# DOPPLER: DUAL-POLICY LEARNING FOR DEVICE ASSIGNMENT IN ASYNCHRONOUS DATAFLOW GRAPHS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

We study the problem of assigning operations in a dataflow graph to devices to minimize execution time in a work-conserving system, with emphasis on complex machine learning workloads. Prior learning-based approaches face three limitations: (1) reliance on bulk-synchronous frameworks that under-utilize devices, (2) learning a single placement policy without modeling the system dynamics, and (3) depending solely on reinforcement learning in pre-training while ignoring optimization during deployment. We propose DOPPLER, a three-stage framework with two policies—SEL for selecting operations and PLC for placing them on devices. DOPPLER consistently outperforms baselines by reducing execution time and improving sampling efficiency through faster per-episode training.

## 1 INTRODUCTION

Existing systems for multi-GPU computing such as PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016), and the JAX-based Google stack (Frostig et al., 2018) proceed through a computation in a lock-step, level-wise fashion. Consider the three-matrix multiplication chain X $\times$ Y $\times$ Z in Fig. 1a, each matrix is partitioned into four submatrices to be distributed to eight GPUs. The resulting additions, multiplications, and data transfers form the dataflow graph in Fig. 1b. When implemented with JAX (Frostig et al., 2018) on a multi-GPU server, $X \times Y$ is first computed through pairwise matrix multiplications (e.g., $X_{11}, Y_{11}, \ldots$) distributed across the server's GPUs. Once these partial results are produced, an all-reduce operation is performed to aggregate them to perform additions. This collective synchronization step forces all GPUs to pause computation and wait until every device has communicated its share of results simultaneously (Li et al., 2020b). Then, $Z$ is computed with the results of $X \times Y$ followed by another collective communication step to synchronize results with all other GPUs with an all-reduce to compute the final round of matrix additions.

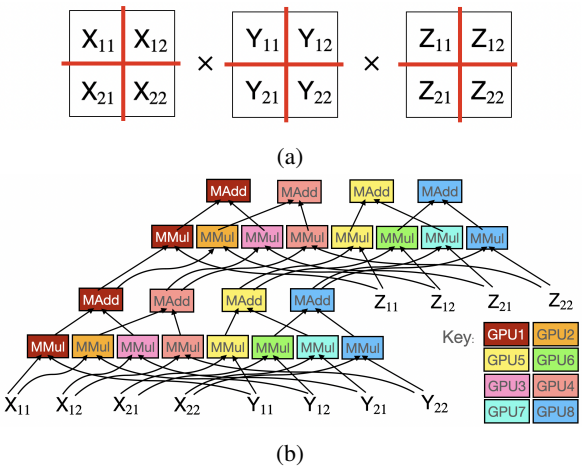


Figure 1: A dataflow graph (b) corresponding to the decomposed chain from (a). Vertices correspond to computation kernel calls, edges data dependencies, and colors the mapping of computations to GPUs.

In deep learning, many operations—as the matrix multiplication example—are executed bulk-synchronously (Valiant, 1990), which enforces barrier-style global synchronization across GPUs. This synchronization leads to idle resources and lost opportunities for speedup: for instance, an all-reduce operation cannot begin until every pairwise multiplication has completed, so one slow multiplication delays the entire step (Li et al., 2020c). Moreover, the all-reduce itself is communication-dominated, during which GPUs are severely underutilized.

A more efficient approach would overlap "reduce" with computation by scheduling operations (transfers and kernel calls) asynchronously, as soon as they can be run. A dynamic scheduler that never willingly allows resources to sit idle and schedules them dynamically is called *work-conserving* (Kleinrock, 1965). Table 1 shows the potential speedup of a work-conserving (WC) system on two workloads: a chain of matrix multiplications and additions, CHAINMM, and a feedforward neural network, FFNN (Configuration details are provided in Appendix F). On a GPU server, the WC system achieves a reduction of 46.3 ms (33%) for CHAINMM and 26.7 ms (53%) for FFNN, compared to PyTorch. These savings should be assessed in the context of long-term deployment, where even small per-query reductions (on the order of a few milliseconds) accumulate into substantial GPU-hour savings. For instance, a 24.2 ms reduction per query for running a Llama model (Yaadav, 2024) amounts to more than 2.4 million GPU hours saved annually at ChatGPT-scale workloads (assuming one million queries per day). Further experimental details are provided in Section 6.

| MODEL | WC SYSTEM | SYNCHRONOUS |
|---|---|---|
| CHAINMM | 139 | 185.3 |
| FFNN | 50.2 | 76.9 |

Table 1: Execution time (in milliseconds) for execution in a work-conserving system (WC) system and a synchronous system.

While work-conserving (WC) systems can improve efficiency, they introduce significant challenges, particularly for GPU assignment. In contrast to bulk-synchronous settings that fix execution order via all-reduce, WC systems rely on asynchronous point-to-point communication, where the lack of global synchronization makes ordering uncontrollable and performance highly sensitive to hardware heterogeneity and resource contention. An effective device assignment must capture this information and balance two competing goals: (1) maintaining GPU load balance and (2) minimizing inter-GPU communication (Harlap et al., 2018; Lu et al., 2017). Traditional placement methods emphasize communication minimization, but under a WC scheduler, the stochastic execution order makes load balancing especially challenging, as **load balancing is inherently temporal** (Saha et al., 2019).

In this paper, we tackle the device assignment problem in an asynchronous WC system by introducing DOPPLER, a reinforcement learning–based framework that adopts a learning-by-doing paradigm to optimize device placement. Prior work Addanki et al. (2019); Zhou et al. (2019); Mirhoseini et al. (2017) learns a single placement policy, DOPPLER learns efficient kernel assignments through a dual-policy sequential decision scheme. The first policy selects the next kernel (vertex in the dataflow graph) to assign by traversing the partially assigned dataflow graph in a manner that approximates the non-deterministic flow of "time," while the second policy determines the GPU placement of that kernel to balance load and minimize communication. This separation captures both execution dynamics and hardware constraints, producing assignments tailored to stochastic WC execution. Moreover, DOPPLER employs a three-stage framework for training its dual policies. Stage I (offline) uses supervised learning to train the policies to follow simple heuristics, such as co-locating neighboring vertices on the same device. Stage II (offline) transitions to reinforcement learning: the policies generate assignments that are "executed" in a simulated WC system, with rewards computed from the simulated runtime. Stage III (online) deploys the trained policies in a real WC system, where they are continuously refined through reinforcement learning using rewards derived from observing runtimes of dual-policy assignments in the system, recursively updating the policies as the system executes.

Our contributions are the following: (1) We investigate the device assignment problem in a multi-GPU system under a work-conserving scheduler; (2) We introduce a dual-policy learning approach to first learn the approximated traversing order of nodes before assigning them to devices; (3) We propose DOPPLER, a three-stage training framework to improve scheduling efficiency on the fly by continuously training it during deployment, along with two pretraining stages to accelerate convergence in deployment; and (4) Our experiments show that DOPPLER achieves up to 52.7% lower execution times than the best baseline. DOPPLER also achieves a significant runtime reduction compared to a stronger baseline that we designed (ENUMERATIVEOPTIMIZER) by up to 13.8%.

## 2 DEVICE ASSIGNMENT IN A WORK CONSERVING SYSTEM

Formally, given a *dataflow graph* $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ with nodes $\mathcal{V}$=$\{v_1, v_2, ..., v_n\}$ representing computations, and edges $\mathcal{E}$ = $\{e_1, e_2, ..., e_m\}$ representing data flows, as well as a set of devices $\mathcal{D}$, we aim to generate a *device assignment* $A$ which is a mapping from $\mathcal{V}$ to $\mathcal{D}$. $A_v$ denotes the device associated with vertex $v$ in $A$. $A$ is chosen to minimize the execution time $\texttt{ExecTime}(A)$.

For an example dataflow graph, consider the matrix multiplication chain $X \times Y \times Z$, which can be decomposed to run on a server with eight GPUs by sharding each matrix four ways in Fig. 1a. The resulting fine-grained dataflow graph contains eight submatrix multiplies associated with both of the two original multiplies, and four matrix additions to aggregate the results. A candidate assignment of the graph to eight GPUs is shown in Fig. 1b. This assignment achieves low execution time as (a) the expensive matrix multiplications that will tend to run in parallel are load-balanced, and (b) communication is minimized by co-locating neighboring nodes.

**A key question is: how to define the execution time of an assignment $\texttt{ExecTime}(A)$?** It is difficult to give a closed formula for $\texttt{ExecTime}(A)$, given the stochasticity of WC systems. As operations are issued dynamically, based on the state of the system, different runs of the same assignment can have very different execution times.

**Algorithm 1** $\texttt{ExecTime}(A)$

```
% rdy[v, d] is T iff the result of vertex
% v is on device d; initially, nothing is ready
rdy[v, d] ← F  ∀(v ∈ V, d ∈ D)
% except inputs: available everywhere
rdy[v, d] ← T  ∀(v ∈ V, d ∈ D) s.t. (v', v) ∉ E
t ← 0 % exec begins at time 0
S ← ⟨⟩ % schedule is empty
while ∃(v ∈ V) s.t. rdy[v, A_v] = F do
  tasks ← EnumTasks(rdy, A, S)
  task ← ChooseTask(rdy, A, S, tasks)
  if task = null then
    % if no task is chosen, just wait
    ⟨t, task⟩ ~ P(.|S, t) % which task done?
    S ← S + ⟨task, t, end⟩ % save completion
    rdy[vertex(task), device(task)] ← T
  else
    S ← S + ⟨task, t, beg⟩ % record initiation
  end if
end while
return t
```

Algorithm 1 describes how an assignment $A$ is executed in a WC system (the subroutine $\texttt{EnumTasks}(rdy, A, S)$ in Algorithm 2 enumerates the tasks that can be taken in each step by the scheduler). The algorithm stochastically simulates the execution of the assignment $A$ via a WC dynamic scheduler and returns the total execution time. It works by repeatedly asking the scheduler to choose the next task to schedule; when there is not a task that can be scheduled, the algorithm waits until an event is stochastically generated. In the algorithm, a *schedule* $S$ is the complete list of events that have occurred up to $t_{in}$. An *event* is a (*task*, *time*, *eventtype*) triple, where *task* is either an execution result `transfer` between devices or an execute (`exec`) of a node on a device, *time* records when the `transfer` or `exec` event happens, and *eventtype* specifies the recorded time as either `beg` or `end` of an event. `EnumTasks` enumerates all `transfer` and `exec` tasks that are ready when `EnumTasks` is called.

**Algorithm 2** $\texttt{EnumTasks}(rdy, A, S)$

```
output ← {}
% get all potential transfers
for (v1, v2) ∈ E do
  if rdy[v1, A_v2] = false and rdy[v1, A_v1] =
  true and transfer(v1, A_v1, A_v2) ∉ S then
    Add transfer(v1, A_v1, A_v2) to output
  end if
end for
% get all potential ops to exec
for v2 ∈ V do
  if rdy[v1, A_v2] = true ∀v1 s.t. (v1, v2) ∈ E
   and exec(v2, A_v2) ∉ S then
    Add exec(v2, A_v2) to output
  end if
end for
return output
```

Algorithm 1 has two key generic components that allow it to serve as a reasonable proxy or digital twin for a real-life scheduler, executed on real-life hardware. First, the distribution $P(\langle t_{out}, a\rangle | S, t_{in})$ governs the "next completed task." Given a schedule $S$ and a current time $t_{in}$, $P$ is a joint distribution over the next task to complete and the time $t_{out}$ at which this task completes. Second, the function `ChooseTask` encapsulates the underlying scheduling algorithm that is implemented by the WC system. It may choose any *task* from *tasks*. As described, it may operate depth-first (seeking to probe deeply into $\mathcal{G}$), breadth-first, or may employ any other applicable strategy.

In practice, Algorithm 1 is implemented by either (a) a simulator where the distribution $P(\langle t_{out}, a\rangle | S, t_{in})$ is realized by a model that takes into account factors such as the number of floating operations in the underlying operation (in the case of an operation such as a tensor contraction) or the number of bytes to be transferred (in the case of a GPU-to-GPU transfer), or (b) by

**Algorithm 3** ASSIGN($\mathsf{SEL}_\theta, \mathsf{PLC}_\theta$)

$A \leftarrow \{\}$
**for** $t' \in \{1, ..., m\}$ **do**
  Choose $v \leftarrow \mathsf{SEL}_\theta(A, \mathcal{G})$
  Choose $d \leftarrow \mathsf{PLC}_\theta(A, \mathcal{G}, v, \mathcal{D})$
  Add $(v \rightarrow d)$ to $A$
**end for**
return $A$

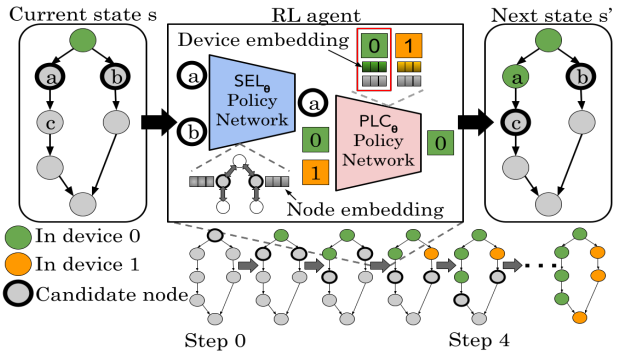


Figure 2: (Left) The ASSIGN algorithm, which sequentially produces an assignment $A$ using $\mathsf{SEL}_\theta$ policy and placed using $\mathsf{PLC}_\theta$ policy. (Right) A graphical depiction of the algorithm's implementation.

actually deploying the assignment $A$ in a real-life work-conserving system, and observing the running time. We use option (a) in Stage II of DOPPLER, and (b) in Stage III of DOPPLER (see Section 5).

