DUET: A Compiler-Runtime Subgraph Scheduling Approach for Tensor Programs on a Coupled CPU-GPU Architecture

Minjia Zhang*
Microsoft
minjiaz@microsoft.com

Zehua Hu*[§]
Beijing University
t-zehhu@microsoft.com

Mingqin Li
Microsoft
mingqli@microsoft.com

***Abstract*—Deep neural networks (DNNs) are currently the foundation for many artificial intelligence tasks. Existing DL frameworks and compilers often focus on optimizing DL inference speed against CPUs and GPUs in isolation while missing the opportunities to reap the benefits of aggregated computation power from both CPU and GPU. We show that there are DNNs that exhibit complex computation patterns, and different components might be suitable for executing on different types of devices to maximize performance gains. Based on this observation, we present a DNN inference engine, called DUET, that explores potential concurrent execution opportunities on heterogeneous CPU-GPU architecture for DNN inference. In particular, we introduce (i) a coarse-grained partitioning strategy that divides a DNN computation graph into subgraphs that retain high computational granularity with relatively low communication volume, (ii) a compiler-aware profiling method to include DL compiler optimization into the loop to improve scheduling decisions, and (iii) a greedy-correction subgraph scheduling algorithm that automatically maps the DNN computation to CPU and GPU without input from model developers. We evaluate DUET against several DNNs that exhibit complex model structures and compare its performance against existing DL frameworks and the state-of-the-art DNN compiler. The experiment results show that DUET is much faster than existing DL frameworks and obtains 1.5–2.3 times and 1.3–6.4 times speed-ups against the optimized code by the state-of-the-art DNN compiler on GPU and CPU alone, respectively.**

***Index Terms*—DNNs, Efficient Inference, Compiler, Heterogeneous Execution**

I. Introduction

The development of deep neural networks (DNNs) is driving an explosion in multiple artificial intelligence domains, such as computer vision, natural language processing, and speech recognition. However, the ability of DNN comes at the cost of high computational complexity. In order to achieve high efficiency for DNN on existing general-purpose hardware such as CPU and GPU, popular deep learning (DL) frameworks such as TensorFlow [10] and PyTorch [32] resort to incorporate highly optimized kernels via hardware vendors such as Nvidia cuDNN [14] or Intel MKL-DNN [3] as backend. However, when a tensor operator is not supported by the pre-optimized vendor library, the computation efficiency decreases dramatically. To keep up with the pace of the fast innovations in DNNs, major players in the industry develop highly optimized in-house operators [43], [47]. However, this raises challenges because of the increasing complexity of tensor operations in DNNs and the volatility of DL algorithms. The complexity further increases when multiple versions of the same model have to be optimized to deploy on different hardware platforms.

Motivated by improving the agility of optimizing DNNs, there has been a great interest in developing automated frameworks to handle the unprecedented amount of innovations. Notably, recent research has developed neural network compilers, such as XLA [9], Halide [35], Glow [38], Tensor Comprehension [41], and TVM [12]. Many of them use static analysis to find pipelined operations that can be fused together for improved performance and generate platform-dependent code efficiently for models trained through popular DL frameworks.

While DL compilers produce highly optimized code for DNNs, existing works often focus on optimizing DNNs for CPUs, GPUs, and other accelerators in isolation. However, servers with coupled CPUs and GPUs are now ubiquitous in data centers and cloud environments. From a practical point of view, combining CPUs and GPU means that the computation power provided by the CPU and the GPU can be aggregated to improve DNN performance. Despite the potential advantages of this strategy, there is, to the best of our knowledge, no DL compiler that can reap the benefit of this approach.

On another aspect, while the standard DNNs consist of a linear task dependence chain with more or less homogeneous components, e.g., ResNet [17] and VGG [39], there is a large number of DNNs that exhibit more complex model structures and diverse computation patterns. For example, some models exhibit higher fan-outs [29], [31], [44], implying more potential for parallel execution, while others contain sub-networks that have very different characteristics [13], where some components are more suitable to execute on one type of device than the other, implying potential benefits for heterogeneous execution. Existing DL compilers lack efficiency in handling these DNNs. For example, they often employ a *Operators-in-Sequence* scheduling, where an operator runs, presumably using multiple threads, but one operator starts running only after the previous one finishes. They also lack the necessary abstractions to support hardware-conscious inference execu-

[§]Work performed as a Microsoft intern. * Equal contribution.

tion over multiple devices to exploit all the resources available on a single server.

To address these limitations, we propose DUET, a DNN inference engine, which supports heterogeneity-conscious and compiler-aware DNN inference on a coupled CPU-GPU architecture. Our main contributions are the following:

- We make the case for heterogeneity- and compiler-aware DNN inference and present DUET, an engine design for concurrent execution of DNN computation on heterogeneous hardware.
- We introduce a coarse-grained partitioning strategy that allows the partitioned subgraphs to retain high computational granularity with relatively low communication volume.
- We show that involving the DL compiler in the heterogeneity optimization loop is beneficial for improving scheduling decisions.
- We introduce a greedy-correction subgraph scheduling algorithm, which automatically partitions the work of DNN inference between the CPU and GPU without input from model developers.

DUET builds on top of a unified graph-level IR and TVM [12], which is framework agnostic. It also makes the hardware implementation and the scheduling invisible to DL/ML practitioners, avoiding the developing cost of algorithms specialized for heterogeneous hardware. If a model does not have much intrinsic parallelism and cannot have performance improvements through the subgraph scheduling, DUET *falls back* to the original best-performing single device execution. The experimental results show that DUET achieves 1.3–6.4 times speed-ups on CPU and 1.5–2.3 times speed-ups on GPU, respectively, on three DNNs – Wide-and-Deep [13], Siamese Network [31], and MT-DNN [29], against the state-of-the-art DNN compiler.

II. Background

A. DNN Inference

The general life-cycle of DNNs from its birth to deployment comprises two major stages. The first stage is the designing and the training of a DNN by a model scientist, with the primary goal of achieving the highest feasible accuracy. The second stage is the deployment of the pre-trained DNN to a target hardware (often on GPU or CPUs [20], [43], [47]) to benefit end users, often done by a deployment engineer. These two stages are iterative processes: model scientists iterate until it reaches the target performance in terms of accuracy, whereas the deployment engineers iterate until the inference speed satisfies a latency SLA (e.g., often a few milliseconds per query). These two stages are most often separate processes, and optimizing the performance of DNNs to meet stringent latency targets can be very time-consuming. Therefore, the goal of DNN inference optimization is to minimize the model inference time while improving optimization agility to accelerate the overall deployment cycle.

