Overflowing Emerging Neural Network Inference Tasks from the GPU to the CPU on Heterogeneous Servers

Adithya Kumar
The Pennsylvania State University
University Park, USA

Anand Sivasubramaniam
The Pennsylvania State University
University Park, USA

Timothy Zhu
The Pennsylvania State University
University Park, USA

ABSTRACT

While current deep learning (DL) inference runtime systems sequentially offload the model's tasks on to an available GPU/accelerator based on its capability, we make a case for selectively redirecting some of these tasks to the CPU and running them concurrently with the GPU doing other work. This new opportunity specifically arises for emerging DL models whose data flow graphs (DFGs) have much wider fan-outs compared to traditional ones which are invariably linear chains of tasks. By opportunistically moving some of these tasks to the CPU, we can (i) shave off service times from the critical path of the DFG, (ii) devote the GPU for more deserving tasks, and (iii) improve overall utilization of the provisioned hardware in the server. However, several factors such as its criticality in the DFG, slowdown when moved to a different hardware engine, and overheads in transferring input/output data across these engines, determine the what/when/how of tasks to be directed. While this is computationally demanding and slow to be solved optimally, through a series of rationales we derive a fast technique for task overflow from GPU to CPU. We implement this technique on a nimble heterogeneous concurrent runtime engine built on top of the state-of-the-art ONNXRuntime engine and demonstrate $> 10\%$ reduction in latency, $> 19\%$ gain in throughput, and $> 9.8\%$ savings in GPU memory usage for emerging neural network models.

SYSTOR '22, June 13–15, 2022, Haifa, Israel

ACM ISBN 978-1-4503-9380-5/22/06…$15.00
https://doi.org/10.1145/3534056.3534935

CCS CONCEPTS

• **Computer systems organization** → Neural networks; **Heterogeneous (hybrid) systems**; • **Software and its engineering** → **Scheduling**.

KEYWORDS

DNN Inference, Heterogeneous systems

ACM Reference Format:
Adithya Kumar, Anand Sivasubramaniam, and Timothy Zhu. 2022. Overflowing Emerging Neural Network Inference Tasks from the GPU to the CPU on Heterogeneous Servers. In *The 15th ACM International Systems and Storage Conference (SYSTOR '22), June 13–15, 2022, Haifa, Israel.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3534056.3534935

1 INTRODUCTION

This paper adapts the evolution of modern system architectures to the characteristics of emerging Deep Learning (DL) applications. The growing popularity and adoption of deep neural networks [16] (DNNs) for making accurate inferences/predictions is leading to increasingly complex DL models. Specifically, tasks in emerging models such as Faster-RCNN [52] have larger Data-Flow Graph (DFG) fan-outs than conventional neural networks such as ResNet [23]. On the hardware side, heterogeneity continues to grow in modern servers [22, 48], which are provisioned today with GPUs, TPUs [34], FPGAs [11], and other accelerators [9] for meeting the demands imposed by neural network computations. This paper marries these two emerging trends by opportunistically mapping tasks in large fan-out neural networks to multiple devices within heterogeneous servers, thereby accelerating inference queries while boosting the hardware utilization.

DNNs have become commonplace for numerous applications - image and voice recognition [10, 58], natural language processing, conversational systems, and several scientific [30, 39] applications. Earlier generations of DNNs, such as the widely studied [8, 12] ResNet model [23], can be viewed as a linear chain of data dependent tasks. Each task by itself is quite demanding and is usually offloaded to a specialized accelerator (TPU, FPGA, etc.) or to one or more

GPUs with thousands of cores. Such an execution model, which runs one task after another on these specialized engines, is usually close to optimal for a linear task dependence chain. However, newer DNN models for these applications are growing in complexity to improve accuracy. For example, models such as FasterRCNN can no longer be viewed as linear chains of tasks. The DFG between its tasks exhibit higher fan-outs, implying more potential for parallel execution. This paper will show how to assign these tasks to execute in parallel across heterogeneous devices to improve performance over the conventional sequential execution style that is implemented in today's DNN execution engines such as ONNXRuntime [27], TensorRT [59], etc.

Today's servers offer an eclectic mix of compute engines beyond the many/multi-core CPUs. While software infrastructures/compilers [8, 38] exist that readily take high level DNN models and compile/synthesize them to such diverse hardware devices, the question still remains as to which hardware engine is the best for executing a given task when there are multiple such tasks waiting for available hardware. This would depend not only on the relative suitability of the hardware for the task but also issues such as "relative criticality" (of the task in the DFG) and inter-task communication. At the application level, there have been several proposals [8, 32] that try to optimize the graph of the model - such as fusing neighboring operators and pruning unused ones. Nevertheless, the optimized graph still needs an efficient runtime system to map its tasks to the diverse hardware engines.

We propose a heterogeneous execution runtime, which sits below the optimizing compilers (which generate backend kernels) and makes informed decisions based on runtime information to choose the *right device* for running the *right kernel* at the *right time* to minimize the overall model execution time. Today's runtime systems (e.g., ONNXRuntime [27]) execute kernels sequentially one after the other on specialized compute units such as the GPU. Our system augments this to explore parallel execution of the kernels/tasks across different (heterogeneous) computational units to speed up execution of the whole model.

The mapping of a neural model's tasks to heterogeneous hardware has been examined to some extent from a theoretical perspective in prior works [2, 41, 47]. In general, these approaches try to use the history of similar task executions (either in this model or in other models) to learn where they should be executed. While these studies focus more on learning techniques to solve the problem, this work aims to navigate the practical challenges in implementing a solution on a real inference engine.To our knowledge, this work is the first attempt to extend a real DNN execution engine to accommodate task assignment in a wide-spanning neural DFG onto heterogeneous compute engines of a server to leverage parallelism for inference workloads. While training is equally important, in this study, we mainly focus on inference workloads, which are user-facing and require careful resource allocation in order to meet stringent latency (especially tail latency) requirements. Our discussion and evaluation focuses on high-end servers containing multi-core CPUs and state-of-the-art GPUs (GPUs are the dominant secondary compute platform in today's datacenters [53]), though our work could be extended to include other devices in the future.

Considering the wide fan-out of emerging neural network DFGs, we show that several of its tasks spend a considerable time waiting for computation engines, even more than the time taken to execute them. While we could redirect/overflow *all* such waiting tasks opportunistically to the abundantly available surplus CPUs, we show that blindly doing so would simply shift the bottleneck there. Instead, we want to be selective about which tasks to redirect - specifically prioritizing tasks that have minimal slowdown when they get executed by the CPU instead of the GPU. Even doing this only marginally improves the overall performance. It is also important to consider the task communication costs since the overheads of moving the dependent data between the memories of the compute engines can be high. Based on these observations, we develop strategic execution plans for these wide-fan-out DFGs, called SIR+ [1] , that opportunistically leverages the CPU for selective tasks while boosting the utilization of the GPU itself. The key contributions of our work are:

- We study emerging wide fan-out DNN models and present the opportunity to parallelize their execution between under-utilized CPUs and the GPU on a heterogeneous server.
- The queueing times of tasks, the slowdown of tasks (on the CPU w.r.t. the GPU), and the communication between tasks are equally important criteria when deciding what tasks to redirect/overflow to the CPU. We develop an effective and fast algorithm that leverages these criteria and show that this can significantly reduce the execution time while boosting GPU utilization. We also show that our approach performs close to an optimal placement solution derived from a constrained optimization problem formulation.
- We build the necessary system support by extending the popular production inference engine (ONNXRuntime [27]) to implement SIR+. We also show SIR+ is extendable to TensorRT. This requires the concurrency design to be fine-tuned at the lower levels in order to make this work efficiently. The challenges posed by μs scale tasks of these

