SystemX: A Scalable and Optimized Data System for Large Multi-Model Deep Learning

ABSTRACT

In many deep learning (DL) applications, the desire for higher accuracy has led to a marked increase in the popularity of very large model architectures, especially on unstructured data. Thus, the memory capacity of GPUs is now a bottleneck in DL. Training larger-than-GPU-memory models is a vexed process today, forcing DL users to get multiple GPUs to partition a large model. Such "model-parallel" approaches actually offer little to no speedups because of inherent sequential dependencies within a neural network's computations. Often they under-utilize GPUs massively and have long runtimes. In this work, we take a fresh database-inspired view of this problem by decoupling scalability from parallelism. We devise a set of techniques–some novel and some adapted from classical RDBMS ideas–to enable seamless training of very large DL models in both single-GPU and multi-GPU cases. This includes automated model partitioning and spilling to DRAM. We then exploit a hitherto unexplored avenue for parallelism in this context, namely, multi-model execution such as during model selection, to devise a novel hybrid parallelism approach combining model- and task-parallelism to raise efficiency. We formalize our overall scheduling problem, study alternative algorithms using simulations, and devise a simple but effective greedy scheduling heuristic. We prototype our ideas into a system we call SystemX on top of PyTorch. Experiments with real benchmark large-model DL workloads show that SystemX is over 7x faster than regular model parallelism and over 50% faster than state-of-the-art industrial tools for pipeline parallelism, while enabling seamless scalability and parallelism.

1 INTRODUCTION

Deep learning (DL) has revolutionized predictive analytics on unstructured data. Its high-profile successes at Web giants has led to growing adoption of state-of-the-art DL at more Web firms, enterprises, domain sciences, and even digital humanities. A major recent change in DL is the popularity of ever larger neural architectures such as Transformers. Natural language processing (NLP) is now awash with multi-billion parameter models such as BERT-Large [4], GPT-3 [2], and Megatron-LM [31]. Moreover, transfer learning by fine-tuning such pre-trained public models is often good enough instead of training from scratch, e.g., as seen in the popularity of HuggingFace [34]. Interest in such large models is also growing in computer vision (e.g., [5]) and for tasks bridging relational data and NLP [35]. *Unfortunately, GPU memory capacity is trailing DL model sizes, making this a new systems bottleneck for DL users* [32].

Example. Consider a data scientist at a small Web company training a text classifier to monitor reviews. They download a state-of-the-art BERT-Large model for this. They want to finetune the whole model on their (likely smaller) application-specific labeled dataset. However, the GPUs they have do not have enough memory to hold the full model, causing PyTorch or TensorFlow to crash and impeding their application.

Existing Approaches and Drawbacks. There is a recent flurry of work in the ML systems world aimed at mitigating the above bottleneck. The primary paradigm is so-called "model parallelism" [1]. It works as follows. *Partition* the model and places the "shards" across *multiple* devices (GPUs), as Figure 1(A) illustrates. Such partitioning is typically done in a layer-wise way. During execution, intermediates between model shards are exchanged between GPUs to emulate regular training as if on a single device. Alas, the majority of DL models (especially large models) have *sequential dependencies* inside their architecture, which means only one device is fully active at a time. This leads to massive GPU under-utilization and no speedups from adding GPUs. Besides, it forces users to get multiple GPUs in the first place, which may not be available or not affordable (even in pay-as-you-go clouds) for many kinds of DL users.

Two recent work that improve upon regular model parallelism are FlexFlow [14], which hybridizes model- and data parallelism, and "pipeline parallelism" [10, 11]. But FlexFlow does not work easily out of the box for larger-than-GPU-memory training and was designed more for just getting extra parallelism. It could shard a model in more complex ways, e.g., width-wise or even within a layer. But it creates a tradeoff between model parallelism and data parallelism, which might necessitate even more devices because data parallelism copies shard across multiple devices. In this sense, FlexFlow's goals are not aligned with our core problem.

Finally, pipeline parallelism exploits the fact that DL training uses mini-batch stochastic gradient descent (SGD) wherein subsequent mini-batches are independent samples of the training dataset. So, it stages out successive mini-batches across devices to reduce idling, as Figure 1(B) illustrates. This is indeed quite effective in practice but it still leaves a lot on the table due to "bubbles" in the pipelining, which still under-utilizes all GPU devices.

Observations and System Desiderata. Overall, we observe two issues with prior art. First, *they conflate scalability with parallelism.* Parallelism is not a fix for scalability but complementary. One must address the scalability bottleneck from first principles. Second, they all fail to recognize or exploit a key source of higher degree of parallelism in DL workloads: training multiple models in one go, e.g., during model selection such as hyperparameter tuning or neural architecture engineering [15]. Based on these observations, we aim for a new approach and system for large-scale multi-model DL training with the following desiderata.

- **Out-of-the-box Model Scalability.** We desire an approach that can scale to very large models even if the user has only one GPU. It should not force users to get multiple GPUs. This requires a more holistic and principled use of the *memory hierarchy*, akin to relational DBMS design [27].
- **Efficient Multi-Model Parallelism.** Given the ubiquity of needing to train multiple models, we desire an approach that exploits this degree of parallelism more effectively to

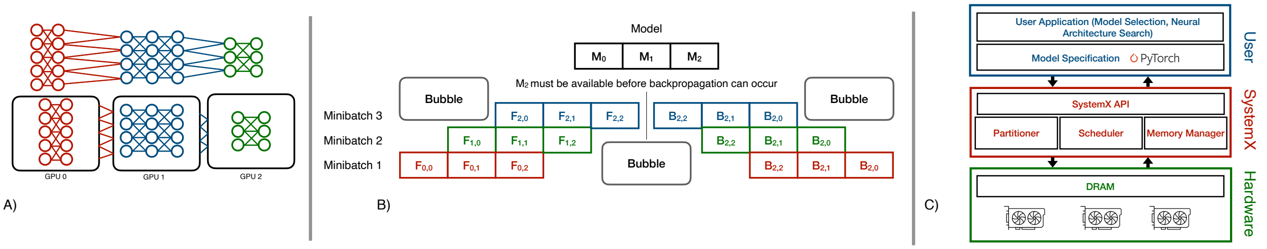


Figure 1: A) Traditional model parallel execution on a feedforward architecture. B) Illustration of pipeline parallelism of prior art on three devices. A model M is partitioned into shards M_i. A mini-batch is split into micro-batches, defined by the compute stages of each device on the minibatch. $F_{i,j}$ is the forward pass of shard j on micro-batch i; likewise $B_{i,j}$. Note how data mini-batches are pipelined through the shards, improving resource utilization by introducing some degree of parallelism for computation. But "bubbles" where no parallelism is possible persist. Imbalances in the runtimes of the shards can lead to resource idling. C) SystemX in the typical DL application stack. SystemX executes large-model training for DL application users by automatically using available system resources efficiently.

achieve higher overall resource utilization. In turn, this can help reduce runtimes and costs.

Our Approach. We present SystemX, a new system for large DL training that satisfies both of the above desiderata. Its design is inspired by the architecture of RDBMSs, specifically, using more of the memory hierarchy to meet the first desideratum, multi-query optimization to meet the second, and exposing higher level APIs to shield ML users from having to manually optimize the execution. Figure 1(C) illustrates our high-level architecture and its placement in between the user-facing APIs and the hardware (more details in Section 3). The user provides a set of model specifications in a popular DL tool (we focus on PyTorch). SystemX rewrites the neural computational graphs to satisfy the GPU memory constraints *automatically*. If there are multiple models, SystemX automatically decides what model to place on which device when and manages all intermediates.