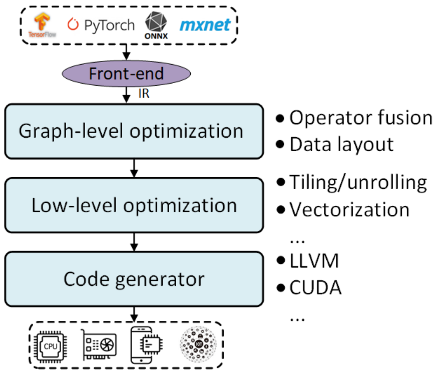

Fig. 1. The compilation pipeline for DNNs.

B. DNN Compilation

There has been recent work on optimizing DNN performance through *DL compilers*, which emit optimized code that runs the model efficiently on a target hardware [9], [12], [35], [38], [41]. These DL compilers work in the context of high-level DNN specifications, provided by deep learning frameworks such as TensorFlow [10], PyTorch [32], and MXNet [11]. The optimization passes are applied at different stages of the compilation process. Fig. 1 shows a typical processing flow of these DL compilers, which consists of five layers: 1) front-end, 2) intermediate representation (IR), 3) graph-level optimization, 4) low-level optimization, and 5) back-end.

The front-end transforms high-level DSL of DNNs into compiler-specific IRs. These IRs are usually in the form of data flow graphs, in which each node represents a tensor operator, and each edge denotes the data dependency between operators. Based on these IRs, graph-level optimizations can be applied to fuse operations and optimize data layouts. Low-level optimizations perform hardware-dependent optimizations (e.g., tiling size, vectorization) against the fused operators to improve data locality and utilization of the target hardware (e.g., CPU or GPU). Finally, the back-end is responsible for generating hardware-dependent executable instructions using LLVM (for CPU) or CUDA (for GPU).

III. Challenges and Motivations

This section first discusses the challenges of performing DL inference on both CPU and GPU efficiently, and then it presents several studies that guided the design of the approach introduced in Section IV.

A. Challenges

First, the CPU and GPU have very different device characteristics. CPU has a smaller number but faster cores, which typically have out-of-order execution with sophisticated branch predictions and deep cache hierarchies for reducing memory access latency. In contrast, GPU has many but slower cores, which are typically in-order and hide memory latency by switching between hardware threads. This dissimilarity leads to significant differences in their execution performance. Given that different NN components may have different computation patterns during inference, some components may execute significantly faster on one device than the other. As a result, executing even a small amount of work on the slower device may hurt performance. Moreover, although there is a big improvement in PCIe bandwidth, CPU-GPU communication may still create a performance bottleneck.

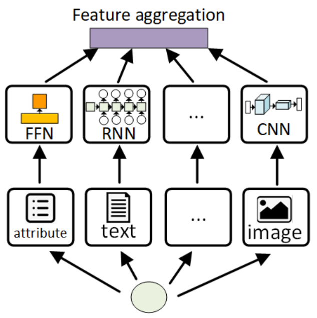

Fig. 2. The architecture of Wide-and-Deep network for heterogeneous contents encoding.

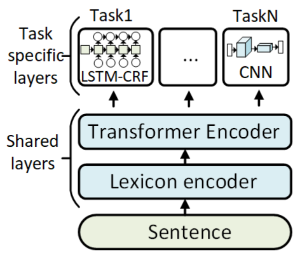

Fig. 3. The architecture of MT-DNN for natural language understanding.

Second, the DNN execution is heavily affected by the model characteristics, framework, compiler optimization, and the hardware (i.e., the HW/SW stack). Without accurate information about a NN component's behavior on a target hardware, it is hard to optimally divide work between the CPU and GPU automatically. Previous work use FLOPs to approximate the execution time of operators [40]. However, FLOPs is often an inaccurate proxy as operations with the same FLOPs can result in very different latencies on CPU and GPU, and factors such as compiler optimizations may change the execution time significantly as well. Recent work proposes to use profiling to characterize the performance of NN models [28]. However, there have been fewer studies on how to leverage the profiled statistics to make informed DNN heterogeneous execution decisions.

Third, existing DL compilers often contain inefficiencies in parallel execution, making executing on both CPU and GPU challenging. For example, we observe that TVM [12], the state-of-the-art DL compiler, employs a sequential execution schedule of computation graphs, where the executable operators are executed synchronously in topological order. This scheduling strategy generally works well for models with a sequential chain of tasks, such as ResNet [17], VGG [39], and SqueezeNet [23]. However, the structure of DNNs is more diverse and complex than just a sequential chain. For example, Fig. 2 shows the structure of a Wide-and-Deep network [46], which combines convolutional neural network (CNN), recurrent neural network (RNN), feed-forward neural network (FFN), etc. for heterogeneous contents encoding, and Fig. 3 shows a multi-task DNN model [29] used for natural language understanding. Both contain independent submodules, yet existing DNN frameworks often miss the opportunity of executing these independent components concurrently without violating dependencies.

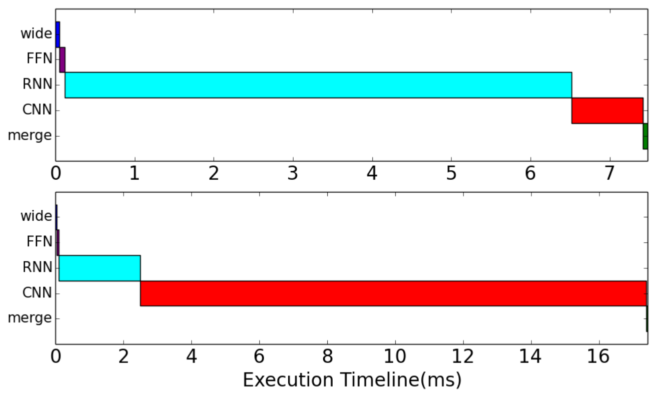

Fig. 4. The execution timeline of Wide-and-Deep models on GPU (upper) and CPU (lower).

B. Opportunities

Despite the aforementioned challenges, we still identify opportunities for heterogeneous computation of DNN inference on both CPU and GPU.

First, DNN computation exhibits diverse patterns, and GPU is not always faster than CPU for DNN inference. Although DNN training is often conducted on GPUs, which is throughput-oriented, CPU sometimes provides competitive performance compared to GPU for DNN inference. During inference, latency is the most important metric. The batch size at inference is often small (e.g., one or at most a few), which limits the amount of parallelism that can be leveraged by the massive cores on GPU. Furthermore, operators that contain sequential dependencies, such as recurrent neural networks (e.g., LSTM [19], GRU [15]), are also difficult to parallelize on GPU across sequential steps. Figure 4 shows an example of the execution timeline of the Wide-and-Deep [13] model using TVM on both CPU and GPU. As shown, although GPU takes less time to execute the model, the RNN execution time on GPU is much longer than on CPU. Simply executing the model on CPU does not work either because the CNN computation is extremely slow on CPU. Existing DL frameworks do not explicitly optimize for execution on both CPU and GPU, resulting in sub-optimal performance in this case.