[1] SIR+ enables the use of "Surplus" (CPU) resources for DNN Inference Runtime systems

neural networks, requires the use of agile lock-free approaches and memory saving techniques.
- Extensive evaluation across four different heterogeneous server platforms for three emerging DNN models (FasterRCNN, MaskRCNN, SSD) indicate that SIR+ provides ~15% latency savings under light load, upto 90% latency savings under high load, and 45%-79% savings in tail latency (at high load regions).
- SIR+ also provides higher throughput (by at least 19.8%) on a single GPU and (by at least 11%) on multi-GPU settings over the state-of-the-art ONNXRuntime. It dedicates the GPUs for more demanding tasks by shifting some of the load to the CPU, thus handling more load for the same responsiveness.
- By moving some tasks to the CPU, SIR+ also reduces the demand on "valuable" GPU memory by at least 9.8%. [2]

2 DNN INFERENCE

Platform: DNN inference workloads are usually run on servers with CPUs alone or in conjunction with GPUs and other accelerators to speed up tasks that are amenable to using the hardware [22, 48]. While there is no dearth of commodity CPUs in the datacenter, GPUs [53] are the most prevalent form of accelerators that are typically paired with these "brawny" CPUs. This makes our work particularly relevant today, and readily adoptable, for exploiting these two intra-server heterogeneous computing engines (CPU+GPU) for neural network inference services. The hardware platform, used in our discussions, comprises a server running Xeon CPU with an Nvidia Titan RTX GPU (See *PF*1 in Table 5 for full details). We use the Microsoft ONNXRuntime [27] execution engine, a state-of-the-art neural network execution platform that incorporates many optimizations [29] including compiling optimized tasks for both CPU and GPU devices. We will study both latency (i.e., the time taken to produce the result from the time a single query and its input is given to the server) as well as throughput (i.e., the number of queries served by the system per second) in terms of performance. Both metrics will not only be influenced by the service time to perform their respective tasks, but also by the waiting/queueing time to start executing them on the compute engine (either the CPU or GPU).

Emerging DNN models: Conventional neural models such as AlexNet/ResNet are well understood and extensively studied in the systems community [8, 12, 13, 18, 24, 32]. In contrast to these models, recent proposals in the ML/Vision community [36, 47, 61, 65] indicate an emergent trend towards complex models with inherent differences in the structures of their data-flow graphs (DFG). On comparing traditional ResNet like models which have a low (in/out) degrees to emerging DNN models such as FasterRCNN, we observe a clear distinction in the fan-out characteristics of the DFGs of the emerging class of neural networks. While the fan-out of traditional models is as low as <= 4 for ResNet50, with most of them having just 2 edges (1 incoming and 1 outgoing edge), it is as high as 83 incoming and/or outgoing edges for emerging DNN models such as in the case of FasterRCNN. This wide fan-out characteristic of emerging models offers the possibility of executing several paths of the DFG that could be executed in parallel without violating dependencies. As explained earlier, our interest in this paper is beyond the well studied conventional neural models, and instead, we focus on the inherently more complex ones. We will use the FasterRCNN (trained on COCO [37] images) model in the next few sections to illustrate the issues behind our solution.

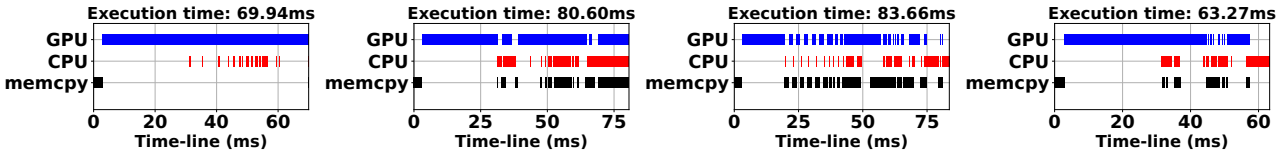
[2] SIR+ is available at https://github.com/minus-one/sir_plus.

3 EXISTING SEQUENTIAL EXECUTION

DNN inference computations can be depicted as a Data-Flow-Graph (DFG) of executions. A neural network execution engine, such as ONNXRuntime, takes this DFG and executes the nodes (tasks) of the DFG one after another in a sequential fashion while obeying the data dependencies (e.g., using a topological ordering of the graph). A static plan of the execution is created when the model is first loaded onto the engine. The plan contains the order of execution of all the nodes as well as the choice of the device (CPU/GPU) to be used for each node when serving a query.

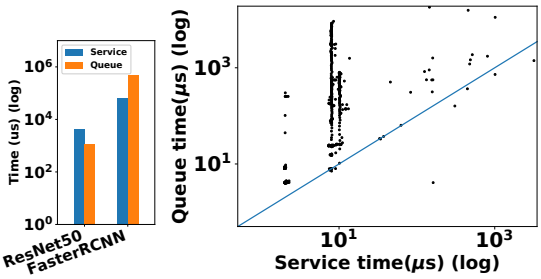
Figure 1a visually depicts the execution of the FasterRCNN model for the baseline ONNXRuntime as a Gantt chart. As can be seen, most nodes (tasks) of the DFG for this model run on the GPU with very few nodes (< 10% of the tasks) relegated to the CPU. While this is shown specifically for the ONNXRuntime, other neural execution engines such as PyTorch [49] also execute these models similarly. Execution engines that sequentially schedule these DFG nodes would perform (near) optimally for linear style DFGs (such as ResNet50), but fall short for emerging DNN models such as FasterRCNN.

To illustrate this, we plot the task wait time - referred to as *queueing time (Q.T.)* henceforth - representing the delay in starting a task after it becomes ready to execute (i.e., once all its input data has become available). We can clearly see in Figure 2a that cumulative Q.Ts are a lot shorter for ResNet50 compared with FasterRCNN, which has very high Q.Ts (18% of the nodes must wait). Consequently, the current practice of scheduling the sequential chain of tasks one by one on the GPU would be close to optimal for older style models (e.g., ResNet50). But in newer style models (e.g., FasterRCNN), one could possibly do better by scheduling the tasks in parallel using other available computer engines instead of waiting for the GPU.



(a) Baseline (b) Queueing aware (QD) (c) Slowdown aware (SD) (d) Comm+SD (Proposed)

Figure 1: Gantt chart of executing FasterRCNN on a system with 1 CPU and 1 GPU using different placement strategies.



(a) Cumulative (b) Q.T. vs. S.T. - FasterRCNN

Figure 2: Queueing times (Q.T) and Service times (S.T)

On closely examining the Q.Ts of FasterRCNN tasks in Figure 2b, we see that *many tasks are waiting longer for the GPU than the time taken to service them.* This suggests that one could selectively move these tasks to the largely idle host CPU cores to reduce the make-span of the DFG.

4 SIR+: CONCURRENT DNN EXECUTION

We next explore different strategies to help alleviate the high queueing delays experienced by tasks in the DFG by opportunistically utilizing the CPU for their execution while concurrently executing other tasks with the GPU. Note that runtime environments such as ONNXRuntime come up with a static plan of the order and placement to execute the tasks a priori, and we would like to simply modify this plan so that selected tasks get re-routed/overflowed for our desired goals. SIR+ profiles the execution and applies a combination of strategies to optimize the future re-routes of the tasks. As these models are deployed to run for extended periods, we simply use a short duration before model deployment for profiling the baseline execution (Figure 2b).

