, we adopt an approximated decomposition of Eq. (2) in order to simplify the problem. Intuitively, the device prediction of one node is usually highly influenced by the device assignments of its upstream nodes. Therefore, if we could always have a node $d_v$'s upstream nodes assigned before it (e.g. ordering the nodes via breadth-first traversing of the graph), we can have the following approximation:

$$P(G_y|G_x) = \prod_t P(d_{v_t}|D^{(up)}(v_t), G_x), \quad (3)$$

where $D^{(up)}(v_t)$ refers to the assignments of all the upstream nodes of $v_t$. Our proposed graph-aware decoder is based on this decomposition.

**States Representation in Decoder** To deal with the intertwined dependencies among the new assignment $d_{v_t}$, all previous assignments and the $G_x$, the DRL model learns a state representation $\mathbf{s}_t$ to encode the information associated

---

weights. Thus concatenating the edge weight features does not help. We leave investigating richer edge features to future work.
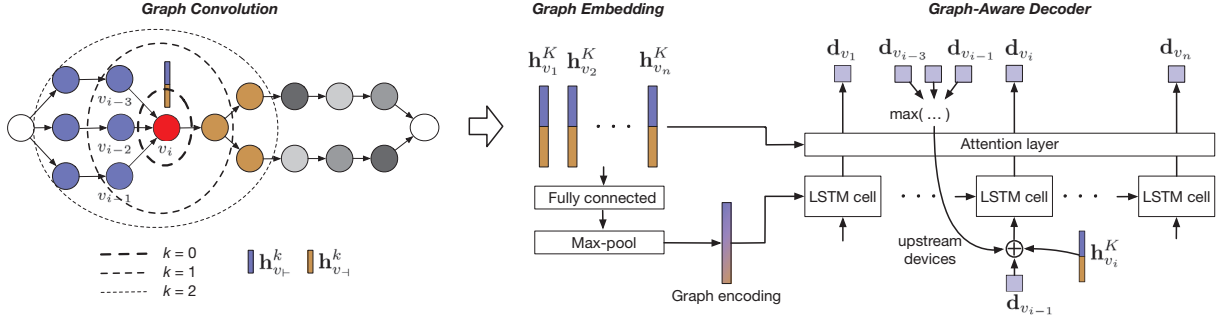
**Figure 3:** Overview of the proposed model architecture. Our model is composed of graph encoding to learn the structured information of stream graphs and graph-aware decoder for device allocation prediction.

with $\{d_{v_{t-1}}, d_{v_{t-2}}, \cdots, d_{v_1}, G_x\}$. This can be implemented with an LSTM (Hochreiter and Schmidhuber 1997):

$$\mathbf{s}_t = \text{LSTM\_Cell}(\mathbf{h}_{v_t}, \mathbf{s}_{t-1}, d_{v_{t-1}}).$$

However, although LSTM is proposed to memorize long-term temporal dependency, in practice it is difficult to learn an LSTM to memorize the history well without further inductive biases. To further encourage the state $\mathbf{s}_t$ be aware of the assignments of the nodes related to $v_t$ to help its assignment prediction, inspired by the decomposition in Eq. (3), we model the state of our decoder as:

$$\mathbf{s}_t = \text{LSTM\_Cell}(\mathbf{h}_{v_t}, D^{(up)}(v_t), \mathbf{s}_{t-1}, d_{v_{t-1}}).$$

In our implementation, we convert all the device assignments $d_v$s to their (learnable) device embedding vectors. The set $D^{(up)}(v_t)$ is thus represented as the mean-pooling results of the device embeddings to concatenated to the LSTM inputs. The above vectors are concatenated with the device embedding of $d_{v_{t-1}}$ as input to the LSTM cell.

**Prediction Layer**   Finally, the prediction for node $v_i$ becomes $P(d_{v_i}|\mathbf{s}_i)$. We model this step with an attention-based model (Bahdanau, Cho, and Bengio 2014) over all the graph nodes $v_j$s. Each node receives an attention score at step $t$ as $\alpha_j = \mathbf{s}_t^T \mathbf{h}_{v_j}^K$. Then all $\alpha_j$s get normalized with softmax, leading to $\tilde{\alpha}_j$ for each $j$. Finally, to make the device prediction, we feed the concatenation $[\mathbf{s}_t : \sum_j \tilde{\alpha}_j \mathbf{h}_{v_j}^K]$ to a multi-layer perceptron (MLP) followed by a softmax layer.