## 3 PROBLEM DEFINITION AND A SOLUTION VIA REINFORCEMENT LEARNING

As there is no closed-form objective function, learning-by-doing is a promising approach for determining the optimal assignment $A^*$. Stages II and III of DOPPLER formulate choosing the assignment as a bandit problem. For an assignment $A$ at time tick $t$, we obtain a reward $r_t \sim R_A$ where $R_A$ is the reward distribution for $A$ (in practice, $r_t$ is sampled by invoking $\texttt{ExecTime}(A)$ and observing the runtime). Let $R^* = R_{A^*}$ where $A^* = \arg\max_A \mathbb{E}[R_A]$. Our goal is to minimize the regret:

$$\rho = \sum_{t=1}^{T} (\mathbb{E}[R^*] - r_t) \tag{1}$$

This is a bandit problem with $\mathcal{D}^{|\mathcal{V}|}$ arms. Our problem is not amenable to classic solutions because the number of arms is so large. For example, if we are producing an assignment for an 8-GPU server that is to execute a dataflow graph with 100 vertices, there are $\approx 2^{300}$ possible assignments.

Fortunately, there is a combinatorial structure to the assignment problem that can allow us to deal with the very large set of possible assignments (Cesa-Bianchi & Lugosi, 2012; Chen et al., 2013). Note that if two assignments $A_1$ and $A_2$ differ only in how a few vertices have been assigned to devices, it is likely that the two reward distributions $R_{A_1}$ and $R_{A_2}$ will be similar. Thus, it may be possible to systematically search the possible assignments.

Our approach builds the assignment at each time tick using a sequential process controlled by two policies $\mathsf{SEL}_\theta$ and $\mathsf{PLC}_\theta$. If we use ASSIGN($\mathsf{SEL}_\theta, \mathsf{PLC}_\theta$) (Algorithm 3) as the mechanism for choosing the assignment $A$ at each time tick $t$ of the bandit problem of Equation 1, this process can be reformulated as an episodic Markov decision process (MDP) as shown in Liu et al. (2024), where each episode executes ASSIGN($\mathsf{SEL}_\theta, \mathsf{PLC}_\theta$) (shown in Figure 2) and a reward is obtained once the assignment is completed at the end of each episode. Therefore, any suitable reinforcement learning algorithm can be used to learn the policies $\mathsf{SEL}_\theta$ and $\mathsf{PLC}_\theta$. In our implementation, we apply a graph neural network (GNN) along with message passing to encode graph $\mathcal{G}$ and use a feedforward neural network for decoding actions for $\mathsf{SEL}_\theta$ and $\mathsf{PLC}_\theta$. Details on GNN architectures are in Section 4.2.

## 4 DOPPLER DUAL POLICY IMPLEMENTATIONS

We describe our episodic Markov Decision Process formulation (Section 4.1), along with the graph neural network architectures that we used to implement the dual policy learned during DOPPLER training (Section 4.2) and an efficiency analysis of the GNN message-passing (Section 4.3).

### 4.1 EPISODIC MDP FORMULATION

We formulate device assignment as an episodic Markov Decision Process (MDP) $(\mathcal{S}, \mathcal{A}, H, \mathcal{P}, \mathcal{R})$ where $\mathcal{P}$ is the transition function, and H is the horizon that equals the number of nodes in the graph.

**States** Each state $s_h \in \mathcal{S}$ is a tuple $(X_\mathcal{G}, \mathcal{C}_h, X_{\mathcal{D},h})$, where $X_\mathcal{G} = (X_\mathcal{V}, X_\mathcal{E})$ represents the static graph features including node and edge features such as bottom-level paths (i.e., to entry nodes), top-level (i.e., to exit nodes) paths, and communication costs. $\mathcal{C}_h$ is the dynamic set of candidate nodes and $\mathcal{C}_0$ is defined as the entry nodes in the graph. Finally, $X_{\mathcal{D},h}$ represents the dynamic device features (e.g., total computing time and the end time for computations in each device). Details about the sets of features can be found in the Appendix D.

**Actions** At each time-step $h$, the agent takes an action $a_h \in \mathcal{A}$ where $a_h = (v_h, d_h)$. Each action selects a node $v_h$ and places it into device $d_h$ using policies $\mathsf{SEL}_\theta$ and $\mathsf{PLC}_\theta$ as shown in Figure 2. Each episode is composed of $|\mathcal{V}|$ iterations—i.e., one iteration per node in $\mathcal{V}$.

**Reward** We calculate rewards using the execution time $R_{s_H} = (-1) * \texttt{ExecTime}(s_H)$ derived from either the real system or a simulator. The reward is computed at the end of each episode ($h = H$), with intermediate rewards set to zero for efficiency. To enhance stability, we subtract a baseline reward equal to the average execution time observed across all previous episodes ($\overline{R_{s_H}}$). The final reward is computed as $r_H = R_{s_H} - \overline{R_{s_H}}$.

### 4.2 DUAL POLICY GRAPH NEURAL NETWORK ARCHITECTURES

We will describe the policy networks for computing $\mathsf{SEL}_\theta$ and $\mathsf{PLC}_\theta$ in Algorithm 3. The symbol $\theta$ is used here to emphasize that these functions have parameters that will be optimized as part of the training process. DOPPLER applies a Graph Neural Network (GNN) to encode node information in the dataflow graph. Our GNN is a message-passing neural network (Gilmer et al., 2017) that learns node representations for each node $v$ via $K$ successive iterations:

$$\mathbf{h}_v^{[k]} = \phi(\mathbf{h}_v^{[k-1]}, \bigoplus_{u \in N(v)} \psi(\mathbf{h}_u^{[k-1]}, \mathbf{h}_v^{[k-1]}, \mathbf{e}_{uv}))$$

where $\mathbf{h}_v^{[k]}$ are representations learned at the $k$-th layer, $\mathbf{h}_v^{[0]} = X_\mathcal{V}[v]$, $\psi$ and $\phi$ are functions, $N(v)$ are the neighbors of $v$, and $\bigoplus$ is a permutation-invariant operator. We will use $\mathsf{GNN}(\mathcal{G}, X_\mathcal{G}) = [\mathbf{h}_1^{[K]}; \mathbf{h}_2^{[K]}; \dots \mathbf{h}_n^{[K]}]$ to refer to the representations of all nodes in $\mathcal{G}$.

We also apply an $L$-layer feedforward neural network (FFNN) to encode node information:

$$\mathbf{x}_v^{[l]} = W^{[l]} \mathbf{x}_v^{[l-1]} + \mathbf{b}^{[l]}$$

where $\mathbf{x}_v^{[l]}$ are representations at the $l$-th layer, and $W^{[l]}$ and $\mathbf{b}^{[l]}$ are weights and biases, respectively. Let $\mathsf{FFNN}(X) = [\mathbf{x}_1^{[L]}; \mathbf{x}_2^{[L]}; \dots \mathbf{x}_n^{[L]}]$ be the representations of all nodes in $\mathcal{G}$ with $\mathbf{x}_v^{[0]} = X[v]$.

**Node policy network ($\mathsf{SEL}_\theta$):** Selects a node from the candidate set $\mathcal{C}$ based on observed graph state $X_\mathcal{G}$ using the $\epsilon$-greedy approach. Let $b(v)$ and $t(v)$ be the b-path and t-path for $v$, where a b-level (t-level) path for $v$ is the longest path from $v$ to an entry (exit) node in $\mathcal{G}$. We aggregate information from these critical paths via GNN embeddings $H[u]$ for each node $u$ along them. Nodes are selected according to probabilities estimated from a graph embedding matrix $H_\mathcal{G}$, which is a result of the concatenation of critical path ($\mathbf{h}_{v,b}$ and $\mathbf{h}_{v,t}$), GNN ($H[v]$), and feature ($Z[v]$) representations.

$$
\begin{array}{lll}
H = \mathsf{GNN}(\mathcal{G}, X_\mathcal{G}) & \mathbf{h}_{v,b} = \sum_{u \in b(v)} H[u] & \mathbf{h}_{v,t} = \sum_{u \in t(v)} H[u] \\
Z = \mathsf{FFNN}(X_\mathcal{V}) & \mathbf{h}_v = [H[v] \parallel \mathbf{h}_{v,b} \parallel \mathbf{h}_{v,t} \parallel Z[v]] & H_\mathcal{G} = [\mathbf{h}_1; \mathbf{h}_2; \dots \mathbf{h}_\mathcal{C}] \\
H'_\mathcal{G} = \mathsf{LeakyReLU}(\mathsf{FFNN}(H_\mathcal{G})) & Q_\mathcal{G}(v) = \mathsf{softmax}(\mathsf{FFNN}(H'_\mathcal{G})) &
\end{array}
$$

$$\mathsf{SEL}_\theta(\mathcal{G}, X_\mathcal{G}) = \begin{cases} \arg\max_v Q_\mathcal{G}(v) & p = 1 - \epsilon \\ \text{random } v \in \mathcal{C} & p = \epsilon \end{cases}$$

where $p$ is the probability of the event and $\epsilon$ is a parameter.

**Device policy network ($\mathsf{PLC}_\theta$):** Places a node $v$ into one of the devices in $\mathcal{D}$ based on the composed

state observation $(v, X_{\mathcal{D}}, X_{\mathcal{G}})$. Devices are selected based on an embedding matrix $H_{\mathcal{D}}$ for the set of devices generated by concatenating representations for the node ($H[v]$), node features ($Z[v]$), device features ($Y[d]$,) and for nodes already placed into the device ($\mathbf{h}_d$).

$$H = \mathsf{GNN}(\mathcal{G}, X_{\mathcal{G}}) \qquad \mathbf{h}_d = \textstyle\sum_{u:d_u=d} H[u] \qquad Y = \mathsf{FFNN}(X_{\mathcal{D}})$$
$$Z = \mathsf{FFNN}(X_{\mathcal{G}}) \qquad \mathbf{h}_{v,d} = [H[v] \parallel \mathbf{h}_d \parallel Y[d] \parallel Z[v]] \qquad H_{\mathcal{D}} = [\mathbf{h}_1; \mathbf{h}_2; \dots \mathbf{h}_m]$$
$$H'_{\mathcal{D}} = \mathsf{LeakyReLU}(\mathsf{FFNN}(H_{\mathcal{D}})) \qquad Q_{\mathcal{D}}(d) = \mathsf{softmax}(\mathsf{FFNN}(H'_{\mathcal{D}}))$$

$$\mathsf{PLC}_\theta(v, \mathcal{D}, \mathcal{G}, X_{\mathcal{D}}, X_{\mathcal{G}}) = \begin{cases} \arg\max_d Q_{\mathcal{D}}(d) & p = 1-\epsilon \\ \text{random } d \in \mathcal{D} & p = \epsilon \end{cases}$$

### 4.3 Efficient Message Passing Approximation

Implementing the MDP described in Section 4.1 requires performing message-passing on the dataflow graph $\mathcal{G}$ when calling $\mathsf{SEL}_\theta$ and $\mathsf{PLC}_\theta$ policies at each MDP step $h$. We found this to be prohibitive for large graphs since we may apply up to 8k episodes $\times$ 261 steps = 2m steps in our experiments. PLACETO (one of our baselines) suffers from this issue, being very inefficient during training as it performs one message-passing round per MDP step. Instead, we propose performing message passing on the graph only once per MDP episode and encoding updated assignment information at each step $h$ in device $X_{\mathcal{D},h}$ without message passing. We found empirically that this modification has a negligible impact over DOPPLER's convergence but leads to a significant reduction in training time, especially for large neural networks (we show an ablation study in Appendix G.3).

## 5 DOPPLER: Cost-Effective Training

DOPPLER adopts an efficient three-stage framework (A detailed illustration in Appendix I):

**Imitation Learning stage (Stage I).** We propose using imitation learning for teaching the dual policy to replicate the decisions of an existing heuristic (teacher) before deploying it to the real system. We apply the decisions of CRITICAL PATH (Kwok & Ahmad, 1999) ($a_{cp}$):

$$J(\theta) = \mathbb{E}_{a_{cp}\sim\Pi_{cp}(s), s\sim T(s',a), a\sim\Pi_\theta(s')}[\nabla_\theta \log \Pi_\theta(a_{cp}|s)] \tag{2}$$

**Simulation-based reinforcement learning stage (Stage II).** Even after pre-training with a teacher, we may have a policy that is too low-quality for deployment in the real system, where longer running times or slow convergence due to exploration may be unacceptable. Thus, we also train DOPPLER using a software-based simulator that implements Algorithm 1. The dual-policy networks are updated using the policy gradient method (Sutton et al., 1999). We maximize the following objective function:

$$J(\theta) = \mathbb{E}_{a\sim\Pi_\theta(s)}\left[\nabla_\theta(\log \Pi_\theta(a|s))R(s,a)\right] \tag{3}$$

**Real-system reinforcement learning stage (Stage III).** Because the dual policy is of reasonably high quality before it is deployed in a user-facing environment, actual users need not suffer through long wait times due to low-quality assignments. Moreover, the dual policy can be continuously improved after its deployment in a real WC system. Further, the reward signal for optimizing Equation 3 in this stage is obtained "for free" by observing real-life runtimes (`ExecTime`), so this continuous improvement of the assignments produced is possible with no additional cost.