Second, it is possible to make good use of aggregated computation capacity without incurring too much CPU-GPU inter-device communication delay. To validate the performance characteristics of the CPU-GPU communication, we use a micro-benchmark to measure the bandwidth and latency of CUDA point-to-point bulk transfer with respect to different message sizes. We conduct the experiment on a machine with Intel Xeon Gold 6152 CPU, Titan V GPU, connected through PCIe 3.0. From the results shown in Figure 5, the latency increases almost linearly as the message size increases. This increased communication latency can limit the amount of communication links between CPU and GPU. However, given that the latency delay (e.g., from passing inputs/outputs of tensor operators) is often less than tens of milliseconds, which

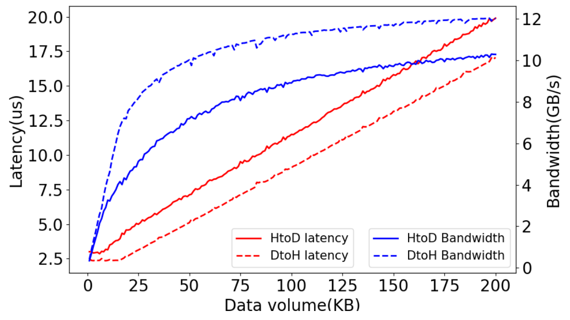

Fig. 5. Communication cost between CPU and GPU device.

is orders of magnitude smaller than many NN operator (e.g., LSTM, CNN) execution time, we may minimize the impact from the communication overhead by retaining a relatively high computational granularity.

Finally, coarse-grained partitioning allows the DL compiler to perform more graph-level optimizations that improve single device efficiency. Graph-level optimizations have been demonstrated to be one of the best ways to achieve lower execution time on a single device. As an example, the fusion pass fuses multiple operators in the computation graph and generates a rewritten graph with fused ops to improve the temporal data locality and computation intensity. By exploiting the observation that graph-level optimizations help improve single device efficiency, we can minimize the effect of heterogeneity by partitioning the NN computation into coarse-grained subgraphs that may still benefit from the compiler graph-level optimizations.

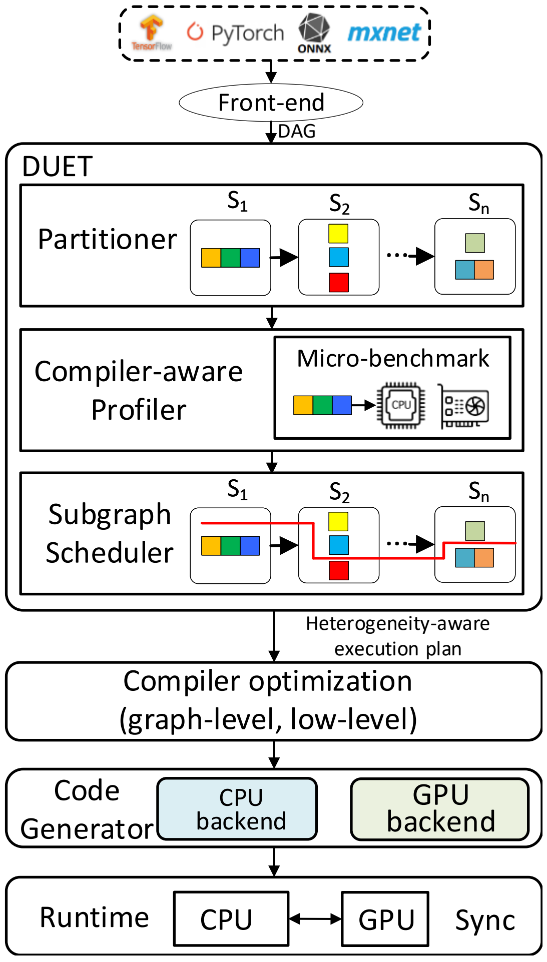

Fig. 6. The DUET Architecture.

IV. Compiler-Aware Heterogeneous DNN Inference

DUET allows DNN inference to take advantage of heterogeneous hardware present in modern servers by encapsulating heterogeneity and CPU-GPU parallelism. DUET is composed of three major parts, as shown in Figure 6. The input of DUET is a pre-compiled DNN model. The first part is a coarse-grained graph partitioner, which divides the DNN computation graph into multiple subgraphs that still allow DL compiler to apply graph-level optimizations. The second part is a compiler-aware profiler, which is responsible for providing the runtime execution statistics of subgraphs based on compiler optimized code on target devices. The third part is a profiling-based online subgraph scheduler, which maps subgraphs to their specialized hardware based on the actual run time.

A. *Coarse-Grained Multi-Phase Graph Partitioning*

DNN inference computations is often transformed into compiler-specific IRs, in the form of directed acyclic graphs (DAG). For a given DAG G, each node $v_i \in G$ is an operator (e.g., matmul, softmax) in the DNN, and each edge $(v_i, v_j) \in G$ establishes a dependency between the output of operator v_i and the input of operator v_j. A valid execution schedule of the DAG determines an execution order of its nodes that satisfies all the dependencies.

In this work, we consider those valid schedules that are composed of phases: A phased schedule executes a DAG in a sequence of phases $S_1, S_2, S_3, ..., S_t, ...$, where each phase S_t represents a non-overlapping subset of nodes and $S = \sum_t S_t$ consists of all nodes. There is a total ordering between phases such that if $t < t'$, then all nodes in S_t must be executed before $S_{t'}$. We further divide a phase into two categories: i) If a phase consists of a subgraph with a sequential chain of operators, we call it a *sequential phase*. ii) Otherwise, if a phase contains multiple independent subgraphs, we call it a *multi-path phase*. Phases can be either sequential or multi-path, and phase_type(S_i) != phase_type(S_{i+1}). Figure 7 shows an example of a schedule with three phases, where S_1 and S_3 are sequential phases and S_2 is a multi-path phase.

According to the definition of the phased schedule, DUET partitions a DAG into multiple subgraphs[1]. A *schedule* S on a coupled CPU-GPU architecture includes a mapping for each

[1]Please note that it is possible to have a nested partition of subgraphs. However, doing so will decrease the computation granularity and incur more CPU-GPU communication overhead. For simplicity, we assume a one-level partition scheme, and we leave multi-level partitioning as future work.