## Training with Reinforcement Learning

In our task, it is difficult to get the ground truth allocation $G_y$ for an input $G_x$. However, given any allocation $G_y$, we can obtain its relative quality by calculating the throughput. Therefore, our task fits the reinforcement learning setting, where the model makes a sequence of decisions (i.e., our decoder) and gets delayed reward $r$ (i,e., the throughput of the predicted graph allocation).

In this work, we seek to maximize the ***relative throughput*** $r(G_y) = \frac{T(G_y)}{I(G_x)}$, which is defined as the ratio of throughput $T(G_y)$ to the source tuple rate $I(G_x)$. This is because the objective is to ensure the tuple processing rate (throughput) catches up with the source tuple rate, i.e., no backpressure due to bad resource allocation. Hence, the range of reward $r$

is between $0$ and $1$. We train a stochastic policy to maximize the following objective, in which $\pi_\theta$ is a distribution over all possible resource allocation schemes $Y$:

$$J(\theta) = \sum_{G_y \in Y} \pi_\theta(G_y) r(G_y)$$

We apply the REINFORCE algorithm (Williams 1992) to compute the policy gradients and learn the network parameters using Adam optimizer (Kingma and Ba 2014)

$$\nabla(\theta)J(\theta) = \frac{1}{N} \sum_{n=1}^{N} \nabla \log \pi_\theta(G_y^n) \big[ r(G_y^n) - b \big] \quad (4)$$

In each training update, we draw a fixed number of on-policy samples. We also draw random samples to explore the search space, and the number of random samples exponentially decays. Due to the sparsity of good resource allocation schemes over the large search space, for each training graph, we maintain a memory buffer (Liang et al. 2018) to store the good samples with reward higher than $0.8$. Extra random samples are included to fasten the exploration if the memory buffer is empty. $N$ samples in Eq. 4 are composed of both on-policy samples and those from the memory buffer. Baseline $b$, the average reward of the $N$ samples, is subtracted from the reward to reduce the variance of policy gradients.

**Remark**   Our proposed method learns from training data that are representative of real streaming graphs in terms of similar topologies and distributions but not the same graphs as in testing data. If the topologies in testing are not commonly seen in streaming workloads and hence not sufficiently covered by the training data, our model could be further trained. For example, upon deployment in operation, we could perform periodic incremental training with additional graphs seen in history. We leave this study to future work.

## Speeding up the Reward Calculation

To compute the reward $r(G_y)$, each sampled allocation $G_y$ needs to be deployed on the stream processing system, however, the process may take up to a few minutes for the system to stabilize and calculate the throughput. The total time and computing resource required for training in this way is simply intractable, given that DRL relies on the evaluation of numerous resource allocation trials.
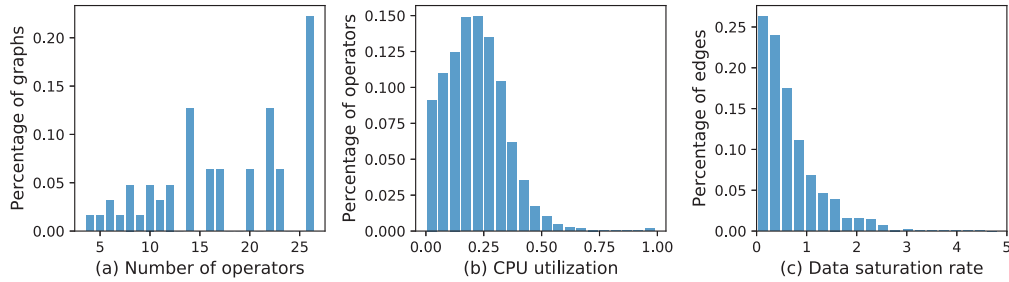
Figure 4: Statistics of graphs in our benchmark data set.

Therefore, for fast training, we adopt a simulator ***CEPSim*** for complex event processing and stream processing (Higashino, Capretz, and Bittencourt 2016) to evaluate each allocation sample. CEPSim is a simulator for cloud-based complex event processing and stream processing system that can be used to study the effect of different resource allocation, operator scheduling, and load balancing schemes. In CEPSim, DAGs are used to represent how input event streams are processed to get complex events. CEPSim provides the flexibility for users to specify the number of instructions per tuple for each operator in the DAG. To simulate stream processing, users can give a mapping function to allocate parts of the DAG to different virtual machines (VMs), and these VMs can communicate with each other using a network. We extend CEPSim to allow users to specify the payload of tuples emitted by each operator, combined with the tuple submission rate, as such information is important to derive the amount of communication over edges. We further extend CEPSim to use LogP model (Culler et al. 1993) to simulate the network delays between VMs.