Placing and scheduling the tasks (nodes) of the DFG of a DL model is an NP-hard problem [26, 57]. We formulate and present details of this problem in Section 5. Considering DFGs of emerging DNN models with hundreds/thousands of tasks (and this number is only expected to increase with newer models), these problems take an excessive amount of time (order of days) to produce optimal solutions. Instead, we first seek fast techniques before comparing them to the MILP formulation. In the forthcoming discussions, we assume a batch-size of 1 for the inference as these newer models are large enough to fully utilize the GPU and higher batch-sizes generally incur higher latencies.

Scheme	Tasks ⟶ CPU	Avg. Q.T. (μs) GPU	Avg. Q.T. (μs) CPU	Avg. Q.T. (μs) Overall	Time (μs) Overall
Baseline	0	188	16	172	69.94
QD_{all}	544	244	2136	670	213.46
QD	519	236	504	255	80.6

Table 1: Performance breakdown for QD vs. Baseline.

4.1 Queueing Delay-based Overflow to CPU (QD)

Rationale: The question that arises is *which tasks waiting on the GPU must overflow to the CPU.* A straw-man is to overflow all the long waiting tasks to the CPU rather than wait for the GPU.

Strategy: The naïve approach is to move *all tasks* with a non-zero queueing delay (18% of tasks in the DFG) to the CPU (denoted as QD_{all}). This might simply shift the bottleneck to the CPU and defeat the whole purpose of employing a GPU for compute intensive tasks. To avoid this problem, rather than divert all tasks with non-zero queueing delays to the CPU, we migrate only those tasks that can complete earlier on the CPU. We estimate and compare the task finish times on the CPU and the GPU and move it to the CPU if the former is earlier than the latter. That is, if the service times (S.T) of the task on the CPU < (Q.T + S.T) on the GPU, we re-route the task to the CPU. This enhanced policy - termed as QD - selectively overflows tasks based on whether they can finish earlier on the CPU as compared to their original completion time on the GPU.

Results with this Strategy: Table 1 shows the results on employing the naïve QD_{all} and the enhanced QD techniques to re-route tasks from the GPU to the CPU. As surmised, migrating *all* tasks which queue up on the GPU (QD_{all}) dramatically worsens the execution time as the migrated tasks take significantly more time on the CPU. Surprisingly, even selectively choosing a subset of tasks which finish earlier on the CPU (termed as QD) results in a worse performance than the baseline sequential execution.

In general, there are 2 contrasting drivers. On the one hand, moving from the GPU to the CPU can reduce the task's Q.T. to execute and also ease the load on the GPU for another waiting task. On the other hand, moving a lot of tasks to the CPU can shift the bottleneck there, causing even longer waiting times as the CPU is usually less efficient at servicing a task.We see that for both the naïve QD_{all} and the enhanced *QD* policy, the performance worsens giving credence to the importance of the latter effect.

Figure 1b shows the Gantt chart for the *QD* strategy. As a consequence of higher CPU utilization (compared to the baseline Figure 1a), tasks now start to queue on the CPU. Although the Q.Ts of the migrated tasks may have reduced, this comes at the cost of a possibly higher Q.T for other tasks both on the CPU and GPU. The overall Q.T (Table 1) shows a marked increase. These tasks, and those dependent on them, get inordinately delayed because of this new schedule, resulting in an overall increase in latency for the query - from 69.94ms in the baseline to 80.6ms with this approach.

These results suggest that while we can move tasks off the GPU to reduce their waiting time, it is not profitable overall as they cause other tasks to suffer from significant slowdown.

4.2 Slowdown based Overflow to CPU (SD)

Rationale: As we carefully analyze the slowdown for each task and avoid the inordinate slowdown caused by moving them to the CPU, we observe the following about candidate tasks that can be moved to the CPU. **Not all tasks *equally* slow down when they move from the "faster" GPU to the "slower" CPU**. For FasterRCNN we see that the spread of the slowdown is very large - ranging from less than $10^0\mu s$ to as large as $10^4\mu s$. Compute heavy tasks like *Convolution*, *Add*, and *GEMM* incur large slowdowns (order of $10^4\mu s$). On the other hand, a substantial chunk (≈80-90%) of computationally light-weight tasks like *NonMaxSuppression*, *Softmax*, *Square-Root*, etc., take almost the same time on the CPU and the GPU by leveraging the SIMD capabilities of the CPU (when applicable). The top tasks contributing to the slowdown (e.g., Convolution, GEMM, Relu, etc.) account for a small number of tasks, while there is a long tail of tasks that have much smaller contribution to slowdown. This encourages us to apply the *SD* strategy, which moves these long tail of tasks to the CPU.

Strategy: A clear trade-off is emerging when performing re-routes of tasks to the CPU from the GPU. Redirecting too many tasks can shift the bottleneck to the CPU while redirecting too few tasks will achieve a similar result to the baseline. We go against the *conventional wisdom of always preferring the faster device and explore this trade-off by nudging tasks with marginal slowdowns to the CPU.* We start with a conservative threshold of $100\mu s$ (motivated by GPU kernel launch overheads that are usually in the region of 10-100 of μs) and assign tasks with a lower S.T increase less than this threshold to the CPU.

Scheme	Time (μs) CPU	Time (μs) GPU	Time (μs) Memcpy	Util. (%) CPU	Util. (%) GPU
Baseline	370	66597	2974	0.53	95.25
SD	16515	47816	34848	18.36	57.54

Table 2: SD vs. Baseline. Migrating short tasks boosts CPU utilization but also introduces costly memory copy operations.

Results with this Strategy: As we can see from the Gantt chart in Figure 1c, the CPU is clearly more utilized than the baseline. Table 2 gives a detailed breakdown of the time spent on each device. Although we have only migrated the tasks with a marginal slowdown, both the time spent on the CPU and its utilization (column 2 and 5 in Table 2 respectively) have been boosted significantly. Unfortunately, the performance of this strategy is still worse than the baseline sequential execution. This is explained as follows. Until now, we have considered each task to be self-contained, with its service time dependent only on the type of the computation engine (CPU or GPU) where it runs. However, tasks operate on *input* data to produce *output* data, and the costs of data transfers depend on where these tasks are run. When data needs to be transferred between devices (from GPU memory to host RAM or vice-versa), there is an additional cost to perform this transfer (using DMA engines over the PCIe bus) rather than simply accessing the local memory as is the case when they are running on the same device. In fact, if we consider the time spent in data transfers (see column 4 in Table 2 and in the Gantt charts of Figure 1c), we see that a considerable amount of time ($\approx 41\%$) elapses in performing such data transfers before the tasks can run, thereby increasing the execution time in this scheme.

While selectively choosing tasks which do not suffer an increase in S.T. is imperative to improve the utilization of CPU, it is additionally important to consider the "cost of data transfers" while migrating tasks to the CPU.

4.3 Communication+Slowdown aware Overflow to CPU (Comm+SD)

Rationale: We set out with a goal to find fast and profitable techniques to overflow tasks to the CPU and observed that data transfer costs play a significant role in determining task placements. Consequently, we develop an algorithm (Algorithm 1) - which runs in the order of seconds - to take communication costs into consideration on top of the SD strategy.