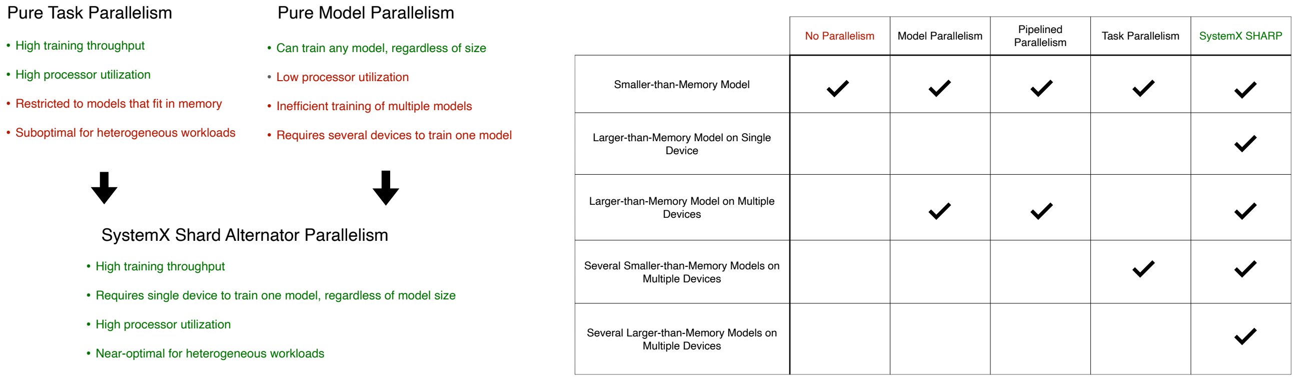
Techniques in SystemX.. SystemX achieves the desiderata with a suite of data systems techniques, some novel and some old (inspired by RDBMSs), albeit repurposed to DL systems. But our main novelty is also in how we assemble these "right" set of techniques to build a fully automated system for large model DL workloads. First, SystemX features a *model spilling* technique to "break up" a large model into manageable shards. Only some model shards are loaded onto GPUs, while the rest sit in DRAM. This is akin to sharding a large table in an RDBMS and loading only the active shard from disk to buffer manager. Second, we fully *automate the partitioning* of all models to respect GPU memory constraints with a lightweight and highly general approach.

Third, we devise a novel hybrid form of parallel DL execution that *blends task parallelism and model parallelism* to improve overall resource efficiency in multi-GPU multi-model cases. Figure 2 positions our hybrid parallelism approach, which we call Shard Alternator Parallelism (SHARP). The table in the figure also contrasts SHARP with pipeline parallelism. Basically, model parallelism's main con is that it keeps only one GPU busy at a time, while task parallelism's main con is that it requires a model to fully fit in a GPU's memory. SHARP obviates both these issues. While Section 4 will present the details of how SHARP works, the our intuition is to "break models down and put them back together in a better way." To do so, SHARP relies on the model spilling and partitioning mentioned before.

Fourth, we adopt the classic RDBMS trick of *double buffering* to reduce the latency in between execution of model shards on a device. It overlaps GPU computation with loading from DRAM and works in lockstep with SHARP. Finally, we put all these techniques together to formulate a formal scheduling problem for our setting. We propose a simple greedy algorithm we call *Sharded-LRTF* to tackle it. We show using simulations that Sharded-LRTF offers near-optimal results for both homogeneous and heterogeneous sets of models.

We prototype all of our ideas on top of PyTorch to create SystemX. We evaluate it empirically on two key large-model benchmark workloads and datasets: hyperparameter tuning for BERT-Large on the WikiText-2 [20] dataset and neural architecture evaluation for Vision Transformer [5] on the CIFAR-10 dataset. SystemX substantially outperform all prior approaches, yielding near-linear speedups on an 8-GPU machine for both workloads. In particular, it offers almost 7.5x speedups over regular model parallelism, while being almost twice as fast as FlexFlow. SystemX also report the highest GPU utilization. We then dive deeper into the behavior of SystemX by varying model scales, number of models trained together, and number of GPUs. We also report an ablation study showing the impact of our two key optimization techniques: SHARP and double buffering. In summary, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to study the union of scalability and parallelism from first principles for multi-model training of very large DL models.
- Inspired by RDBMSs, we present a suite of scaling and efficiency techniques combining automated model partitioning, model shard spilling, and double buffering.
- We devise a novel hybrid form of DL execution called SHARP combining task parallelism and model parallelism that mitigates the major cons of both.



Pure Task Parallelism

- High training throughput
- High processor utilization
- Restricted to models that fit in memory
- Suboptimal for heterogeneous workloads

Pure Model Parallelism

- Can train any model, regardless of size
- Low processor utilization
- Inefficient training of multiple models
- Requires several devices to train one model

SystemX Shard Alternator Parallelism

- High training throughput
- Requires single device to train one model, regardless of model size
- High processor utilization
- Near-optimal for heterogeneous workloads

	No Parallelism	Model Parallelism	Pipelined Parallelism	Task Parallelism	SystemX SHARP
Smaller-than-Memory Model	✓	✓	✓	✓	✓
Larger-than-Memory Model on Single Device					✓
Larger-than-Memory Model on Multiple Devices		✓	✓		✓
Several Smaller-than-Memory Models on Multiple Devices				✓	✓
Several Larger-than-Memory Models on Multiple Devices					✓

Figure 2: SystemX hybridizes task parallelism and model parallelism to achieve more scalability and efficiency than either. Compared to prior approaches, SystemX's hybrid approach enables us to layer it on top of DL tools to achieve optimizations prior work has not explored.

- We cast our multi-model shared training as a form of multi-query optimization and build a simple scheduler featuring an efficient greedy algorithm called Sharded-LRTF.
- We implement all our ideas in a system we call SystemX. A thorough empirical evaluation with real multi-model large DL workloads shows that SystemX substantially outperforms prior state-of-the-art open source and industrial systems.

2 BACKGROUND AND PRELIMINARIES

We start with some background on key technical concepts and notation needed for the rest of this paper.

Basic ML Terms. In this paper, we use the term *model* to refer to a neural computational graph (also called the *neural architecture*) and its parameters. These are specified and trained using popular deep learning (DL) tools such as PyTorch or TensorFlow. The *training* of a model involves an optimization procedure called stochastic gradient descent (SGD). It samples *mini-batches* of data from the training dataset, performs updates to the parameters of the model, and repeats this for all mini-batches. A full pass through the dataset is called an *epoch*. Training a model requires many epochs of SGD. Popular DL tools implement many variants of SGD (e.g., Adam, AdaGrad, RMSProp, etc.) but their data access patterns are identical. The size of a DL model can be measured through the number of parameters and/or its memory footprint. Updates to the parameters of a DL model involve two passes: *forward*, which transforms input (data features) to output (target) with a series of computations, and *backward*, which transforms a loss calculated with the label back into updates on parameters through a reverse series of computations. The backward pass is also called *backpropagation* or backprop for short.

Model Shards. Most DL models can be seen as a series of groups of tensor operators, also called "layers" [7]. Such models are also called "feedforward" architectures. Given such a layout, we can sub-divide a model's layers into contiguous subsets of layers; each subset is known as a *model shard*. Thus, a model can be partitioned into a series of non-overlapping shards. Some architectures such as RNNs and ResNets do not fit this layout. We omit support for non-feedforward models in this paper for tractability sake and leave it to future work; see Section 6 for more discussion. But most of the largest DL models that are popular are feedforward architectures, especially the so-called Transfomer architectures such as BERT [4].

Note that our definition of shards assumes that partitioning occurs between layers. However, in some cases, layers themselves can be partitioned without affecting the output. This setting enables model parallelism to actually use more than one GPU simultaneously — the layer itself can be parallelized to enable parallel compute. Some frameworks [14] use this to produce a "partial" model parallel scheme, where some layers are parallelized in this way while others use data parallelism. However, this technique can only be applied effectively if the layer operation is amenable to parallelization. By contrast, our sharding scheme which splits between different layers can be applied to any multi-layer network.

Pipeline Parallelism and Micro-batches. model parallelism approaches typically split apart a mini-batch's computation further to achieve more parallelization across devices via *pipeline parallelism*. A *micro-batch* is a shard's stage of compute on a minibatch. Note that the term does not refer to data-wise partitioning within a minibatch — it refers to compute stages on a minibatch [11].

Recall that a mini-batch update for SGD includes both a forward pass and a backward pass. These passes are split across micro-batches within a mini-batch-compute stage and interleaved across devices. Figure 1(B) offers an illustration of this approach. Note, however that linear speedup is impossible with pipeline parallelism due to the inevitability of so-called "bubbles" when interleaving the micro-batch passes.

Task Parallelism. Put simply, task parallelism is the concurrent execution of multiple tasks. In the ML context, it refers to the concurrent execution of training of multiple independent models, e.g., in a model selection context such as hyper-parameter tuning or neural architecture selection [15]. The most common form of task parallelism is one where an entire compute device (say, a GPU for DL) is assigned to one task for uninterrupted execution. In other words, a task is indivisible.

Data Parallelism. Data parallelism can be seen as an abstraction on top of task parallelism wherein a single model is trained on multiple compute devices wherein each device operates with a subset of the dataset. The model copies are identical and refer to a shared pool of parameters that they will infrequently update and pull from. Two common variants of data parallelism are Parameter Server [18] and Horovod [29]. The latter is increasingly popular in DL practice and its basic mechanism is now natively supported in PyTorch as the Distributed Data Parallel library [19]. Data parallelism ignores the higher degree of parallelism available in multi-model workloads such as model selection.

