Exact Memory- and Communication-aware Scheduling of DNNs on Pipelined Edge TPUs

Jiaqi Yin[1], Zhiru Zhang[2], Cunxi Yu[1]
[1]University of Utah
[2]Cornell University
{jiaqi.yin@utah.edu, cunxi.yu@utah.edu}

***Abstract*—Deep neural networks (DNNs) represent the state-of-the-art in many applications but have substantial computational and memory requirements, which greatly limit their training and deployment in real-world systems. In particular, the deployment challenges further increase on edge systems with much more restricted resource-constrained (e.g., computation and memory bounded), which recently attracted significant interest in many application scenarios. Such devices like Edge TPUs usually provide limited on-chip storage and memory bandwidth, where the heuristic-based ahead-of-time compilation techniques are highly limited in optimizing the inference performance due to the lacks of performance guarantees. This work proposes a novel exact pipeline scheduling framework that enables model parameter caching, data dependency, and device-to-device communication-aware multi-objective optimizations. The framework is powered by novel versatile SDC+ILP formulations supporting both propositional logic and non-equality constraints. The experimental results demonstrate that the proposed scheduling frameworks consistently outperform commercial Edge TPU Compiler with up to more than $4\times$ speedups on eleven ImageNet models in physical pipelined Edge TPU setups. In addition, we have demonstrated consistent real-world energy efficiency improvements measured with high precision power meter. Finally, the proposed framework has also demonstrated the capability in multi-model co-deployment on pipeline Edge TPU system, which is not supported by Edge TPU Compiler.**

I. Introduction

Targeted specialization of functionality in hardware has become arguably the best means for enabling improved compute performance and energy efficiency. However, as the complexity of modern hardware systems explodes, fast and effective hardware deployments of high computation density algorithms are more and more challenging. Deep neural networks (DNNs) represent the state-of-the-art in many applications but introduce substantial computational and memory requirements, which greatly limit their training and deployment in resource-constrained (e.g., compute and memory resources) environments. To efficiently deploy DNNs on the hardware platforms, it usually requires designated compilers that take in front-end DNN models and map them to the platforms. As the size of DNN models rises, it becomes more challenging to deploy the models onto edge devices with small on-chip buffer sizes using static and heuristic-based execution scheduling methods, specifically for edge computing ecosystems, such as Google Edge TPU ([6], [41]), Microsoft Azure ML ([4]), etc.

To efficiently utilize those hardware platforms, scheduling algorithms implemented in deep learning (DL) compilers are critical in deploying such hyper-dimensional computationally-intensive workloads, which is a classical *NP-hard* combinatorial optimization problem ([24], [27]). Mostly, vendor-specific libraries such as Nvidia cuBLAS, TVM, and TF-Lite ([1], [8], [25], [36]), rely on hand-crafted domain-specific heuristics to optimize the executions, which trades the execution performance for scheduling runtime. An alternative solution is the tensor compiler, which expresses the processing of tensors in its own intermediate representations (IRs) [1], [8], [32], [35]. These compilers separate the computation definition (i.e., what to compute) and the scheduling (i.e., how to compute) to focus on the scheduling part for performance optimization, including loop transformation, tiling, thread binding, etc. While recently there has been significant progress in advancing DL compilers, the challenges for large-scale DL executions are still rising up.

One of the critical limitations is that executions of DL algorithms optimized by heuristic-based compilation lack quality guarantees, which can cause significant performance degradation. As state-of-the-art neural networks are growing wider and deeper, the number of tensor operations growing from millions to trillions ([14], [34]). Meanwhile, the resource constraints become stricter, especially on the edge devices such as mobile devices and customized edge platforms for smart homes. Such devices usually provide limited on-chip storage and memory bandwidth, which makes it more challenging to fit the growing models without losing too much performance. Failures of executing deep learning algorithms on edges devices without exact guarantees can cause significant performance degradation, which can be safety-critical, e.g., autonomous driving [17] and Face-ID on mobile devices [18]. Thus, we argue there is a great need to develop scalable and versatile exact compilation methods for heavily customized edge devices. Most edge-device scheduling methods such as Google Edge TPU compiler [6], [41] and dynamic scheduler [3] utilize iterative metaheuristics [31], which either optimize schedule efficiently without performance guarantees or require significant long optimization runtime. On the other hand, there have been many learning-based scheduling approaches [7], [30], [37], which lack guarantees in determinism and solution quality and require expensive training and data collection. This work aims to develop a deterministic, optimal, and scalable constraint solving-based scheduling method for edge DNN deployment.

This work aims to develop a novel exact memory caching aware and communicate aware pipeline scheduling framework,

targeting a multi-stage pipeline Edge TPUs system for DNNs inference acceleration. Specifically, the contributions can be summarized as follows:

- First comprehensive experimental studies on multi-stage Edge TPU pipelining are provided in Sections II-B and III, which post the limitations of heuristic-based pipeline scheduling and domain-specific knowledge for exact Edge TPU optimizations.
- We propose a novel exact pipeline scheduling framework that enables exact model parameter caching, data dependency, and device-to-device communication cost optimization, supported by novel versatile SDC+ILP formulations supporting both propositional logic and non-equality constraints (Section IV).
- With the novel SDC+ILP formulations, we introduce a novel incremental ILP solving strategy for multi-objective exact optimization, which balances both quality-of-results of scheduling and solving runtime.
- Our approaches are evaluated with 2,3,4,5, and 6-stage physical pipeline Edge TPU setups, using eleven popular ImageNet models, with commercial Edge TPU Compiler as the baseline. The proposed approaches demonstrate consistent inference runtime speedups across all pipeline setups, with up to 4× against Edge TPU Compiler.
- In addition, comparisons and evaluations on real-world power consumption and energy efficiency (Joules/fps) are provided to demonstrate the advantages of the proposed scheduling approaches.
- Finally, we demonstrate a case study of the first multi-model co-compilation on pipeline Edge TPUs, which is not supported by Edge TPU Compiler.

Experimental methodology and artifact availability – The proposed system has been integrated with IBM CPLEX, Google Edge TPU and Tensorflow-Lite, to provide complete solution of DNN scheduling on Edge TPU and offers extensibility on other edge platforms using SDC+ILP scheduling. Experimental results are conducted on real-world Edge TPU system obtained from `Coral.ai` and power measurements are performed with high precision USB power meter. The software system will be released in public repository after acceptance.