## 6 Experiments

We focus on the following questions: Q1: How does DOPPLER compare against alternative approaches in terms of execution time? Q2: What are the individual contributions of the $\mathsf{SEL}_\theta$ and $\mathsf{PLC}_\theta$ policies on DOPPLER's performance? Q3: How can imitation learning (Stage I), simulation-based RL (Stage II), and real system RL (Stage III) be combined to improve DOPPLER's training? Q4: How does DOPPLER's training and inference cost scale with the size of graphs? Q5: How can DOPPLER's execution time be interpreted based on the assignments it produces? Q6: Can DOPPLER be trained on one dataflow graph and be generalized to new graphs and to new hardware architectures?

| | 4 GPUs | | | | | | Runtime Reduction | |
|---|---|---|---|---|---|---|---|---|
| Model | Crit. Path | Placeto | GDP | EnumOpt. | Doppler-sim | Doppler-sys | baseline | EnumOpt. |
| ChainMM | $230.4 \pm 4.3$ | $137.1 \pm 2.2$ | $198 \pm 3.3$ | $139 \pm 10.0$ | **122.5** $\pm 4.0$ | $123.4 \pm 2.5$ | 10.7% | 11.9% |
| FFNN | $217.8 \pm 11.3$ | $126.3 \pm 5.8$ | $100.3 \pm 3.2$ | $50.2 \pm 2.5$ | $49.9 \pm 1.1$ | **47.4** $\pm 0.7$ | 52.7% | 5.6% |
| Llama-block | $230.9 \pm 8.7$ | $411.5 \pm 19.7$ | $336.5 \pm 8.4$ | $172.7 \pm 5.0$ | $191.5 \pm 5.97$ | **160.3** $\pm 4.30$ | 30.6% | 7.2% |
| Llama-layer | $292.6 \pm 5.8$ | $295.1 \pm 7.0$ | $231.5 \pm 5.1$ | $174.8 \pm 4.7$ | $167 \pm 3.4$ | **150.6** $\pm 4.2$ | 48.5% | 13.8% |

Table 2: Real engine execution times (in milliseconds) for assignments identified by our approaches (EnumOpt. and Doppler) compared against existing baselines. During training, both Placeto and Doppler-sim rely on a simulator to find good solutions, while others use a real system.

### 6.1 Experimental Setup

**Neural network architectures.** We test dataflow graphs from four types of neural network architectures in our experiments: a feed-forward neural network (FFNN), chain matrix multiplications (ChainMM), a Llama transformer block (Llama-block), and a complete Llama transformer layer (Llama-layer). Further details can be found in Appendix C.

**GPU systems.** We compare the assignment approaches using 4 NVIDIA Tesla P100 GPUs and 16GB of memory each. Moreover, we did ablation studies on 1) 8GB out of 16GB restricted GPU memory, 2) 8 NVIDIA V100 GPUs with 32GB memory, and show results in Appendix H.

**Baselines.** We compare our approach against four baselines. Critical path Kwok & Ahmad (1999) is a popular (non-learning) heuristic for DAG device assignment. Placeto Addanki et al. (2019) is a recent RL-based alternative that applies a single (device) policy and is trained using simulations (see ablation study on the simulator in Appendix G.1). GDP Zhou et al. (2019) is another recent RL-based method that consists of graph embedding and sequential attention. EnumerativeOptimizer is a baseline we developed—it is our best effort at exploiting the structure of a sharded tensor computation to produce a high-quality assignment (described in detail in the Appendix A).

| | 4 GPUs | | |
|---|---|---|---|
| Model | Doppler-sys | Doppler-sel | Doppler-plc |
| ChainMM | $123.4 \pm 2.5$ | $127.0 \pm 0.8$ | **121.6** $\pm 0.7$ |
| FFNN | **47.4** $\pm 0.7$ | $59.1 \pm 7.6$ | $63.2 \pm 1.6$ |
| Llama-block | **160.3** $\pm 4.3$ | $175.6 \pm 4.1$ | $172.9 \pm 4.3$ |
| Llama-layer | **150.6** $\pm 4.2$ | $161.7 \pm 4.1$ | $159.5 \pm 4.9$ |

Table 3: Real engine execution time (in milliseconds) of our approach (Doppler-sys) against only applying $\mathsf{SEL}_\theta$ (Doppler-sel) or only $\mathsf{PLC}_\theta$ (Doppler-plc). The results show that both contribute to the performance improvements.

**Hyperparameters.** For RL-based methods, we run 4k episodes for ChainMM and FFNN and 8k episodes for Llama-block and Llama-layer. We tried different learning rate schedules for each method (initial values $\{1e{-}3, 1e{-}4, 1e{-}5\}$) and found that $1e{-}3$ decreasing linearly to $1e{-}6$ works best for Placeto and $1e{-}4$ linearly decreasing to $1e{-}7$ works best for all versions of Doppler and GDP. We apply a $0.5$ exploration rate linearly decreasing to $0.0$ for Placeto and $0.2$ linearly decreasing to 0.0 for Doppler and GDP. An entropy weight of $1e{-}2$ is applied for all RL methods. For Critical path, we run 50 assignments and report the best execution time. The reported execution time (and standard deviation) using the real system is the average of 10 executions.

### 6.2 Results and Discussion

**Comparison between our solutions and existing alternatives (Q1).** Table 2 reports execution times across neural network architectures on 4 GPUs. Doppler-sys outperforms all baselines in most settings, with Doppler-sim often second best. For example, Doppler-sys reduces runtime by up to 78.2% over Critical Path, 62.5% over Placeto, and 52.7% over GDP, while both Doppler-sys and Doppler-sim surpass EnumerativeOptimizer by up to 13.8%. Additional ablations with different seeds are in Appendix G.2.

**Ablation study for select and place policies (Q2).** Table 3 presents an ablation study isolating the effects of the node and device policies. In the ablated variants, we replace our policies with Critical Path strategies: Doppler-sel assigns selected nodes to the earliest-available device, while Doppler-plc selects nodes with the longest path to an exit. Overall, combining both policies

yields the best performance. For ChainMM, DOPPLER-PLC slightly outperforms DOPPLER-SYS by a few milliseconds, but for more complex models the combined policies provide clear gains.

**Improving training using Stage I, Stage II, and Stage III (Q3).** Fig. 3 shows execution times for DOPPLER-SYS when trained using different combinations of imitation learning (I), simulations (II), and real system executions (III) for the LLAMA-LAYER dataflow graph. As we hypothesized, training using the real system only leads to slower convergence due to the need for exploration starting from a poor initial model, and leads to unstable performance. Imitation learning and simulations enable faster convergence and lower execution times. Ablation studies are conducted on pre-training PLACETO in Appendix G.4.

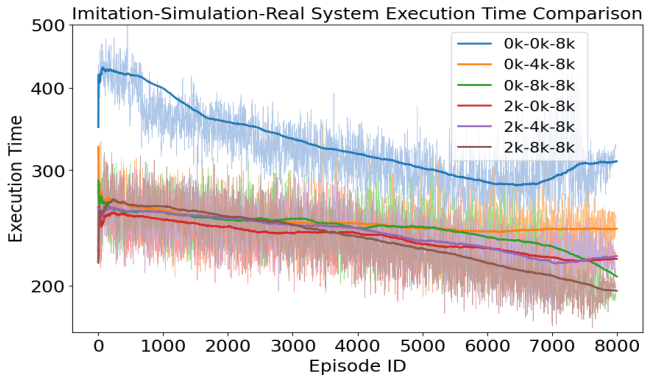


Figure 3: Real engine execution times (in milliseconds) for DOPPLER-SYS using different combinations of three training stages for the LLAMMA-LAYER dataflow graph.

**DOPPLER training and inference scalability (Q4).** We analyze DOPPLER's scalability in both training and inference time for dataflow graphs with increasing size in Fig. 4. The figure shows that DOPPLER scales linearly with the size of graphs and, compared to RL-based baselines such as GDP, achieves the lowest training and inference times. Detailed discussion is provided in Appendix J

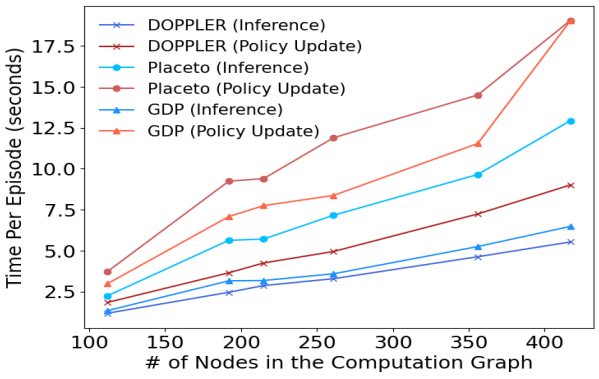


Figure 4: Inference time and RL policy update time as the number of nodes in the dataflow graph increases.

**Visualizing DOPPLER-SYS's assignments (Q5).** Fig. 5 shows the assignments produced by DOPPLER-SYS for the FFNN dataflow graph, which achieves both GPU load balancing and communication minimization along the critical path. By further profiling how the assignments are scheduled in the system (see Appendix B), we find that DOPPLER schedules often enable overlapping communication and computation across GPUs, minimizing stalling and maximizing GPU utilization. Additional analyses are provided in Appendix E.

**DOPPLER's transfer ability across graphs and architectures (Q6).** We evaluate transfer learning (1) from simple architectures (FFNN, CHAINMM) to Llama-structured graphs on the same hardware and (2) across hardware architectures for the same graph. With 2K fine-tuning episodes, DOPPLER adapts to new architectures and outperforms baselines. With 4K episodes (less than half of the original training), it achieves assignments comparable to full target training. In our hardware adaptability experiments, we train a policy for the FFNN graph with four P100 GPUs and transfer it to eight V100 GPUs. The zero-shot setting yields 82.7% of communication intra-GPU, 6.7% within the same

Figure 5: Assignments for FFNN found by DOPPLER. Colors show the mapping of computations to GPUs. DOPPLER is effective at both load balancing and communication minimization across GPUs.

| | | 4 GPUs | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TRAIN MODEL | TARGET MODEL | ZERO-SHOT | 2K-SHOT | 4K-SHOT | DOPPLER-SYS | CRIT. PATH | PLACETO | ENUMOPT. |
| FFNN | LLAMA-BLOCK | $251.0 \pm 2.9$ | $165.3 \pm 4.1$ | **159.4** $\pm 4.8$ | **160.3** $\pm 4.3$ | $230.9 \pm 8.7$ | $411.5 \pm 19.7$ | $172.7 \pm 5.0$ |
| CHAINMM | LLAMA-BLOCK | $242.3 \pm 6.7$ | $184.9 \pm 4.3$ | **174.0** $\pm 4.4$ | **160.3** $\pm 4.3$ | $230.9 \pm 8.7$ | $411.5 \pm 19.7$ | $172.7 \pm 5.0$ |
| FFNN | LLAMA-LAYER | $206.1 \pm 4.5$ | $158.2 \pm 4.1$ | **155.8** $\pm 5.0$ | **150.6** $\pm 4.2$ | $292.6 \pm 5.8$ | $295.1 \pm 7.0$ | $174.8 \pm 4.7$ |
| CHAINMM | LLAMA-LAYER | $338.2 \pm 5.0$ | $164.4 \pm 3.3$ | **156.4** $\pm 4.4$ | **150.6** $\pm 4.2$ | $292.6 \pm 5.8$ | $295.1 \pm 7.0$ | $174.8 \pm 4.7$ |

Table 4: Real engine execution times (in milliseconds) for assignments identified by DOPPLER under different few-shot settings (Zero-shot, 3k-shot, 4k-shot) compared against baselines. The results show that DOPPLER finds comparable results to full training (DOPPLER-SYS) in 4k-shot.

GPU group (with all-to-all NVLink), and 10.6% across GPUs without direct NVLink. After 2K episodes, the policy improves assignments to 94.7% intra-GPU, 1.9% within NVLink groups, and only 3.4% across GPUs without NVLink (see detailed results in Appendix K).

## 7 RELATED WORKS

**Classical Approaches for Device Placement and Scheduling.** List scheduling (LS) heuristics, such as CRITICAL PATH, decompose the problem of computing a schedule into a sequence of *select* and *place* steps (Kwok & Ahmad, 1999). DOPPLER can be seen as a neural LS heuristic that learns to *select* and *place* directly from observations using an MDP. Our experiments show that DOPPLER outperforms CRITICAL PATH. Graph partitioning (Kernighan & Lin, 1970; Kirkpatrick et al., 1983; Fiduccia & Mattheyses, 1988; Johnson et al., 1989; Hagen & Kahng, 1992; Karypis, 1997) can also be applied for device placement but previous work has shown that RL is a better alternative for the problem (Mirhoseini et al., 2017).

**Reinforcement Learning for Combinatorial Optimization.** Traditional algorithms for combinatorial optimization problems often rely on hand-crafted heuristics that involve sequentially constructing a solution. Recently, there has been a growing interest in applying RL (and deep learning more broadly) to learn heuristics for these problems (Mazyavkina et al., 2021). For instance, (Bello et al., 2016) introduced a policy gradient method for the Traveling Salesman Problem (TSP). Subsequent studies extended RL to problems beyond TSP (Khalil et al., 2017; Cappart et al., 2019; Drori et al., 2020; Emami & Ranka, 2018; Lu et al., 2019; Mazyavkina et al., 2021; Nazari et al., 2018; Kool et al., 2018; Drori et al., 2020; Abe et al., 2019; Manchanda et al., 2019; Chen & Tian, 2019; Li et al., 2020a; Laterre et al., 2018; Gu & Yang, 2020; Cai et al., 2019). Our work is unique in how it leverages the combinatorial structure with list scheduling heuristics and direct access to the target system during training to address the device assignment problem using RL.

**RL for Device Placement and Scheduling.** Early work introduced a sequence-to-sequence RNN trained with policy gradients for device placement, showing RL can outperform heuristics (Mirhoseini et al., 2017; Pellegrini, 2007). PLACETO (Addanki et al., 2019) replaced the RNN with a GNN, while Paliwal et al. (2019) combined RL with a Genetic Algorithm, though at high evaluation cost. Zhou et al. (2019) proposed a graph-embedding approach with sequential attention and a single placement policy. DOPPLER introduces dual-policy learning with three-stage training for faster convergence and continuous optimization. More recent work includes end-to-end optimization from graph construction to placement (Duan et al., 2024) and improved node representations using cosine phase position embeddings (Han et al., 2024), which are complementary to our approach.