Strategy: The SD policy (Section 4.2) already subsets candidate tasks for running on the CPU based on their increase in

Algorithm 1 Comm+SD

Input: G = DFG of the model with N tasks
$T[] = T_i$ is the choice of device for executing task i
$\mu[i][d]$ = Service time (S.T.) of task i on device d
F_i = Device on which task i has lowest service time
$Data[i][j]$ = Data transfer size from task i to task j
$Rate[d1][d2]$ = Data transfer rate from $d1$ to $d2$
α = Threshold for switch to a slower device ($100\mu s$)
max_iter_count = Maximum number of iterations
Initialize $iter\_count = 0$
repeat
  for $i \in$ Topological ordering(G) **do**
    $CommTime = 0$
    for $j \in Pred(G, i)$ **do**
      if $T_j \neq T_i$ **then**
        $CommTime += Data[j][i] * Rate[T_j][T_i]$
      end if
    end for
    for $j \in Succ(G, i)$ **do**
      if $T_j \neq T_i$ **then**
        $CommTime += Data[i][j] * Rate[T_i][T_j]$
      end if
    end for
    $FastDeviceTime = \mu[i][F_i]$
    $CurrDeviceTime = \mu[i][T_i]$
    if $(CurrDeviceTime + CommTime) > (FastDeviceTime + \alpha)$ **then**
      $T_i = F_i$
    end if
  end for
  $iter\_count += 1$
until (T changes) && ($iter\_count < max\_iter\_count$)

Policy	All tasks to CPU	Common Tasks	Avg. Q.T. (us) Common Tasks	Avg. S.T. (us) Common Tasks	Comm. Cost (us)
SD	2316	2238	139	6	34848
Comm+SD	2238	2238	124	6	6293

Table 3: Performance breakdown, Comm+SD vs. SD.

S.T. We analyze the predecessors/successors of these tasks. If they are placed to run on the GPU, then the costs for the data transfer are included in the service time to re-evaluate whether this task still satisfies the SD threshold. We perform this iteratively till the set does not change any more (i.e., it converges). In our experiments, we find that it converges very fast (less than 10 iterations) and works very well even for large graphs (1000s of nodes).

Results with this strategy: Figure 1d shows the Gantt chart with this placement strategy, where we see a significant improvement in the overall response time compared to the baseline (by 9.5%) as well as compared to SD (by 24%). Of the 20.3ms (24%) improvement in the overall response time over SD, the reduction in communication overheads constitute the bulk, validating the need for considering communication overheads in placement. Table 3 compares these two schemes showing the average queueing and service times in the respective schemes for the tasks common to both these schemes. These values are nearly the same suggesting that the main savings are from the lower time spent in communication (shown in last column). The total size of the data flow - both in and out - for tasks that incur communication cost under the SD and CommSD schemes is 580MB (Avg.: 5.13MB) and 152MB (Avg.: 1.97MB) respectively. The *Comm+SD* policy deliberately avoids moving tasks with significantly large data-flows from the GPU to the CPU.

To effectively utilize all available devices on a heterogeneous server for executing the DFG of a DNN model, one must account for both service time differences (section 4.2) as well as the communication costs incurred on the data flows (section 4.3).

5 MIXED INTEGER LINEAR PROGRAM (MILP) TO MINIMIZE EXECUTION TIME OF DEEP LEARNING INFERENCE

We pose the problem to find the optimal placements for the tasks of a model as an MILP. Our goal is to determine the optimal placement for each task in a model's DFG consisting of V nodes (i.e., tasks of the model) on a heterogeneous server with D devices. The objective is to minimize the total execution time (ET) taken to complete all the tasks.

Let μ_i^d represent the time taken taken by task i to run on a device d. The binary variable D_i^d is used to indicate the placement of a node (i) on a specific device (d). We include variables ST_i^d to denote the time at which task i starts on device d if assigned to it, and FT_i^d as the time at which task i finishes on device d. We assume that μ_i^d is constant and known a-priori for all tasks i on all devices D, which is reasonable given that task service times on a device do not vary considerably. Data transfers between any 2 tasks (i and j) executing on the same device is assumed to incur zero cost, while the transfer between tasks running on different devices (d and l) is measured by a fixed transfer rate - $Rate_{dl}$. Dividing the size of the transfer $Data_{ij}$ by this rate would yield the transfer time. Table 4 lists all the variables used in the MILP model.

The goal of the optimization is to minimize the total inference time of the model as:

Objective: *Minimize ET*, where execution time is bounded by the completion time of the last task.

$$ET \geq FT_i^d \qquad (i \forall V)(d \forall D) \qquad (1)$$

The finish time of task i on device d is determined by adding the start time to the service time of this task i on device d

Variable	Description	Type
V	Nodes in the DL Model	Set
D	Devices available on the server	Set
ET	Execution Time	Variable
ST_i^d	Start Time of node i on device d	Variable
FT_i^d	Finish Time of node i on device d	Variable
D_i^d	Placement of node i on device d	Indicator
μ_i^d	Time taken by node i on device d	Constant
$Data_{ij}$	Tensor transfer size b/w nodes i and j	Constant
$Rate_{dl}$	Data transfer rate between devices d and l	Constant
$Pred(j)$	Nodes preceding node j in V	Constant set

Table 4: Model variables. Decision variables are inside the box.

(μ_i^d) multiplied by the decision variable D_i^d to ensure it is added only for the device on which it actually runs.

$$FT_i^d = ST_i^d + (D_i^d \times \mu_i^d) \qquad (i \forall V)(d \forall D) \tag{2}$$

We now describe the various constraints to enforce placement with exclusion guarantees (i.e., any device cannot service two tasks simultaneously at any time) and ordering constraints to respect dependencies specified between tasks.

Placement constraint: Each task should be assigned to exactly 1 device:

$$\sum_{d \in D} D_i^d = 1 \qquad (i \forall V) \tag{3}$$

Exclusion constraint: At any time, a device can execute at most 1 task (exclusion) and there is no preemption of tasks. Hence for every task i running on a device d there should be no other task j whose start time (ST_j^d) lies between its start time (ST_i^d) and its finish time (FT_i^d).

$$\neg (ST_i^d \leq ST_j^d \leq FT_i^d)(i, j \forall V,\ d \forall D) \tag{4}$$

Precedence constraint: To enforce dependencies in the DFG along with the communication costs, we need to ensure that the start time of a task i is greater than the finish time (FT_j^l) along with the data transfer overheads, over all its precedent tasks ($j \in Pred(i)$).

$$ST_i^d \geq FT_j^l + Data_{ij}/Rate_{dl}$$
$$(i \forall V)(d \forall D)(j \in Pred(i))(l \forall D - d) \tag{5}$$

We suitably modify the equations using the well-known Big-M method to construct the MILP and generate optimal solutions using the Gurobi [20] solver.

6 SIR+ RUNTIME IMPLEMENTATION

Having presented our Comm+SD and MILP techniques for efficient placement of tasks, we next discuss the challenges when implementing this on a production DNN engine - ONNXRuntime [27]. The runtime system must run tasks of the DFG concurrently across different devices by automatically creating the data and control workflows while satisfying the data dependencies between tasks. We first discuss the support already available in ONNXRuntime that we tap into and then illustrate the different system optimizations that are necessary to incorporate the placement strategies.

6.1 Challenges with existing Runtimes

Inference serving frameworks [12, 54] use two main APIs - *load* and *run* [28] - to load and execute the model. In ONNXRuntime, the *load* API first parses an ONNX [4] (protobuf) file and constructs an in-memory graph representation of the model. This graph is subsequently passed through an optimizer that assigns the nodes of the graph to the best available (CPU/GPU) device. This stage also performs various optimizations like node fusion, constant folding etc., similar to the ones proposed by TASO [32], TVM [8], and other such systems [56, 66, 67]. Once the graph has been optimized, a buffer (tensor) allocation plan for each task is generated on a memory arena using the Best-Fit-With-Coalescing (BFC) trace allocator. Note that memory allocation (*malloc*) is avoided on the critical path. The final plan of execution is generated containing (i) the order of execution of each task on the appropriate device, (ii) fence operations to enforce control flow dependencies using CUDA events API, (iii) tensor allocation plan of weights and inputs, (iv) data transfer operations between the devices (if any). When an inference is initiated using the *run* API, the execution engine calls the corresponding executor with this execution plan which executes them sequentially.