II. Background

A. Edge TPU and Software Stack

Following the success of TPUs ([1], [23]), Google extends the specialized systolic array architecture to deep learning acceleration at edge, namely Edge TPUs. The main components in Edge TPUs include a 2D array of processing elements (PE), where each PE contains a single or multiple cores. One critical feature that has high impacts on performance and energy efficiency in edge accelerators is the memory design. In Edge TPUs, each PE has a memory shared across all the compute cores on a single Edge TPU device, which is mainly used to store model activations, intermediate computing results, and outputs. The cores within each PE feature an on-chip cache memory that is mainly used for storing model parameters. When the model parameters cannot be fully fit in the cache, parameters are partially loaded in the off-chip DRAM. Similar to many accelerator designs, the cache and DRAM utilization distribution will be critical for achieving high throughput of such highly parallel computing architecture. Currently, Edge TPU platforms only support TensorFlow Lite (TFLite [28]) compiled and quantized models trained with TensorFlow as input to the Edge TPU software stack. Specifically, the Edge TPU runtime library is developed to support TFLite APIs, where the TFLite models are compiled ahead-of-time before deployment using Edge TPU Compiler. This compiler maps the operators in the computational graphs of the DNNs models to customized operations supported on the Edge TPU hardware and optimizes the execution of all the operations.

B. Edge TPU Pipelining

Due to the aforementioned memory design characteristics, Edge TPU system offers a pipeline compilation option to accelerate the runtime of deploying large models in Edge TPU ecosystem. While the model parameters cannot be fully fit into the Edge TPU cache and the overall throughput needs to be improved, Edge TPU compiler offers pipelined compilation that segments the model into separate computational subgraphs. At the deployment stage, each subgraph runs in a pipeline on separate Edge TPUs. For example, as shown in Figure 1, we have built a 6-stage pipeline Edge TPU system, where the one or multiple models can be deployed after partitioning into six computational subgraphs, where each subgraph will be deployed to devices T0 to T5. For example, we perform a simple multi-stage pipeline evaluation shown in Table I, which shows that inference speed can be greatly improved by pipelining the models on multiple Edge TPU devices. This is because if model parameters cannot be fit to on-chip cache, the overall throughput becomes a bottleneck.

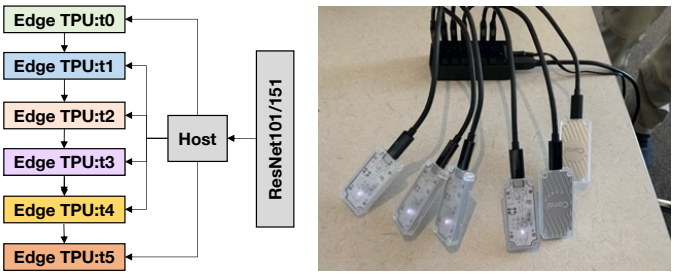


Fig. 1: Illustration of our multi-stage (up to 6) pipeline Edge TPUs that is connected to CPU host via USB 3.0 Hub. The USB 3.0 Hub is equipped with external power supply to maintain stable 5.2 V power supply cross all Edge TPU devices.

TABLE I: Execution time (millisecond) of ResNet inference on single Edge TPU and pipeline Edge TPUs.

Stages	1	2	3	4	5
ResNet101	80.7	68.6	48.6	32.8	14.2
ResNet152	115.1	104.5	84.5	66.8	50.4

While Table I shows that the performance can be improved by pipelining, there are several limitations in the Edge TPU compiler. First, Edge TPU Compiler pipeline compilation strategy uses a heuristic that tries to evenly distribute the model parameters loading across all the Edge TPU devices, without considering the optimality and global runtime impact. Second, Edge TPU compiler does not consider the communication cost from earlier stage to later stage in the pipeline Edge TPU system. Note that the communication that mostly communicates the intermediate results to next stage is conducted on device-to-device communication via USB 3.0 connection, which could become runtime bottleneck when all devices load subgraphs using cache only. Moreover, to maintain the correctness of execution, the pipeline scheduling of the DNNs computational graph has to follow topological order to avoid dependency violation. However, there exists a large number of different topological orders for a given computational graph but result in very different performance on the physical Edge TPU system.

C. Resource-constrained scheduling

Resource-constrained scheduling (RCS) has been the subject of extensive study, resulting in a line of heuristics, including Hu's Algorithm, List Scheduling, and Force-Directed Scheduling, to solve the problem efficiently ([33], [40]). Iterative metaheuristics, such as simulated annealing, ant colony, and dynamic programming optimizations, have also been demonstrated as viable options [31]. For example, [3] proposed a dynamic programming based adaptive budgeting scheduling technique, which can either optimize schedule efficiently without performance guarantees or requires significant long optimization runtime. On the other hand, while resource-constrained scheduling maps to a constraint satisfaction problem consisting of logical connectives of linear constraints, it can also be solved with modern SMT solvers, which integrate specialized solvers with propositional satisfiability search techniques to achieve conflict-driven learning [5], [12], [13], [15], [29], [39].

System of difference constraints (SDC) is a system of inequality constraints in the integer difference form $x_i - x_j \leq b_{ij}$, where b_{ij} is an integer, and x_i and x_j are variables. The system is feasible if there exists a solution that satisfies all inequalities in the system. Because of the restrictive form of the constraints, SDC can be solved efficiently. For SDC-based scheduling [10], [11], [26], [43], [44], a schedule variable s_i is declared for each operation i in the DFG to denote the clock cycle at which operation i is scheduled. All SDC scheduling constraints are then expressed in the integer difference form so that the system consists of a totally unimodular constraint matrix over which an optimal integer solution can be guaranteed in polynomial time, which significantly improves the classic integer linear programming (ILP) scheduling ([16], [31]). However, SDC-based formulations have not yet been applied to neural network domain scheduling, and yet have been extended to support a hybrid form of propositional and non-equality constraints.

III. Pipelined Edge TPU Runtime

A. DNNs Computational Graph Scheduling

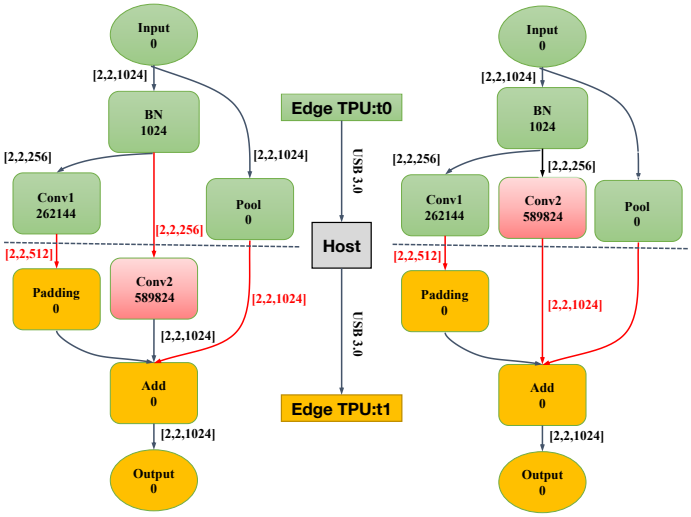


Fig. 2: Two scheduling examples of a simple DNNs computation graph that differ only in the Conv2 node allocation – Left scheduling achieves better runtime, since it optimizes memory utilization and communication cost.