## 8 CONCLUSION

In this paper, we have considered the problem of assigning computations in a dataflow graph to devices to minimize execution time in a work-conserving system. We have proposed DOPPLER, a dual-policy learning framework for learning device assignment in three stages. Some of the key innovations are (1) DOPPLER explicitly tries to learn an approximate node traversing order, to make the assignment problem easier, (2) DOPPLER adopts two pre-training stages using imitation learning and simulation-based learning to speed up policy convergence, and (3) DOPPLER continues dual-policy training during deployment, achieving a gradual reduction in execution time over time.

## REPRODUCIBILITY STATEMENT

We provide details of experimental settings and hyperparameters used for training in Section 6.1 to reproduce our results and experiments. We have included an anonymous link in Appendix L with our code and data used for running all experiments.

## REFERENCES


Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving np-hard problems on graphs with extended alphago zero. *arXiv preprint arXiv:1905.11623*, 2019.

Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv preprint arXiv:1906.08879*, 2019.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Daniel Bourgeois, Zhimin Ding, Dimitrije Jankov, Jiehui Li, Mahmoud Sleem, Yuxin Tang, Jiawen Yao, Xinyu Yao, and Chris Jermaine. `EinDecomp`: Decomposition of declaratively-specified machine learning and numerical computations for parallel execution. *Proceedings of the VLDB Endowment*, 18(7):2240–2253, 2025.

Qingpeng Cai, Will Hang, Azalia Mirhoseini, George Tucker, Jingtao Wang, and Wei Wei. Reinforcement learning driven heuristic optimization. *arXiv preprint arXiv:1906.06639*, 2019.

Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 1443–1451, 2019.

Nicolo Cesa-Bianchi and Gábor Lugosi. Combinatorial bandits. *Journal of Computer and System Sciences*, 78(5):1404–1422, 2012.

Wei Chen, Yajun Wang, and Yang Yuan. Combinatorial multi-armed bandit: General framework and applications. In *International conference on machine learning*, pp. 151–159. PMLR, 2013.

Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. *Advances in neural information processing systems*, 32, 2019.

Zhimin Ding, Jiawen Yao, Brianna Barrow, Tania Lorido Botran, Christopher Jermaine, Yuxin Tang, Jiehui Li, Xinyu Yao, Sleem Mahmoud Abdelghafar, and Daniel Bourgeois. Turnip: A" nondeterministic" gpu runtime with cpu ram offload. *arXiv preprint arXiv:2405.16283*, 2024.

Iddo Drori, Anant Kharkar, William R Sickinger, Brandon Kates, Qiang Ma, Suwen Ge, Eden Dolev, Brenda Dietrich, David P Williamson, and Madeleine Udell. Learning to solve combinatorial optimization problems on real-world graphs in linear time. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 19–24. IEEE, 2020.

Shukai Duan, Heng Ping, Nikos Kanakaris, Xiongye Xiao, Peiyu Zhang, Panagiotis Kyriakis, Nesreen K Ahmed, Guixiang Ma, Mihai Capota, Shahin Nazarian, et al. A structure-aware framework for learning device placements on computation graphs. *arXiv preprint arXiv:2405.14185*, 2024.

Patrick Emami and Sanjay Ranka. Learning permutations with sinkhorn policy gradient. *arXiv preprint arXiv:1805.07010*, 2018.

Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, pp. 241–247. 1988.

Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pp. 1263–1272. PMLR, 2017.

Shenshen Gu and Yue Yang. A deep learning algorithm for the max-cut problem based on pointer network structure with supervised learning and reinforcement learning strategies. *Mathematics*, 8 (2):298, 2020.

Lars Hagen and Andrew B Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(9):1074–1085, 1992.

Meng Han, Yan Zeng, Jilin Zhang, Yongjian Ren, Meiting Xue, and Mingyao Zhou. A novel device placement approach based on position-aware subgraph neural networks. *Neurocomputing*, 582: 127501, 2024.

Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

David S Johnson, Cecilia R Aragon, Lyle A McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations research*, 37(6):865–892, 1989.

George Karypis. Metis: Unstructured graph partitioning and sparse matrix ordering system. *Technical report*, 1997.

Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

Leonard Kleinrock. A conservation law for a wide class of queueing disciplines. *Naval Research Logistics Quarterly*, 12(2):181–192, 1965.

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Torbjorn S Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. *arXiv preprint arXiv:1807.01672*, 2018.

Dongda Li, Changwei Ren, Zhaoquan Gu, Yuexuan Wang, and Francis Lau. Solving packing problems by conditional query learning. 2020a.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020b.

Shigang Li, Tal Ben-Nun, Giorgi Nadiradze, Salvatore Di Girolamo, Nikoli Dryden, Dan Alistarh, and Torsten Hoefler. Breaking (global) barriers in parallel stochastic optimization with wait-avoiding group averaging. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1725–1739, 2020c.

Xutong Liu, Siwei Wang, Jinhang Zuo, Han Zhong, Xuchuang Wang, Zhiyong Wang, Shuai Li, Mohammad Hajiesmaili, John Lui, and Wei Chen. Combinatorial multivariant multi-armed bandits with applications to episodic reinforcement learning and beyond. *arXiv preprint arXiv:2406.01386*, 2024.

Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *International conference on learning representations*, 2019.

Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*, pp. 553–564. IEEE, 2017.

Sahil Manchanda, Akash Mittal, Anuj Dhawan, Sourav Medya, Sayan Ranu, and Ambuj Singh. Learning heuristics over large graphs via deep reinforcement learning. *arXiv preprint arXiv:1903.03332*, 2019.

Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International conference on machine learning*, pp. 2430–2439. PMLR, 2017.

Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. *arXiv preprint arXiv:1905.02494*, 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

François Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *Euro-Par 2007 Parallel Processing: 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007. Proceedings 13*, pp. 195–204. Springer, 2007.

Sujan Kumar Saha, Yecheng Xiang, and Hyoseung Kim. Stgm: Spatio-temporal gpu management for real-time tasks. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–6. IEEE, 2019.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8): 103–111, 1990.

Vignesh Yaadav. Exploring and building the llama 3 architecture: A deep dive into components, coding, and inference techniques, 2024.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.

Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint arXiv:1910.01578*, 2019.

# A Enumerative Assignment Algorithm (Algorithm 4)

In this section, we describe our enumerative assignment algorithm, which uses a level-by-level, exhaustive enumeration in an attempt to find an assignment $A$ for the vertices in a graph $\mathcal{G}$ to minimize $\texttt{ExecTime}(A)$. This algorithm uses a greedy approach that first groups vertices based on the graph structure and then attempts a subset of possible assignments for the operations within each group and selects the assignment with the minimum estimated cost.

This algorithm requires that the vertices in the graph $\mathcal{G}$ have been organized into a list of "meta-ops" called $M$. As our input dataflow graph has been created by sharding a compute graph using a framework such as Alpa Zheng et al. (2022), each operation in the input dataflow graph $\mathcal{G}$ is descended from some operation that has been sharded. For example, consider Figure 1. All of the eight `MMul` ops at the lowest level of the graph, as well as the four `MAdd` ops at the next level were created by sharding $X \times Y$. We group all of these twelve operations and term them a "meta-op". Further, we can topologically order these meta-ops so that if $m_1$ comes before $m_2$ in $M$, it means that none of the vertices in $m_2$ can be reached from some vertex in $m_1$ by traversing $\mathcal{E}$.

Note that each meta-op has two subsets—one of which may be empty: a set of computationally expensive operations (such as the `MMul` ops) that result directly from sharding the original operation, and a set of less expensive operations needed to aggregate and/or recompose the results of the first set of operations. For a meta-op $m$, we call these $m.\texttt{shardOps}$ and $m.\texttt{reduceOps}$. Note that the original operation is always sharded so that if there are $n$ devices, there are $n$ items in $m.\texttt{shardOps}$, and load-balancing of these shards is crucial. Thus, our tactic will be to always partition $m.\texttt{shardOps}$ across the $n$ devices, and never assign two operations in $m.\texttt{shardOps}$ to the same device. Likewise, if the meta-op is sharded into $n$ there will always be at most $n$ items in $m.\texttt{reduceOps}$. Therefore, we always partition them across (possibly a subset of) the devices.

Given this, our algorithm "EnumerativeOptimizer" proceeds through the list $M$ of meta-ops in order. For each $m$, it exhaustively tries all assignments of $m.\texttt{shardOps}$. Each assignment is costed by computing the time required to transfer all of the items in $m.\texttt{shardOps}$ to where they will be consumed. These times are estimated using statistics gathered by testing transfers on the actual hardware. Once $m.\texttt{shardOps}$ is placed, then $m.\texttt{reduceOps}$ is placed, using the same cost model. Because of the ordering of the meta-ops in $M$, and because we process $m.\texttt{shardOps}$ before $m.\texttt{reduceOps}$, we always know the assignment of the input to $m.\texttt{shardOps}$ or $m.\texttt{reduceOps}$ before we place it, and so it is easy to compute the cost.

This algorithm is greedy in the sense that it processes meta-ops one at a time, from start to finish, using the topological ordering. Thus, if the cost model is correct, it will be optimal only as long as each $m.\texttt{shardOps}$ and $m.\texttt{reduceOps}$ is run in lock-step, with a barrier before each set is executed. Obviously, this is not the case in reality with a dynamic scheduler, but one might expect the algorithm to produce a good assignment in practice.

# B System Implementation

The underlying system runtime Bourgeois et al. (2025); Ding et al. (2024) that executes our graphs is written in C++. The dataflow graph $\mathcal{G}$ is executed asynchronously by a single-threaded event loop whose job is to monitor when the dependencies implicit in the graph are satisfied. When the inputs to a vertex are found to be available, the event loop checks whether the resources necessary to execute the vertex are also available (a "resource" may be an open GPU stream or an open communication channel). If resources are available, the event loop may choose to execute the vertex. We use the term "event loop" because the loop asks the necessary resources to execute the vertices in $\mathcal{G}$ in response to "events." An event occurs whenever a graph dependency is satisfied, or when a previously used resource becomes available. The main event loop is notified of this via an asynchronous callback. In our implementation, we use CUDA version 11.8.0 and cuTensor version 2.0.1.

**Algorithm 4** EnumerativeOptimizer

**Input:** Sorted list of meta-ops $M$ containing all vertices in $\mathcal{G}$
**Output:** Assignment $A$
A $\leftarrow \langle\rangle$ % assignment is empty
% loop thru meta-ops
**for** $m \in M$ **do**
  % First deal with the shared op
  $A \leftarrow$ `getBestAssign`$(m.\texttt{shardOps}, A)$
  % now place the reductions
  $A \leftarrow$ `getBestAssign`$(m.\texttt{reduceOps}, A)$
**end for**
return $A$

% computes the best assignment for a set of vertices that are expected to run in parallel on all devices
**subroutine** `getBestAssign`$(vertices, A)$
$bestCost \leftarrow \infty$; $bestAssign \leftarrow$ `null`
% loop through all possible device assignments
**for** $D \in$ `allPerms`$(\mathcal{D})$ **do**
  $whichDev \leftarrow 0$; $cost \leftarrow 0$
  % loop through the ops created by sharding this meta-op
  **for** $v \in vertices$ **do**
    % loop through inputs to this op and add network cost
    **for** $v_1 \in \mathcal{V}$ s.t. $(v_1, v) \in \mathcal{E}$ **do**
      $cost \leftarrow cost +$ `getNetworkTime`$(v_1, a_{v_1}, D_{whichDev})$
    **end for**
    $whichDev \leftarrow whichDev + 1$
  **end for**
  **if** $cost < bestCost$ **then**
    $bestCost \leftarrow cost$
    $bestAssign \leftarrow D$
  **end if**
**end for**
% we have the best assignment for this meta op, so record it
$whichDev \leftarrow 0$
**for** $v \in vertices$ **do**
  $a_v \leftarrow D_{whichDev}$
  append $a_v$ to $A$
  $whichDev \leftarrow whichDev + 1$
**end for**
return $A$

# C Computation Graphs Used in the Experiments

## C.1 ChainMM

- **Input matrices**: A,B,C,D,E $\in \mathbb{R}^{10000\times10000}$
- **Neural Network Function**: $(A \times B) + (C \times (D \times E))$
- **Number of nodes in the graph**: 112

## C.2 FFNN

- **Input batch matrix**: $X \in \mathbb{R}^{2^{15}\times2^5}$
- **First layer weight matrix**: $W^{(1)} \in \mathbb{R}^{2^5\times2^{16}}$
- **First hidden layer bias vector**: $b^{(1)} \in \mathbb{R}^{2^{16}}$
- **Output layer weight matrix**: $W^{(2)} \in \mathbb{R}^{2^{16}\times2^5}$

- **Output layer bias vector**: $b^{(2)} \in \mathbb{R}^{2^5}$
- **Number of nodes in the graph**: 192
- $\mathrm{ReLU}(\cdot)$ denotes the element-wise rectified linear activation function.
- Softmax $(\cdot)$ denotes the softmax activation function applied over the output dimensions to obtain class probabilities.
- **Neural Network Function**:
  - **Hidden layer computation**: $H = \mathrm{ReLU}\left(X \cdot W^{(1)} + b^{(1)}\right)$ Here, $H \in \mathbb{R}^{2^{15} \times 2^{16}}$.
  - **Output layer computation**: $Y = \mathrm{Softmax}\left(H \cdot W^{(2)} + b^{(2)}\right)$ Here, $Y \in \mathbb{R}^{2^{15} \times 2^5}$ represents the output probabilities after applying softmax.