3 OVERVIEW OF SYSTEMX

SystemX is designed to be a lightweight wrapper around the popular DL tool PyTorch.[1] We do not need any internal code of the DL tool to be altered, which can help ease practical adoption. Figure 2 illustrates the overall architecture of SystemX and how it handles the models. There are 4 main components: *API*, *Automated Partitioner*, *Memory Manager*, and *Scheduler*. We briefly explain the role of each component.

- **API.** The user specifies a set of models to be trained using standard PyTorch APIs. Figure 4 provides an example. Note that SystemX can also scale the training of single model on a single device. The neural architectures are *fully automatically* inferred by SystemX; no custom annotations are needed. The user just needs to provide the model(s), a data loading function, and model selection specification, e.g., hyperparameter search grid or metaheuristics. SystemX then readies the training jobs of all models and .
- **Automated Partitioner.** SystemX automatically ascertains the memory size(s) of the GPU(s). Then it automatically partitions the model(s) given into model shards that respect the GPU memory constraints. At a given point in time, a GPU runs computations for only only one model shard. Section 4.1 explains our sharding process in more detail.
- **Memory Manager.** After the model shards are constructed, SystemX puts them all in the machine's DRAM. It then moves shards up the memory hierarchy into the GPU based on the schedule produced by the *Scheduler*. When a shard's computation is completed, it is moved back to DRAM. Intermediate outputs within a model across shards are also written to DRAM by this component. All of this happens transparently to the DL user. This is the crux of how SystemX achieves seamless scalability to very large models. Section 4.2 explains this "model spilling" technique in more detail.
- **Scheduler.** This component is the core orchestrator of what shard gets placed on what GPU and when. It uses SHARP, our novel hybrid of task- and model parallelism. It ensures that the shards of a given model are scheduled in a way that respects the *sequential dependency* inherent in the DL model's forward pass and backward pass. We formalized this scheduling problem as an MILP, compared a few alternative scheduling algorithms, and devised a simple new algorithm that best suits our system setting. We also devised a buffering technique to further raise resource utilization. Sections 4.3–4.5 explain our ideas in more detail.

4 TECHNIQUES IN SYSTEMX

We now dive into the techniques in SystemX to achieve seamless scalability and resource-efficient parallelism for training multiple large DL models in one go. Our techniques are inspired by a suite of classical ideas in RDBMSs, viz., spilling, sharding, multi-query optimization, and double buffering [27, 28], but our work is among the first to study them in the context of DL training. While the individual techniques may not be highly novel in the context of data management systems, the way we identify the right set of techniques, adapt them for DL, and synthesize them in SystemX is novel. This enables SystemX to offer state-of-the-art results in this important DL systems setting.

4.1 Model Spilling

In traditional model parallelism, a model is divided into shards; each shard is placed on a different (GPU) device. But all shards must be loaded across multiple GPUs for a forward/backward pass to work at all. We observe that this is an overkill: due to the sequential dependency across layers inherent in DL models, only one device is typically "active" with computation at any point in time. The other devices are merely repositories for inactive model shards.

Exploiting the above observation, we use a simple idea in SystemX to avoid making all shards active: *spill to DRAM.* Only an active shard is promoted to a GPU's memory, while the rest "wait" in DRAM. This is akin to sharding a large table and staging reads between disk and DRAM in RDBMSs, except we focus on a higher level in the memory hierarchy and apply it to a large model instead. All this means SystemX scales to arbitrarily large models *on a single device.* So, even a trillion-parameter DL model can now be trained on a single GPU out of the box. This can already makes a qualitative difference for DL users with limited resources.

The above poses two open questions: Who does the sharding and how? How to efficiently train multiple models together with sharding in a multi-GPU setup? Our next two techniques tackle precisely these questions.

4.2 Automated Model Partitioning

It is a painful for DL users to manually partition models for more parallelism. Thus, we want to fully automate this stage based on model sizes and GPU memory constraints. There are two main alternative approaches: static and dynamic. We first explain the system design tradeoffs of these and then explain our algorithm.

A static approach gets all models' layers' memory footprints, all GPU memory capacities, etc. upfront to perform discrete optimization. Prior art [14, 24] does use some heuristics of this sort, albeit restricted to a specific class of models. Unfortunately, their approach is not general enough on model architectures. While one

[1] It is a relatively simply engineering effort to add support for TensorFlow too but we skip it in our current version for tractability.

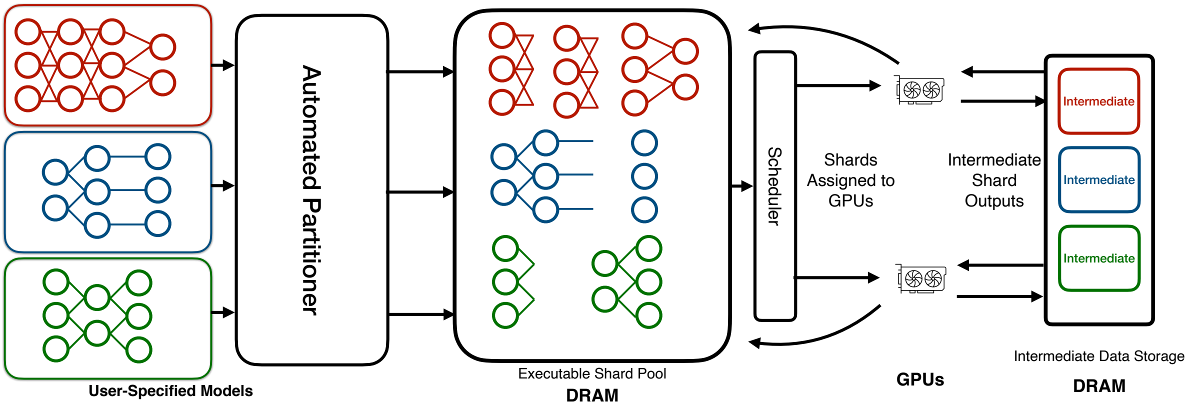


Figure 3: Overall system architecture of SystemX. It takes as input a user-specified set of models. It partitions the models automatically and assigns shards to GPUs according to its Scheduler. DRAM is used to temporarily store inactive model shards and inter-shard outputs.

```
task_0 = ModelTask(model_0, loss_fn, dataloader_0, lr_0, epochs_0)
task_1 = ModelTask(model_1, loss_fn, dataloader_1, lr_1, epochs_1)
orchestra = ModelOrchestrator([task_0, task_1])
orchestra.train_models()
```

Figure 4: Example usage of the API of SystemX.

Algorithm 1: Our dynamic partitioning algorithm for a model

```
Input: Model as a sequence of cut-point layer indices $L$; data mini-batch $B$
Result: Set $S$ of start indices of model's shards
Append 0 to $S$;
Select the GPU $G$ with smallest memory in the set;
for Layer index $l$ in $L$ in increasing order do
    Place $l$ and $B$ on $G$;
    $B' \leftarrow$ Forward pass through $l$ on $B$;
    $T \leftarrow$ New tensor with same shape as $B'$;
    Backpropagate $T$ through $l$ without freeing its memory;
    if $G$ out of memory then
        Append $l$ to $S$;
        For all layers before $l$, release memory in $G$;
    else
        continue;
    end
end
Return $S$;
```

could use sophisticated graph partitioning algorithms for “optimal” sharding, we find this is not worthwhile for two reasons. First, this stage is anyway only a tiny part of the overall runtime, which is dominated by the actual training runs. Second, due to the marginal utility of over-optimizing here, it will just make system engineering needlessly complex.

We prefer simplicity that still offers generality and good efficiency. Thus, we use a *dynamic greedy approach* based on toy “pilot runs.” Algorithm 1 presents it succinctly. The basic idea is to pack as much of a model as possible on to a GPU. If the set of GPUs is heterogeneous, we use the smallest-memory GPU to ensure cross-device compatibility of shards. We treat a DL model as an ordered list of layer indices, with the layers being “cut-points” in the graph to enable smooth partitioning. SystemX then iterates through these indices to run “toy” passes with a *single mini-batch once.* If the run is successful, the Partitioner raises the shard size by appending the next set of layers. If the GPU throws an out-of-memory error, we remove the set of layers appended last. Thus, in this dynamic way, we find the near-maximum set of layers that fit in GPU memory; this set is then cut off from the model as its own shard. The Partitioner continues this process for the remaining the layers, until that model is fully partitioned. Repeat this for all other models the user gave. We record all pilot runs’ runtime statistics for later use by our Scheduler.