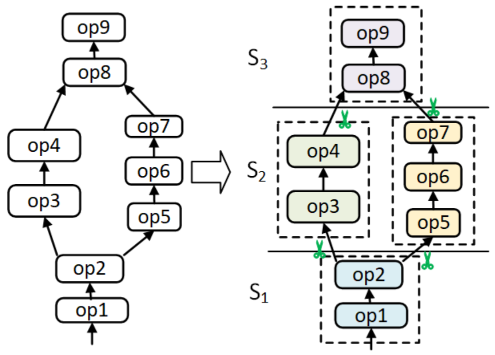

Fig. 7. The multi-phase execution schedule.

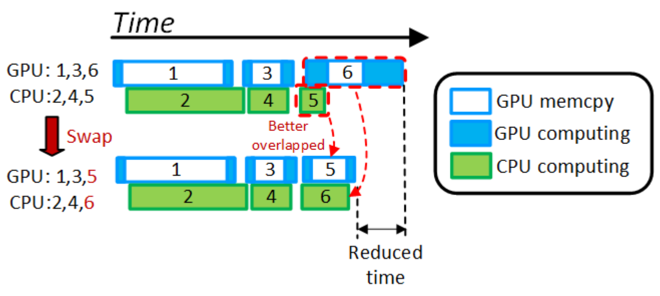

Fig. 8. The greedy-correction scheduling for a case of concurrently executing multiple subgraphs on CPU and GPU.

subgraph to either CPU or GPU while satisfying dependencies. Our goal is to find a parallelization schedule S such that the per inference time is minimized.

When doing partitioning, we note that there are cases where multiple nodes consume the same input, i.e., a shared node in the DAG. We handle this situation by creating replicated placeholders in different branches but let them all point to the same input stream.

B. Profiler for Compiler-Optimized Subgraphs

Existing DL frameworks such as TensorFlow provides profilers to profile model execution time. However, these profilers assume a general, non-optimized compiler, and the profiled statistics are quite different from the true statistics from compiler optimized code, which do not help make informed decisions. Low-level profilers are provided by NVIDIA's nvprof [4] or Intel's VTune [2]. However, these are at the hardware and kernel execution level, which do not easily map to the execution of DNN subgraphs.

To obtain accurate execution time of each subgraph in the DAG, we take an end-to-end approach and build a profiler for compiler-optimized subgraphs. For a given subgraph, the profiler builds a micro-benchmark by treating that subgraph as a standalone DNN model and going through the DL compilation pipeline, including generating the target-dependent code through the back-end. The profiler then runs each micro-benchmark on both CPU and GPU for several runs and records the information, including start and end time, the input/output data size, as well as the device running it. The execution time helps improve the scheduling decision. The input/output data size helps analyze the communication overhead. We note that profiling is only done during the offline phase and is therefore a one-time cost.

C. Greedy-Correction Subgraph Scheduling and Mapping

Given the profiling results, the next step decides how to map each subgraph to CPU and GPU. To better tackle the unpredictable variations at run time, we introduce a profiling-based online scheduling scheme: *greedy-correction*, where DUET prioritizes the subgraphs in the critical path of the DAG and then corrects the decision based on the actual running time. The approach consists of three steps:

Step 1. Placing the critical path on the fastest device(s). The critical path is the path in the dataflow graph that has the longest computation time from source to sink vertex. Speeding up the processing of the critical path, therefore, would speed up the overall computation time of the dataflow graph. For each subgraph in the sequential phase, DUET prioritizes to select the device (e.g., between CPU and GPU) that the subgraph has a faster execution time to be on the critical path. For subgraphs within a multi-path phase, we select the shortest execution time between CPU and GPU as the *cost* of a subgraph, and add the subgraph that has maximum *cost* in that phase to the critical path.

Step 2. Greedily placing remaining subgraphs to CPU and GPU. At this step, we sort the subgraphs by their execution time on different devices. Then, for each independent subgraph in a multi-path layer, we make the placement decision in the order of the sorted subgraphs list. In each iteration, we place the subgraph to the device that minimizes the increase of the critical path.

Step 3. Correcting the placement decision by considering the communication cost. The first two steps provide an initial placement, but it might not be optimal because it does not consider the potential added communication overhead. In this step, we refine the previous decisions by performing series of experiments to understand which subgraph should be placed on which device, and how to arrange the computations so that the communication is optimized. In particular, we use iterative refinement to fine-tune the placement, in a way that swapping the subgraphs from one device to another to minimize the execution time. Figure 8 shows an example of an initial subgraph placement of GPU = {1,3,6} and CPU = {2,4,5}. The algorithm maintains and improves a schedule, In one pass, the algorithm selects a pair of subgraphs (6,5) in GPU and CPU so that switching the paired subgraph from one side of the device to the other will optimize the performance (e.g., by avoiding some excessively high communication overhead between CPU and GPU). Notes that one of the subgraphs could be empty, which represents moving only a single subgraph to the other side. The correction step terminates when x round of swaps are performed without decreasing the execution time.

Note that we perform the third step for each multi-path layer, so it may need to run multiple times as we may have several multi-path layers in a model. Algorithm 1 provides the details of the subgraph scheduling. We remark that this idea of subgraph refinement is similar to the Kernighan-Lin refinement in existing literature on graph partitioning, which dates back to 1970's [27]. Different from KL refinement, which finds equal-sized subsets with the minimal edge-cut, the criterion we use is to minimize latency. We also note that it is possible to analytically decide the placement strategy based on the profiled subgraph computation and communication cost, similar to the dynamic programming based method [24]. However, profiling communication in existing DL frameworks often introduces estimation errors, due to potential inefficiencies or unexpected behavior [33]. Therefore, we take an approach that refines the subgraph placement based on actual end-to-end latency.

D. Executor

Once the scheduling decision has been made, DUET instantiates an executor to run the decided schedule, as shown in Figure 9. The executor spawns two child processes to run compiled subgraphs concurrently on CPU and GPU [2]. Each process works in a busy loop: it polls for input data from its own synchronization queue, executes the corresponding subgraph, and triggers the subgraph's dependencies. The synchronization queue is implemented as a *shared memory* queue for high efficiency.

V. Implementation

DUET is built on top of TVM [12]. The main reasons for choosing TVM are its wide adoptability for DL inference optimization and its support for multiple DL frameworks. However, the implementation can also be migrated to other deep learning compilers.