There are *three key challenges* to realize a concurrent execution of an inference with the above workflow. **(i)** Unlike in vanilla ONNXRuntime, tasks across different devices must execute concurrently, with low overheads for synchronization. **(ii)** The buffer allocation plan, must be amended to account for concurrent execution. The allocation is easy to determine a-priori in sequential execution, as once a task is complete the tensors consumed by the task can be deallocated and be re-used by another downstream task. With a concurrent execution strategy, it is challenging to the re-use memory buffers as downstream tasks can re-use tensors only when a task is specifically marked as complete. **(iii)** Finally, as DNN inference tasks are at the scale of μs, minor overheads cause a significant impact in performance. We will next see how SIR+ mitigates these challenges effectively.

6.2 SIR+: System for concurrent execution

We first describe the behavior of the concurrent executors and subsequently discuss a few optimizations.

Multi-threaded device executor: To enable parallel execution spanning multiple devices, SIR+ uses device specific queues to en-queue tasks. This serves as the driver to implement our concurrent placement strategies by en-queueing each task into the appropriate device queue with minimal

concurrency control. As opposed to a single sequential executor in ONNXRuntime, SIR+ launches executors on a threadpool [17]. They poll their respective device queues and execute tasks as and when tasks become ready (arrive on their queue). On task completion, they consult the graph for whether each successor node is ready (i.e., all its dependent tasks are complete) and correspondingly push the tasks on their respective device queues if dependencies are met.

Lock-free DFG execution: There are two important regions of synchronization that need to be efficiently handled. On task completion, each executor (i) should safely query the graph for task readiness, and (ii) add the ready task appropriately into the specific device queue. For the former, we use atomic variables to track the completed dependencies for a task. For the latter, we use a lock-free concurrent queue [6]. Together they provide light weight lock-free synchronization for orchestrating the data-flow execution. Lastly, we enhance the events based control flow mechanism by using a pre-created CUDA event pool to avoid additional CUDA induced synchronization overheads.

Memory re-use optimization: The next addition in SIR+ relates to the buffer allocator component. As described previously, the sequential executor determines re-use of buffers and de-allocates the buffer for a tensor when all its (re)uses are completed. In SIR+, each executor tracks and updates the use of the buffers allocated using atomic variables and de-allocates them upon the final use of the buffer.

Context-switch avoiding chain execution: On closely analyzing the above framework, one can see that when executing a long chain of tasks (i.e., each task has one predecessor and successor), unnecessary queue manipulation operations are performed resulting in additional contention with hidden context switch overheads. To avoid this, we perform an optimization where we detect linear-chains of tasks and avoid adding those tasks to the queue, which in effect fuses several tasks into one. This has two advantages: (i) we perform the DFG synchronization operations only when necessary, which reduces contention on the lock-free data structures, and (ii) for linear models like ResNet50, we do not impose any additional overheads, which generalizes our solution to all types of models.

6.3 Impact of system optimizations:

While the use of atomics and/or lock-free synchronization primitives may seem excessive, we find that they are essential for handling microsecond scale tasks. If we do not employ these mechanisms and instead go with conventional locking mechanisms, *we find a 16% degradation (i.e., increase) in the overall model execution time and a 4× increase in the memory usage for the FasterRCNN model* that we have studied so far.

PF1	PF2	PF3	PF4
Titan RTX 24 GB Xeon Gold 6230 188 GB DDR4	Quadro RTX 48GB Xeon Gold 6230 188GB DDR4	Tesla T4 16 GB Xeon Gold 6230 188 GB DDR4	RTX 2060 8GB Ryzen 2700X 16 GB DDR4

Table 5: Evaluated hardware platforms

Mean Latency Savings (%)	PF1		PF2		PF3	
	Low RPS=6	High RPS=15	Low RPS=6	High RPS=15	Low RPS=2	High RPS=6
FasterRCNN	9.44	51.90	8.53	42.84	4.99	15.5
MaskRCNN	12.21	82.8	11.56	68.68	6.07	26.23
SSD	12.79	84.89	14.10	90.36	25.9	45.3

Table 6: % savings in mean latency for SIR+ over ORT

The optimization to avoid context-switching is important to maintain the performance for DNN models with linear DFGs. Thus, these low level design choices are critical for performance.

7 EVALUATION

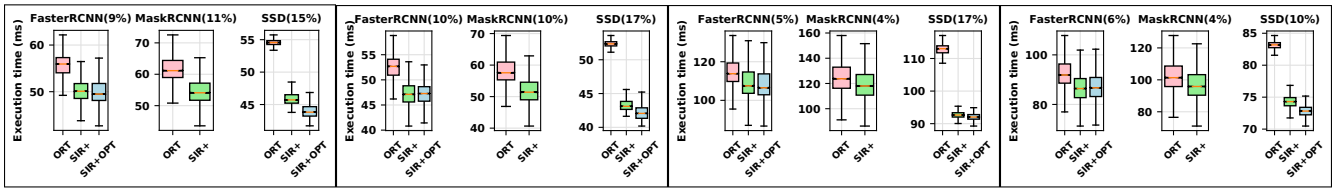
The goal of this work is to opportunistically utilize the CPU, beyond simply using the GPU, to (i) reduce the latency and (ii) increase the throughput of DNN inference on heterogeneous servers. We evaluate on four different heterogeneous servers comprising different CPUs/GPUs as described in Table 5. We compare ONNXRuntime version 1.6.0 - compiled with CUDA 10.2, CuBLAS, CuDNN (ORT) with SIR+ incorporating our *Comm+SD* placement algorithm. We consider 3 emerging models - FasterRCNN [44], MaskRCNN [45], SSD [46] - all trained on the COCO dataset [37] taken from the ONNX model zoo.

In experiments measuring execution time, we use a closed loop load generator calling ORT directly. For latency and throughput measurements, we use an open loop load generator running on a separate CPU socket of the same server with Poisson Process arrivals and inputs chosen at random to create the inference requests/queries. The request-response to this service is carried out using gRPC.

7.1 Latency for Emerging DNN Models

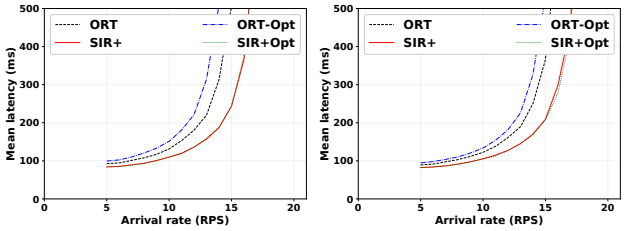
Execution time: We first evaluate the execution time for the emerging DNN models (FasterRCNN, MaskRCNN, and SSD) over the entire COCO dataset. We show the execution time distribution in Figure 3 on 4 different server platforms (see Table 5). We observe savings (see parenthesis in Figure 3) of 4%-15% in the average execution time across these platforms when using our SIR+ heterogeneous runtime over the state-of-the-art ONNXRuntime (ORT).