Our above approach is general, flexible, and easy to implement, but it does have two (minor) cons. First, the shard may be too tiny at the end of a model, perhaps even just one layer. But this is not a showstopper in SystemX because it has little effect on overall efficiency in the mix of things. Second, since our pilot runs use the smallest-memory GPU, the actual training later may under-utilize the memory of larger GPUs (if applicable). Nevertheless, Section 5 shows that our approach substantially boosts GPU utilization relative to all prior art anyway.

4.3 SHARP

We now present one of our key novel techniques: Shard Alternator Parallelism (SHARP), a hybrid of classical model parallelism and task parallelism to improve resource efficiency. We define our basic unit of computation, *shard unit*, as follows: the subset of computations of a forward or backward pass on a model’s shard. Thus, a full forward or backward pass of a model is a *sequence* of shard

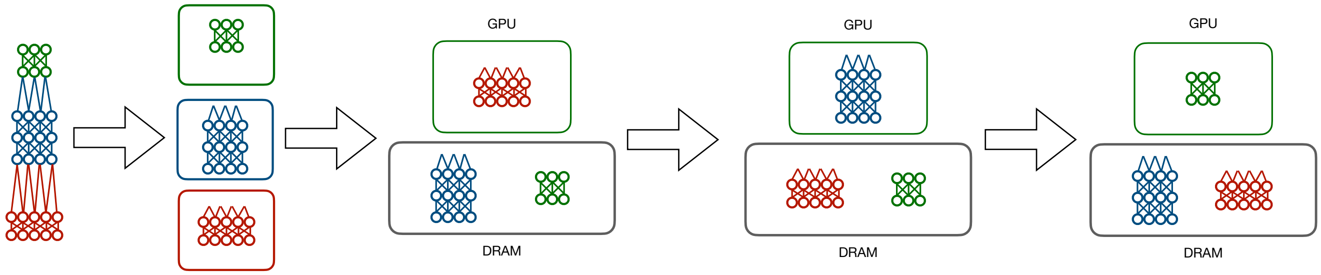


Figure 5: Illustration of model spilling as a temporal schematic. SystemX places inactive shards at a lower level of the memory hierarchy (DRAM here), from which they are re-activated later.

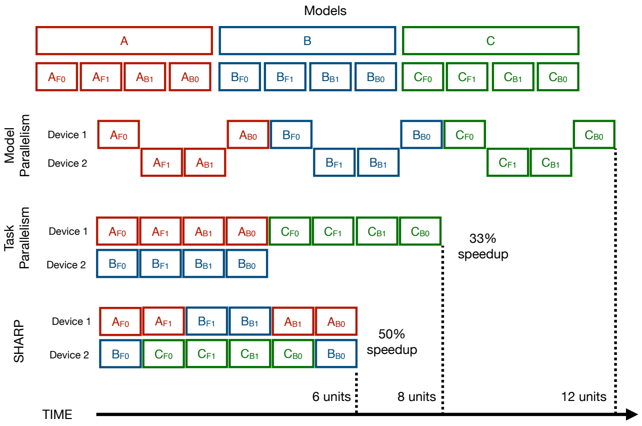

Models
A
B
C
A_{F0} A_{F1} A_{B1} A_{B0} B_{F0} B_{F1} B_{B1} B_{B0} C_{F0} C_{F1} C_{B1} C_{B0}
Model Parallelism
Device 1 A_{F0} A_{B0} B_{F0} B_{B0} C_{F0} C_{B0}
Device 2 A_{F1} A_{B1} B_{F1} B_{B1} C_{F1} C_{B1}
Task Parallelism
Device 1 A_{F0} A_{F1} A_{B1} A_{B0} C_{F0} C_{F1} C_{B1} C_{B0}
Device 2 B_{F0} B_{F1} B_{B1} B_{B0}
33% speedup
SHARP
Device 1 A_{F0} A_{F1} B_{F1} B_{B1} A_{B1} A_{B0}
Device 2 B_{F0} C_{F0} C_{F1} C_{B1} C_{B0} B_{B0}
50% speedup
6 units
8 units
12 units
TIME

Figure 6: Illustration of SHARP and contrasting with regular task parallelism and model parallelism for training 3 models A, B, and C, each with 2 shards. Forward and backward pass are shown separately, e.g., $A_{1,F}$ is the forward pass on shard 1 of A. Note that while task parallelism uses the device efficiently for one model, it requires each model to fit entirely in single-device memory. Model parallelism scales to larger models but it severely under-utilizes the devices. SHARP offers the best of both worlds: efficiency of task parallelism and scalability of model parallelism.

units.[2] Overall, the scheduling goal is to execute all shard units of all models given by the user for all epochs.

Figure 6 illustrates the basic idea of SHARP contrasted with both task- and model parallelism. After a model's shards are created (Section 4.2), shard units are naturally set. The key difference in SHARP is that a given model's shard units do not necessarily run *immediately* after one another, i.e., they may be staggered over time. This is the key reason for SHARP's higher efficiency–it breaks things down and puts them back together better. Strictly speaking, such flexibility is technically feasible with regular task parallelism too but it will be very tedious for an ML user because they must manually break each mini-batch shard unit into its own task. In contrast, in SystemX exploits such fine-grained scheduling flexibility automatically, thus *freeing the user to focus on their higher level ML goals instead of worrying about optimizing system internals*.

While the idea of SHARP is fundamentally simple (but novel), realizing it in a working system faces two bottlenecks: (1) the sheer number of shard units and (2) the latency of swapping shards between device memory and DRAM. First, note that the number of shard units to be handled by SystemX is multiplicative in four quantities: number of models given by the user, number of shards per model, number of mini-batches per epoch, and number of epochs per model training run. In realistic DL scenarios, one can easily hit *tens of millions of shard units*! Thus, next we answer two questions regarding the above bottlenecks: How to automate the orchestration of large numbers of shard units? Is it possible to reduce latency of swapping shard units?

4.3.1 ***Automated Shard Orchestration***. To realize SHARP in an system, we must handle 3 kinds of "data" before, during, and after a shard unit: (1) training data mini-batch, (2) model parameters, and (3) intermediate data/outputs of a shard unit. Thankfully, DL tools like PyTorch offer APIs that enable data bytes to be transferred from GPU memory to DRAM and vice versa. We use those APIs in SystemX under the hood to automate shard orchestration. Note that we are still discussing the *mechanisms* of how SHARP works, not the *policy* that picks what shard unit runs when–the latter will be explained in Section 4.4.

Each model is defined as a "queue" of shards in DRAM, ordered according to the neural computational graph. Each shard maintains the prepared data that will need to be used with its "next" shard. This data could include the training data mini-batch, intermediate data exchanged *between* shards, and/or gradients sent backward. The shard at the front of the queue is transferred to GPU memory along with its associated data to begin running that shard unit.

After a shard unit is over, the inputs it used and the intermediate data it produced during execution are discarded. The shard parameters (possibly updated) are returned to DRAM. In addition, the shard's intermediate outputs, say, a gradient vector or a loss value are also written to DRAM and attached to the model. They will be used as inputs for the "next" shard of that model in the future. The last shard of a model concludes a full mini-batch training pass; after that, the old mini-batch is discarded and the next mini-batch of the prepared data will be used subsequently.

[2] In recent ML literature, this unit is also called a "microbatch" [11]. We prefer to use the more standard terminology of "unit" from the operations research and systems literatures instead because the term "microbatch" may cause confusion on whether the mini-batch *data* is split further, which is not the case. A shard unit splits the *computations* (not data) of a forward/backward pass of a whole mini-batch.

Note that our above orchestration never "materializes" any information about the millions of shard units together upfront. Rather we run them in a pay-as-you-go fashion, one mini-batch (per model) at a time. Thus, SystemX remains a lightweight framework even as it achieves high efficiency. The minor con is that our scheduling approach is not necessarily globally optimal. But as we show later in Section 4.4.3, this is not a major issue for efficiency anyway.

4.3.2 ***Double Buffering.*** A common trick used in RDBMSs, e.g., for external merge sort, is double buffering [27]. The basic idea is this: the processor's memory (higher in the memory hierarchy) is split into two regions: one for active processing and the other as a "loading zone" for the next task. We bring this trick to the DL systems world for the first time. SystemX uses it to mask the latency of loading shards. We deduct a "buffer space" in GPU memory during model partitioning itself (Section 4.2) to offer a guarantee that so much buffer memory will be available during training. The buffer size is 15% by default, but users can adjust it if they like.

