Summary of "RAMMER: Enabling Holistic Deep Learning Compiler Optimizations with rTasks"

Drake Svoboda (drakes), Anshul Aggarwal (aanshul)

Problem and Motivation The computation of deep neural networks (DNNs) can be represented as a data-flow graph (DFG) of operators and their data dependencies. Modern deep learning frameworks schedule these DFGs to execute on hardware accelerators like GPUs. There are two levels of available parallelism in this representation. At the highest level, multiple operators can be scheduled in parallel if they are data independent; this is known as **inter-operator** parallelism. At a lower level, individual operators (like matrix multiply) typically have inherent internal data parallelism called **intra-operator** parallelism. Hardware accelerators like GPUs are can be used to leverage this inherent parallelism.

Existing frameworks like Tensorflow exploit **inter-operator** parallelism by emitting a schedule of operators that can be executed when their data dependencies are met. These frameworks treat the hardware accelerator as a black-box and rely on a second layer of scheduler (implemented in lower level libraries like cuDNN or directly in the hardware) to schedule each operator to the accelerator to exploit **intra-operator** parallelism. The two layer scheduling approach does not efficiently exploit the available parallelism. There are two points of inefficiency. (1) Two layer scheduling has overheads that reduce accelerator usage. Existing frameworks like TensorFlow and PyTorch emit operators to be run by the accelerator at runtime. If the accelerator can execute the operators faster than they are emitted, then the program is bottlenecked and the accelerator will be underutilized. (2) The two layer approach does not consider the interplay between both levels of parallelism. Figure 1 shows an example where two operators are scheduled to an accelerator. In the two layer approach, the intra-operator scheduler greedily selects a kernel for the first operator that exhausts the accelerator's available execution units blocking the execution of other data independent operators. A holistic scheduling approach could instead choose to share the accelerator's resources across multiple operators reducing overall execution time.

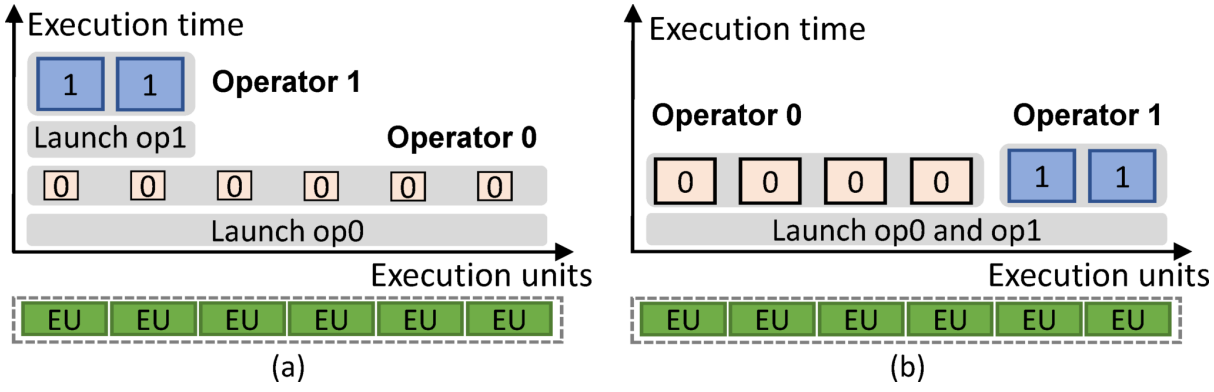


Figure 1: An illustration of (a) the inefficiency scheduling in existing approach; and (b) an optimized scheduling plan.

Hypothesis A holistic scheduling approach at both the inter- and intra-operator level can improve utilization and performance across DNN workloads.

Solution Overview The main contribution of this paper is clarifying the problem of scheduling DNNs by defining their computation using powerful abstractions. In this section we will first describe how the authors abstracts the computation of a DNN, then we describe RAMMER, a DNN compiler that schedules DFGs to exploit both levels of available paralellism.

Each DNN operator is abstracted as an rOperator consisting of multiple homogeneous rTasks; for example, a matrix multiplication rOperator can be represented as a set of rTasks that each compute a different tile in the output matrix. The hardware accelerator is abstracted as a vDevice with multiple homogeneous virtual execution units (vEUs) that can each execute rTasks independently. Thus, a scheduler using these abstractions can schedule fine-grained rTasks holistically leveraging both levels of parallelism.