Mean latency: To evaluate a real-world serving scenario, we measure the mean latency as a function of arrival rate for ORT and SIR+ on $PF1-3$. The Figures 4a-4f (for $PF1$ and $PF2$) show the latency and the savings with SIR+ are

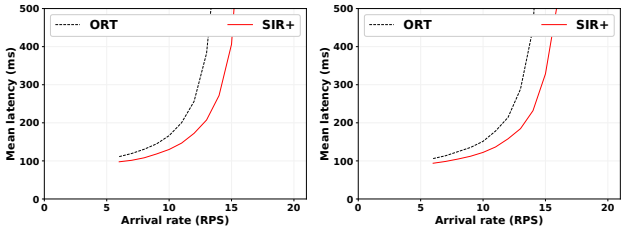


(a) $PF1$ (b) $PF2$ (c) $PF3$ (d) $PF4$

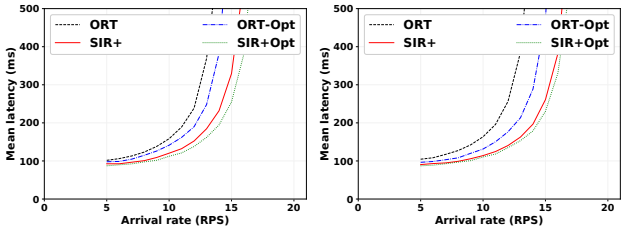
Figure 3: Execution time of ORT vs. SIR+ on different heterogeneous server configurations.



(a) FasterRCNN on $PF1$ (b) FasterRCNN on $PF2$

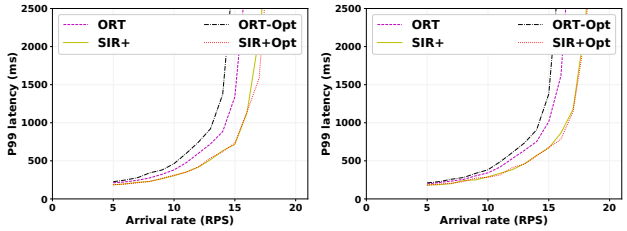


(c) MaskRCNN on $PF1$ (d) MaskRCNN on $PF2$

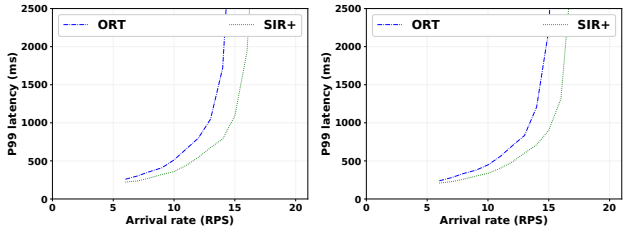


(e) SSD on $PF1$ (f) SSD on $PF2$

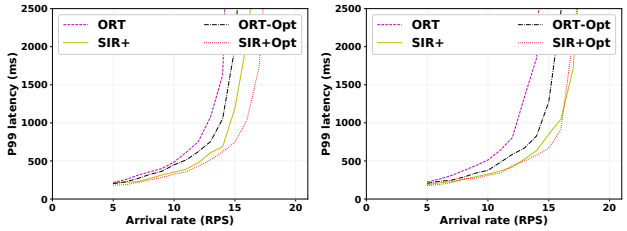
Figure 4: Mean latency performance of ORT vs. SIR+ with the Comm+SD/Opt placement.



(a) FasterRCNN on $PF1$ (b) FasterRCNN on $PF2$



(c) MaskRCNN on $PF1$ (d) MaskRCNN on $PF2$



(e) SSD on $PF1$ (f) SSD on $PF2$

Figure 5: P99 Tail latency performance of ORT vs. SIR+ with the Comm+SD/Opt placement.

summarized in Table 6. With SIR+, all three models show significant savings (at least 5-10%) at low arrival rates and are substantial (as much as 90%) at higher loads.

Tail latency: For latency critical services, the tail latency becomes as important as the mean latency. As seen in Figures 5a- 5f, SIR+ does significantly better in terms of 99^{th} percentile (P99) of the latency across different arrival rates. At an arrival rate of 15 RPS, there is at least 45% ($PF1$) and 47% ($PF2$) tail latency reduction when using SIR+ compared to vanilla ORT.

Comparison to Opt: We next compare Comm+SD with the MILP formulation that generates the optimal execution plan. It is expensive/impractical to generate a fully optimal solution for large graphs for this MILP problem. After two days of optimizing the 3 models, only FasterRCNN and SSD yielded a (marginally) better solution than our approach (Comm+SD). We compare this optimal placement on ORT and SIR+ (terming it as ORT-Opt SIR+Opt respectively) in Figures 4a-4b (FasterRCNN),4e-4f(SSD). SIR+ improves the mean latency at low load regions over ORT-Opt by 16% and 6.28%, and comes very close - less than 1% and 2.87% of the overall optimal solution (SIR+Opt) for the two models FasterRCNN and SSD respectively. Even if one is willing to spend an inordinate amount of time generating optimal

Model	Scheme	PF1 1 GPU TP (RPS)	PF1 Avg. GPU SM-Util (%)	PF2 1 GPU TP (RPS)	PF2 2 GPU TP (RPS)	PF2 4 GPU TP (RPS)
FasterRCNN	ORT	15.6	89	17.6	34.9	69.6
	SIR+	18.7	93	19.5	38.4	77.3
Mask RCNN	ORT	14.2	89	16.0	31.6	63.4
	SIR+	17.2	94	17.9	35.3	70.6
SSD	ORT	16.11	86	18.8	37.8	75.6
	SIR+	22.04	94	23.9	46.8	94.2

Table 7: Peak throughput comparison. ORT vs. SIR+

placements, the system design of SIR+ is necessary to realize any gain in performance.

These results reiterate the need to opportunistically leverage the surplus CPU devices for DNN inference instead of relying on the GPU only. This can be achieved by using a communication-and-slowdown aware placement strategy implemented on a low overhead runtime like SIR+.

7.2 Throughput for Emerging DNN Models

Datacenters hosting these inference services aim to maximize the throughput of requests served to boost their profitability. We measure the peak throughput that can be extracted from these executions (both ORT and SIR+) and show the results for the 3 emerging DNN models in Table 7 on single and multi-GPU settings. We measure peak throughput both on *PF*1 and *PF*2. On *PF*1, we first measure the peak throughput on single GPU settings, and scale SIR+ up to 4 GPUs on *PF*2. We run these models with a batch-size of 1 to satisfy their real-time latency requirements. As a result, the only option to measure the peak throughput is by running multiple instances of these inference services which has been done to obtain the results in Table 7.

When measured on *PF*1 with a single GPU, we see that the peak throughput has increased by 19.8%, 21.1%, and 36.8% respectively for FasterRCNN, MaskRCNN, and SSD models with SIR+ over ORT. Since SIR+ moves some tasks to the CPU from the GPU, the CPU utilization (not shown) obviously increases. Interestingly, the GPU utilization, obtained using NVML [42], has also increased under SIR+. This is because SIR+ assigns tasks that utilize the GPU capabilities more effectively to the GPU while the less effective tasks are redirected to the CPU. On scaling to multiple GPUs on *PF*2, we observe near linear scaling of throughput across 2 and 4 GPUs where the peak throughput increases by 11%, 11%, and 24% for the 3 models respectively when fully using the 4 GPUs available on the machine. This shows that SIR+ can generally be deployed on dense servers where the surplus CPUs generally tend to be wasted and under utilized.