When our Scheduler picks the next shard to be run, we transfer it to that GPU's buffer space *even as the previous shard unit is running there*. We load only what will fit into the buffer space. Usually, this is feasible because a model shard alone is often not *that* big– it is the combination with training data mini-batch and massive intermediate outputs (created during execution) that causes high memory pressure in large-model DL.

Interestingly, our double-buffered DL training in SystemX also offers a serendipitous new bonus: we can avoid spilling (to DRAM) altogether in some cases. When a model's shard unit it active, if its next shard is double-buffered on another GPU, we send intermediates with a fast GPU-to-GPU transfer instead of a slower GPU-to-DRAM-to-GPU route. Going further, if the next shard happens is double-buffered on the same GPU, intermediates need not move out of the GPU at all, substantially reducing latency.

Overall, while we focus on the GPU memory-DRAM dichotomy, our above techniques are general enough to be applicable across the entire memory hierarchy: between DRAM and local disk, local and remote disk, etc. We hope our work inspires more research on such cross-layer caching and prefetching ideas from the RDBMS world to benefit ML/DL systems as well.

4.4 Scheduling Formalization of SHARP

The sheer number and variable runtimes of shard units across models necessitates a rigorous automated Scheduler. An immediate issue is that the different models may train for different numbers of epochs, say, due to convergence-based stopping criteria for SGD. Moreover, it is possible different devices have different compute capacities. Furthermore, devices may drop over time, say, due to faults, or get added, say, due to elasticity. Due to all these reasons, we choose a *dynamic scheduling* approach instead of a globally static joint optimization up front. That is, we place shard units on devices as and when a device becomes available over time. This design decision makes our problem more tractable for both unified reasoning and for simplicity of implementation. Specifically, we formalize our problem for a given set of epochs per model at a time. This may mean one epoch at a time or a pre-fixed number of epochs the user gives per model. Regardless, we treat each model to be trained as a *queue of shard units* unifying reasoning of division within a mini-batch, across mini-batches within an epoch, and across epochs.

Table 1: Notation for our scheduling formalization.

Symbol	Description
T	List of models specified by the user to be trained
P	List of devices (GPUs) available for training.
$M_i \in \mathbb{Z}^+$	M_i is the total number of shard units for model $T_i \in T$. Note that this covers all mini-batches (and potentially epochs).
$S_i \in \mathbb{R}^{M_i}$	S_i is a variable-length list of shard unit runtimes for model T_i. The runtime of shard unit j is denoted as $S_{i,j}$.
$X_i \in \mathbb{R}^{\|P\|\times\|M_i\|}$	X_i is a variable-shape matrix of start times of shard units of model T_i across workers. The start time of shard unit j on worker p is denoted as $X_{i,p,j}$. Note that this linear ordering covers not just the model's forward and backward passes but also ordering across mini-batches (and potentially epochs).
$Y_i \in \{0,1\}^{\mathrm{\|P\|\times L\times L}}$	L is the total number of shard units across all models, i.e., $L = \sum_i M_i$, indexed cumulatively by the index of model i and its shard unit j (denoted i_j). $Y_{p,i\_j,i'\_j'} = 1 \Leftrightarrow X_{i,p,j} < X_{i',p,j'}$.
U	An extremely large value used to enforce boolean logic.

4.4.1 ***Formal Problem Statement as MILP.*** The scheduling problem is as follows. At a given point in time, when a device (GPU) becomes available, a shard unit must be selected from one the model's queues to be placed upon that device. Shard units become *eligible* for scheduling if they have no pending dependencies, i.e., they are at the front of their queue and no other shard unit of that same model is still running on another device. The Scheduler's job then is to pick a shard unit from the set of eligible shard units. Double-buffered training is already factored into this formulation as follows: the Scheduler is actually picking shard units for double-buffering, and they get promoted from the buffer to compute.

The runtimes of all shard units are given as input. Recall from Section 4.2 that we do get this information from the pilot runs during automated model partitioning using a single mini-batch. While these runtimes may not be fully accurate due to differing system factors during execution, they are accurate enough for devising effective scheduling. We now present the formal scheduling problem as an MILP. Table 1 explains our notation.

$$\texttt{Objective:}\quad \min_{X,Y} C \tag{1}$$

Algorithm 2: Our Sharded-LRTF algorithm for scheduling.

```
Model{
    Remaining epochs $e$
    Minibatches per epoch $b$
    Remaining minibatches in current epoch $ce$
    Minibatch training time $t$
    Remaining train time in current minibatch $cm$
};
Result: Selected Model $m'$
Input: Idle Models $[M]$
$MaxTrainTime = 0$;
$MaxModel$;
for Index i, Model m in $[M]$ do
    $ModelTrainTime = ((m_e - 1) \times m_b + m_{ce} - 1) \times m_t + m_{cm}$;
    if $ModelTrainTime > MaxTrainTime$ then
        $MaxTrainTime = ModelTrainTime$;
        $MaxModel = i$;
    else
        continue;
    end
end
Return $MaxModel$;
```

`Constraints:`

$$\begin{aligned}
&\forall t, t' \in [1, \ldots, |T|] \qquad \forall p, p' \in P\\
&(a)\ \ \forall j \in [2, \ldots, M_t]\ X_{t,p,j} \geq X_{t,p',j-1} + S_{t,j-1}\\
&(b)\ \ \forall j \in [1, \ldots, M_t] \qquad \forall j' \in [1, \ldots, M_{t'}]\\
&\qquad X_{t,p,j} \geq X_{t',p,j'} + S_{t',j'} - (U \times Y_{p,t\_j,t'\_j'})\\
&(c)\ \ \forall j \in [1, \ldots, M_t] \qquad \forall j' \in [1, \ldots, M_{t'}]\\
&\qquad X_{t,p,j} \leq X_{t',p,j'} - S_{t,j} + (U \times (1 - Y_{p,t\_j,t'\_j'}))\\
&(d)\ \ \forall j \in [1, \ldots, M_t]\\
&\qquad X_{t,p,j} \geq 0\\
&(e)\ \ \forall j \in [1, \ldots, M_t]\\
&\qquad C \geq X_{t,p,j} + S_{t,j}
\end{aligned} \tag{2}$$

The objective is to pick a shard unit that can minimize makespan (completion time of the whole workload at this granularity). Constraints (a) simply enforce the *sequential ordering of shard units* within a model. Note that this set per model here is unified within a mini-batch, across mini-batches within an epoch, and potentially across epochs too–they are all sequentially dependent. Constraints (b) and (c) enforce *model training isolation*, i.e., only one shard unit can run on a device at a time. Constraints (d) is just non-negativity of start times, while Constraints (e) define the makespan.

Using a MILP solver such as Gurobi [9] enables us to produce an "optimal" schedule in this context. But the above task is a variant of a general job-shop scheduling problem described in [33], and it is known to be NP-complete. Given that the number of shard units can span thousands to tens of millions, solving it optimally will likely be impractically slow. Thus, we look for fast and easy-to-implement scheduling algorithms that can still offer near-optimal makespans.

4.4.2 *Intuitions on Scheduling Effectiveness.*

We observe that there are 2 primary cases encountered by a scheduler in our setting:

(1) The number of models is equal to or greater than the number of available devices.
(2) The number of models is less than the number of available devices.

In case (1), there will always be at least one eligible shard unit for each device at every scheduling decision. Any shard-parallel scheduling algorithm that accounts for all devices can easily keep all devices busy most of the time, i.e., *busy waiting* is unlikely. In case (2), all models can be trained simultaneously. Since each model's shard unit uses at most one device in SHARP, and since there are more devices than models, there is no contention for resources here. In this case, regular task parallelism-style scheduling suffices and the makespan will just be the runtime of the longest "task."

In both the cases above, even basic randomized scheduling might yield reasonable makespans. However, what it will not take into account is that case (1) is not static. Over time, as models finish their training, our setting may "degrade" from case (1) to case (2). Thus, two different schedulers that operate on a workload in case (1) may differ in their effectiveness based on how gracefully they degrade to case (2). As noted before, the makespan in case (2) scenario is determined solely by the longest-running remaining model. This gives us an intuition for a simple scheduler that can often do better than randomized: *minimize the maximum remaining time among the remaining models.*

In most realistic DL workloads, overall completion time will be dominated by case (1). Thus, the marginal utility of pursuing an overly optimized Scheduler is low given the likely higher implementation complexity. However, if degradation to case (2) occurs earlier on, and if there is a substantial differences in task runtimes post-degradation, the overall completion times can differ more significantly based on the scheduling. Such degradation can arise in model selection workloads that "kill" underperforming models in earlier epochs, e.g., Hyperband [**?**] or by manual intervention. Thus, we aim for a scheduling algorithm that can address such cases too.