### C.3 LLAMA-BLOCK AND LLAMA-LAYER

Hyperparameters for llama structure include:

- **Number of parameters**: 7B
- **Max sequence length**: 4096
- **Embedding dimension**: 4096
- **Number of tokens**: 32000
- **Batch size**: 1
- **Number of layers**: 1
- **Number of nodes in the graph**: 215
- **Neural Network Function**: Figure 6 shows the structure of Llama represented in the computation graphs Llama-block and Llama-layer.

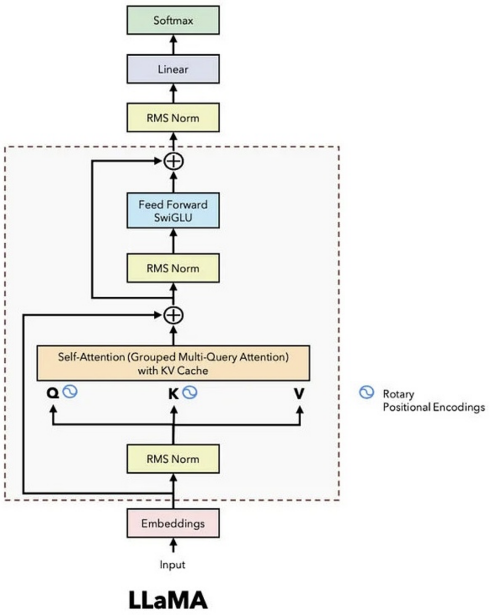


Figure 6: Llama-block and Llama-layer architecture. Figure from Yaadav (2024)

# D GRAPH FEATURES $X_{\mathcal{G}}$ AND DEVICE FEATURES $X_{\mathcal{D}}$.

Given a computation graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \{v_1, v_2, ..., v_n\}$ and $\mathcal{E} = \{e_1, e_2, ..., e_m\}$, we define the following:

- **Computation cost for $v$.** We use floating point operations per second of $v$ as the computation cost.
- **Communication cost for $e_{i,j} = (v_i, v_j)$.** The number of bytes for the output tensor of $v_i$ times a communication factor. In our case, we set the communication factor equal to 4. We benchmark the execution time of a simulator versus the real execution engine. We tried the values communication factor from 1 to 10 and found 4 to be the closest for the simulator with respect to the real execution engine.

## D.1 STATIC GRAPH FEATURES $X_{\mathcal{G}}$

The graph features matrix $X_{\mathcal{G}}$ is a $n \times 5$ matrix where each row contains the following five features:

- **Computation cost for $v_j$.**
- **Sum of communication cost from predecessor nodes to $v_j$.** The sum of communication cost for all $e_{i,j}$ such that $(v_i, v_j) \in \mathcal{E}$.
- **Sum of communication cost from $v_j$ to descendant nodes.** The sum of communication cost for all $e_{j,k}$ such that $(v_j, v_k) \in \mathcal{E}$.
- **t-level cost of $v_j$.** The sum of all computation costs and communication costs on a t-level path for $v_j$. A t-level path is defined in Section 4.2.
- **b-level cost of $v_j$.** The sum of all computation costs and communication costs on a b-level path for $v_j$. A b-level path is defined in Section 4.2.

## D.2 DYNAMIC DEVICE FEATURES $X_{\mathcal{D}}$ FOR DEVICE $d$ AT TIMESTEP $t$ GIVEN NODE $v$

The device features matrix $X_{\mathcal{D}}$ is a $|D| \times 5$ matrix where each row contains the following five features:

- **Total node computation cost.** Sum of the computation costs for all the nodes that have been assigned to device $d$ at timestep $t$.
- **Total predecessor nodes computation cost.** Sum of computation costs for all predecessor nodes of target node $v$ at timestep $t$ that are currently being assigned on device $d$.
- **Min start time of all input.** Earliest time to start execute a predecessor node of $v$ on each device $d$ at timestep $t$.
- **Max end time of all input.** Latest time for all predecessor nodes of $v$ to finish on each device $d$ at timestep $t$.
- **Earliest start time to compute $v$.** Earliest time for a device $d$ to finish receiving inputs on $d$ and start execute $v$.

# E More Device Assignment Analysis and Visualizations

## E.1 Computation node details

- **input:** input tensors
- **matmul:** matrix multiplications on two matrices
- **input elemwise.** elementwise operations(eg. ReLU) on an input tensor.
- **straight elemwise:** elementwise operations(eg. ReLU) with two inputs having the same dimensions.
- **bcast elemwise:** takes two inputs of different shapes (eg. a matrix and a vector) performing an elementwise operation(eg. ReLU) with one element of the vector on an entire row of the matrix.
- **max reduction:** reduce one dimension by finding max.
- **min reduction:** reduce one dimension by finding min.
- **sum reduction:** reduce one dimension by finding the sum along that dimension.
- **product reduction:** reduce one dimension by finding the product along that dimension.
- **formation:** a placeholder operation that forces aggregations in joinAgg groups to form a single tensor.
- **complexer:** a conversion between floats and complex tensors.
- **fill:** an operation to create tensors with all the same scalar, or to assign all lower or upper diagonal elements with provided scalar values.
- **squeezer:** adding or removing singleton dimensions of a tensor.
- **selec:.** an operation to copy a subset of several input tensors into an output tensor–a generalization of tensor subset and tensor concatenation.

### E.2 ASSIGNMENT PROFILING

This section examines the performance results from Table 2, where the DOPPLER algorithm achieved a lower runtime compared to CRITICAL PATH, PLACETO and the expert-designed ENUMERATIVEOPTIMIZER.

During the development of the ENUMERATIVEOPTIMIZER algorithm, it became clear that for meta-ops (described in Appendix A) containing many computations, the devices should be fully utilized and load-balanced. *Put more succinctly, it is expected that good assignments should minimize data transfers while maximizing computational resource utilization.*

**CHAINMM**: The assignments in Figure 7 for DOPPLER show all four devices used whereas for ENUMERATIVEOPTIMIZER, Figure 8, only two of the devices are used for the latter computations. The corresponding device utilization plots are shown in Figure 9 and Figure 10, respectively. It appears that, indeed, ENUMERATIVEOPTIMIZER does not fully utilize available compute resources towards the end of the computation. On the contrary, DOPPLER does well from the beginning to the end as shown in Figure 7.

**FFNN**: The assignments for FFNN with DOPPLER and PLACETO are shown in Figure **??**; the corresponding device utilization are shown in Figure 11 and Figure 12. In the DOPPLER assignments, subsequent vertices typically share the same device assignment. In turn, this should lead to a lower amount of data transfer. For the PLACETO assignments, however, subsequent vertices do not tend to have the same device assignment, possibly leading to a large amount of data transfer. In the device utilization figures, this is indeed borne out, where the PLACETO execution is three times slower, with most time spent on data transfers across GPUs.

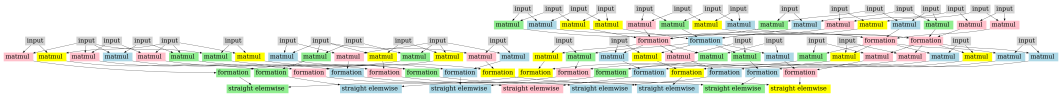

Figure 7: Assignment found by Doppler for ChainMM

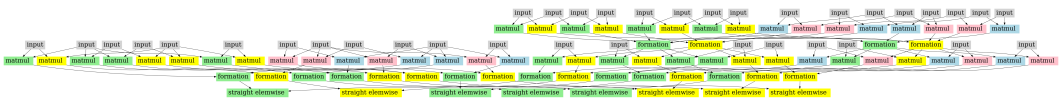

Figure 8: Assignment found by EnumerativeOptimizer for ChainMM

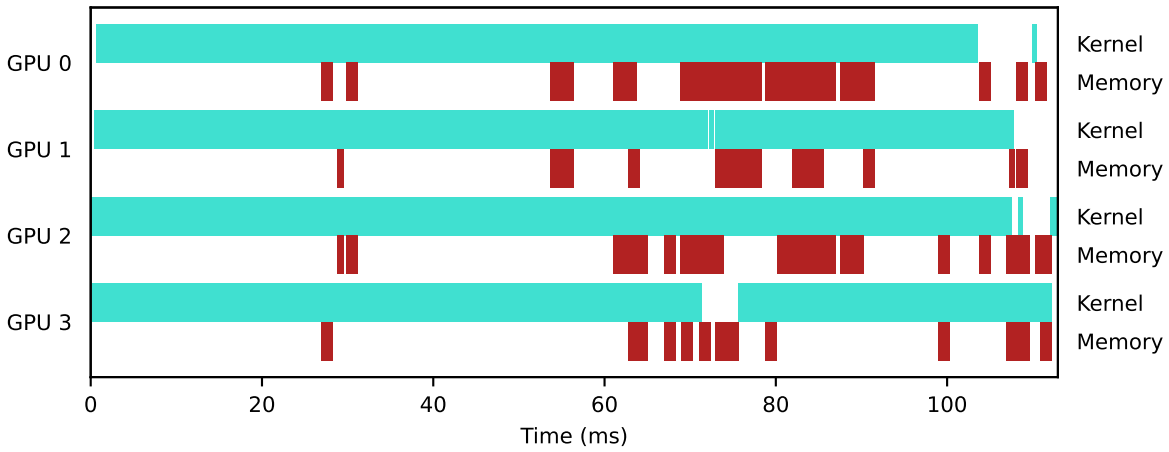


Figure 9: Device and transfer utilization for DOPPLER, CHAINMM.

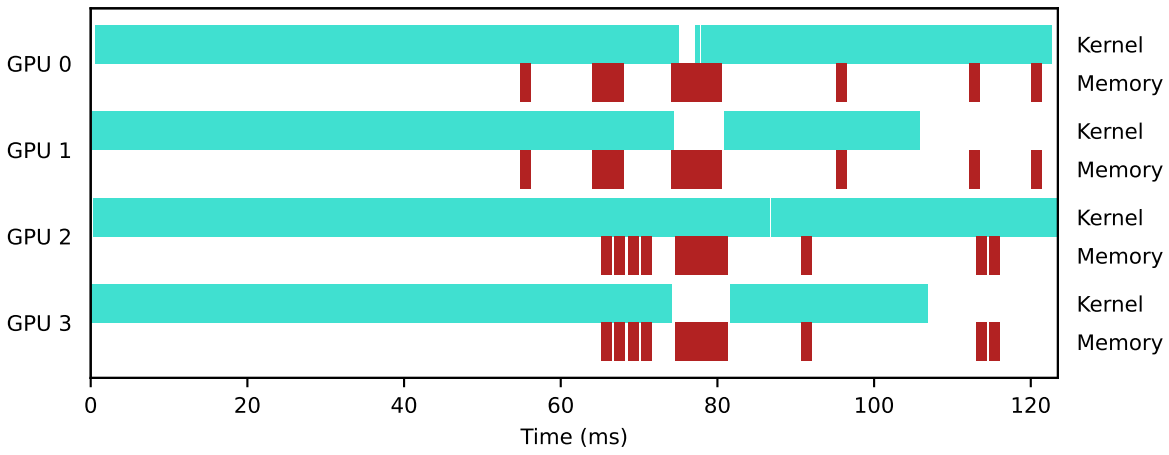


Figure 10: Device and transfer utilization for ENUMERATIVEOPTIMIZER, CHAINMM.

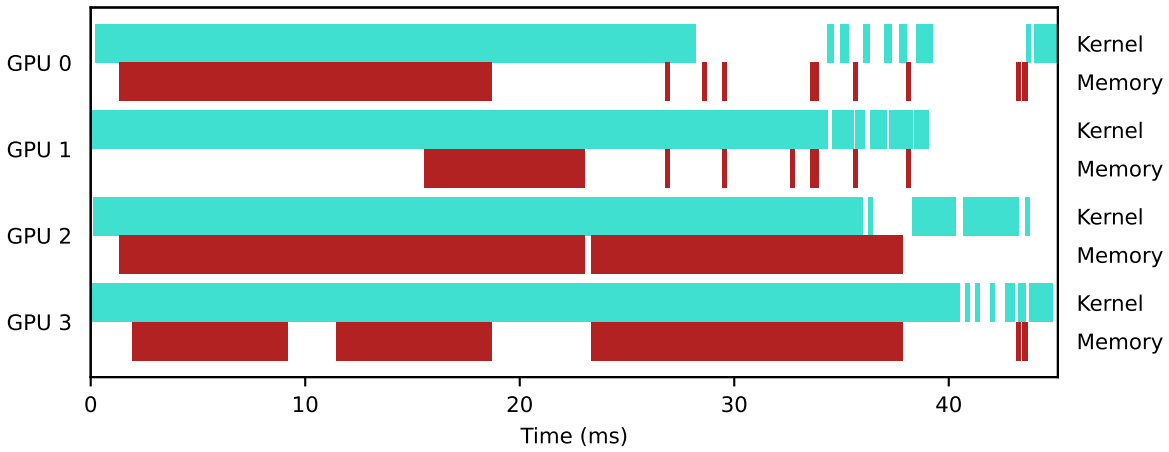


Figure 11: Device and transfer utilization for DOPPLER, FFNN.