7.3 GPU Memory Usage

Reducing the GPU memory footprint is an often noted concern in deploying DNN models to production due to prohibitive costs of memory (GDDR6) per GB [7, 25, 47, 50]. An important impact of our task placement is in reducing the GPU memory footprint. As Table 8 shows, we save over 9.8% of the GPU memory for the 3 models.

By migrating tasks to the CPU, the model parameters and tensors of these tasks get allocated in host DRAM instead of GPU memory.

Model	ORT (MB)	SIR+ (MB)	% Savings
FasterRCNN	1705	1517	11.0%
MaskRCNN	1715	1513	11.7%
SSD	1122.25	1012.25	9.8%

Table 8: GPU memory usage - *PF*1

7.4 Extending SIR+ to TensorRT:

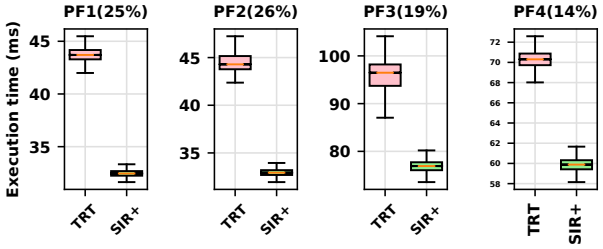


Figure 6: TensorRT (TRT) vs. SIR+ - SSD model

We now implement and compare SIR+ with TensorRT [59] (TRT) v7.0. TensorRT implements proprietary techniques that generate backends for different deep learning operators that extract maximum performance on Nvidia GPUs. We piggyback on ORT's capability to use TensorRT under-the-hood, and integrate it with our SIR+ system components. By applying our Comm+SD placement strategy, we profile and selectively overflow tasks from the TRT engine to be run on the CPU. Figure 6 shows the gain in execution engine by using SIR+ with TRT for the SSD model. The other two models are excluded as they perform better with vanilla ORT/SIR+ than with TRT. We see at least 14% gain (on average) in execution time for this model over the entire COCO dataset demonstrating that SIR+ and the Comm+SD placement strategy can be ported to different runtime engines.

8 RELATED WORK

DL inference services: DL inference has been optimized across the stack ranging from hardware [9, 11, 38] to software solutions [8, 56, 64, 66]. Inference serving frameworks [12, 18, 43, 54] all treat the execution engine as a black box and provide higher level support - in terms of cost [19, 21, 63]/SLO [62] - for deploying DL solutions at scale. They all stand to gain by adopting SIR+ as the execution engine on heterogeneous servers.

Device placement problem: The learning community [2, 14, 40, 41, 47, 60] calls the problem of placing tasks of a model on different devices as the *device placement problem* and have proposed learnings approaches to place tasks

automatically across devices. In the systems community, Baechi [31] and BytePS [33] propose to place tasks of ML training on various devices. Our work specifically proposes a practical placement solution for inference tasks and navigates the system challenges at μs scale on a real-world *inference* engine (Section 6.2).

DFG execution on heterogeneous devices: ETF [26] deals with executing DFGs on multiple processing engines. The ETF heuristic accounts for communication delays among multiple homogeneous processing engines by first assigning tasks to processing elements and subsequently accounting for the communication delays. Recently, GETF [57] proposes a scheduling framework generalizing ETF across heterogeneous processing engines with provable approximation guarantees. With the goal of extending such approaches to real system implementations, we have designed and developed SIR+ - by coupling a heuristic derived from empirical insights with a practical implementation that mitigates system side challenges - and shown substantial benefits in performance.

Heterogeneous task execution systems: In contrast to prior heterogeneous task execution systems comprising compilers [55]/runtimes [15, 35, 55]/schedulers [3] targeting HPC or other application domains, in this work, we study the inference of large DNNs with complex DFGs on a heterogeneous CPU + GPU system, with a focus on reducing their latency for user-facing scenarios. Many existing runtimes [1, 55] including ONNXRuntime [27] provide support for heterogeneous execution by compiling optimal backends for individual tasks on CPUs/GPUs, executing most (if not all) tasks on the GPU, and simply relegating tasks not implemented on the GPU to the CPU. Instead of this conventional approach, SIR+ proposes a scheduling policy (Comm+SD) that carefully migrates tasks scheduled to run on the GPU to run (concurrently) on the under-used CPUs. For DFGs with wide fan-outs, this can significantly reduce the overall execution time of the DNN model.

As we have highlighted in section 6.3, executing large DFGs composed of 100s of μ-scale sub-tasks can result in poor performance with ordinary support for concurrency, as is the case with existing scheduling engines [3, 27]. To efficiently execute the tiny sub-tasks of large DNN models concurrently across the CPU and the GPU, SIR+ builds a light-weight concurrency mechanism with added support to minimize memory usage.

SIR+ is the first execution engine for DNN inference that exploits the heterogeneity by navigating the system challenges at the microsecond scale. By opportunistically assigning tasks to otherwise idle devices on the server, it reduces the execution time even for a single DNN model.

9 CONCLUDING REMARKS

SIR+ paves the way to use surplus CPU devices inside increasingly heterogeneous datacenters when performing DNN inference, and demonstrates an holistic approach combining CPUs and GPUs to yield lower execution times and higher throughput. By opportunistically leveraging the surplus CPUs for CPU-efficient tasks, we allow more GPU-efficient tasks to better utilize the GPUs and reduce queueing times on them. In addition to efficiently matching tasks to devices, our Comm+SD placement strategy also accounts for communication costs between the devices. We implement this placement on the state-of-the-art ONNXRuntime execution engine by adding enhancements to eliminate synchronization bottlenecks for effectively supporting parallel execution of μs scale tasks on heterogeneous devices. Results show execution latency improvements of over 10% and throughput improvements of over 19% in emerging DNN models (Faster-RCNN, MaskRCNN, SSD) with wide DFG fan-outs. We have laid the foundation for leveraging intra-server heterogeneity (CPUs and GPUs that are widely present in today's datacenters) for such workloads, and we hope to extend it to a richer diversity of accelerators [5, 51] such as FPGAs, and customized ASICs in future work.

ACKNOWLEDGMENTS

We thank our shepherd Haris Volos, and the anonymous reviewers at SYSTOR for their helpful feedback. We also thank Iyswarya Narayanan, Saambhavi Baskaran, and Balachandran Swaminathan for reading through drafts of this paper and providing useful suggestions and feedback. This research was supported by National Science Foundation grants NSF-1909004,1714389,1912495,1763681.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: A System for {Large-Scale} Machine Learning. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Boston, MA, 265–283.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2018. Placeto: Efficient progressive device placement optimization. In *NIPS Machine Learning for Systems Workshop*. NeurIPS, San Diego, CA, USA.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[4] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[5] Saambhavi Baskaran and Jack Sampson. 2020. Decentralized Offload-Based Execution on Memory-Centric Compute Cores. In *The International Symposium on Memory Systems* (Washington, DC, USA) *(MEMSYS 2020)*. Association for Computing Machinery, New York, NY, USA,

61âĂŞ76. https://doi.org/10.1145/3422575.3422778

[6] Cameron. 2016. Moodycamel::ConcurrentQueue. https://github.com/cameron314/concurrentqueue. [Online; accessed 27-May-2021].

[7] Chia-Hao Chang, Adithya Kumar, and Anand Sivasubramaniam. 2021. To Move or Not to Move? Page Migration for Irregular Applications in over-Subscribed GPU Memory Systems with DynaMap. In *Proceedings of the 14th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '21)*. Association for Computing Machinery, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/3456727.3463766

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. USENIX Association, Boston, MA, 578–594.