We validated that our designed simulator can mimic the behavior of the real streaming systems, by empirically comparing the relative performance rank given by our simulator and a real streaming platform. More details can be found from our GitHub repository.

## Experiments

This section first describes our benchmark and baselines, and then presents our evaluation results.

### Experimental Settings

**Simulation Environment**    We create a cluster in *CEPSim* with 5 homogeneous devices. The computing capacity of each device is 2.5E3 ***million instructions per second (MIPS)***. The link bandwidth between devices is 1000 Mbps.

**Benchmark Data Set Construction and Statistics**    We create a new benchmark with $3,150$ graphs in the data set. Graphs vary in the graph topology and number of operators. For example, the graph in Figure 2 illustrates the structure of the graphs with 3 branches and 8 operators in total. Both the number of branches and the longest length of a branch in a graph vary from 2 to 12. Figure 4(a) shows the distribution of the number of operators in the data set ranging from 4 to 26. Within each graph, we randomly assign the CPU utilization

and payload to each operator. Note that the CPU utilization of an operator is calculated as $\frac{IPT*R}{MIPS}$, where $IPT$ is the number of instruction per tuple and $R$ is the tuple rate of this operator. Figure 4(b) presents the CPU utilization distribution of all the operators in the 3,150 graphs. As shown in Figure 4(c), we also calculate the data saturation rate at each edge using $\frac{P*R}{BW}$, where $P$ is the payload and $BW$ is the link bandwidth. In practice, the average CPU utilization and payload of an operator can be approximated from a profiling run of the stream processing graph. Figure 4 demonstrates that our data set provides good coverage of different operator count, CPU utilization, and payloads, which represents a wide range of real-world streaming applications. For example, our dataset covers many data-parallel and pipeline-parallel streaming graphs similar to the building blocks of many large real-world stream processing workloads.[5] We show that the trained model on the stream graph dataset using our method can capture the graph topologies mostly seen for stream processing.

**Baselines**    In the evaluation, we compare our proposed approach with the following baselines:

• ***Encoder-decoder*** (Mirhoseini et al. 2017) is an LSTM-based *sequence-to-sequence model* designed for device placements in Tensorflow graphs. This scheduling work for Tensorflow graphs learns one model for each separate graph. When a new graph comes, it needs to repeat the whole time-consuming learning process for the new graph. In comparison, we train our model on many different streaming graphs. Our model learns to capture the meta-information about the graph topologies. We adopt an adapted version of the Tensorflow scheduling work for our task.

• ***METIS*** (Karypis and Kumar 1998) is a graph partitioning library, which takes the input graph, the computational cost of each operation, the amount of data flowing through each edge, and the number of partitions to produce a mapping of operators to partitions. We set the number of partitions to 5, same as the number of available devices.

• ***IBM Streams*** (IBM Streams 2019) is a streaming platform used in production. We use the *fuser* component from IBM Streams to partition the stream graphs. *Fuser* uses simple

---

[5]For a graph that is significantly larger than those in our dataset, a common approach (Mirhoseini et al. 2017) is to hierarchically segment it into groups and reduced it to a smaller one covered in our dataset, where each group is treated as a single node.

rules to balance the number of operators and reduce edge cut among partitions. Same as METIS, it requires the number of partitions as an input parameter, which is set to 5.

One advantage of our method is that it can *automatically determine the best number of devices to use*, which makes it more suitable to the varying types of graphs at testing. To demonstrate this advantage, we also compare our results with the two baselines with their oracles on the best number of devices to use. Specifically, we vary the number of partitions of METIS or IBM Streams from 2 to 5, run their results with each number, compute the throughput, and select the highest throughput to report. The two approaches are denoted as *METIS Oracle* and *IBM Streams Oracle*.

