We thank the reviewers for their careful reading and insightful comments. We are encouraged that they all appreciated the relevance, significance, clarity, and demonstrated efficiency of our solution. Reviewers 1 and 4 recognize the novelty of our algorithms; Reviewer 4 points out the generality of our framework. Below we address one general point and some specific questions, but will incorporate all feedback in the final version.

General comment on scalability. We evaluated our algorithms on several modern DNN workloads; our algorithms are efficient for all graphs that we tried, which are of size up to 2012 nodes. In addition, after the submission we have discovered a new heuristic that allows one to heavily restrict the search space by linearizing the input graph (essentially adding an auxiliary Hamiltonian path found using a DFS-based topological sort). When this is done, the number of ideals becomes just the number of nodes (plus one), and the Dynamic Programming approach is *guaranteed* to handle graphs of size, say, **50K nodes within minutes**. Strikingly, we have found that with this restriction of the search space, our algorithms *still find the optimal split* for all of the tested workloads! We will add this in the final version.

Looking towards the future and even larger graphs, one can take advantage of the repetitive layer structure of modern (e.g. Transformer-based) models to further shrink the search space.

Reviewer 1. *Number of nodes in the graphs seems to be quite low (200 for GNMT). GDP reports several tens of thousands of nodes? Is there some manual grouping operation performed on the computational graph? Does a "node" correspond to an op run on a device?* To showcase the flexibility of our approach, we were able to obtain computation graphs of many state-of-the-art models, either on the operator granularity level (where each node corresponds to an operation) or on the layer granularity level (where each node corresponds to a DNN layer / a PyTorch module). Our solvers and cost model are agnostic to the granularity used. Our operator-level graphs are obtained from ONNX Runtime, which does not support GNMT. Instead, we obtain GNMT from PyTorch, which yields coarser layer-level graphs. Note that GDP runs experiments on GNMT-8 (operator graph), whereas we do so for GNMT-**16** (layer graph).

Few more details on how the graphs are pre-processed could be helpful. Details about the graph preprocessing are provided in Appendix E.1 (in the supplementary material).

Evaluation does not compare with other learning based approaches (...) We compare directly to past work that uses a cost model and employs optimization algorithms to find a split statically before the execution. We also provide a qualitative comparison to the Reinforcement Learning approaches in Section 2. In short, they are online methods that treat the objective function as a black box; while no profiling is needed to build a cost model, this necessitates lots of retraining to evaluate various placements. In contrast, our static technique does not have runtime overhead or require runtime hardware execution changes, which could be very costly on some hardware platforms. Further, our methods find *provably optimal* splits, whereas the RL ones are mostly heuristic, without optimality or approximation guarantees.

This approach makes use of a fairly sophisticated cost model. How hard is this to build for different configurations? It is not difficult and in fact quite practical to build an instance of our cost model (see Section 3), as one need only know (for every node) the CPU/accelerator processing time, memory usage, and output activation sizes; for the network we need to know the bandwidth; finally, the number of accelerators and available memory. Most of these parameters can be automatically profiled, measured or estimated. Profiling has been shown to be effective and accurate, as well as practical, for example in PipeDream (which runs profiling and builds a cost model automatically).

What if we would like to extend this to devices across multiple machines? Our framework is flexible and readily extends to scenarios exactly like the one you mentioned (hierarchical interconnect topologies); see Appendix H.3.

Reviewer 2. *The algorithms are lack of novelty (...) somewhat trivial (...) the algorithm just put different mini-batches on different devices and performs feed-forward and back-forward propagation.* We believe that would be data parallelism; our work in fact addresses the very different notion of [pipelined] *model* parallelism, where the DNN operators are partitioned onto different devices. We would also like to stress that the focus and the contribution of our paper are not the pipelining schemes (due to e.g. GPipe and PipeDream), but novel algorithms that *determine which operators/layers of the DNN should be assigned to which machines*. These algorithms are based on the sound algorithmic principles of Dynamic Programming and Integer Programming and are not trivial. The other reviewers acknowledge their novelty.

The only reason why we discuss pipelining schemes is to argue that our cost model (and the combinatorial optimization problem that we are solving) indeed corresponds precisely to the average time-per-sample that arises when running inference or training using these pipelining schemes. The combinatorial optimization problem that we are solving is also anything but trivial. It is NP-hard for multiple reasons, and its difficulty is further reflected in the performance of multiple baselines that we tested (both simple and more complex ones).

I am confused about Figure 3(a). Why the time of sample on Device 2 takes much more time compared the sample on Device 1? The different devices hold disjoint parts of the computation graph, which is partitioned among them. This figure corresponds to an example partitioning with skewed (higher) load on one device (i.e., the time it takes to execute, for one minibatch, those parts of the DNN computation graph that have been placed on that device). We intentionally show a sub-optimal split to highlight that maximum load is indeed the crucial factor that determines the throughput of the system when pipelining is used (being equal to the average time-per-sample – see Sections 5.1–5.3).