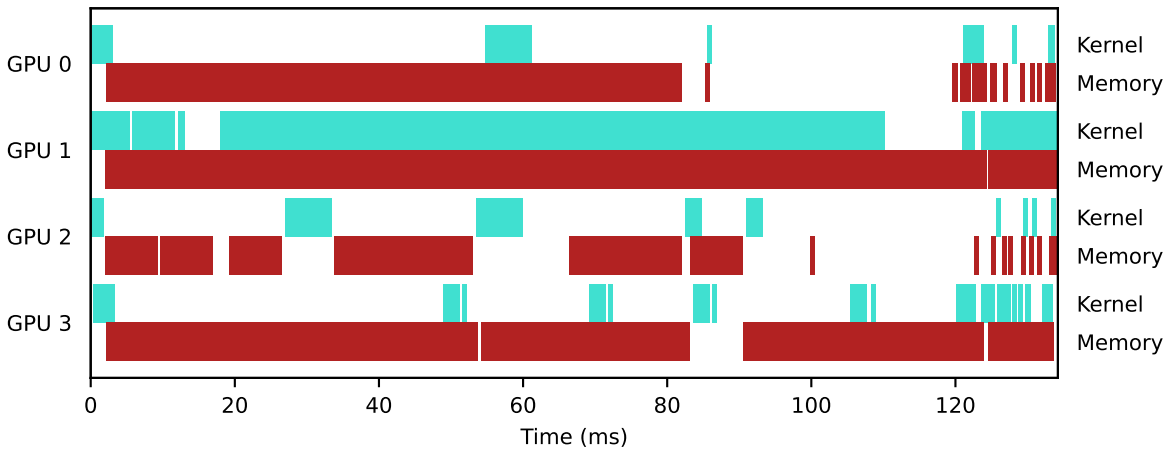


Figure 12: Device and transfer utilization for PLACETO, FFNN.

### E.3 LLAMA-BLOCK ASSIGNMENT ANALYSIS

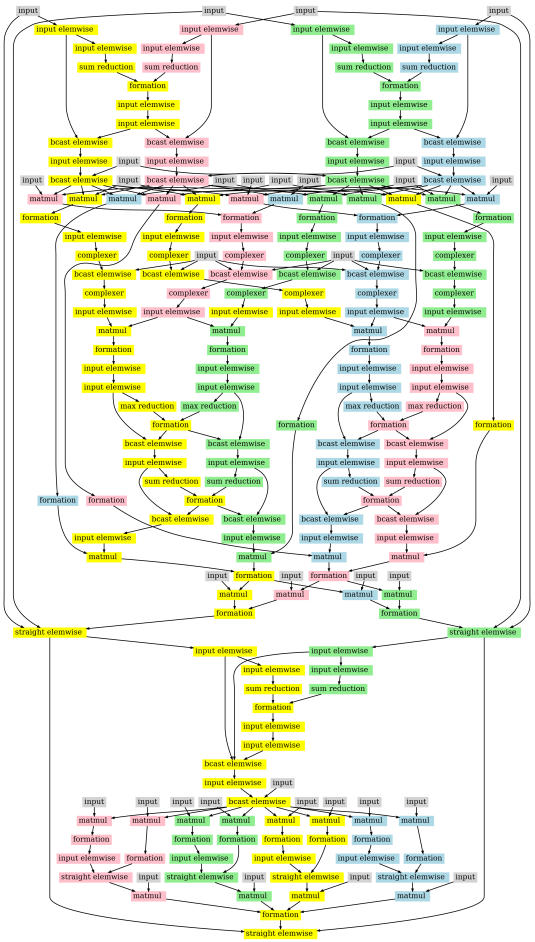

Figure 13: Assignment found by DOPPLER for LLAMA-BLOCK

The experiments demonstrate DOPPLER's capability to efficiently achieve load balancing in a distributed computing environment, particularly when multiple GPUs are utilized for computation. Across varied workloads, we observed that DOPPLER distributes workloads more evenly across available GPUs, ensuring that no single device is over-utilized while others remain under-utilized. Additionally, further analysis has provided deeper insights into DOPPLER's ability to make better assignment decisions, ensuring that computational tasks are assigned in a way that maximizes efficiency and minimizes stalling. This improves system performance and enables more effective resource utilization in distributed GPU computing environments as shown in Figure 13.

For instance, when executing a single LLAMA-BLOCK, PLACETO demonstrates minimal load balancing by assigning the majority of the computation to a single GPU while distributing only a small fraction of the workload to the remaining GPUs. As a result, one GPU becomes heavily burdened with the computation, while the other three GPUs remain largely idle for most of the execution process. This imbalance leads to inefficient resource utilization, causing the distribution computation to become nearly sequential. Consequently, PLACETO exhibits the slowest execution time among the tested approaches.

CRITICAL PATH makes a better attempt at load balancing by distributing the computational workload more evenly across multiple GPUs. However, its performance is negatively impacted by inefficient assignment decisions. Specifically, CRITICAL PATH does not take into account the previous data locations when determining where to assign computations. As a result, it frequently places a node's

computation on a GPU different from the one holding its input data. This misalignment introduces unnecessary data transfers between GPUs, as all input data must first be moved to the newly assigned GPU before computation can begin. The added communication overhead and the delays caused by data transfers contribute to a slowdown in execution time, ultimately reducing the overall efficiency of the system.

Both ENUMERATIVEOPTIMIZER and DOPPLER effectively mitigate the inefficiencies observed in CRITICAL PATH by ensuring that the consumer of an operation is assigned to one of the GPUs where the corresponding input data is already located. This strategy reduces unnecessary data transfers, leading to improved execution performance. However, ENUMERATIVEOPTIMIZER adopts a more conservative strategy by prioritizing data locality over load balancing. Its cost model assigns a high penalty to communication overhead, often favoring keeping computations and their subsequent consumers on the same GPU. While this minimizes data transfer costs, it sometimes results in an uneven workload distribution, with certain GPUs handling more tasks than others.

In contrast, DOPPLER demonstrates the most effective load-balancing strategy among all four approaches. As evidenced in the performance profile, after an initial stage of communication, DOPPLER achieves consistently higher GPU utilization throughout the majority of the execution process. By intelligently distributing workloads while considering both communication cost and GPU availability, DOPPLER maximizes efficiency, ensuring that all GPUs contribute effectively to the computation. This balanced approach allows DOPPLER to outperform other methods in terms of overall execution speed and resource utilization.

### E.4 LLAMA-LAYER ASSIGNMENT VISUALIZATIONS

We show the best assignment we found for four methods on the LLAMA-LAYER computation graph: CRITICAL PATH (Figure 17), PLACETO (Figure 16), ENUMERATIVEOPTIMIZER (Figure 14), DOPPLER (Figure 15), and GDP (Figure 18).