The authors demonstrate the power of their abstractions by implementing RAMMER, a compiler that schedules rTasks. RAMMER provides two scheduling interfaces, `Wait` and `Append`. `Append` can be used to schedule an rTask to a vEU given their global ids. `Wait` can be used to wait for a particular rTask to execute. `Wait` implicitly appends a special **barrier-rTask** that is used to wait for a set of rTasks to finish execution.

RAMMER implements a *Wavefront Scheduling Policy* using these interfaces. The fringe nodes of the DFG, whose data dependencies are met, are selected as a *wave* of operators to be scheduled. RAMMER chooses the fastest rKernel for each operator in the wave if there are available resources; otherwise, it performs profiling to choose the most efficient set of rKernels for the wave. Thus, RAMMER considers the interplay of inter- and intra-operator parallelism for the operators within a wave. After the kernels are selected, each rTask belonging to these kernels is scheduled to a vEU by selecting the vEU that can execute the rTask at the earliest. Since scheduling is done at compile time, many of the overheads associated with runtime scheduling are eliminated.

RAMMER's abstractions can be implemented for multiple accelerators including Nvidia GPUs, AMD GPUs, GraphCore IPUs and x86 CPUs. The authors describe in detail how RAMMER is implemented for Nvidia GPUs using CUDA. rOperators and rTasks can be implemented in CUDA code; the key difference between an rTasks written in CUDA and traditional CUDA code is that an rTask uses an `rtask_id` managed by RAMMER's software scheduler rather than a `blockIdx` managed by the GPU's hardware scheduler. To implement vEUs on CUDA GPUs, RAMMER uses a presistant thread block (PTB). A PTB contains a group of continuously running threads that can be pinned to a streaming multiprocessor (SM). RAMMER compiles a DFG into a CUDA program where each thread in the PTB executes rTasks according to the sequence specified by the scheduler. There is some additional complexity needed to execute heterogeneous rTasks and to implement barrier-rTasks.

RAMMER shows significantly reduced runtime and increased utilization for many DNNs over existing frameworks. Figure 2 compares model inference time between RAMMER's CUDA implementation and existing frameworks. The improvement is significant especially when batch-sizes are small and when there is a lot of available inter-operator parallelism. RAMMER does not improve performance for DNNs without much inter-operator parallelism like AlexNet. RAMMER also has considerably higher GPU utilization with a roughly $4\times$ improvement on average over Tensorflow. The authors also show that RAMMER has considerably less scheduling overhead than runtime schedulers.

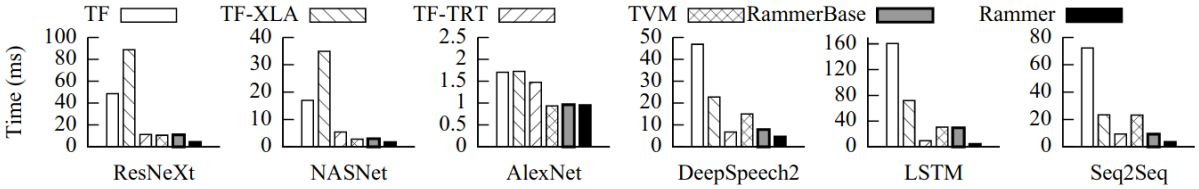


Figure 2: End-to-end model inference time with batch size of 1 on NVIDIA V100 GPU.

Limitations and Possible Improvements RAMMER requires that a DNN's DFG is static and available at compile time. Compile time scheduling reduces the overheads of run-time scheduling; however, I'm not sure what would limit these abstractions to static DFGs. The authors suggest that RAMMER could compile each static sub-graph of a dynamic DFG separately and leave this idea to future work.

Performance improvement is predicated on the DFG having significant inter-operation parallelism; this is not the case for many models like AlexNet. This is a really minor limitation since the performance on these benchmarks is roughly the same as the best performing alternatives.

The authors focus their evaluation on inference, however, they do show improvement for a single training workload. They note that nothing is limiting RAMMER from model training except requiring that more kernels be implemented as rKernels. Future work could develop more operators and evaluate performance improvements on training workloads.

Additionally, different scheduling algorithms could be used to further boost performance.

Summary of Class Discussion A piazza poster asked about the downsides of compile-time scheduling. The major downside is that the DFG for the neural net must be static and available at compile-time.