Scheduling of DNNs computational graph is a critical step to achieve desired runtime performance for DNNs computing platforms [2], [8], [22], [23], [36]. Specifically, the optimization objectives of scheduling computational graphs can be defined as follows: **Given:** *(1)* A DAG $G(V, E)$ where V represents the set of operations in the DNNs computational graphs, and E represents the set of edges; *(2)* A set of scheduling constraints, which may include dependency constraints (E), resource constraints, execution time, memory allocations, etc. **Objective:** Construct an exact optimal schedule S = s_0,s_1,...s_n, $n \leq |V|$, where n represents the number of scheduling stages. The operation set in the computational graph V will be allocated to S that satisfies all scheduling constraints (E). For example, in a two-stage pipelined Edge TPU system in Figure 2, resulted schedule assigns computation node N_{Input}, N_{BN}, N_{Conv1}, and N_{Pool} to s_0 (Edge TPU:0), N_{Padding}, N_{Add} and N_{Output} to s_1 (Edge TPU:1), etc.

Figure 2 illustrates two different scheduling solutions on a DNNs synthetic computational graph, in which the number of parameters is shown in the vertices. While both schedules are valid, they differ in the scheduling assignment of node N_{Conv1}. Considering a two-stage pipelining system, the time to execute the given computational graph (T) is equal to sum of execution time among all stages ($T = T_0 + T_1$). Because T_0 and T_1 are dominated by the off-chip memory usage, the scheduling solution that has the lower off-chip memory usage will likely execute faster. Specifically, the left scheduling in Figure 2 has lower memory upper bound (589824). In addition, the communication cost, i.e., tensors communicated between the neighbor stages, is another critical factor for the runtime. In Figure 2, the left scheduling offers smaller communication cost

([2,2,512]+[2,2,256]+[2,2,1024]) than the right one ([2,2,512], [2,2,1024] and [2,2,1024]). Therefore the left scheduling has better memory usage and smaller communication cost, which leads to better runtime. We extend the analysis on an ImageNet ResNet50 model in the following section to motivate the studies in exact scheudling of DNNs computation graph, particularly on Edge TPUs.

B. Motivating Example – ResNet50 on Pipelined Edge TPUs

In this section, a comprehensive analysis of the performance variations with different scheduling of pipeline execution is provided in order to learn domain-specific knowledge for producing optimal schedules. Specifically, we aim to analyze the runtime differences in three aspects: *1) Parameters caching* – One critical optimization on edge hardware is optimizing the model parameter caching, as on-chip memory size is very limited. For inference on multi-stage pipeline Edge TPUs, the execution efficiency could benefit significantly from minimizing per device parameters loading in DRAM. *2) Data dependency* – Give a DNNs computational graph, there could exist many execution schedules that evenly split out the cache/DRAM usage, but result in very different performance. This is mostly caused by data dependency. Note that the pipeline Edge TPU system involves device-to-device communication, such that the effects of data dependency could be more significant. *3) Device-to-device-communication* – Note that device-to-device-communication is performed off-chip (e.g., via USB 3.0 I/O), which is much slower than any on-chip interface. A motivating example to demonstrate the importance of these three optimization aspects in the pipeline Edge TPU system is provided below.

TABLE II: **Motivating example of pipelined Edge TPU execution scheduling using ResNet50v2 model with nine different scheduling choices w.r.t partial computation graph shown in Figure 3.** Note that execution time is measured as average per frame execution runtime with 5,000 inference iterations. And we pick pipeline option IV as baseline to measure the speedup.

Choice	Execution Time(ms)	*Stage-0 (MiB)*		*Stage-1 (MiB)*	
		DRAM	Cache	DRAM	Cache
I	12.6 (+18.7%)	1.85	6.72	0	5.81
II	13.3 (+21.1%)	1.82	6.72	3.13e-2	5.81
III	11.9 (+11.8%)	1.56	6.72	3.13e-2	6.07
IV	10.5 (Best)	0.993	6.72	3.13e-2	6.64
V	11.5 (+8.7%)	0.738	6.72	3.13e-2	6.89
VI	11.5 (+8.7%)	0.738	6.72	3.13e-2	6.89
VII	15.5 (+32.3%)	1.82	6.72	3.13e-2	5.81
VIII	13.3 (+21.1%)	0.738	6.72	3.13e-2	6.89
IX	10.9 (+3.7%)	0.707	6.72	6.25e-2	6.89

a) Motivating example: Here, we provide an experimental example to demonstrate the impacts of different pipeline execution schedules, with specific cases corresponding to the aforementioned three items. Specifically, we build a CPU central-hosted pipeline Edge TPU system to execute the pipeline models and evaluate the performance (see Figure

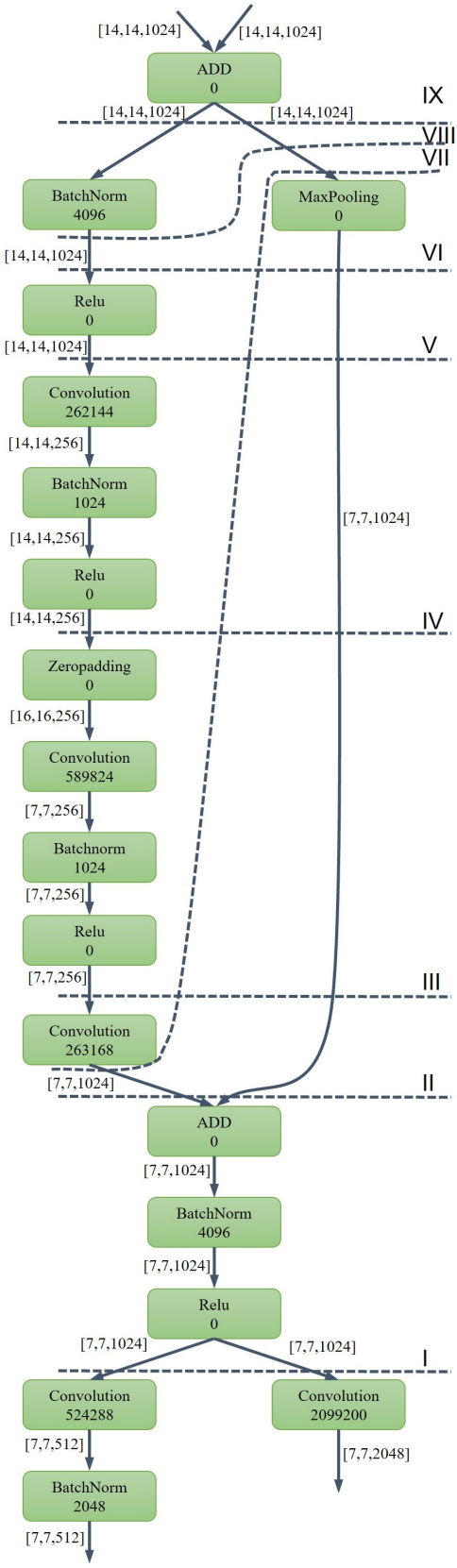


Fig. 3: **Partial structure of ResNet50v2 used in the motivating example with detailed partitions for first two stages in 3-stage pipeline Edge TPUs.** Partial computational graph of ResNet50V2 with nine different pipeline options shown in dash lines I – IX, which is used as partition point for stage-0 and stage-1.