TVM uses Relay as an intermediate representation, which is a pure, expression-oriented language and employs the BNF Grammar [37], as shown in Listing 1. To facilitate graph partitioning and to debug, we perform a translation of this representation to an adjacency-list representation, as shown in Figure 10. In particular, we iterate the Relay IR using the visitor pattern and obtain the inputs/outputs of each operator to build a graph with adjacency-lists. We apply phased partitioning against this adjacency-list graph, and we translate the subgraphs back to a sequence of Relay statements, which can be readily optimized through the TVM compiler.

VI. Evaluation

A. Experiment Setup

Environment. Our evaluation is conducted on a server with a 2.10 GHz Intel(R) Xeon(R) Gold 6152 CPU processor and an NVidia TITAN V GPU, connected through PCIe V3.0 interconnect. The server has 128GB RAM, running 64-bit Linux Ubuntu 16.04.

[2]Please note that it is possible to further improve the performance by allowing multiple subgraphs to execute concurrently within one device (e.g., CPU). For simplification, we assume a sequential execution of subgraphs on CPU, because the small number of cores can be largely occupied by most subgraphs.

Algorithm 1 **Subgraph scheduling algorithm**

```
Input: A set of subgraphs obtained through the graph partitioning method described in Section IV-A.
Output: Fine-tuned subgraphs placement $S_{cpu}$ and $S_{gpu}$
Step 1: Placing the critical path on the fastest device.
Step 2: Greedily placing the remaining subgraphs to CPU and GPU.
Step 3: Refining the subgraph placement decision. For each multi-path phase, assume that the subgraphs are separated into sets $S_{cpu}$(subgraphs placed to CPU) and $S_{gpu}$(subgraphs placed to GPU),
$T_{old} \leftarrow$ measure_latency($S_{cpu}, S_{gpu}$)
do
    $gain \leftarrow 0$
    $S'_{cpu} \leftarrow S_{cpu}$, $S'_{gpu} \leftarrow S_{gpu}$
    while $S'_{cpu} \neq \emptyset$ or $S'_{gpu} \neq \emptyset$ do
        find swapping pairs of subgraphs ($s_i \in S'_{cpu}, s_j \in S'_{gpu}$ or movement of individual subgraph) that maximize the reduction of the expected latency
        $S'_{cpu} \leftarrow S'_{cpu} - s_i, S'_{gpu} \leftarrow S'_{gpu} - s_j$
        $T_{new} \leftarrow$ measure_latency($S_{cpu} - s_i + s_j, S_{gpu} - s_j + s_i$)
        if $T_{new} < T_{old}$ then
            $S_{cpu} \leftarrow S_{cpu} - s_i + s_j$
            $S_{gpu} \leftarrow S_{gpu} + s_i - s_j$
            $T_{old} \leftarrow T_{new}$
        end if
        $gain \leftarrow max(T_{old} - T_{new}, gain)$
    end while
while $gain > 0$
```

Workloads. The experiments compare the inference speed on three neural networks. The first one is Wide-and-Deep Network [13], which is trained with wide linear layers and deep neural networks together and can simultaneously have the benefits of memorization and generalization as well as heterogeneous contents encoding, with a lot of applications in recommender systems [16], [22], [48]. We choose the open-sourced PyTorch implementation based on [7], which has a structure that consists of wide linear layer, FFN, RNN, and CNN, as shown in Figure 2.

The second is Siamese Network [31], which is a neural network with two independent RNN branches used for similarity ranking (e.g., similarities between queries and passages). We choose the TensorFlow implementation from [1].

The third one we use is a Transformer-based neural network called MT-DNN [29]. The model is used for natural language understanding. It has a shared layer that consists of a lexicon encoder and a multi-layer bidirectional Transformer encoder. It then has an arbitrary number of task-specific output layers, which are independent from each other, as shown in Figure 3.

Table I shows the model parameters used for the evaluation. We choose batch size of 1 to represent a common case in DNN inference. For RNNs, sequence lengths refer to maximum sequence length. To make reliable measurement, we run each

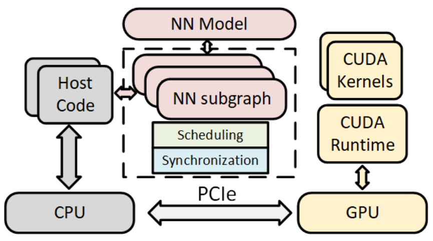

Fig. 9. The heterogeneous execution engine.

Listing 1. Example of Relay expressions

```
fn (%Const: Tensor[(1, 2)],
  %Const_1: Tensor[(2, 1)],
  %Const_2: Tensor[(1, 2)],
  %Const_3: Tensor[(2, 1)])
  -> Tensor[(1, 1)] {
    %0 = transpose(%Const_1, axes=[1, 0]);
    %1 = nn.dense(%Const, %0);
    %2 = transpose(%Const_3, axes=[1, 0]);
    %3 = nn.dense(%Const_2, %2);
    add(%1, %3)
}
```

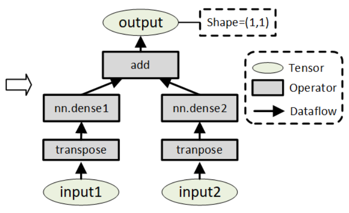

Fig. 10. An example of translated graph representation.

Model	Parameters
Wide-and-Deep	Wide: –input_wide (1,363) FFN: –input (1,13); –embedding 192; –hidden [128,64] CNN: –input (1,3,224,224); –model ResNet-18 RNN (stacked LSTM): –input (1,80); –num_layers 2; –hidden 64
Siamese Network	2 Stacked LSTM: –input (1,64); –sequence_length 64; –num_layers 2; –hidden 128
MT-DNN	Stacked LSTM: –embedding 768; –sequence_length 32; –layers 2; –hidden 256 Num_tags (10 CRF): 29,3,3,7,9,11,9,25,7,13

TABLE I
THE MODEL PARAMETERS OF WIDE-AND-DEEP, SIAMESE, MT-DNN.

	Subgraph	**Comp. Time**		**Placement**	
		CPU	GPU	CPU	GPU
Wide-and-Deep	Wide	0.03	0.05		✓
	FFN	0.07	0.07		✓
	RNN	2.4	6.4	✓	
	CNN	14.9	0.9		✓
	merge	0.03	0.06	✓	
Siamese Network	merge3	0.03	0.05	✓	
	Stacked-RNN-1	2.74	3.22		✓
	Stacked-RNN-2	2.72	3.24	✓	
MT-DNN	bert-base	289.7	7.8		✓
	LSTM_CRF-1	3.18	2.04		✓
	LSTM_CRF-2	3.21	2.03		✓
	LSTM_CRF-3	3.2	2.03		✓
	LSTM_CRF-4	3.19	2.05		✓
	LSTM_CRF-5	3.18	2.03	✓	
	LSTM_CRF-6	3.19	2.04	✓	
	LSTM_CRF-7	3.17	2.05	✓	
	LSTM_CRF-8	3.18	2.04		✓
	LSTM_CRF-9	3.19	2.03	✓	
	LSTM_CRF-10	3.2	2.04		✓

TABLE II
COMPUTATION COST AND DEVICE PLACEMENT DECISIONS.

configuration 5000 times to report average and tail latency, with the warm-up time excluded.