**Hyperparameters** We randomly select $2,520$ graphs for training and the remaining $630$ graphs for testing. The number of hops $K$ in graph embedding is 2, and the length of node embeddings is $512$. The network is trained for 40 epochs using Adam optimizer with learning rate $0.001$. At each training step, only one graph is fed to the network. The number of samples $N$ for a training graph varies from 3 to 6 (with 3 on-policy samples and up to 3 samples from memory buffer). These settings are selected via cross-validation.
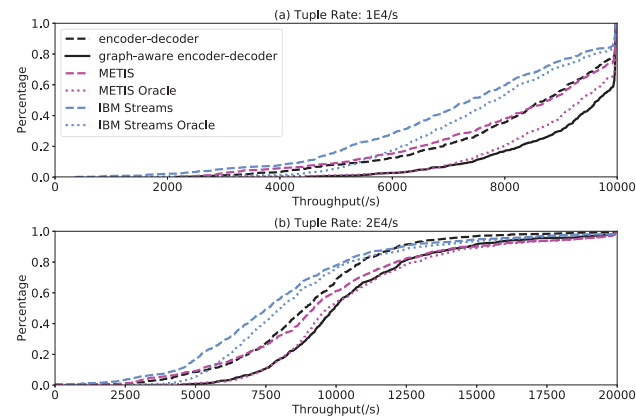


Figure 5: CDF of the throughput comparison of our approach with baselines. For instance, $(8000, 0.17)$ means that $17\%$ of graphs have throughput less than $8000/s$. If the CDF plot is more skewed towards right, the overall throughput is better.

## Experimental Results

Figure 5 shows the comparison of the throughput obtained using resource allocations predicted by our graph-aware encoder-decoder model and other baselines over the test data. The source tuple rates of the systems (set in *CEP-Sim*) are $1E4/s$ for a typical workload and $2E4/s$ for stress testing to show that our allocation scheme is reasonable for different tuple rates. In Figure 5(a), the minimum throughput of our proposed model is $4468/s$ which is significantly better than other baselines (IBM Streams: $360/s$ METIS: $2220/s$ encoder-decoder: $2140/s$). More than $50\%$ graphs have throughput above $9700/s$ using our approach. In contrast, for the same throughput value, IBM Streams, METIS

and encoder-decoder drops to $16\%$, $30\%$ and $27\%$, respectively. Figure 5(b) shows similar performance trend between different approaches. This results also confirm the importance of graph-aware encoding and graph-aware decoding in predicting the resource allocation for stream processing graphs. The encoder-decoder model by itself, which is used in Tensorflow scheduling work, fails to capture the general graph topologies in our dataset and hence our model outperforms it by a large margin.
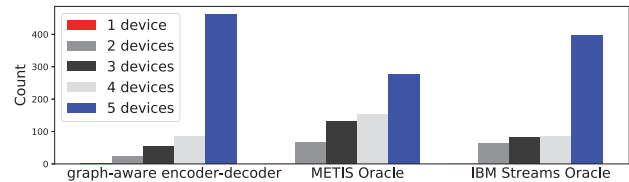


Figure 6: Device count distribution of different approaches.

Moreover, the graph-aware encoder-decoder model exceeds the performance of METIS Oracle and IBM Streams Oracle without human interference to select the optimal partition number. Figure 6 presents the device count comparison between these approaches. Unlike METIS and IBM Streams, our model is able to automatically find the right number of devices based on the workload of stream graphs.

**Ablation Study** Table 1 shows the ablation study of components in the graph-aware encoder-decoder model. B refers to the baseline LSTM encoder-decoder model. F1 replaces the LSTM encoder in the baseline with graph embedding while decoder remains the same. F2 further enhances the model with the graph-aware decoding. The numbers in the first three columns are relative throughput for source tuple rates $1E4/s$ ($2E4/s$). The last column shows the percentage of graphs that have better performance in comparison to METIS[6]. As can be seen from Table 1, both the graph encoding and graph-aware decoding improve the performance of resource allocation predictions. Our holistic approach is able to learn better representations of different stream graphs and their complex dependencies. As a result, our model outperforms METIS in $76\%$ ($65\%$) of the test cases.

**Qualitative Study** Finally, we show examples of different resource allocations from different models, providing details on why the allocation generated by our approach performs better. For each resource allocation scheme, we also give the corresponding throughput number on the top.

Figure 7 gives an example where the streaming graph has a relatively larger number of branches, while the communication costs between operators have equal weights. In this situation, an approach that is aware of the global topology is more likely to find the optimal solution, i.e., colocating the operators in the same branch. Our results confirm that our graph encoder is able to capture such topological information, given its perfect prediction. The METIS method, which also benefits from the graph topology information, outperforms the other baselines as well. More examples can be

---

[6]We choose METIS since it outperforms other baselines except for the artificial oracle ones.

Table 1: Ablation study of individual components in our model.