1), and this case study is conducted on deploying an ImageNet ResNet50v2 model in a 3-stage execution. To be specific, this study includes nine different manually generated scheduling choices on partitioning the first and second stages of ResNet50v2 model, shown in Figure 3 and Table II. As shown in Figure 3, the choices evaluated here are located in one of the residual blocks. Specifically, the number of parameters for each operation and the output volume size of each operation are annotated in Figure 3. Table II records the core memory (Cache) usage and DRAM usage after deploying different pipelined models, as well as the physical computing execution runtime. The main observations can be summarized as follows:

- **Parameter caching has great impacts on execution runtime in ahead-of-time compilation**. Specifically, the execution speed is determined by the cache and DRAM usage. For example, pipeline choice IV offers the best execution time for 3-stage pipeline ResNet50v2, where it maximizes the cache usage and simultaneously minimizes the DRAM usage per device. Note that the third stage (stage-2) excluded in Table II are partitioned identically cross all the choices. Other pipeline choices such as I, II, III, and VII need to have similar DRAM parameter loading on stage-1 but require more than 1 MiB DRAM usage on stage-0, which significantly degrades the performance by $\sim$14%. However, we can see that choices IV (10.5 ms) and VIII (13.3 ms) have very similar parameter caching statistics but result in $\sim$12% difference in execution time, which leads to our next domain-specific observation.
- **Data dependency has to be considered in compiler optimization, particularly the partition points and their structure between subgraphs**. The partition structure needs to be specially taken care considering operation fanouts, i.e., the output of one operation is used as the inputs of multiple nodes. For example, pipeline choice VIII put the child nodes of Add into different stages and need 13.3 ms to inference. However, we can see a speedup if we schedule the child nodes into the same stage. Pipeline choices IX, VI schedule the child nodes of ADD into the same stage and respectively have a speedup of $\sim$1.22$\times$ (10.9 ms) and $\sim$1.16$\times$ (11.5 ms).
- **Minimizing communication cost leads to faster pipeline inference.** For example, given pipeline choices IV and V, DRAM usage in IV is 0.993 MiB, which is more $\sim$25% more than DRAM usage of V 0.738 MiB. However, we can see that IV performs 8.70% faster than V, due to a much smaller communication cost. Specifically, in the pipeline choice V, the tensors passed from `ReLU` operation to `Convolution` are in shape of (14,14,1024), while in IV the tensor size is (14,14,256), i.e., 4$\times$ smaller in communicating the intermediate output from the first device to the second device (Figure 3).

IV. Approach

In this section, we present the formulations of exact scheduling for optimizing the pipeline schedule on Edge TPU system, with DNNs computational graphs as inputs. Specifically, we propose a novel SDC+ILP-based formulation that combines both propositional logic and non-equality constraints for optimal Edge TPU scheduling in n-stage pipeline settings ($n \geq 2$), which enable multi-objective scheduling that leverage domain-specific knowledge summarized in Section III. Noted that all constraints are automatically generated for any given computational graph in our framework.

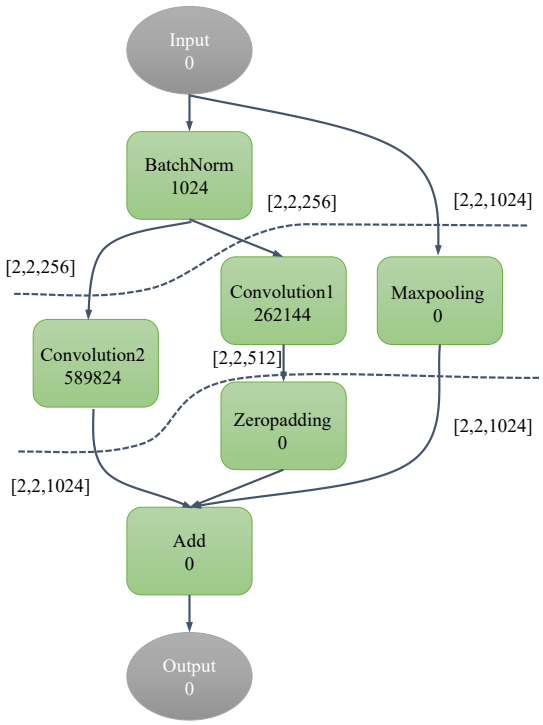


Fig. 4: **Synthetic DNNs computational graph representation for formulation illustrations.**

A. Multi-objective Optimization Formulations

In the modern deep learning frameworks, machine learning algorithms are represented as computational graphs, where each graph is a directed graph $G(V, E)$ where V, E represent operations and the dataflow, respectively. Given the computation graph, the edge device scheduling maps the operators into different devices. Specifically, the SDC-ILP based scheduling builds the formulation and different sets of constraints to optimize the objectives. The constraint includes dependency constraint, pipeline constraints, memory caching constraint, etc.

As demonstrated in the ResNet50v2 motivating example, we can see that multi-objective optimization is necessary to generate the optimal schedules to cover all performance factors influencing edge device performance. However, directly solving a global multi-objective SDC+ILP formulation can be very challenging due to the nature computational complexity. In particularly, while elaborating all domain-specific knowledge in optimizations with a large computational graph, the formulation size and complexity can explode very fast. Therefore, we integrate incremental multi-objective solving in our scheduling

framework to rank the solving priorities of the three optimization objectives.

1) Dependency and Pipeline Constraint: The dependence constraints formulated in this section guarantee the execution correctness of the given computational graph, which has the highest priority among all constraints. Specifically, for each edge $e_{i,j} \in E$ that directs v_i to v_j, dependency formulation is required to make sure the optimization is aware of v_j can be executed if v_i is complete. Let s_{i} be the pipeline stage for node v_i, and s_{j} be the pipeline stage for node v_j, the dependency encoded by edge $e_{i,j}$ can be formulated as

$$s_{\mathrm{i}} - s_{\mathrm{j}} \leq 0 \tag{1}$$

where s_i and s_j are the stages of nodes which should be an integer variable. In addition, given an n-stage pipeline Edge TPU system, the scheduling space s_i of each operation v_i are limited to one of the n stages. Note that we assume that all the operations in the computational graphs are executed only once in this work. Thus, let s_i^k be the scheduling variable of node v_i scheduled at k stage ($k < n$), Equation 2 combined with Equation 1 completes the dependency constraints for a given pipeline system.

$$\sum_{\mathrm{k=0}}^{\mathrm{n\text{-}1}} s_{\mathrm{i}}^{\mathrm{k}} = 1 \tag{2}$$