Comparison framework. We compare the performance with TVM, which is widely accepted as the state-of-the-art compiler for DNN inference, on both CPU and GPU. We also include comparison with the original PyTorch [32] or TensorFlow [10] implementation. We let the framework to decide the appropriate number of threads used for computation.

B. DNN Inference Performance Comparison

Figure 11 shows the execution time of different models by PyTorch/TensorFlow, TVM, and DUET. We make the following observations. First, DUET achieves 1.5–2.3 times and 1.3–15.9 times speed-ups compared with TVM-GPU and TVM-CPU, respectively. DUET achieves speed-ups because it optimizes by leveraging the computation power from both CPU and GPU. Second, DUET is significantly faster than the inference time using existing DL frameworks, achieving 2.1-8.4(on GPU) times and 2.3-18.8(on CPU) times speed-ups than TensorFlow/PyTorch on GPU and CPU, respectively. DUET offers much higher performance improvements than DL frameworks, because it combines heterogeneous execution with DL compiler optimizations to maximize the gains on CPU-GPU.

Computation cost breakdown and placement decisions. To see why DUET obtains speed-ups compared with using just GPU or CPU, Table II shows the computation cost (column 3 and 4) and final scheduling decision (column 5 and 6) of subgraphs (column 2) from the three models. The computation cost is collected through the DL compiler-aware profiler (Section IV-B), where a fixed, small number of profiling runs (e.g., 500) is sufficient to obtain statistically stable measurement. As shown, in the first line of Table II, For Wide-and-Deep, the RNN subgraph takes 2.4ms on CPU but 6.4ms on GPU, while the CNN subgraph takes 14.9ms on CPU but only 0.9ms on GPU. Due to this heterogeneity compute pattern, running the model either on CPU or GPU entirely does not lead to the optimal latency. In contrast, DUET exploits hardware heterogeneity and place subgraphs to their suitable hardware, which minimizes the overall end-to-end latency.

Tail latency. For online inference, tail latency is as important, if not more, as the mean latency. To see if DUET provides steady speed-ups, we collect the 50th (P50), 99th (P99), and 99.9th (P99.9) percentile latency at batch size 1 from running TVM-GPU and DUET on the three models, as shown in Figure 12. The results show that in most cases, the P99 and P99.9 latencies only increase moderately, and DUET obtains 1.3–2.4 times and 1.1–2.1 times speed-ups against TVM-GPU at P99 and P99.9, respectively. The speed-ups at P99.9 is slightly smaller, especially for MT-DNN, because the CPU-GPU interconnect communication adds additional performance variation.

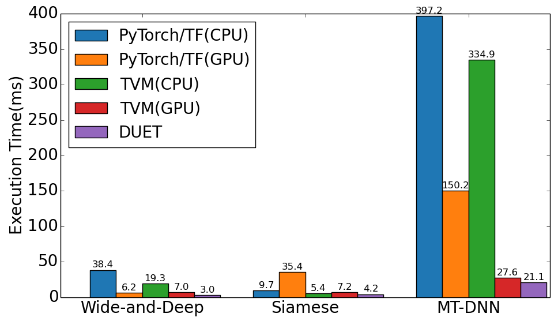

Fig. 11. The end-to-end latency of different frameworks on Wide&Deep, Siamese and MT-DNN model.

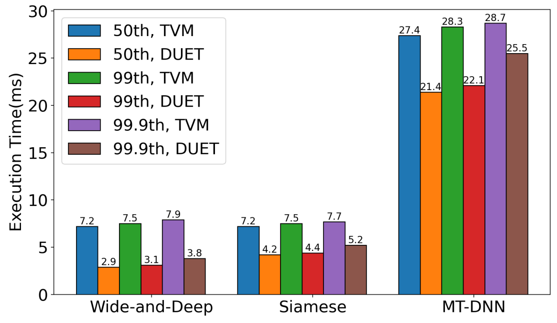

Fig. 12. The comparison of 55th, 99th, 99.9th percentile latencies between TVM(GPU) and DUET (CPU-GPU) on Wide-and-Deep, Siamese network, and MT-DNN.

C. Comparison of Scheduling Algorithms

In this part, we evaluate the effectiveness of DUET's subgraph scheduling algorithm by comparing the following schemes:

- Random: randomly assigns a subgraph to devices.
- Round-Robin: assigns subgraphs to CPU and GPU alternatively.
- Random + Correction: first randomly assigns subgraphs and then performs the correction (step 3) as described in Section IV-C.
- Greedy + Correction: our scheduling algorithm described in Section IV-C.

We use Wide-and-Deep as an example. Fig. 13 presents the model execution time from the above schedules. We observe that both Round-robin and Random scheduling yield relatively higher execution time than the two correction-based scheduling algorithms. This is expected as the former two schedule subgraphs in an arbitrary topological order, where a global optimization strategy cannot be imposed. In contrast, the two correction-based scheduling algorithms yield much lower execution time, because they make subgraph placement decisions by taking into account the execution time of subgraphs and GPU-CPU communication cost. We choose greedy-correction because the greedy placement provides a good initialization for the correction algorithm, which requires

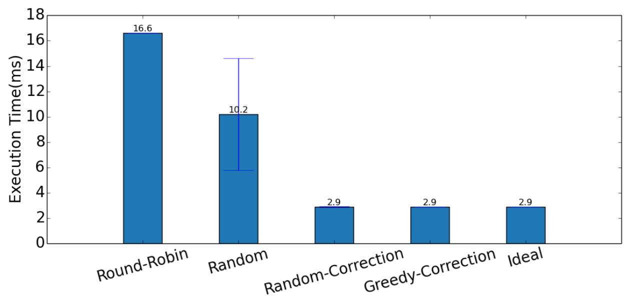

Fig. 13. The comparison of execution time with different scheduling algorithms.

fewer iterations for the correction algorithm to converge. To verify the correctness of our scheduling algorithm, we enumerate all possible schedules (which may not be always feasible given that finding the optimal schedule is NP-hard) to find the exact optimal schedule (*Ideal*). We empirically show that the greedy-correction methods finds the optimal schedule, at least when the number of subgraphs is relatively small.