4.4.3 *Our Scheduling Algorithm: Sharded-LRTF.*

We propose a simple and practical greedy heuristic we call Sharded Longest Remaining Time First (LRTF) based on our above intuitions. Algorithm 2 explains our algorithm. Sharded-LRTF simply selects the model training task with the *longest total remaining train time* at every possible scheduling decision time. The fine-grained nature of scheduling individual shard units means that it is possible to minimize the longest remaining task time with a high degree of accuracy. As we explained before, the main differentiating factor between alternative dynamic scheduling algorithms is in when case (2) scenarios arise over time, at which point makespan is decided entirely by the longest individual task. Thus, by aiming to minimize this value, the total makespan will be rather close to optimal.

To quantitatively understand the effectiveness of Sharded-LRTF, we compare it using simulations against a basic randomized schedule and a Gurobi-output "optimal" schedule. For tractability, we set a timeout of 100s for Gurobi. We who both a homogeneous setting (all neural architecture are identical) and a heterogeneous setting, wherein they differ significantly. We assume all GPU devices are

Table 2: Details of end-to-end workloads. *Architectures similar to BERT-Large and ViT, scaled up for demonstration.

Dataset	Model Architectures	Model Sizes	Batch Size	Learning Rate	Epochs
WikiText 2	BERT-Large*	1B	8, 16, 32	$10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}$	4
CIFAR-10	Vision Transformer (ViT)*	600M, 300M, 800M, 1B, 1.5B, 2B	64, 128	10^{-3}	5

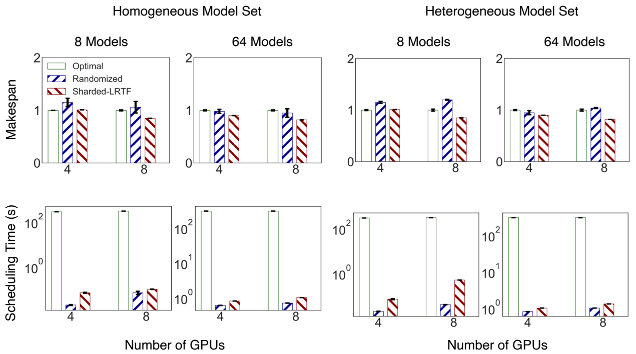


Figure 7: Comparison of various scheduling algorithms. Makespans are normalized to Optimal.

identical for simplicity, but that is also common in practice. Per-epoch runtimes of a model in the homogeneous setting are all fixed to 2 hours each, with 2000 shard units each. For the heterogeneous setting, per-epoch model runtimes are set between 30 minutes to 4 hours; number of shard units are set between 100 to 10,000. We randomly sample an initial set and report the average and standard deviations of 3 runs on the fixed set. Variance occurs due to non-deterministic scheduling behaviors from random selection and Gurobi timeout. Figure 7 shows the results.

We note that MILP “optional” has higher makespan in some cases because Gurobi did not converge to the global optimal in the given time budget. The randomized approach matches it or performs worse in many cases. But Sharded-LRTF matches or significantly outperforms the other approaches in many cases, especially in the heterogeneous setting. This is in line with the intuition we explained that being cognizant of longer running models in the mix is helpful. Also note that the runtime of Sharded-LRTF is in the order of tens of milliseconds, ensuring it is practical for us to use in SystemX repeatedly for scheduling shard units on devices dynamically. Note that the actual mini-batch training computations on the device are the dominant part of the overall runtime.

5 EXPERIMENTS

We now compare SystemX against state-of-the-art and industrial tools aimed at training larger-than-GPU-memory DL training: PyTorch Distributed, Microsoft’s DeepSpeed, FlexFlow from Stanford/CMU, and Google’s GPipe idea. We also show multiple variants with DeepSpeed, including superimposing a hybrid task parallelism (note that regular task parallelism is not applicable) and a hybrid data parallelism offered by DeepSpeed. We then dive into how SystemX performs when various workload and system parameters are varied.

Workload Details. We use two popular DL model selection scenarios: hyperparameter evaluation and neural architecture evaluation. Table 2 lists the details. For hyperparameter evaluation, we focus on masked-language modeling with the Transformer architecture BERT-Large [4]. This is a common use case now with such pre-trained models being finetuned for downstream NLP tasks. We use the benchmark WikiText-2 dataset. The neural architecture is fixed and we vary batch size and learning rate as key hyperparameters to create a total of 12 models to train, each with 1B parameters.For neural architecture evaluation, we focus on an emerging computer vision task with variants of the Vision Transformer (ViT) model [5]. We use the benchmark CIFAR-10 dataset. We create models with sizes between 600M parameters and 2B parameters. We also vary batch sizes, leading to a total of 12 models again.

Machine Setup. We focus on the single-node multi-GPU setting, anecdotally the most common among DL practitioners (although nothing in SystemX prevents generalizing to multi-node clusters in the future).We use 8 GPUs, each being GTX 1080Ti’s with 11GB memory. The machine has 512 GB of DRAM and an Intel Xeon CPU with 10 2.20GHz cores, and it runs Ubuntu 18.04.

5.1 End-to-End Workloads

Figure 8 presents overall runtimes and GPU utilization results. We find that the baseline off-the-shelf PyTorch Distributed and DeepSpeed model parallelism report massive resource under-utilization. Thus, their runtimes are the highest. The basic hybrids with data- or task- parallelism, as well as FlexFlow do provide higher utilization and some modest speedups, but the fundamental limitations of model parallelism persist with such approaches, which means they still fall substantially short of ideal linear speedup (8x in this case). GPipe-style pipeline parallelism is much better, about 5x speedup against regular model parallelism. But SystemX is the most efficient approach overall, reaching about 7.5x, close to the physical upper bound. The average GPU utilization of SystemX is also the highest at over 80%.

5.2 Drill Down Analysis

We now dive deeper into the behavior of SystemX when varying key parameters of interest from both ML and system standpoints.

5.2.1 *Impact of Model Scale.*

We vary the scale of the models to see the impact on relative performance of SystemX. We fix the number of GPUs at 8 and the number of models to 12. Figure 9 shows the results. We see that SystemX’s speedups over regular model

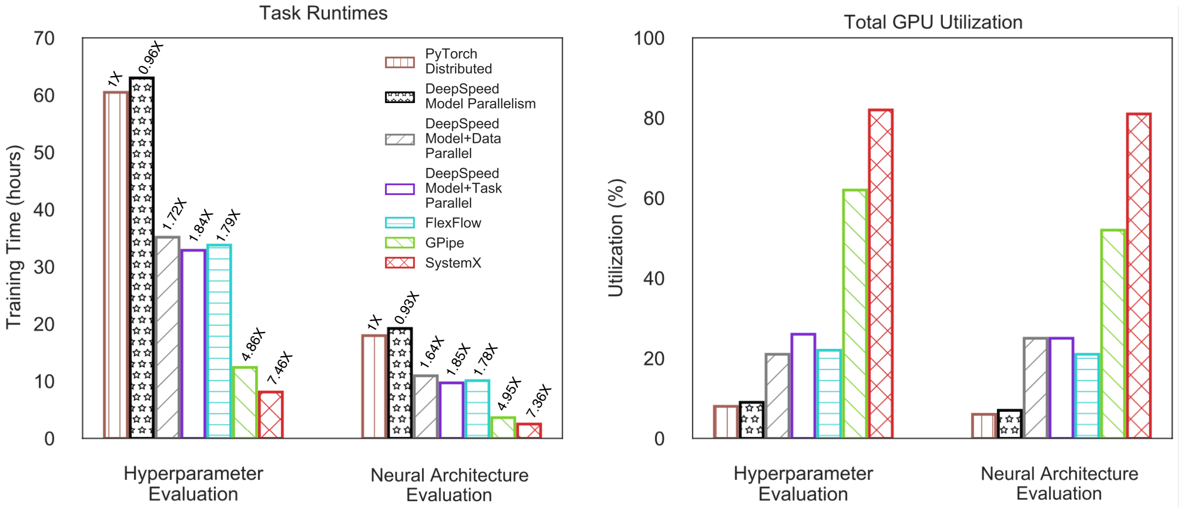


Figure 8: End-to-end workload results: Runtime speedups relative to the baseline PyTorch Distributed and GPU utilization.

parallelism *is fairly consistent* even as the model scale grows. This is because our partitioning approach (Section 4.2) and the dynamic Sharded-LRTF algorithm (Section 4.4) together ensure that shard unit times are similar even as model scale grows; basically, it just leads to more shard units to run.Our SHARP and double-buffering techniques further ensure that having more shard units do *not* cause relatively more resource idling on average.