Example 1 We illustrate the dependency constraints using a synthetic computational graph shown in Figure 4, which consists of six operations, i.e., BatchNorm (BN), two Convolution operations (Conv1 and Conv2), Maxpooling (Pool), Zeropadding (Pad), and Addition (Add).

$$\begin{aligned} & s_{\mathrm{BN}} - s_{\mathrm{In}} \geq 0 \\ \wedge \;\; & s_{\mathrm{Pool}} - s_{\mathrm{In}} \geq 0 \\ \wedge \;\; & s_{\mathrm{Conv2}} - s_{\mathrm{BN}} \geq 0 \\ \wedge \;\; & s_{\mathrm{Conv1}} - s_{\mathrm{BN}} \geq 0 \\ & \ldots \\ \wedge \;\; & s_{\mathrm{Add}} - s_{\mathrm{Pool}} \geq 0 \\ \wedge \;\; & s_{\mathrm{Out}} - s_{\mathrm{Add}} \geq 0 \end{aligned} \tag{3}$$

As shown in Figure 4, there exist execution dependencies from BN to Conv1 and Conv2, such that the formulation imposes constraint $s_{\mathrm{BN}} - s_{\mathrm{Conv1}} \leq 0 \wedge s_{\mathrm{BN}} - s_{\mathrm{Conv1}} \leq 0$ to guarantee the BN is complete before initializing Conv1 and Conv2. Similarly, Equation 3 encodes all execution dependency constraints in the graph. Besides, we introduce $s_{\mathrm{i}}^{\mathrm{k}}$ as a binary variable to denote if operation v_i is scheduled in stage k, which is used for encoding the pipeline constraints. In this example, Equation 4 is sufficient to complete the dependency constraints along with Equation 3.

$$\begin{aligned} \wedge \;\; & s_{\mathrm{BN}}^{\mathrm{k=0}} + s_{\mathrm{BN}}^{\mathrm{k=1}} + s_{\mathrm{BN}}^{\mathrm{k=2}} = 1 \\ \wedge \;\; & s_{\mathrm{Conv1}}^{\mathrm{k=0}} + s_{\mathrm{Conv1}}^{\mathrm{k=1}} + s_{\mathrm{Conv1}}^{\mathrm{k=2}} = 1 \\ & \ldots \\ \wedge \;\; & s_{\mathrm{Add}}^{\mathrm{k=0}} + s_{\mathrm{Add}}^{\mathrm{k=1}} + s_{\mathrm{Add}}^{\mathrm{k=1}} = 1 \end{aligned} \tag{4}$$

2) Parameter Caching Optimization: To optimize the parameter caching, a set of formulations that encode the per pipeline stage parameter memory consumption is needed. Specifically, we introduce additional constraints on top of dependency constraints discussed in Section IV-A1 to exhaustively encode the per stage memory consumption w.r.t all scheduling search space. Let m_{k} be the memory consumption of pipeline stage k, m_{k} is the sum of parameter memory caching cost P_{k} of all operations to be scheduled in stage k. To exhaustively cover all scheduling options, we use an explicit formulation for m_{k}

$$m_{\mathrm{k}} = \sum_{\forall v \in V} p_v \cdot s_v^k \tag{5}$$

where p_v is the memory consumption estimated based on the number of parameters in operation v. Notes that s_v^k is a binary variable and only one of the variables for $k \in [0, n-1]$ evaluates to `True`.

Example 2 In this example, we first illustrate the memory consumption modeling in ILP formulation. Assume the target system is 3-stage pipeline Edge TPUs, there is total of three scheduling choices for each operation in Figure 4, e.g., s_{BN}^0, s_{BN}^1, s_{BN}^2 encoding that BN node can be theoretically scheduled at either of the three stages. Similarly, in the explicit formulation, other nodes can be theoretically scheduled at any stage. Thus, according to Equation 5, the memory consumption per pipeline stage can be formulated as

$$\begin{cases} V = \{\text{BN, Pool, Conv1, Conv2, Pad, Pool, Add}\} \\ \wedge \quad m_0 = \sum_{\forall v \in V} p_v \cdot s_v^0 \\ \wedge \quad m_1 = \sum_{\forall v \in V} p_v \cdot s_v^1 \\ \wedge \quad m_2 = \sum_{\forall v \in V} p_v \cdot s_v^2 \\ \wedge \quad \text{Equations (4) and (5)} \end{cases} \tag{6}$$

where p_v is denoted in Figure 4. These ensure the ILP formulations precisely encode the per-stage memory consumption over the whole scheduling space. For example, we can verify the memory consumption using the trivial partitions shown in Figure 4 (dash lines). For m_0, the only schedule variable is evaluated to `True` is s_{BN}^0, such that $m_0 = 1024$. Note that s_{BN}^1 and s_{BN}^2 will be automatically determined to `False` according to Equation 4. Similarly, m_1 includes only s_{Conv1}^1, s_{Conv1}^1, and s_{Pool}^1, where the rest of the variables are evaluated to `False`.

$$\begin{cases} \textbf{min} \quad m_{\text{limit}} \\ \wedge \quad m_0 \leq m_{\text{limit}} \\ \wedge \quad m_1 \leq m_{\text{limit}} \\ \wedge \quad m_2 \leq m_{\text{limit}} \\ \wedge \quad \text{Equations (4), (5), (7)} \end{cases} \tag{7}$$

The per-stage memory consumption modeling enables flexible optimization objective definitions in the entire formulation. Specifically, we are interested in two exact parameter caching optimization objectives: **1)** minimize per device parameter caching, and **2)** maximize cache utilization as well as minimize DRAM usage. For objective **1)**, as shown in Equation 7, we introduce a new memory bound variable m_{limit}, which is the upper bound of the per stage memory consumption. While giving an optimization objective that minimizes m_{limit}, the solver is able to find the minimum per stage memory cost scheduling solution that satisfies all other constraints.

Enabling memory optimization objective is more complex as a conditional *greater-equal* ILP formulation that encodes

the DRAM utilization. For example, in Edge TPU devices, the on-chip memory is 8MiB, and per stage memory usage of a 3-stage pipeline model is 6 MiB, 8.5 MiB, and 9 MiB. Then, the DRAM usage of the first stage is 0 since it is smaller than the cache size. For the second and third stages, while the size is greater than 8, the DRAM usage will be 0.5 and 1 MiB. Thus, the DRAM utilization at stage k is $\texttt{max}(0, m_k - m_{\text{cache}})$, which can be formulated using a novel ILP greater-equal formulation $\text{ILP}_{\geq}()$, shown in Equation 8, where m_{k} denotes the memory consumption at stage k, m_{cache} is cache memory size, and $\text{ILP}_{\geq}(m_{\text{k}}, m_{\text{cache}})$ is a binary variable to represent if $m_k \geq m_{\text{cache}}$. To express $\text{ILP}_{\geq}(m_k, m_{\text{cache}})$ in ILP formulation, $\text{ILP}_{\geq}(m_k, m_{\text{cache}}) = \text{ILP}_{\geq}(m_k, m_{\text{cache}}, k, U)$, where k and U are constants. Specifically, we want to **1)** evaluate $\text{ILP}_{\geq}$ to `True` if $m_k - m_{\text{cache}} \geq k$, and **2)** evaluate $\text{ILP}_{\geq}$ to `False` if $m_k - m_{\text{cache}} < k$. U is a user-defined upper bound that $U >> m_k, m_{\text{cache}}$. Therefore, we can simply set $k = 0$ and U to be a very large integer.