D. Evaluation of Model Variations

In practice, model scientists may constantly experiment with new model architectures, e.g., varying depths of a model, and it is important to have the agility to adapt to different architectures for DNN inference optimization. In this part, we evaluate how DUET performs when the model architecture of Wide-and-Deep changes.

Varying the stacked RNN layers. Figure 14 shows the comparison of the execution time, varying the number of stacked RNN to have 1, 2, 4, 8 layers. Compared with TVM-GPU and TVM-CPU, DUET achieves 2.3–2.5 times and 2.9–9.8 times speed-ups, respectively. The execution time of all configurations increases as the number of RNN layers increases, but the execution time on GPU increases more substantially. This is because RNN is relatively slow to execute on GPU. With GPU only, RNN computation becomes a dominant part of the execution time, creating a performance bottleneck as the RNN layers keep increasing. On the other hand, both TVM-CPU and DUET have a relatively slower increase in execution time as the stacked RNN increases its depths. However, DUET achieves a much lower execution time because it maps the CNN computation, which is slow to run on GPU, to GPU, resulting in overall reduced execution time.

Varying the CNN depths. Figure 15 shows the execution time of the same network, but varying the depth (e.g., 18, 34, 50, 101) of the ResNet encoder. This time, TVM-CPU observes a much larger latency increase as the ResNet increases its depth. This is because ResNet dominates the total execution on CPU. For DUET, the execution time remains almost the same when the depth of the ResNet is relatively small (e.g., 18, 24). This is because when CNN is shallow, the RNN computation on the CPU side dominates the total execution time, which can hide the computation of CNN on the GPU side. As the depth of ResNet keeps increasing, the

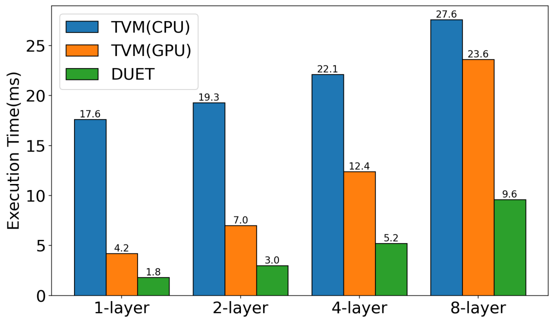

Fig. 14. Inference latency and speedup on Wide&Deep Model with different layers in the RNN component.

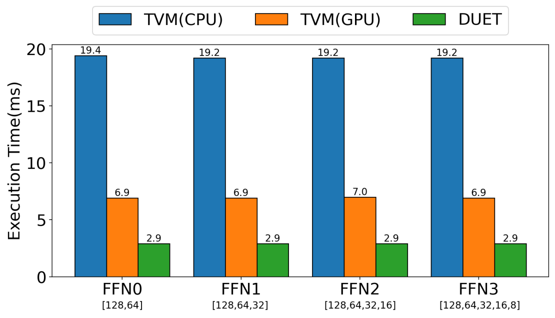

Fig. 16. Inference latency and speedup on Wide&Deep Model with different config in the Deep component.

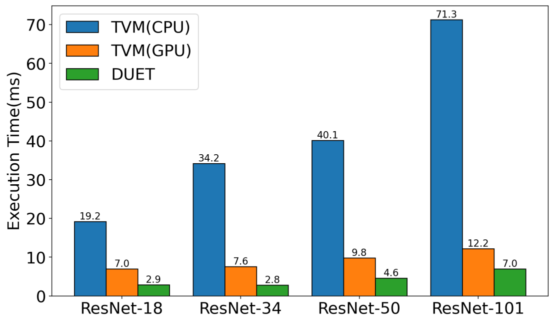

Fig. 15. Inference latency and speedup on Wide&Deep Model with different layers(18/34/50/101) in the CNN component.

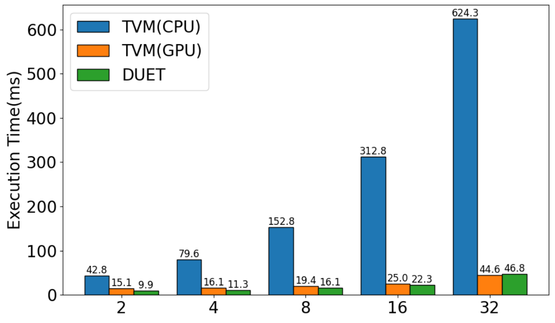

Fig. 17. Comparison of different configurations with different batch sizes.

execution time of both TVM and DUET starts to slowly increase, because as ResNet starts to dominate the computation, there are not enough parallel subgraphs to utilize both CPU and GPU simultaneously, resulting CPU being idle while the execution time on the GPU side keeps increasing.

Varying the FFN depths. Figure 16 reports the comparison of the execution time, varying the number of hidden layers in FFN. As shown, the execution time does not change much as we increase the FFN depths. This is because FFN consists of mostly GEMMs, which have been highly optimized on both CPU and GPU. As a result, FFN only takes a very small amount of execution time despite its hidden depth increases.

Varying the batch sizes. Another factor that may affect the effectiveness of DUET is the batch size of the input data. Since TVM does not support dynamic batch size yet, we freeze the model with a fixed batch size range from 2, 4, 8, 16, 32. Overall, as shown in Figure 17, the speed-ups from DUET are more pronounced when the batch size is small (e.g., 1.5 times speed-up at batch size 2) but gradually diminish as the batch size increases, compared with TVM-GPU. This is expected, because GPU is overall more suitable for large batch execution. However, as discussed earlier, batch size is often rather small for inference scenarios due to the stringent latency target.

E. Applicability to Traditional Models

So far, we have evaluated DNN workloads with complex structures that exhibit heterogeneity where existing DNN frameworks are less efficient to optimize. One may concern about how DUET would perform for existing models that have been well-optimized on a specific hardware. We conduct experiments on ResNet [17] and the results are shown in Table III. As shown, DUET offers the same performance as the best performing baseline, which is TVM-GPU in this case. This is expected, because not only ResNet has a relatively sequential structure but it also consists mostly of CNNs, which have been heavily optimized by TVM on GPUs. Given that the model is mostly sequential and does not present much heterogeneity, its partitioned subgraphs cannot be executed in parallel efficiently because it introduces additional communication overhead and the CPU does not make CNNs run faster. In this case, DUET falls back to the single-device execution mode and simply chooses the device where the model runs the fastest.

	PyTorch-CPU	PyTorch-GPU	TVM-CPU	TVM-GPU	DUET
Time(ms)	17.4	2.2	15.8	1	1

TABLE III
THE END-TO-END LATENCY ON RESNET-18.