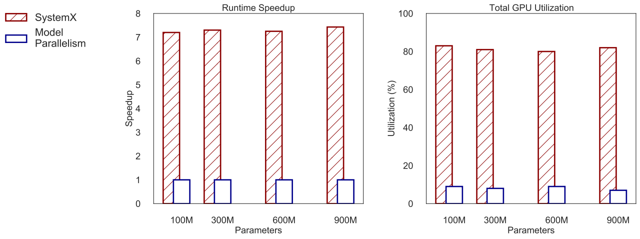


Figure 9: Impact of model scale. Runtimes normalized to the first instance of regular model parallelism for clarity.

5.2.2 *Impact of Number of GPUs.*

We now study how varying the number of GPUs affects SystemX's speedup behavior. We fix the workload to 4 Transformer models, each with 250M parameters. We choose only 4 models to showcase both regimes: when the number of devices is less than models and vice versa. Figure 10 shows the results. We see that SystemX exhibits a roughly linear speedup when there are more models than devices. And when that flips, since SystemX has not more models to schedule, the speedup flattens as the degree of parallelism is limited. This flattening in the fewer-models regime is inherited from task parallelism by SHARP. We believe further hybridization of SHARP with data parallel training can help boost speedups and resource utilization in this regime; due to its complexity, we leave it to future work.

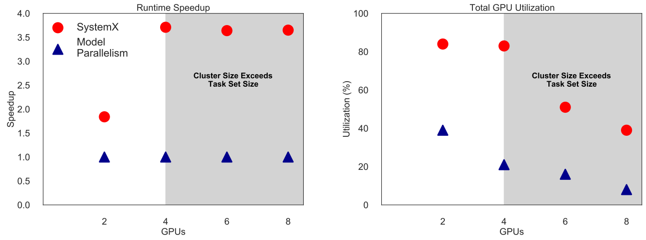


Figure 10: Varying the number of GPUs. Speedups are normalized to standard model parallelism.

5.2.3 *Impact of Number of Models.*

We now vary the number of models that are trained together. The number of GPUs is set to 8; all models have are uniformly large, at 250M parameters (same Transformer workload as before). Figure 11 shows the results. We see that SystemX exhibits close to 8x speedups when the number of models is 8 or more but lower than that, the speedup is capped close to the actual number of models. As before, this too is due to SHARP inheriting the degree of parallelism from task parallelism. The GPU utilization numbers vary proportionally to the speedups seen.

5.2.4 *Ablation Tests.*

In this experiment, we explore the effect of system components on framework performance. The number of devices is fixed to 8, with 16 Transformer models. All optimization levels include model spilling as a baseline, as this technique is critical to SystemX's basic operations. Table 3 demonstrates the results. Pure model spilling dramatically slows down model training. This is only to be expected, given that it introduces a dependency on DRAM. SHARP's throughput improvements dominate the slowdowns of model spilling, but it is important to note that SHARP's speedups are workload-dependent. Double-buffering largely eliminates the cost of model spilling, enabling further speedups.

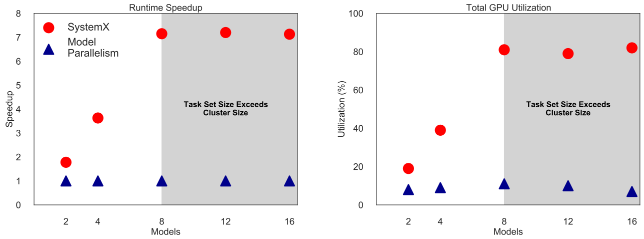


Figure 11: Varying the number of models trained together. Speedups are normalized to standard model parallelism.

Optimization Level	Runtime (hrs)	Runtime relative to SystemX
SystemX without SHARP or double-buffering	41.5	13.05X
SystemX without double-buffering	4.8	2.3X
SystemX	1.85	1X

Table 3: Runtimes and slowdowns of SystemX when our two key optimizations are disabled one by one.

6 CURRENT LIMITATIONS AND FUTURE OPPORTUNITIES

Non-Sequential Neural Architectures. As mentioned earlier, SystemX focuses on neural computational graphs that can be represented as sequences of layers or groups of layers (prior work on pipeline parallelism also assumes this). The most popular classes of GPU-memory-bottlenecked models in DL practice today, viz., Transformers, as well as most convolutional neural networks and multi-layer perceptrons do satisfy the assumption. Some not-fully-sequential models such as Inception, ResNets, and DenseNets are easily handled because residual or skip connections can be linearized with a single "super-vertex" in the graph specification given to SystemX. As long as the user defines the graph in that way using the DL tool's API, SystemX works out of the box for such models too. But for recurrent neural networks (RNNs) and graph neural networks (GNNs), SystemX would need to be extended to account for non-trivial dependencies across shard units of a model. Backpropagation through time, maintaining memory cells, and cross-layer global connections all require non-trivial extra implementation machinery and modifications to our Scheduler. We leave such extensions to future work.

Large Model Inference. This paper focused primarily on training of large models. But a trained model is then used for inference in an application. If one wants to use a GPU for inference, the same GPU memory bottleneck exists. Fortunately, SystemX's model spilling, automated partitioning, and automated shard orchestration all suffice already for out-of-the-box large model inference too. While we have not implemented an inference API as of this writing, it is a straightforward addition we plan to do soon.

DL Tool Generality. SystemX is currently implemented as a wrapper around PyTorch. But all of our techniques described in Section 4 are generic enough to be used with, say, TensorFlow or MXNet as well. We just chose to prioritize the depth of our system over generality of DL tools as of this writing. But nothing in SystemX prevents support for additional DL tools in the future.

CUDA-level Optimization. We designed SystemX to operate on top of PyTorch to ensure backward compatibility as PyTorch evolves. This means we did not exploit any low-level optimizations for GPU-to-GPU transfers. One could technically imagine using multiprocessing in CUDA to reduce this latency, including for our double-buffering technique. But all this will require us to write new kernels in CUDA for memory management and hook them into the DL tool. We leave such ideas to future work.

More Hybrid Parallelism. When there are fewer models than devices, SystemX may under-utilize the devices due to the limitation it inherits from task parallelism (so will all alternative approaches). But one could do better by hybridizing data parallelism and pipeline parallelism with SystemX to raise overall resource utilization. This is feasible because both of those approaches are technically *complementary* to SHARP and SystemX's other techniques. We leave such sophisticated hybrid-of-hybrids parallelization to future work. But we note that in cases where there are more models than devices, SHARP is already rather close to optimal utilization, as our empirical results show.

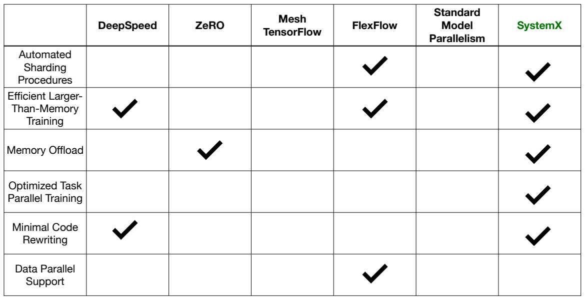
Static Partitioning. Our current pilot run-based dynamic partitioning is an overhead, albeit not that high compared to actual training runtimes. But in the future, it may be worthwhile to adopt a more static off-the-shelf model size estimator such as DNNMem [6] to reduce the runtime of the partitioning phase even further.

7 RELATED WORK

Figure 12 provides an overview of how SystemX qualitatively differs on key technical attributes relative to prominent related prior work. We discuss more in terms of groups of contributions.

Larger-than-GPU-Memory DL Training. As Sections 1 and 2 already explained, model parallelism approaches have been studied extensively in recent years to enable DL users to train larger-than-GPU-memory models, e.g., Mesh-TensorFlow [30] and FlexFlow [14]. SystemX's focus is similar but also complementary because we also study multi-model training. We approach the problem from a first principles standpoint in terms of decoupling scalability from parallelism by exploiting the memory hierarchy better. We also present a novel hybrid-parallel approach, SHARP, for multi-model execution. That said, our automated partitioning scheme can be replaced by the techniques in some of these prior systems in the future to obtain more efficiency.