$$\begin{aligned} & m_{\text{DRAM}^k} = \textstyle\sum_{\text{k}}(m_{\text{k}} - m_{\text{cache}}) \cdot \text{ILP}_{\geq}(m_{\text{k}}, m_{\text{cache}}) \\ \wedge \quad & m_k - U \cdot \text{ILP}_{\geq}(m_k, m_{\text{cache}}) \leq m_{\text{cache}} - 1 \\ \wedge \quad & m_k - U \cdot \text{ILP}_{\geq}(m_k, m_{\text{cache}}) \geq m_{\text{cache}} - U \end{aligned} \tag{8}$$

3) Communication-aware optimization: Similarly, ILP formulations that explicitly encode the device-to-device communication cost are needed to enable communication-aware optimization. Thus, such formulations involve determination of whether and where there exists device-to-device communication. For each edge $e_{i,j}$ from node v_i to v_j, if the stage of v_i and v_j are not the same, i.e., $s_i \neq s_j$, the intermediate tensor will introduce a communication cost. Let $com_{k,k+1}$ express the communication cost from stage k to stage $k+1$, and t_{e} represents the volume to communicate on edge e from device k to device $k+1$.

$$com_{k,k+1} = \sum_{e} t_{\text{e}} \cdot \alpha_{\text{e}} \tag{9}$$

where α_{e} is a binary variable to denote if the edge e introduces a communication overhead to stage $k+1$. Specifically, $\alpha_{e_{i,j}}$ is `True` if $s_i \neq s_j \wedge s_j = k+1$. Otherwise, $\alpha_{e_{i,j}}$ evaluates to `False`. As we can see, α_{e} is the logical AND of $\text{ILP}_{\geq}(s_{\text{j}}, s_{\text{i+1}})$ and $\text{ILP}_{=}(s_{\text{j}}, k+1)$. $\text{ILP}_{\geq}(s_{\text{j}}, s_{\text{i+1}})$ is True if s_{j} is greater or equal to $s_{\text{i+1}}$ and $\text{ILP}_{=}(s_{\text{j}}, k)$ is True if s_{j} is equal to $k+1$.

$$\alpha_{\text{e}} = \text{ILP}_{\geq}(s_{\text{i+1}}, s_{\text{j}}) \wedge \text{ILP}_{=}(s_{\text{j}}, k+1) \tag{10}$$

To complete the formulations for α_{e}, it needs ILP formulations for propositional logic, i.e., $\text{ILP}_{\geq}$ (discussed in Equation 8), $\text{ILP}_{=}$ and $\text{ILP}_{\wedge}$. Thus, here we introduce logical AND and Equal in the format of ILP formulations.

Given $y = x_1 \wedge x_2$, where x_1, x_2, y are binary, we have Equation 11, where $y = \text{ILP}_{\wedge}(x_1, x_2)$.

$$\begin{cases} y \geq x_1 + x_2 - 1 \\ y \leq x_1 \\ y \leq x_2 \end{cases} \tag{11}$$

Given $y = \text{ILP}_{=}(x, k, U)$, with integer x, binary variable y, constant k and upper bound U, and we define $y = 1$ if $x = k$ and $y = 0$ if $x \neq k$. Let $-U < x < U, -U < k < U$, we introduce a new binary variable σ to express $y = \text{ILP}_{=}(x, k, U)$, where σ is the inversion of y.

$$\begin{cases} x + y + (U+1)\sigma \geq 1 + k \\ -x + y - (U+1)\sigma \geq -U - k \\ x + Uy - (U+1)\sigma \leq U + k \\ -x + Uy - (U+1)\sigma \leq U - k \end{cases} \tag{12}$$

Example 3 We use the example in Figure 4 to demonstrate the formalism of communication cost for the edge Convolution2 to Add in stage 2. We use com_{e} to represent the communication cost of this edge from Stage 1 to Stage 2, where $com_{\text{e}} = t_e \cdot \alpha_{e_{\text{Conv1,Add}}}$. The complete formulation is shown in Equation 13, where $k = 2$, and e represents $e_{\text{Conv1,Add}}$.

$$\begin{cases} \alpha_{\text{e}} \geq \text{ILP}_{=}(e) + \text{ILP}_{\geq}(e) - 1 \\ \alpha_{\text{e}} \leq \text{ILP}_{=}(e) \\ \alpha_{\text{e}} \leq \text{ILP}_{\geq}(e) \end{cases} \\ \begin{cases} s_{\text{Add}} + \text{ILP}_{=}(e) + (U+1)\sigma \geq 1 + 2 \\ -s_{\text{Add}} + \text{ILP}_{=}(e) - (U+1)\sigma \geq -U - 2 \\ s_{\text{Add}} + U \cdot \text{ILP}_{=}(e) - (U+1)\sigma \leq U + 2 \\ -s_{\text{Add}} + U \cdot \text{ILP}_{=}(e) - (U+1)\sigma \leq U - 2 \end{cases} \\ \begin{cases} s_{\text{Add}} - s_{\text{Conv2}} - U \cdot \text{ILP}_{\geq}(e) \leq 1 - 1 \\ s_{\text{Add}} - s_{\text{Conv2}} - U \cdot \text{ILP}_{\geq}(e) \geq 1 - U \end{cases} \tag{13}$$

4) Data Dependency Optimization: Finally, we introduce the formulation to encode the last domain-specific observation about data dependency in pipeline Edge TPU scheduling. To be specific, we observe that the execution time can be significantly improved if operations that share the same input execute in the same stage, which improves the efficiency of dataflow execution. In other words, for a given operation with out degree ≥ 2, the child nodes need to be scheduled at the same stage.

Example 4 For the example in Figure 4, to add this domain-knowledge, we add extra constraints shown in Equation 14 that restrict (Conv2, Conv1) and (BN, Pool) to be in the same stage.

$$\begin{aligned} & s_{\text{Conv2}} - s_{\text{Conv1}} = 0 \\ & s_{\text{BN}} - s_{\text{Pool}} = 0 \end{aligned} \tag{14}$$

B. Multi-objective Scheduling

With the formulations discussed previously, multiple critical pipeline cost metrics can be used as optimization objectives. According to the motivating example shown in Section III, we will focus on optimizing parameter caching as well as communication cost, which forms a multi-objective optimization problem in ILP solving. Specifically, we formulate three optimization cost functions in the final optimization SDC+ILP formulations: ① maximum per stage memory consumption for loading parameters (cache and DRAM), ② peak memory footprint per stage, and ③ maximum communication cost in n-stage pipelining. Note that while solving the proposed SDC+ILP formulations using ILP solver, the optimal solution (if exists) is produced w.r.t a single minimization or maximization objective. However, it is possible to apply an incremental ILP solving, which optimizes the solution iteratively w.r.t to a sequence of optimization objectives, where the later iteration further refines the solution space on top of previously obtained solution space.