There was also a question regarding the *operator emitting* overhead mentioned in the paper. One key problem the paper tries to address is that if an accelerator can execute operators faster than the CPU emits them to be run, then the accelerator will be underutilized and performance is bottlenecked. This problem is pronounced when batch sizes are low and operators can be computed quickly by the accelerator. TensorFlow and PyTorch both schedule operators at runtime. Before the accelerator can begin execution the operator must be selected when its dependencies are met, the appropriate kernels dispatched to the GPU, GPU memory allocated, and the function arguments prepared. By scheduling at compile time, we can reduce some of these overheads.

During class, the professor asked who decides which specific kernel to use for an operator. Existing solutions use low level library like cuDNN. In RAMMER, the software scheduler chooses a specific rKernel.

The professor also asked who creates an rKernel. These kernels have to be programmed manually for each operator. RAMMER uses a source code transformation parser to transform existing kernels to rKernels.

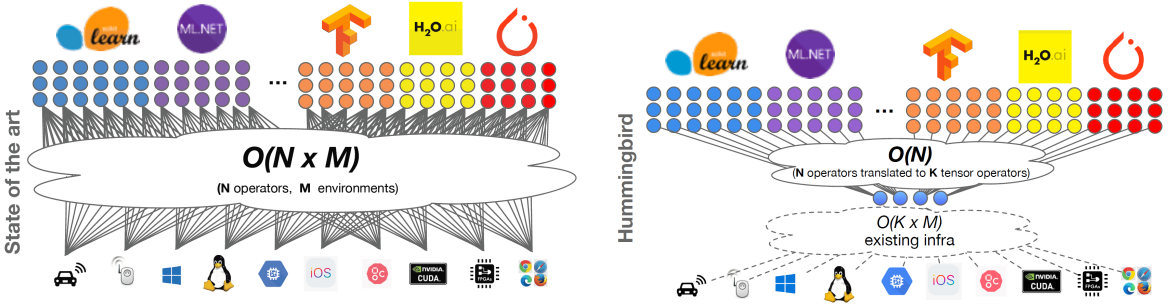
Summary of "A Tensor Compiler for Unified Machine Learning Prediction Serving"

Drake Svoboda (drakes), Anshul Aggarwal (aanshul)

Problem and Motivation Machine learning can be broadly classified into 2 categories—"traditional ML" and "deep neural networks". The majority of ML applications fall under traditional ML. One popular traditional ML framework is scikit-learn which is used about 5 times more than PyTorch and TensorFlow combined. Additionally, many DNN frameworks are extending their scope to cover traditional ML algorithms.

In general, ML models are trained only once but are used multiple times for scoring i.e. obtaining prediction from a trained model on some new data. Model scoring accounts for 45-65% of the total cost of ownership of data science solutions, and hence is an important concern for enterprises using ML.

The design of a traditional ML model consists of not just the model, but a predictive pipeline comprising of operators which fall into 2 main categories—*models* (decision tree ensembles, linear models, etc.) which are fit to the data and *featurizers* (string tokenization, normalization, etc.) which may be stateless or fit to the data. The entire pipeline is required for scoring. Figure 3 shows how this leads to a huge increase in the number of implementations required to support different frameworks on different platforms (deployment environments). Further, the number of libraries being used in data science applications has increased 4x over the last 2 years, which is directly correlated to the growth in the number of operators in different frameworks (N in figure 3a).



(a) state-of-the-art (b) HUMMINGBIRD

Figure 3: Prediction serving complexity

Finally, most traditional ML implementations are optimized for CPUs and do not take advantage of the investments in neural network compilers and hardware accelerators.

The authors of this paper describe a novel model scoring framework—HUMMINGBIRD (HB)—which compiles featurization operators and traditional ML models into small tensor operators. This allows the compiled tensor operators to make use of the existing neural network compilers and hardware accelerators, while reducing the complexity of the overall infrastructure and the burden of supporting different frameworks on different platforms.

Hypothesis Operations in traditional ML algorithms can be re-formulated as tensor operations to leverage existing optimizations for DNN compilers and specialized hardware. Not only

could this improve runtime, but could also reduce software complexity and improve model portability.

The main advantage the authors achieve is that the tensor computations can be executed over DNN frameworks. This allows HB to leverage the existing optimizations in DNN compilers, runtimes and specialized hardware.

Solution Overview HB relies on the key observation that once a model is trained it can be represented as a prediction function mapping input features to a prediction score which depends on the model (e.g., 0 or 1 for binary classification) but is independent of the training algorithm used. This observation can be naturally extended to featurizers in traditional ML pipelines. Hence, HB only has to compile the final prediction function parameters into tensors (and not the training logic) for each operator in a pipeline.