We previously presented an early version of our work at a non-full length (2 page) publication venue [25]. In that article, we presented the basic principles of our model spilling and partitioning ideas, touched upon how one can exploit task parallelism more for large multi-model training, and presented an early vision for our system. This paper builds upon that article by fully implementing our vision and fleshing out all the techniques, including our novel hybrid-parallel execution SHARP. This paper also dives deeper into our scheduling formulation, proposed the Sharded-LRTF heuristic,



	DeepSpeed	ZeRO	Mesh TensorFlow	FlexFlow	Standard Model Parallelism	SystemX
Automated Sharding Procedures				✓		✓
Efficient Larger-Than-Memory Training	✓			✓		✓
Memory Offload		✓				✓
Optimized Task Parallel Training						✓
Minimal Code Rewriting	✓					✓
Data Parallel Support				✓		

Figure 12: Comparison of existing model parallelism frameworks and SystemX on key attributes.

and the double buffering trick. Finally, this paper presents a comprehensive empirical evaluation of our entire system on realistic DL workloads, as well as a deep dive into its empirical behavior.

Other Parallelism Approaches. As explained before, FlexFlow [14] explores hybrids of data parallelism and model parallelism *within* layers of a single model. But this approach is limited as the model scale increases and does not offer scalability out of the box. It is also primarily oriented toward speeding up in-GPU-memory compute. Likewise, pipeline parallelism [10, 11] also aims to reduce idle times on devices by staging multiple mini-batch computations across devices. However, it forces uses to use multiple devices for scalability. Finally, as explained before, neither of these prior parallelism approaches exploit the degree of parallelism of multi-model training, while SHARP is designed from first principles to exploit that possibility. Nevertheless, as Section 6 explains, it is possible to hybridize SHARP with these other parallelism approaches in future work.

Reducing Model Memory Footprints. There is prior work on minimizing the GPU memory footprints of models by changing the training procedures [3, 8, 12, 13, 17]. Their focus is complementary to ours; in fact, their techniques can be infused into SystemX's to reduce the number of shards during partitioning. ZeRO and DeepSpeed [26] propose a technique for reducing memory footprints by sharing model state across data parallel instances. But this is restricted to data parallelism and does not tackle our core problems of scalability and multi-model training. Also, their approach introduces a tradeoff between model parallelism and data parallelism on how to apportion devices across models, which could be a usability burden for ML users to tune. In contrast, SystemX transparently and efficiently blends task- and model parallelism across devices under the hood due to its higher level API and automated scheduling. Thus, SystemX frees the ML user to focus on their ML application goals instead of system tuning. All that said, once again it is possible to combine DeepSpeed's memory-efficient implementation of data parallelism with SystemX, as Section 6 explains; we leave such complex integration to future work.

Multi-Query Optimizations for DL Systems. A recent line of work in the database and systems literatures studies new multi-query optimization (MQO) techniques for DL systems, both for training and for inference. Our work contributes to this general growing direction but ours is the first to study MQO for *large model* DL training. Cerebro [16] and ModelBatch [23] showed that multi-model execution, primarily due to model selection workloads, are amenable to MQO to improve overall runtimes and resource utilization. In particular, Cerebro is perhaps closest to our work in that it too proposes a hybrid-parallel technique named MOP combining task parallelism and data parallelism. But it is *complementary* to SystemX because Cerebro does not support larger-than-GPU-memory training and is aimed at multi-node clusters. Thus, we believe it is beneficial to combine MOP and SHARP in the future. ModelBatch is meant for the opposite issue of models being too small to use a GPU fully; thus its focus is *orthogonal* to ours.

On the inference front, Krypton [21] performs MQO by combining multiple DL inference requests and rewriting the neural computational graphs with materialized views for features. HummingBird [22] also performs MQO for inference, albeit for classical ML pipelines compiled to DL tool backends to exploit GPUs more. Our focus in this paper is not on optimizing DL inference.

8 CONCLUSION AND FUTURE WORK

Training larger-than-GPU-memory DL models is an increasingly critical need for DL users. Yet, the existing landscape of "model parallelism" tools is sub-par on scalability and parallelism, often hard to use, and often massively wastes GPUs. We present SystemX, a new system for large model DL training that is inspired by the design and implementation RDBMSs in that way it uses the memory hierarchy consciously. We identify a judicious mix of data systems techniques–some novel and some classical RDBMS ideas adapted to DL (such as sharding, spilling, and double buffering)–to enable large-model training even on a single GPU. By further exploiting the high degree of parallelism in multi-model training, we devise a novel hybrid parallel execution technique inspired by multi-query optimization. Our work shows that the DL systems world can benefit from learning from the RDBMS world on data systems techniques that enable more seamless scalability and parallelism for DL users.

As for future work, we would like to improve the efficiency of SystemX in cases where there are fewer models than devices. This will require further hybridization of our ideas with data parallelism and pipeline parallelism. We also plan to expand support in SystemX for more complex non-sequential neural architectures such as RNNs and GNNs. Finally, we aim to expand support in SystemX for other popular DL tools and emerging DL-oriented compute hardware beyond GPUs.

REFERENCES

[1] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR* abs/1802.09941 (2018). arXiv:1802.09941 http://arxiv.org/abs/1802.09941

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[3] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174 [cs.LG]

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2020. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *CoRR* abs/2010.11929 (2020).

[6] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating GPU Memory Consumption of Deep Learning Models. (November 2020), 1342–1352. https://www.microsoft.com/en-us/research/publication/estimating-gpu-memory-consumption-of-deep-learning-models-2/ The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Industry Track.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[8] AudrÅńnas Gruslys, Remi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation Through Time. (2016). arXiv:1606.03401 [cs.NE]

[9] LLC Gurobi Optimization. 2021. Gurobi Optimizer Reference Manual. "http://www.gurobi.com

[10] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *CoRR* abs/1806.03377 (2018). arXiv:1806.03377 http://arxiv.org/abs/1806.03377

[11] Yanping Huang, Youlong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *CoRR* abs/1811.06965 (2018). arXiv:1811.06965 http://arxiv.org/abs/1811.06965

[12] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. [n.d.]. Checkmate: Breaking the Memory Wall with Efficient Tensor Rematerialization. ([n. d.]).

[13] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. *SOSP '19* (2019).

[14] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. *CoRR* (2018).

[15] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. 2016. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Rec.* 44, 4 (May 2016), 17âĂŞ22.

[16] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. [n.d.]. Cerebro: A Layered Data Platform for Scalable Deep Learning. Conference on Innovative Data Systems Research.

[17] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. 2019. Efficient Rematerialization for Deep Networks. 32 (2019), 15172–15181.

[18] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 583âĂŞ598.

[19] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[20] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. *CoRR* abs/1609.07843 (2016). arXiv:1609.07843 http://arxiv.org/abs/1609.07843

[21] Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou. 2020. Incremental and Approximate Computations for Accelerating Deep CNN Inference. *ACM Trans. Database Syst.* 45, 4, Article 16 (Dec. 2020), 42 pages. https://doi.org/10.1145/3397461

[22] Supun Nakandala, Gyeong-In Yu, Markus Weimer, and Matteo Interlandil. 10‘9. Compiling Classical ML Pipelines into Tensor Computations for One-size-fits-all Prediction Serving. *Conference and Workshop on Neural Information Processing Systems* (10‘9). https://arxiv.org/abs/2005.08314

[23] Deepak Narayanan, Keshav Santhanam, and Matei Zaharia. 2018. Accelerating Model Search with Model Batching. *Proceedings of Fourth Conference on Machine Learning and Systems (MLSys'18)* (2018).

[24] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters. *CoRR* abs/2104.04473 (2021). arXiv:2104.04473 https://arxiv.org/abs/2104.04473

[25] Details omitted for double-blind reviewing. [n.d.].

[26] Samyam Rajbhari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. (2020).

[27] Raghu Ramakrishnan and Johannes Gehrke. 1996. *Database Management Systems.*

[28] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23âĂŞ52. https://doi.org/10.1145/42201.42203

[29] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. (2018). arXiv:1802.05799 [cs.LG]

[30] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, and Ashish Vaswani. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. *CoRR* abs/1811.02084 (2018). arXiv:1811.02084 http://arxiv.org/abs/1811.02084

[31] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[32] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. 2019. Low-Memory Neural Network Training: A Technical Report. *CoRR* abs/1904.10631 (2019). arXiv:1904.10631 http://arxiv.org/abs/1904.10631

[33] J.D. Ullman. 1975. NP-Complete Scheduling Problems. *Journal of Computer , System Sciences.* 10, 3 (June 1975), 384–393.

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, RÃľmi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[35] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. *CoRR* abs/2005.08314 (2020). arXiv:2005.08314 https://arxiv.org/abs/2005.08314