[9] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[10] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. 2015. Attention-Based Models for Speech Recognition. In *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc., NY, USA. https://proceedings.neurips.cc/paper/2015/file/1068c6e4c8051cfd4e9ea8072e3189e2-Paper.pdf

[11] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[12] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. USENIX Association, Boston, MA, USA, 613–627.

[13] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, New York, NY, USA, 492–506.

[14] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*. PMLR, Princeton University, Princeton, NJ, USA, 1676–1684.

[15] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. 2013. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, IEEE, New York, NY, USA, 1299–1308.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press, Cambridge, MA, USA.

[17] Gaël Guennebaud and Benoit Jacob. 2010. Eigen. http://eigen.tuxfamily.org.

[18] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Boston, MA, USA. https://www.usenix.org/conference/osdi20/presentation/gujarati

[19] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2022. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI)*. USENIX, Boston, MA, 1041–1057.

[20] Incorporate Gurobi Optimization. 2018. Gurobi optimizer reference manual. http://www.gurobi.com.

[21] Matthew Halpern, Behzad Boroujerdian, Todd Mummert, Evelyn Duesterwald, and Vijay Janapa Reddi. 2019. One Size Does Not Fit All: Quantifying and Exposing the Accuracy-Latency Trade-Off in Machine Learning Cloud Service APIs via Tolerance Tiers. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, IEEE, New York, NY, USA, 34–47.

[22] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, IEEE, New York, NY, USA, 620–629.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. IEEE, New York, NY, USA, 770–778.

[24] Yitao Hu, Swati Rallapalli, Bongjun Ko, and Ramesh Govindan. 2018. Olympian: Scheduling Gpu Usage in a Deep Neural Network Model Serving System. In *Proceedings of the 19th International Middleware Conference*. ACM, New York, NY, USA, 53–65.

[25] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. 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*. ACM, New York, NY, USA, 1341–1355.

[26] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. 1989. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Journal Computing* 18, 2 (April 1989), 244âĂŞ257. https://doi.org/10.1137/0218016

[27] Microsoft Inc. 2016. ONNX Runtime inference engine. https://github.com/Microsoft/onnxruntime/. [Online; accessed 27-May-2021].

[28] Microsoft Inc. 2016. ONNX Runtime inference engine C APIs. https://github.com/microsoft/onnxruntime/blob/master/docs/C_API.md. [Online; accessed 27-May-2021].

[29] Microsoft Inc. 2016. ONNXRuntime Graph Optimizations. https://github.com/microsoft/onnxruntime/blob/master/docs/ONNX_Runtime_Graph_Optimizations.md. [Online; accessed 27-May-2021].

[30] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B Shah, and Will Tebbutt. 2019. A Differentiable Programming System to Bridge Machine Learning and Scientific Computing. arXiv:cs.PL/1907.07587

[31] Beomyeol Jeon, Linda Cai, Pallavi Srivastava, Jintao Jiang, Xiaolan Ke, Yitao Meng, Cong Xie, and Indranil Gupta. 2020. Baechi: fast device placement of machine learning graphs. In *Proceedings of the International Symposium on Cloud Computing (SoCC)*. ACM, New York, NY, USA, 416–430.

[32] 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. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, New York, NY, USA, 47–62.

[33] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous GPU/CPU Clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. USENIX Association, Boston, MA, USA, 463–479.

[34] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris

Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. IEEE, New York, NY, USA, 1âĂŞ12.

[35] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T Lewis, and Keshav Pingali. 2014. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, IEEE, New York, NY, USA, 151–162.

[36] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. 2017. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. IEEE, New York, NY, USA, 2117–2125.

[37] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, Cham, 740–755.

[38] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} Model Inference on CPUs. In *2019 {USENIX} Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Boston, MA, 1025–1040.

[39] Polina Mamoshina, Armando Vieira, Evgeny Putin, and Alex Zhavoronkov. 2016. Applications of deep learning in biomedicine. *Molecular pharmaceutics* 13, 5 (2016), 1445–1454.

[40] Ruben Mayer, Christian Mayer, and Larissa Laich. 2017. The tensorflow partitioning and scheduling problem: it's the critical path!. In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*. ACM, New York, NY, USA, 1–6.

[41] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *International Conference on Learning Representations*. Openreview.net, Online.

[42] Nvidia. 2020. nvml. https://docs.nvidia.com/deploy/nvml-api/nvml-api-reference.html. [Online; accessed 27-May-2021].

[43] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. arXiv:cs.DC/1712.06139

[44] ONNX. 2016. FasterRCNN Model. https://github.com/onnx/models/tree/master/vision/object_detection_segmentation/faster-rcnn. [Online; accessed 27-May-2021].

[45] ONNX. 2016. Mask-RCNN Model. https://github.com/onnx/models/tree/master/vision/object_detection_segmentation/mask-rcnn. [Online; accessed 27-May-2021].

[46] ONNX. 2016. SSD Model. https://github.com/onnx/models/tree/master/vision/object_detection_segmentation/ssd. [Online; accessed 27-May-2021].

[47] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2020. Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs. arXiv:cs.LG/1905.02494

[48] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. arXiv:cs.LG/1811.09886

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. Curran Associates, Inc., NY, USA, 8026–8037.

[50] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 891–905.

[51] Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Adithya Kumar, Prasanna Venkatesh Rengasamy, Vijaykrishnan Narayanan, Ameen Akel, and Sean Eilert. 2021. Design Space for Scaling-in General Purpose Computing within the DDR DRAM Hierarchy for Map-Reduce Workloads. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*. Association for Computing Machinery, New York, NY, USA, 113âĂŞ123. https://doi.org/10.1145/3457388.3458661

[52] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*. Curran Associates, Inc., NY, USA, 91–99.

[53] Allied Market Research. 2020. Global Machine Learning Chip Market. https://www.globenewswire.com/news-release/2020/02/18/1986370/0/en/Global-Machine-Learning-Chip-Market-to-Garner-37-85-Billion-by-2025-at-40-8-CAGR.html. [Online; accessed 27-May-2021].

[54] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2020. INFaaS: A Model-less and Managed Inference Serving System. arXiv:cs.DC/1905.13348

[55] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 49–68.

[56] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference. arXiv:cs.PL/2006.03031

[57] Yu Su, Xiaoqi Ren, Shai Vardi, Adam Wierman, and Yuxiong He. 2019. Communication-Aware Scheduling of Precedence-Constrained Tasks. *SIGMETRICS Performance Evaluation Review* 47, 2 (Dec. 2019), 21âĂŞ23. https://doi.org/10.1145/3374888.3374897

[58] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. 2013. Deep neural networks for object detection. In *Advances in neural information processing systems*. Curran Associates, Inc., NY, USA, 2553–2561.

[59] Han Vanholder. 2016. Efficient Inference with TensorRT.

[60] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, New York, NY, USA, 1–17.

[61] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. 2019. Condconv: Conditionally parameterized convolutions for efficient inference. *Advances in Neural Information Processing Systems* 32 (2019).

[62] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, USA.

[63] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, Boston, MA, USA.

[64] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. Deepcpu: Serving rnn-based deep learning models 10x faster. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, 951–965. https://www.usenix.org/conference/atc18/presentation/zhang-minjia

[65] Yikang Zhang, Jian Zhang, Qiang Wang, and Zhao Zhong. 2020. Dynet: Dynamic convolution for accelerating convolutional neural networks. *arXiv preprint arXiv:2004.10694* 1 (2020).

[66] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Boston, MA, USA, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[67] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 859–873.