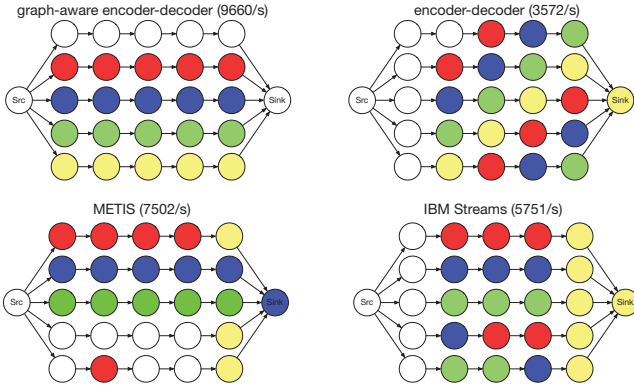|         | average     | first quartile | median      | wrt METIS  |
|---------|-------------|----------------|-------------|------------|
| METIS   | 0.81 (0.48) | 0.70 (0.37)    | 0.88 (0.46) | -          |
| B       | 0.82 (0.45) | 0.73 (0.37)    | 0.87 (0.44) | 51% (39%)  |
| B+F1    | 0.87 (0.48) | 0.80 (0.40)    | 0.91 (0.47) | 59% (47%)  |
| B+F1&F2 | **0.91 (0.53)** | **0.87 (0.43)** | **0.97 (0.50)** | **76% (65%)** |



Figure 7: Example allocation. Our model (top-left) outperforms other baselines by colocating operators in the same branch.

found from our GitHub repository for details on why the allocation generated by our approach performs better.

## Related Work

Here we review and differentiate our approach from the prior work that is most related to this research.

**RL for Job-level Resource Allocation** (Naik, Negi, and Sastry 2015) proposed an RL-based MapReduce scheduler in heterogeneous environments, which monitors the system state of task execution and suggests speculative re-execution of the slower tasks on other available nodes in the cluster. (Liu et al. 2017c) proposed a hierarchical framework to solve the overall resource allocation and power management problem in cloud computing systems with DRL. (Mao et al. 2018) introduced a general-purpose scheduling service for data processing jobs in clusters that converts DAGs of jobs to vectors using graph embedding and calculates priority scores for each job stage.

These studies differ from our work significantly in terms of the granularity of resource allocation. They focused on job-level scheduling without dealing with the internal structures of jobs, while we address the resource allocation inside jobs' computation graphs (i.e., operator-level). As a result, their state representations rely on the hand-designed system- and job-level features, while in this work it is critical to learn state representations from the structured input data.

**RL for Inner-Job Resource Allocation** To the best of our knowledge, (Mirhoseini et al. 2017; 2018; Li et al. 2018) are the only works that focus on a similar setting of inner-job allocation like our work. (Mirhoseini et al. 2017; 2018) use a *sequence-to-sequence model* to predict the de-

vice placements for subsets of operations in a Tensorflow graph. For the device assignment problem for stream processing systems, (Li et al. 2018) proposed an actor-critic approach to minimize the average end-to-end tuple processing time.

Our work differs from these work both on the proposed new model and the new problem. Most importantly, we focus on a different problem of generalizability to unobserved graphs, while the foregoing works train and test their models on the same graph. While they consider different applications with different graphs, such as Inception-V3 for image classification, LSTM for language modeling and machine translation and word count application for stream processing, the model for each application is separately trained and tested on its individual graph. In comparison, our work aims to find a "meta" model that can perform resource allocation on graphs unobserved from training data, which is a more challenging task and is crucial to realistic streaming systems.

**Graph-to-Graph Generation** As discussed in the problem definition section, our task is a special case of the graph-to-graph generation problem. Previous works in this direction mainly fall into two types. First, most works convert the input and output graphs to sequences and reformulate the problem as sequence-to-sequence generation (Iyer et al. 2016; Liu et al. 2017a).The standard sequence models like LSTM-based encoder-encoder can then be directly applied. Such methods may lose important structural information of the graphs and thus limit their performance. Second, some works perform direct graph-to-graph generation with techniques like graph VAE (Jin, Barzilay, and Jaakkola 2018). However, these methods do not handle hard constraints on the topology of the generated graphs. Since the graphs $G_x$ and $G_y$ in our task have a strong node-level one-to-one dependency, a transformation model considering such a requirement is necessary and is important for improving the performance. As the above approaches do not suit the properties of our task well, in this work we adopt a new direction based on the graph-to-sequence generation (Xu et al. 2018) but makes the decoder aware of the generated partial graphs.