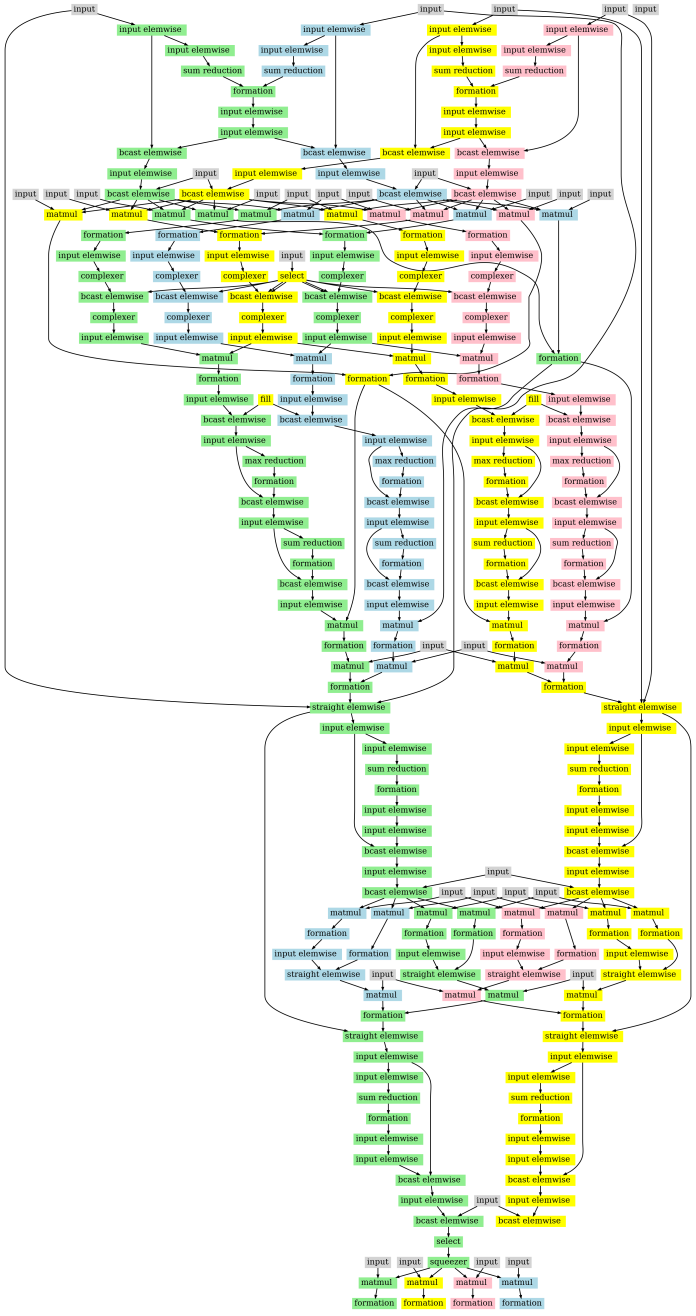

Figure 14: Assignment found by EnumerativeOptimizer for LLAMA-LAYER

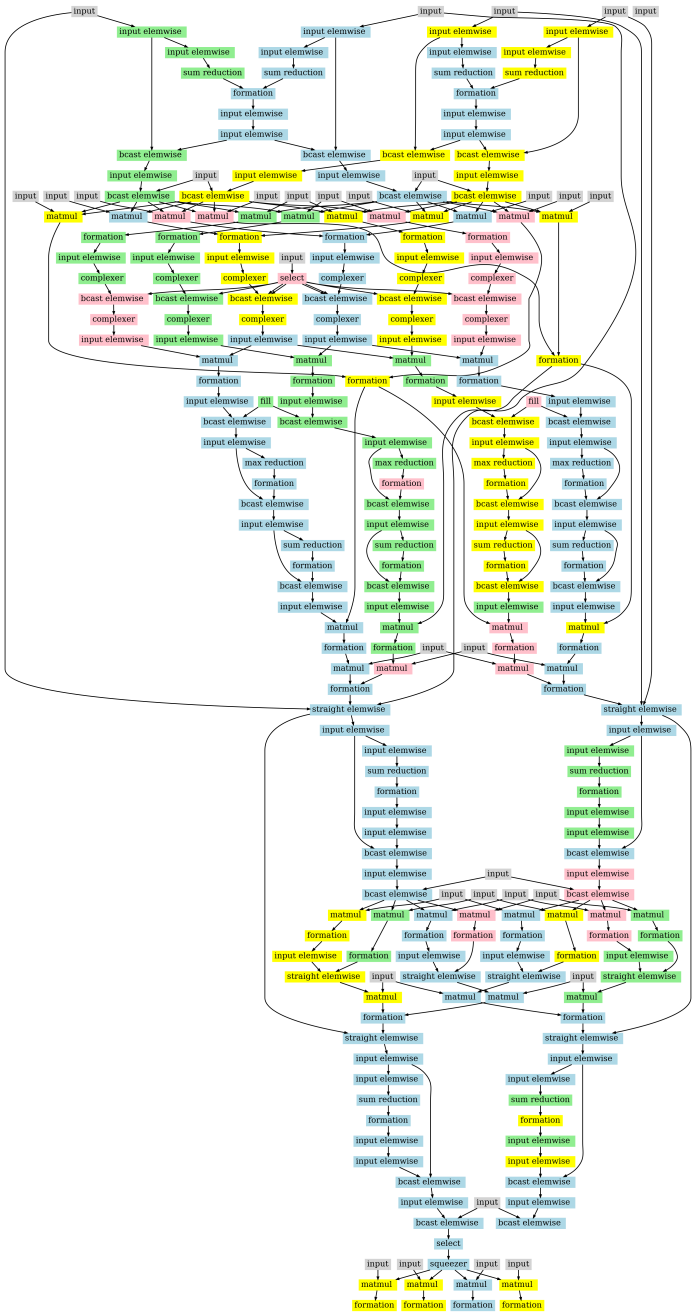

Figure 15: Assignment found by DOPPLER for LLAMA-LAYER

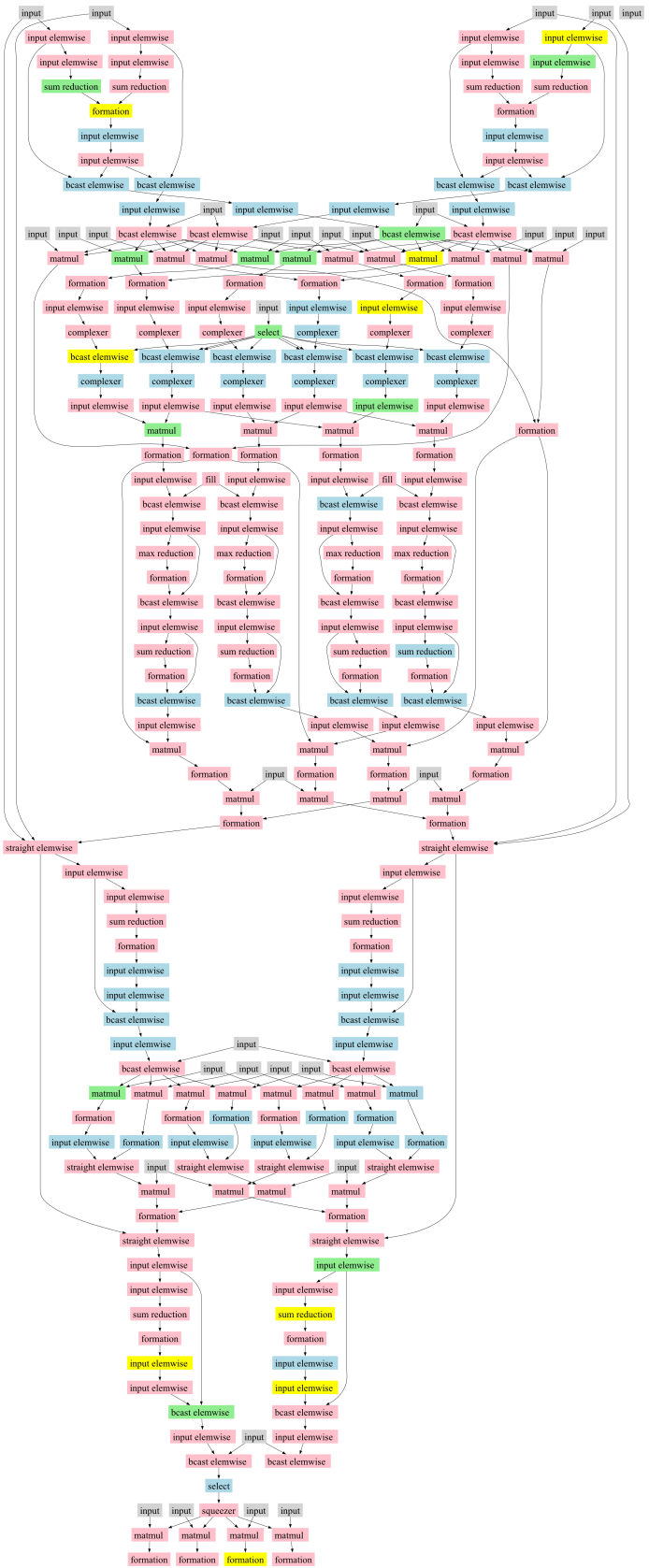

Figure 16: Assignment found by PLACETO for LLAMA-LAYER

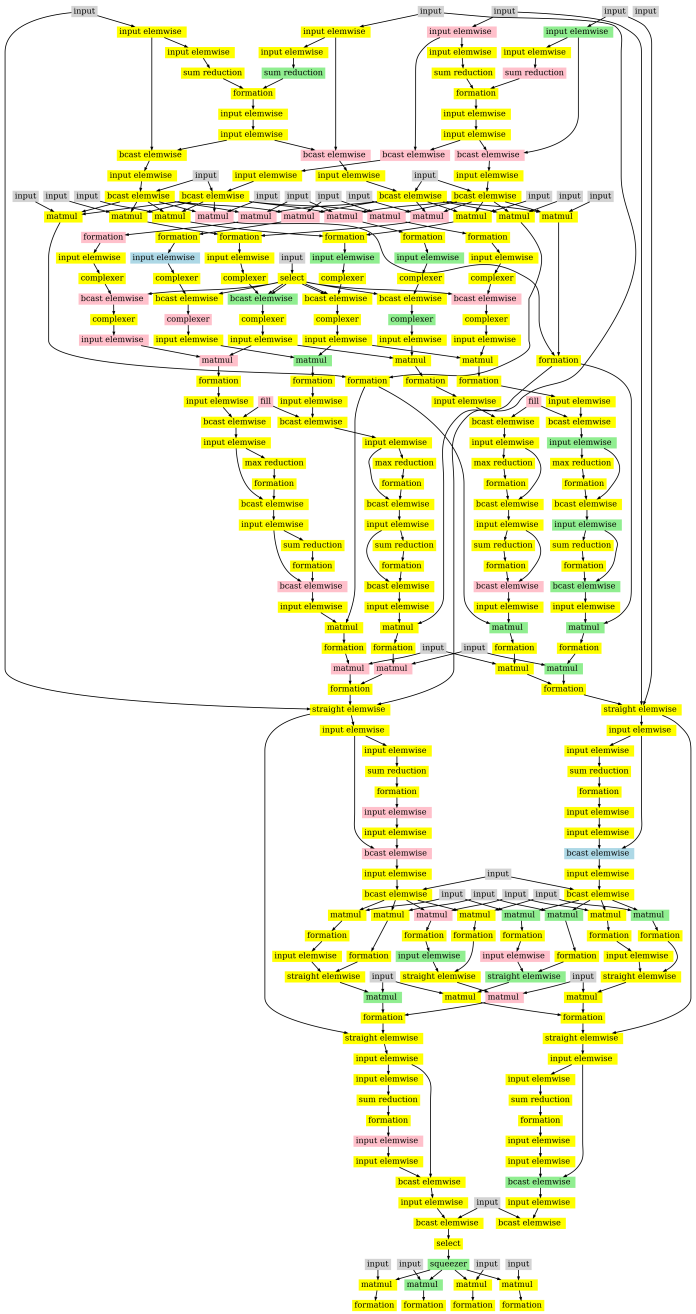

Figure 17: Assignment found by CRITICAL PATH for LLAMA-LAYER

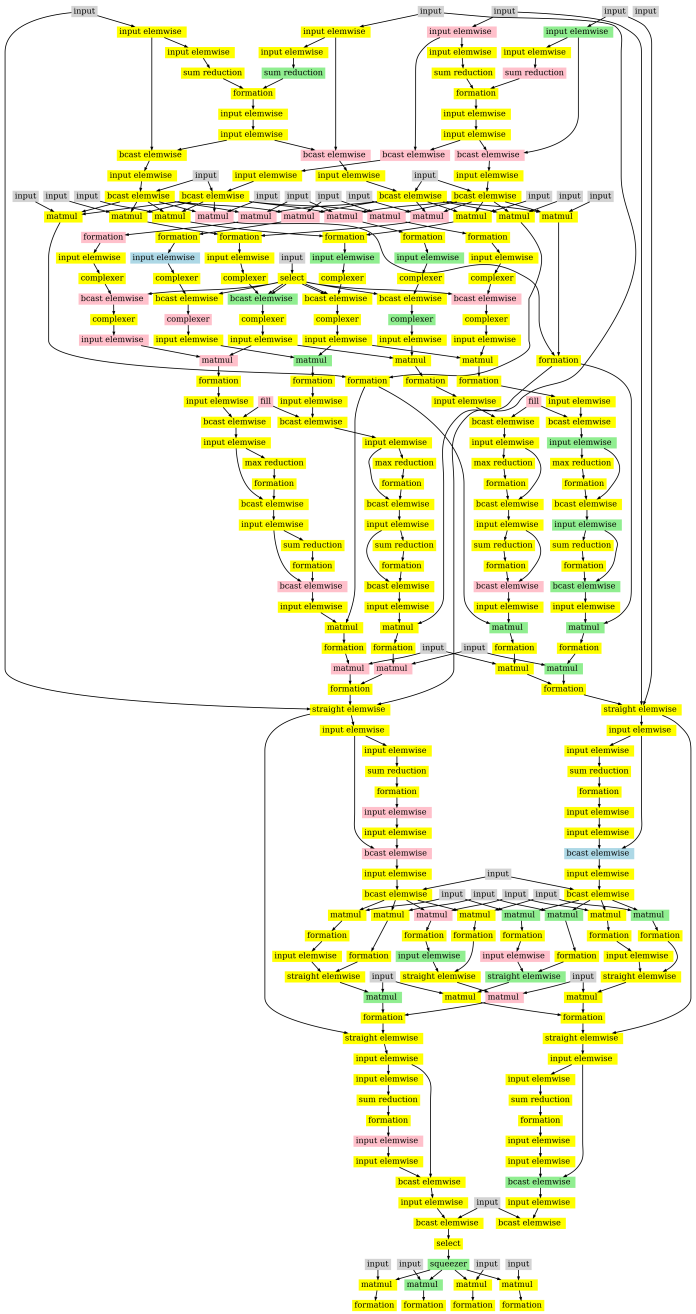

Figure 18: Assignment found by GDP for LLAMA-LAYER

## F SYNCHRONOUS SYSTEM CONFIGURATION

In Table 1, we run CHAINMM using ScalaPack and FFNN using Pytorch Lightning with tensor parallel on 4 NVIDIA Tesla P100 GPUs with 16GB memory each.

# G MORE ABLATION STUDIES

We conduct additional ablation studies on (1) the simulator implementation, (2) random seeds used in training dual policy, (3) the number of message-passing rounds per episode, and (4) the inclusion of pre-training stages on PLACETO. We aim to evaluate the performance and training efficiency of our three-stage training approach with dual policy design compared with alternatives.

## G.1 SIMULATOR ABLATION STUDIES

We compared the simulated and real system execution times for the same device assignments in the following plots. **We hypothesize that the simulator serves as a cost-effective way to approximate system execution times without sacrificing significant precision.** On the top in Figure 19, we show the running times for both the simulator and the real system for training the dual policy on `ChainMM` computation graph via imitation learning. We observe that the simulator tends to overestimate the running time compared to the real system running time and has some trouble differentiating between assignments with similar running times. However, the simulator provides approximate running times that follow the same trend as the real system (eg, high-quality assignments tend to have shorter running times, and low-quality ones exhibit longer running times). On the bottom in Figure 19, we show a statistical analysis comparing the performance of the simulator with the real system under the same setup. We find a Pearson coefficient of 0.79 and a Spearman coefficient of 0.69. This ablation study demonstrates that the simulator offers a cost-effective way to approximate system execution times.

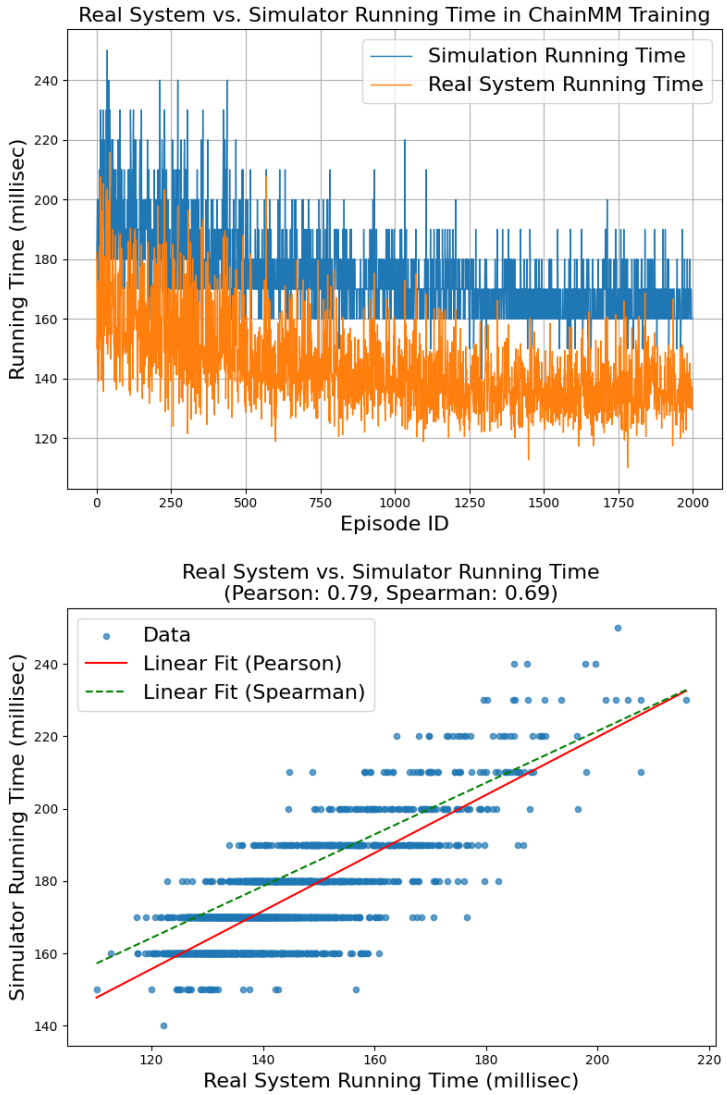


Figure 19: (left) A line chart showing a comparison between the simulator running time and the real system running time throughout training. (right) A scatter plot showing the simulator running time versus the real system running time with Pearson and Spearman fitting lines.

### G.2 Experiment with Random Seeds

In this study, we aim to test **the hypothesis that the best assignment found by our approach DOPPLER remains consistent across different random seeds during training**. Due to the cost of training, we are unable to run experiments multiple times with random seeds for all computational graphs and methods. Therefore, we conduct five training runs of DOPPLER-SYS with different seeds on `ChainMM` computation graph to test this hypothesis. This experimental setup–including computation graph, dual policy architectures, and training hyperparameters–is identical across runs, differing only in the random seeds. For each training run, we evaluate the best-found assignment over 10 system executions and report the mean and standard deviation in Table 5. The results show that DOPPLER achieves consistent performance across different random seeds.

| Model | Run1 | Run2 | Run3 | Run4 | Run5 |
|---|---|---|---|---|---|
| ChainMM | $123.2 \pm 3.7$ | $119.6 \pm 2.2$ | $122.7 \pm 2.1$ | $123.9 \pm 2.5$ | $121.7 \pm 0.9$ |

Table 5: Experiment running DOPPLER using different random seeds on CHAINMM computation graph. We test the best assignment found at the end of the training across different seeds with 10 system runs and report the mean and standard deviation of the system running time (in milliseconds) for each assignment.