More details about our experimental settings for multi-objective optimization are provided in Section V.

C. System Integration

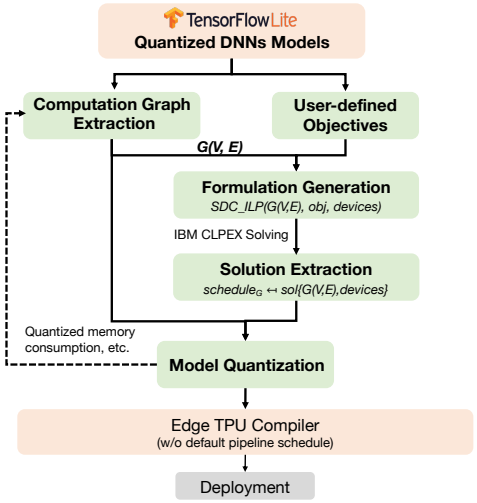


Fig. 5: Overview of end-to-end SDC+ILP scheduling framework for pipelined Edge TPUs system.

We build a framework that integrates the SDC+ILP-based scheduler with TFLite and Edge TPU Compiler to complete the design flow of Edge TPU pipelining (Figure 5). Specifically, the inputs of our framework include a single DNNs model or multiple models for co-deployment, and the number of pipeline stages n in the Edge TPU system. The output of our framework includes n partitioned subgraphs to be deployed on each of the n Edge TPU devices. The flow of our framework includes the following steps:

- *Graph construction* – First, our framework extracts the computation graphs G of the input model(s). If there are more than one input models, our framework constructs a new DAG by introducing a shared sink and source node, which connects the inputs and outputs nodes of all models, respectively. The node and edge attributes are filled up simultaneously.
- *Formulation generator* – Second, multiple sets of formulations are automatically generated with G as inputs w.r.t modeling described in Section IV. Based on the optimization user-inputs, the optimization objectives in the formulation will be adjusted with specific priorities, which enable incremental ILP solving.
- *Solving and solution extraction* – Our framework will deploy IBM ILOG CPLEX to solve the formulations generated in previous step and extract the solutions with a list of tensor matching points w.r.t to the frozen graph of the original model(s). Next, using TFLite `TOCO` converter, our framework produces n subgraphs that will be deployed to specific Edge TPU devices w.r.t the solution.
- *Deployment* – Finally, to map the operations in the subgraphs into Edge TPU specific operations, our framework deploys Edge TPU Complier to deploy the subgraphs to Edge TPUs, without any further optimizations from Edge TPU compiler. Since all Edge TPU models are quantized in INT8 format before deploying to the devices, our scheduling framework takes the INT8 quantization into account while generating the communication and memory caching constrains.

V. Results

In this section, we provide comprehensive evaluations and discussions of the proposed pipeline optimization approaches. Specifically, we compare the optimization performance of three multi-objective optimization approaches enabled by our framework against Coral Edge TPU Compiler (baseline). The experimental results are conducted on eleven popular ImageNet image classification neural network models, using the pipeline Edge TPU system shown in Figure 1. The results are organized as follows: **(1)** We demonstrate the execution runtime improvements over Edge TPU compiler with 2, 3, 4, 5, and 6-stage pipelining setups, using 11 ImageNet models shown in Table III. **(2)** We provide a detailed memory usage among 11 models to explain the runtime speedups obtained with the proposed scheduling approach. **(3)** We physically evaluate the power and energy efficiency improvements over Edge TPU compiler. **(4)** Furthermore, we analyze the complexity and scalability of the proposed approaches. **(5)** Lastly, we demonstrate the application of the proposed framework on multi-model co-deployment on pipeline Edge TPU system.

A. Experimental setups on Edge TPU runtime

The experiments in the rest of this section are conducted on physical computing platforms built with Google Edge TPUs. Specifically, we build a central-hosted pipelined Edge TPU system to execute DNNs inference with configurations as 3-stage, 4-stage, 5-stage, and 6-stage pipelining and evaluate the real-world computation performance improvements for DNNs inference execution. While TPUs can only execute INT8 quantized neural networks, we perform INT8 quantization using TF-Lite with Tensorflow embedded pre-trained models. These models are the inputs to the Edge TPU compiler and our linear formulation scheduling framework. The inference is conducted on Intel 10700K and formulation solving are conducted with Intel Xeon Gold 6230 x20 CPUs. The power and energy measurements are conducted on a precise USB power meter.

B. Inference Performance Evaluation

Our comparison baseline is the commercial version of Edge TPU compiler[1]. The runtime comparisons are evaluated using **eleven** popular ImageNet classification models (Table III), including ResNet50, Xception, ResNet101, ResNet152, ResNet50v2, ResNet101v2, DenseNet121, DenseNet169, DenseNet201, ResNet152v2, InceptionResNetv2. To minimize the impacts of runtime variations in executing DNNs on Edge TPU system, the results included in Figure 6 are the mean runtime of 10 rounds of 1,000 ImageNet inference, using

[1]https://coral.ai/docs/EdgeTPU/compiler/

TABLE III: Statistics of DNNs computational graphs used for evaluating inference runtime on pipelined Edge TPUs Systems

	ResNet50 [19]	Xception [9]	ResNet101 [19]	ResNet152 [19]	ResNet50v2 [19]	ResNet101v2 [42]
$\|\mathbf{V}\|$	177	134	347	517	192	379
$\mathbf{deg(V)}$	2	2	2	2	2	2
Depth	168	125	338	508	184	371
	DenseNet121 [21]	DenseNet169 [21]	DenseNet201 [21]	ResNet152v2 [20]	InceptionResNetv2 [38]	
$\|\mathbf{V}\|$	429	597	709	566	782	
$\mathbf{deg(V)}$	2	2	2	2	4	
Depth	428	596	708	558	571	

the eleven models. Although we select ImageNet models to illustrate the effectiveness of our work, our scheduling framework can be generalized to other neural network models.

