**Summary**

Most compilers for machine learning (ML) frameworks need to solve many correlated optimization problems to generate efficient machine code. Current ML compilers rely on heuristics based algorithms to solve these optimization problems one at a time.

In mapping a computational graph to machine code that executes on a collection of devices, ML compilers need to solve many optimization problems including graph rewriting, assignment of operations on devices, operation fusion, layout and tiling of tensors, and scheduling. Existing learning based approaches in the literature are sample inefficient, tackle a single optimization problem, and do not generalize to unseen graphs making them infeasible to be deployed in practice.

heuristics often lead to sub-optimal configurations especially for previously unseen model architectures. Second, by solving these problems in isolation, the compiler misses out on opportunities for joint optimizations across tasks. To

Many of the graph optimization problems in the compiler stack are inherently coupled. For example, a seemingly well optimized graph partitioning and device placement can lead to poor run time due to bad scheduling decisions that induces a near-sequential execution.

To address these limitations, we propose an end-to-end, transferable deep reinforcement learning method for computational graph optimization (GO), based on a scalable sequential attention mechanism over an inductive graph neural network.

**Tasks and Problem Formulation**

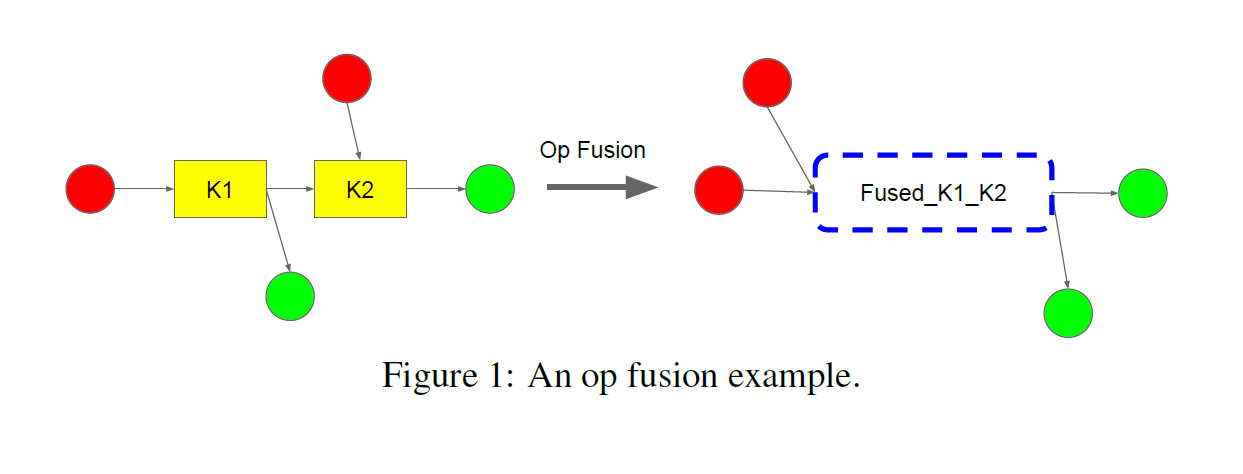
ML computations are usually expressed as computation graphs, G(V;E), where nodes V represent computations and edges E represent data flows.

**Device Placement**: Given a computational graph, the goal of device placement is to learn a policy : G 7! D that assigns a device D 2 D for all nodes in the given graph G 2 G, to maximize a reward rG;D defined based on the run time.

**Operation Scheduling**: An op in a dataflow graph is ready to run when its incoming tensors are present in the device memory. A frequently used scheduling strategy is to maintain a ready queue of operations for each device and schedule operations in first-in-first-out order. However, such schedules can be sub-optimal especially when the operations are distributed across a set of devices.

**Operation Fusion:** Op fusion is the process of merging multiple ops into a single op. Figure 1 showcases an example of op fusion. Op K1 is producing an output which is consumed by op K2. If these two ops are fused, the intermediate data produced by K1 is immediately used for K2

when K1 finishes on the device, without the need to perform read and write transactions with the global memory, thereby reducing the communication overhead.



**Generalization across graphs**: In contrast to prior works that focus on a single graph only, the training objective for GO is to simultaneously reduce the expected run time of the optimization over multiple dataflow graphs.

**Joint optimization across tasks**: Our method can be extended to handle multiple tasks jointly.

**Solution and conclusion.**

One of the key goals of this work is to ensure the generalizability of the the policy network over a wide variety of graphs from potentially different application domains. we propose a feature modulation mechanism similar to parameter superposition [6]. The idea is to re-weight network parameters by generating a feature modulation layer based on the input graph feature to mitigate the potentially undesirable interference among different input graphs.

We propose a recurrent attention policy network that not only applies a segmentlevel recurrent attention to the graph representation spatially, but also generates recurrent actions for multiple tasks through residual connections and parameter sharing across multiple recurrent attention layers.

In this paper, we present a generalized deep RL method for computation graph optimizations that generalizes to arbitrary and held out graphs. We propose recurrent attention layers to jointly optimize dependent graph optimization tasks and demonstrate 33%-60% speedup over TensorFlow’s default strategy on three tasks.