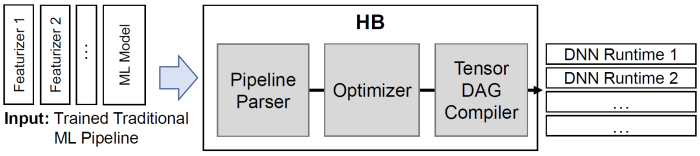


Figure 4: High-level architecture of HUMMINGBIRD

HB has 3 main components as shown in figure 4:

1. **Pipeline parser** – processes the operators one at a time and selects an extractor functions to extract parameters for each operator (weights of a linear regression, thresholds of a decision tree, etc.). At the end of the parsing phase, the input pipeline is represented as a directed acyclic graph of operators storing all the information required for the successive phases.
2. **Optimizer** – executes the extractor functions attached to each operator and then annotates it with a compilation strategy to compile that operator into a tensor operator.
3. **Tensor DAG** – uses the compilation strategies specified by the optimizer to convert the DAG into a small PyTorch neural network module composed of a small set of tensor operators that is exported into the target runtime format.

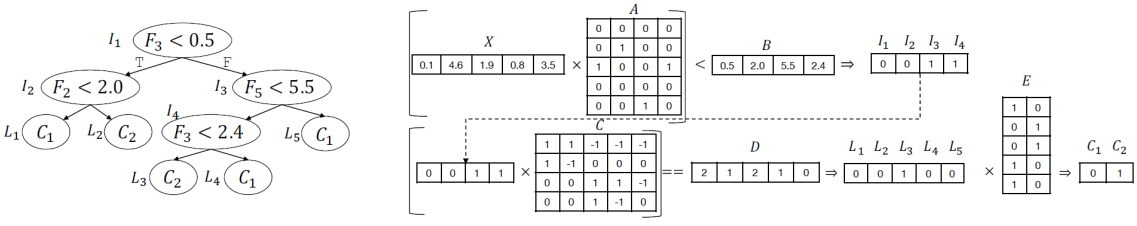


Figure 5: Compiling a decision tree using GEMM

The key challenge for HB is translating the algorithmic operators into tensor operators since algorithmic operators perform arbitrary data accesses and control flow decisions (which can look very different for different inputs), while tensors rely on bulk operations over the entire set of input elements. The authors present methods to convert algorithmic operators into tensor computations that are both correct and efficient. At the time of publication, HB supported

40+ sklearn operators. Some operators are straightforward to convert, other have existing methods.

The authors describe in detail 3 novel strategies for decision tree operators:

1. **GEMM**: They cast the evaluation of a decision tree as a series of three GEneric Matrix Multiplication (GEMM) operations interleaved by two element-wise logical operations as shown in figure 5.
2. **Tree Traversal (TT)**: They reduce the computational redundancy as compared to GEMM by mimicking the typical tree traversal but implemented using tensor operations. The traversal is implemented using the `gather` and `where` operators.
3. **Perfect Tree Traversal (PTT)**: This strategy mimics TT but assumes the decision tree to be a perfect binary tree. This provides some optimizations for shallow and small trees.

The strategy used for a particular decision tree is chosen using a set of heuristics. The authors derive these heuristics by comparing the three strategies for different tree depths and discuss their implementations and speedups on CPUs and GPUs.

The authors also present two strategies for optimizing trained end-to-end ML pipelines for inference:

1. **Feature selection push down** – push featurization steps down in the pipeline to avoid redundant computations such as scaling and one-hot encoding for discarded features. However, this might not always work, for example in the case of "blocking" operators such as normalizers.
2. **Feature selection injection** – add implicit feature selectors, for example L1 regularization performs implicit feature selection in linear models and pruning strategies can be added to trees for similar feature selection.

Limitations and Possible Improvements The authors mention some limitations of HB. First, HB does not work well with sparse data; this is evident from the experimental results. Second, not all operators are supported. For example, text feature extraction such as `TfidfVectorizer` is not supported. Lastly, HB is currently limited to a single GPU memory execution Given the expansion of distributed GPU frameworks being developed for DNNs, it is very important for HB to add this functionality in the future.

Summary of Class Discussion A possible limitation mentioned on Piazza is the use of HB for models that learn over time or have some online training schedule. HB assumes the model learns only once. Our understnading is that HB specifically avoids porting any training related information. This requires the entire model to be rebuilt every time its parameters are updated. Hence, it might not work at all for models that are updated very frequently or have some sort of feedback loop for training.