There are also works (Khalil et al. 2017) about general combinatorial search problems on graphs, such as max-cut and TSP. These works do not address the specific encoding/decoding challenges in our task. For example, their policy does not rely on the global graph topology, and their decoder does not need to work with ordered assignments, which does not well suit the real stream applications.

## Conclusion

In this paper, we present a *generalizable* model to predict resource allocation for stream processing systems. We propose to use graph embedding to better represent the structured information of stream graphs, and a graph-aware decoder to capture the complex dependencies that affect the quality of resource allocation. Deep reinforcement learning is applied to jointly train the model with enhanced search efficiency. The proposed framework predicts better resource allocations than graph partitioning library METIS and LSTM-based encoder-decoder model for more than 70% of unobserved graphs, demonstrating superior generalizability of

our model. As part of the future work, we plan to apply the proposed framework in the real cloud environment and consider heterogeneous devices in the real deployment.

# References

Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Chang, K.-W.; Krishnamurthy, A.; Agarwal, A.; Daume III, H.; and Langford, J. 2015. Learning to search better than your teacher.

Culler, D.; Karp, R.; Patterson, D.; Sahay, A.; Schauser, K. E.; Santos, E.; Subramonian, R.; and Von Eicken, T. 1993. Logp: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, 1–12. ACM.

Flink, A. 2019. Apache Flink. http://flink.apache.org. Retrieved February, 2019.

Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 1024–1034.

Higashino, W. A.; Capretz, M. A.; and Bittencourt, L. F. 2016. Cepsim: Modelling and simulation of complex event processing systems in cloud environments. *Future Generation Computer Systems* 65:122–139.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

IBM Streams. 2019. IBM Streams. https://ibmstreams.github.io. Retrieved February, 2019.

Iyer, S.; Konstas, I.; Cheung, A.; and Zettlemoyer, L. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, 2073–2083.

Jin, W.; Barzilay, R.; and Jaakkola, T. 2018. Junction tree variational autoencoder for molecular graph generation. In *International Conference on Machine Learning*, 2328–2337.

Karypis, G., and Kumar, V. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20(1):359–392.

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 6348–6358.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kipf, T. N., and Welling, M. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Li, T.; Xu, Z.; Tang, J.; and Wang, Y. 2018. Model-free control for distributed stream data processing using deep reinforcement learning. *Proceedings of the VLDB Endowment* 11(6):705–718.

Li, T.; Tang, J.; and Xu, J. 2016. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data* 2(4):353–364.

Liang, C.; Norouzi, M.; Berant, J.; Le, Q. V.; and Lao, N. 2018. Memory augmented policy optimization for program synthesis and semantic parsing. In *Advances in Neural Information Processing Systems*, 10015–10027.

Liu, B.; Ramsundar, B.; Kawthekar, P.; Shi, J.; Gomes, J.; Luu Nguyen, Q.; Ho, S.; Sloane, J.; Wender, P.; and Pande, V. 2017a. Retrosynthetic reaction prediction using neural sequence-to-sequence models. *ACS central science* 3(10):1103–1113.

Liu, C.; Zoph, B.; Shlens, J.; Hua, W.; Li, L.-J.; Fei-Fei, L.; Yuille, A.; Huang, J.; and Murphy, K. 2017b. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*.

Liu, N.; Li, Z.; Xu, J.; Xu, Z.; Lin, S.; Qiu, Q.; Tang, J.; and Wang, Y. 2017c. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, 372–382. IEEE.

Mao, H.; Schwarzkopf, M.; Venkatakrishnan, S. B.; Meng, Z.; and Alizadeh, M. 2018. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963*.

Mirhoseini, A.; Pham, H.; Le, Q. V.; Steiner, B.; Larsen, R.; Zhou, Y.; Kumar, N.; Norouzi, M.; Bengio, S.; and Dean, J. 2017. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 2430–2439. JMLR. org.

Mirhoseini, A.; Goldie, A.; Pham, H.; Steiner, B.; Le, Q. V.; and Dean, J. 2018. A hierarchical model for device placement. In *ICLR 2018*.

Naik, N. S.; Negi, A.; and Sastry, V. 2015. Performance improvement of mapreduce framework in heterogeneous context using reinforcement learning. *Procedia Computer Science* 50:169–175.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8(3-4):229–256.

Xu, K.; Wu, L.; Wang, Z.; Yu, M.; Chen, L.; and Sheinin, V. 2018. Exploiting rich syntactic information for semantic parsing with graph-to-sequence model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 918–924.

Zoph, B., and Le, Q. V. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.