### G.3 MESSAGE-PASSING ABLATION STUDIES

To enhance training efficiency for large computation graphs, we apply a message-passing round on the graph once per MDP episode instead of once per MDP step. For applying message-passing once per MDP step, this means that the number of message-passing rounds per episode equals the number of nodes in the graph. **We hypothesize that this modification has a negligible impact on DOPPLER's convergence but significantly reduces training time–proportional to the number of nodes in the computation graph.** We conduct this ablation study on the `ChainMM` computation graph using the simulator to save time, since per-step message-passing incurs prohibitively high training costs.

| | DOPPLER-SIM | DOPPLER-SIM-mpnn-per-step |
|---|---|---|
| Best assignment | $122.5 \pm 4.0$ | $121.7 \pm 3.2$ |
| Number of episodes | 3425 | 963 |
| Number of message passing | 3425 | 107,856 |
| Run time reduction | 0.7% | |
| Extra message-passing | 3049.1% | |

Table 6: Running time (in milliseconds) for the best device assignment found at the end, along with the number of message-passing steps conducted until finding the best assignment. The reported time for the best assignment includes both the average and standard deviation of the system running time over 10 system rounds. DOPPLER-SIM refers to performing message passing on the computation graph per MDP episode, while DOPPLER-SIM-mpnn-per-step denotes conducting message-passing per MDP step within each episode.

In each MDP step within an episode, we assign a device to the currently selected node. Therefore, for the DOPPLER-SIM-mpnn-per-step approach, the number of message-passing rounds per episode equals the number of nodes in the computation graph. The `ChainMM` computation graph consists of 112 nodes. Table 6 shows that the best assignments–reported in the first row by their running times–were found at episode 3425 for DOPPLER-SIM and at episode 963 for DOPPLER-SIM-mpnn-per-step. Although the best assignment was found in fewer episodes with DOPPLER-SIM-mpnn-per-step, completing 963 episodes took significantly more wall-clock time than 3425 episodes for DOPPLER-SIM, because message-passing took the majority of time during training.

We evaluate the efficiency of the two approaches based on the number of message-passing rounds required to find the best assignment. DOPPLER-SIM performs 3425 message-passing operations (one per episode), while DOPPLER-SIM-mpnn-per-step conducts 963 episodes × 112 nodes = 107,856 message-passing operations (one per MDP step, or equivalently, per node in the computation graph). DOPPLER-SIM-mpnn-per-step achieves a 0.7% reduction in runtime for the best assignment compared to DOPPLER-SIM, but at the cost of 3049.1% more message-passing. Therefore, these results support our hypothesis that the modified approach has greatly reduced the training time with negligible impact on the performance.

### G.4 PLACETO ABLATION STUDIES

We conduct an ablation study on policy design for learning device assignments in both DOPPLER and PLACETO, explicitly including pre-training stages for PLACETO to isolate the effect of the underlying policy design from the benefits of pre-training. **We hypothesize that the dual policy design in DOPPLER outperforms PLACETO regardless of the inclusion of the training stages.** To test our hypothesis, we pre-train PLACETO policy using imitation learning and compare it with DOPPLER-SIM, which is trained using the imitation learning stage (Stage I) and the simulation-based RL stage (Stage II).

| | PLACETO-pretrain | PLACETO | DOPPLER-SIM | DOPPLER-SYS |
|---|---|---|---|---|
| Best Assignment | $99.0 \pm 5.7$ | $126.3 \pm 5.8$ | $49.9 \pm 1.1$ | $47.4 \pm 0.7$ |

Table 7: The mean and standard deviation of the running time (in milliseconds) for the best assignment found for DOPPLER, compared to PLACETO and its pre-training version, PLACETO-pretrain, over 10 system runs. The results indicate that even with the pre-training stage, PLACETO-pretrain performs worse than DOPPLER-SIM.

Table 7 shows that DOPPLER discovers more effective device assignments than PLACETO. This result isolates the impact of the training stages and support the claim that DOPPLER's dual policy design outperforms the policy design in PLACETO.

## H DOPPLER'S EXPERIMENTS ON DIFFERENT HARDWARE CONFIGURATIONS

The experiments so far demonstrate that DOPPLER outperforms the alternatives on four Tesla P100 GPUs, each with 16GB of memory. We hypothesize that DOPPLER could find better device assignments than alternatives across different hardware configurations. In this Section, we conduct experiments with DOPPLER and other methods under 1) varying GPU memory size on four P100 GPUs, and 2) different numbers and types of GPUs. Each experimental setup is described in the following two subsections.

### H.1 EXPERIMENTS WITH RESTRICTED GPU MEMORY

**We aim to test the hypothesis that DOPPLER can adapt to the hardware setups with restricted GPU memory.** Table 8 shows results on four NVIDIA P100 GPUs, each restricted to use 8 GB out of their 16 GB total memory. DOPPLER learns to adapt to memory constraints, achieving up to 49.6% and 18.6% runtime reductions compared to the best baseline and ENUMERATIVEOPTIMIZER, respectively, while heuristics fail to adapt due to dynamic memory allocations in WC systems. These results confirm that DOPPLER can adapt to restricted memory settings.

| | | 4 GPUs | | | | RUNTIME REDUCTION | |
|---|---|---|---|---|---|---|---|
| MODEL | 1 GPU | CRITICAL PATH | PLACETO | ENUMOPT. | DOPPLER-SYS | BASELINE | ENUMOPT. |
| CHAINMM | $439.8 \pm 4.6$ | $310 \pm 4.9$ | $243.5 \pm 5.9$ | $133.5 \pm 10.4$ | **122.6** $\pm 2.2$ | 49.6% | 8.2% |
| FFNN | $148.2 \pm 19.4$ | $225.8 \pm 19.4$ | $126.8 \pm 5.7$ | $49.2 \pm 0.9$ | **46.0** $\pm 0.8$ | 63.7% | 6.5% |
| LLAMA-BLOCK | $465.1 \pm 7.8$ | $216.8 \pm 4.6$ | $433.5 \pm 6.2$ | $233.8 \pm 8.1$ | **190.2** $\pm 11.2$ | 12.3% | 18.6% |
| LLAMA-LAYER | $482.6 \pm 9.4$ | $292.5 \pm 5.1$ | $302.1 \pm 20.2$ | $172.8 \pm 4.3$ | **154.0** $\pm 3.7$ | 47.4% | 10.9% |

Table 8: Real engine execution times (in milliseconds) for assignments identified by our approaches (EnumerativeOptimizer, Doppler-SYS) using 4 GPUs with access to 8G out of 16G GPU memory for each GPU compared against those produced using 1 GPU and by two baselines (CRITICAL PATH and PLACETO). The results show that Doppler-SYS outperforms all baselines across all settings.

### H.2 EXPERIMENTS ON DIFFERENT NUMBERS AND TYPES OF GPUS

**We aim to test the hypothesis that DOPPLER can find better device assignments regardless of the number and type of GPUs in the server.** Table 9 presents results running various computation

graph architectures on eight NVIDIA V100 GPUs, each with 32 GB of memory. The setup consists of two device meshes, each fully interconnected via NVLinks, with a total of four additional NVLinks spanning between the meshes. We observed that DOPPLER effectively leverages NVLink to minimize inter-mesh data transfer. We show that DOPPLER achieves up to 67.7% runtime reduction compared to the existing baseline, and up to 19.3% compared to ENUMERATIVEOPTIMIZER.

| | | 8 GPUs | | | RUNTIME REDUCTION | |
|---|---|---|---|---|---|---|
| MODEL | 1 GPU | CRITICAL PATH | ENUMOPT. | DOPPLER-SYS | BASELINE | ENUMOPT. |
| CHAINMM | $83.5 \pm 4.1$ | $69.6 \pm 2.6$ | $33.5 \pm 2.5$ | **32.1** $\pm 0.7$ | 53.9% | 4.1% |
| FFNN | $51.4 \pm 1.8$ | $50.0 \pm 6.0$ | $20.0 \pm 2.6$ | **16.2** $\pm 2.3$ | 67.7% | 19.3% |
| LLAMA-BLOCK | $154.4 \pm 6.3$ | $117.6 \pm 6.0$ | **109.6** $\pm 4.2$ | $109.7 \pm 3.0$ | 6.7% | -0.1% |
| LLAMA-LAYER | $105.0 \pm 4.8$ | $105.4 \pm 4.2$ | $97.5 \pm 1.1$ | **90.6** $\pm 4.1$ | 13.7% | 7.1% |

Table 9: Real engine execution times (in milliseconds) for assignments identified by our approaches (EnumerativeOptimizer, Doppler-SYS) using 8 V100 GPUs compared against those produced using 1 GPU and by one baseline (CRITICAL PATH). The results show that Doppler-SYS outperforms the alternatives in 3 out of 4 settings.

# I DETAILS ON THE THREE TRAINING STAGES

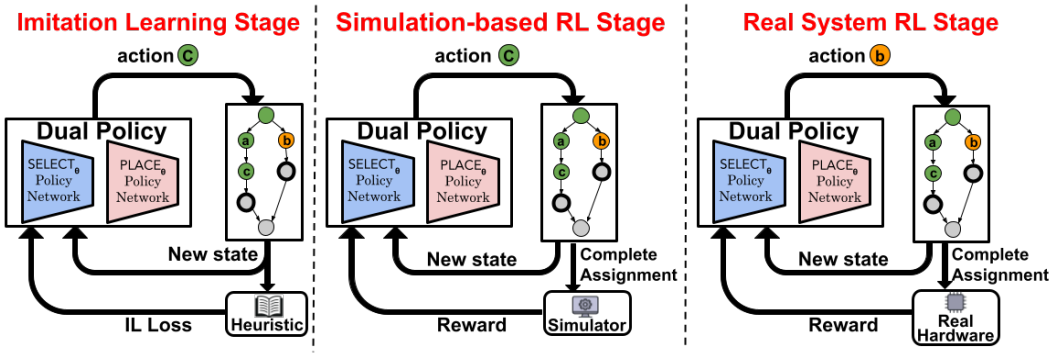


Figure 20: Three-stage framework for training DOOPLER cost-effectively by combining imitation learning, simulation-based RL, and real-system RL.

Figure 20 illustrates the workflow of the three training stages: imitation learning stage, simulation-based RL stage, and real system RL stage. Dual policy is trained sequentially through these three stages.

During the imitation learning stage, the computation graph produces a new state at each step, which describes the current assignment in the graph. This state is provided to both the dual policy and the heuristic method. Dual policy then produces an action consisting of a node and its corresponding device. The computation graph applies the action from the dual policy and transitions to a new state. At each MDP step, a loss is computed based on the action taken by the heuristic. The cumulative loss over the entire episode is then used to update the dual policy at the end of each episode.

During the simulation-based RL stage, the heuristic is replaced by a simulator that provides a reward based on the computation graph with a complete device assignment. The simulator is invoked only at the terminal step, once all nodes in the computation graph have been assigned devices. At this point, the simulator executes the computation graph with the full assignment, following Algorithm 1, and returns a reward to update the dual policy.

In the real system RL stage, the simulator is replaced by the real hardware which executes the computation graph directly–without simulated data–following Algorithm 1. In this stage, the real hardware actively serves real-world user requests. The reward is derived from the observed execution time of the assignment produced by the dual policy.

## J DISCUSSION ON SCALING TO MUCH LARGER DATAFLOW GRAPHS

With respect to much larger graphs, we expect the linear scale-up to continue, but in practice, massive graphs are likely not to be an issue. While we cannot know what companies such as OpenAI are doing, in practice, inference is probably not run on more than a few dozen GPUs (graphs grow linearly in size with increasing GPU counts). Training, while it requires thousands of GPUs, is made by relatively independent access machines through data parallelism and pipelining—each machine gets an identical transformer/MoE layer and a subset of the data—so it is running the same computation as all of the other machines. In this case, one would likely not train a single massive graph. Rather, each repeated block or layer would be used to train a separate, dual policy in parallel and given the same assignment on each machine with repeated structure throughout the cluster (assuming uniform hardware), with the runtimes collected across the cluster would be used to compute a reward.

## K DOPPLER'S TRANSFER ABILITY ACROSS HARDWARE AND DETAILED ANALYSIS

We conducted a transfer learning study that trains DOPPLER on a computation graph with four P100 GPUs of fully NVLink connectivity and generalizes to eight GPUs with partial NVLink connectivity, organized into two groups of four GPUs (GPU 0–3 and GPU 4–7). Each group is fully connected via NVLink, and two GPU groups have in total of four NVLink connections linking to the opposite group. Due to this hierarchical network topology, the communication cost within a 4-GPU group differs substantially from the cost within groups and across groups.

In the assignment found by DOPPLER, we measured how the number and ratio in parentheses of data transfers across two 4-GPU groups, within the same 4-GPU group, and within the same single GPU changes from Zero-shot to 2K-shot for FFNN on eight GPUs in the following table:

| | ACROSS GROUPS | SAME GROUP | SAME GPU |
|---|---|---|---|
| ZERO-SHOT | 22 (10.6%) | 14 (6.7%) | 172 (82.7 %) |
| 2K EPISODES | 7 (3.4%) | 4 (1.9%) | 197 (94.7 %) |

The following table shows the real engine execution times (in milliseconds) for the assignment identified by DOPPLER under different few-shot settings (Zero-shot, 2K-shot) compared against the baseline for the transfer learning study above:

| COMPUTE GRAPH | TRAIN HARDWARE | TARGET HARDWARE | ZERO-SHOT | 2K-SHOT | DOPPLER-SYS | CRIT. PATH | ENUM. OPT. |
|---|---|---|---|---|---|---|---|
| CHAINMM | 4 P100 GPUS | 8 V100 GPUS | 59.2 (1.9) | 26.0 (0.5) | 32.1 (0.7) | 69.6 (2.6) | 33.5 (2.5) |
| FFNN | 4 P100 GPUS | 8 V100 GPUS | 23.1 (2.3) | 14.4 (2.5) | 16.2 (2.3) | 50.0 (6.0) | 20.0 (2.6) |

After 2K episodes, DOPPLER finds better assignments for both ChainMM and FFNN compared to training solely on eight GPUs without generalization (8K episodes) for runtime reduction of 19.0% (ChainMM) and 11.1% (FFNN).

## L CODE

Anonymous GitHub Repo Link: https://anonymous.4open.science/r/Doppler-EA7D/