VII. Related Work

DUET offers a solution that enables (i) heterogeneous execution, with compiler-aware subgraph scheduling, for (2) DNN inference. As such, we discuss the related work from these two independent research directions.

Heterogeneous DNN computation. Prior work studies using CPU memory as an extension of GPU memory to increase memory capacity for DNN workloads [18], [21], [25], [34], [36], [42]. However, most of these work target at optimizing the training process, such as improving the training throughput with larger model/batch sizes, whereas DUET focuses on optimizing the DNN inference, where latency is the most important metric and batch size is often just 1.

Mirhoseini et. al. [30] proposed to use reinforcement learning to learn efficient operator schedules for model parallelism. However, the scheduling is performed at the operator-level and assumes a general, non-optimized compiler, which prevents many graph-level compiler optimizations such as fusion. The use of RL, which requires to train a complex policy network with hyperparameter tuning, also makes it difficult to apply in practice. In contrast, DUET schedules computation at the subgraph-level, which simplifies the design space, while still allowing the DNN compiler to apply the majority of graph-level optimizations, increasing the computation efficiency on a single device.

In a more general context, heterogeneous computing has been studied to make well-orchestrated use of heterogeneous hardware to execute various application [26], [45]. While DUET is inspired by those prior works, unlike them, it is specially tailored for reducing the execution time of DNN inference.

DNN inference. There has been work on optimizing DNN inference through platforms, libraries, and compile-time strategies. Several platforms have been built to facilitate the deployment of DNN models, such as TensorFlow Serving [8], TensorRT [5], ONNXRuntime [6]. To the best of our knowledge, these platforms do not support heterogeneous DNN inference yet, and DUET can be integrated with these platforms to exploit concurrent execution opportunities on multiple devices.

There are libraries for accelerating DNN inference for a specific type of hardware, such as cuDNN [14] and MKL-DNN [3]. DUET can be combined with these libraries by incorporating them as a back-end. Finally, DUET serves as a middleware in between DL frameworks and a DNN compiler [9], [12], [38], [41], which allows an existing DNN to benefit from hardware heterogeneity.

VIII. Conclusion

Although DL compilers produce highly optimized code for individual hardware devices, a drawback is that they miss opportunities to allow DNNs to benefit from the aggregated computation power of CPU and GPU. We introduce DUET, a DNN inference engine that allows DNNs to explore potential concurrent execution opportunities on coupled CPU-GPU architecture. Powered by the coarse-grained graph partitioning, compiler-aware profiling, and a profiling-based online subgraph scheduling algorithm, DUET greatly decreases the end-to-end inference latency for DNNs that exhibit complex model structures and diverse computation patterns. We hope this work will encourage additional studies of heterogeneous execution on the DNN model online serving.

Acknowledgment

The authors appreciate the anonymous IPDPS reviewers for providing very constructive and useful feedback, which has significantly helped improving the quality of this paper.

References

[1] Deep LSTM siamese network for text similarity. https://github.com/dhwajraj/deep-siamese-text-similarity.

[2] Intel vtune. hhttps://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.htmll.

[3] Intel(R) Math Kernel Library for Deep Neural Networks. https://github.com/01org/mkl-dnn.

[4] Nvidia nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[5] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[6] Onnxruntime. https://github.com/microsoft/onnxruntime.

[7] pytorch-widedeep. https://github.com/jrzaurin/pytorch-widedeep.

[8] TensorFlow Serving. https://www.tensorflow.org/serving/.

[9] The Accelerated Linear Algebra Compiler Framework. https://www.tensorflow.org/performance/xla/.

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 265–283, 2016.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv preprint arXiv:1512.01274, 2015.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594, 2018.

[13] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, page 7–10, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. arXiv preprint arXiv:1410.0759, 2014.

[15] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv preprint arXiv:1412.3555, 2014.

[16] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, Xiuqiang He, and Zhenhua Dong. Deepfm: An end-to-end wide & deep learning framework for CTR prediction. *CoRR*, abs/1804.04950, 2018.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[18] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[20] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. GRNN: low-latency and scalable RNN inference on gpus. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 41:1–41:16. ACM, 2019.

[21] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Zhenhua Huang, Guangxu Shan, Jiujun Cheng, and Jian Sun. Trec: an efficient recommendation system for hunting passengers with deep neural networks. *Neural Comput. Appl.*, 31(S-1):209–222, 2019.

[23] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <1MB Model Size. arXiv preprint arXiv:1602.07360, 2016.

[24] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2279–2288. PMLR, 2018.

[25] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Trans. Archit. Code Optim.*, 15(3), September 2018.

[26] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *Proc. VLDB Endow.*, 10(7):733–744, 2017.

[27] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49(2):291–307, 1970.

[28] Cheng Li, Abdul Dakkak, Jinjun Xiong, and Wen-mei Hwu. Benanza: Automatic μbenchmark generation to compute" lower-bound" latency and inform optimizations of deep learning models on gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 440–450. IEEE, 2020.

[29] Xiaodong Liu, Yu Wang, Jianshu Ji, Hao Cheng, Xueyun Zhu, Emmanuel Awa, Pengcheng He, Weizhu Chen, Hoifung Poon, Guihong Cao, and Jianfeng Gao. The microsoft toolkit of multi-task deep neural networks for natural language understanding. In Asli Çelikyilmaz and Tsung-Hsien Wen, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, ACL 2020, Online, July 5-10, 2020*, pages 118–126. Association for Computational Linguistics, 2020.

[30] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2430–2439, 2017.

[31] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. Learning text similarity with Siamese recurrent networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*, pages 148–157, Berlin, Germany, August 2016. Association for Computational Linguistics.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[33] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects. In Varsha Apte, Antinisca Di Marco, Marin Litoiu, and José Merseguer, editors, *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*, pages 209–218. ACM, 2019.

[34] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 891–905, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, 2013.

[36] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, 2016.

[37] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new IR for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 58–68, 2018.

[38] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018.

[39] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[40] Raphael Tang, Ashutosh Adhikari, and Jimmy Lin. Flops as a direct optimization objective for learning sparse neural networks. *CoRR*, abs/1811.03060, 2018.

[41] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[42] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 41–53, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[44] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5987–5995. IEEE Computer Society, 2017.

[45] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. When computing meets heterogeneous cluster: Workload assignment in graph computation. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 154–163. IEEE Computer Society, 2015.

[46] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. Heterogeneous graph neural network. In Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 793–803. ACM, 2019.

[47] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 951–965. USENIX Association, 2018.

[48] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Comput. Surv.*, 52(1):5:1–5:38, 2019.