Specifically, we evaluate three multi-objective optimization formulations: 1) ① → ③ – first minimizes per stage parameter caching m_{limit} then minimizes per stage communication cost; 2) ① → ② → ③ – minimizes per stage total parameter caching, DRAM usage, and communication cost iteratively; 3) ② → ① → ③ – minimizes per stage DRAM usage, total parameter caching, and communication cost iteratively. The results are summarized in Figure 6, where the horizontal axis represents the runtime performance. Due to the actual runtime speedups varies a lot between different models, we present the normalized results w.r.t runtime obtained with Edge TPU Compiler. The multi-objective optimization SDC+ILP formulations are solved using IBM ILOG CPLEX by assigning the priority of solving objectives in the orders discussed above. Compared to the baseline Edge TPU Compiler, the three multi-objective optimizations provide consistent speedups for all pipelining setups shown in Figure 6. Specifically, the proposed approach offers up to 1.17×, 1.15×, 1.29×, 1.15×, and 4.07× speedups on 2 – 6-stage pipeline Edge TPU, respectively. Moreover, we have observed that the proposed approaches offer significantly more speedups on 6-stage pipeline Edge TPU system. For example, for ResNet101v2 and ResNet152, all three optimization approaches offer more than 4× speedups at inference compared to Edge TPU Compiler.

We provide analysis on experimental phenomena as following. Firstly, the results prove that execution runtime is highly dependent on both total per stage memory consumption (①) and DRAM utilization (②), but ① is more dominated. This has been further confirmed by optimizing the pipeline scheduling w.r.t ② → ③ objectives, which mostly perform worse than Edge TPU compiler and the other three strategies. Thus, we exclude ② → ③ results from the comparison. In addition, combining objectives ① and ② also benefits the solving runtime, which greatly reduces the solution space in the formulation. For example, it is almost infeasible (in runtime) to optimize the schedule by using having ② or ③ only as a single optimization objective. Moreover, we find that the performance can be further optimized by optimizing ① and ② in a sequence, however, not the other way around. For example, while deploying ResNet101v2 in 5-stage system, the performance offered by ① → ③ can be significantly improved by adding ② as the second-priority objective. While mostly ② → ① → ③ strategy performs well, we observe that the results are less consistent. For example, in 4-stage and 5-stage systems, it performs the worst overall on ResNet-based models. We have included examples of differences in memory caching between the three multi-objective solutions and Edge TPU compiler in Figure 7.

Finally, pipeline setting dominates in runtime as the number of stages increases. When deployed to more stages, more DNNs can be implemented on Edge TPUs. For example, after the pipeline stage is set over 6, all sub-models can be compiled on Edge TPUs in cache-only execution by either of the three strategies or the default compiler. However, while deploying on more pipeline stages, the device-to-device communication impacts become more dominant in runtime performance, since it communicates via a relatively slow I/O interface (USB 3.0) compared to on-chip communication. Note that Edge TPU compiler is not communication-aware. Therefore, we have observed more significant speedups on 6-stage pipeline system.

C. Memory Allocation Evaluation

To understand the runtime improvements offered by the proposed scheduling framework, we provide a detailed analysis of memory allocation. For edge computing, on-chip memory size is generally limited. Reloading the parameters from off-chip memory is time-consuming for large inference scenarios. Caching partial or full data on on-chip memory will reduce unnecessary data transfer. Specifically, it will lead to better execution performance and energy efficiency. Thus off-chip memory usage dominates the execution performance. Therefore, we comprehensively compare the memory distribution between Edge TPU compiler and the proposed scheduler (Figure 7). The vertical-axis is the memory usage by Megabytes (MB) that represents off-chip and cache usage. Specifically, if its value is ≥ 0, it refers to no off-chip memory usage (all allocated to cache); otherwise, it means the cache is fully loaded and DRAM usage is reflected in negative value (e.g., -2 MB means 8 MB cache load and 2 MB off-chip DRAM load). Noted that the Edge TPU on-chip cache size is 8 MB.

There are two key conclusions that can be summarized in Figure 7: **(1) The proposed three exact methods (① → ② → ③ , ② → ① → ③ , ① → ③) all produce scheduling solutions that reduce and balance the off-chip memory usage than Edge TPU compiler.** We can see that these three solving strategies have the similar memory distribution and all have lower memory upper bound than Edge TPU compiler, which is the main reason for the runtime

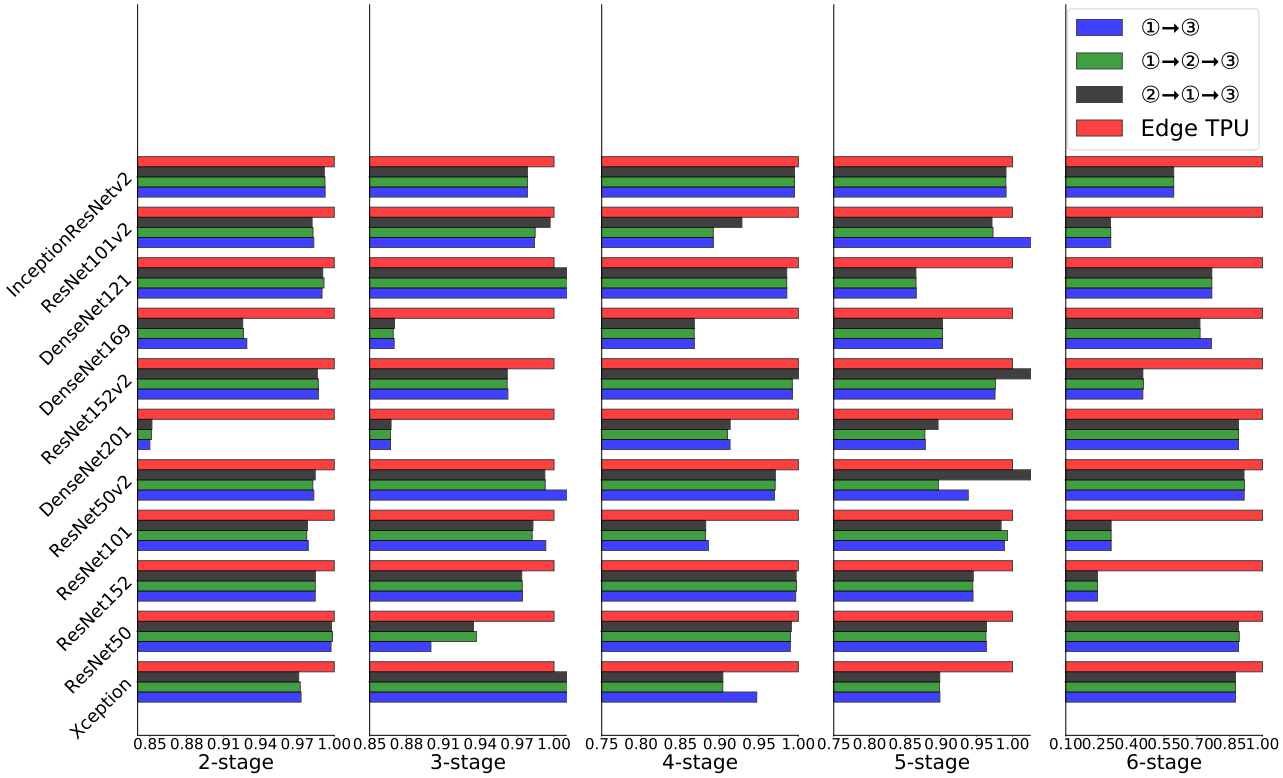
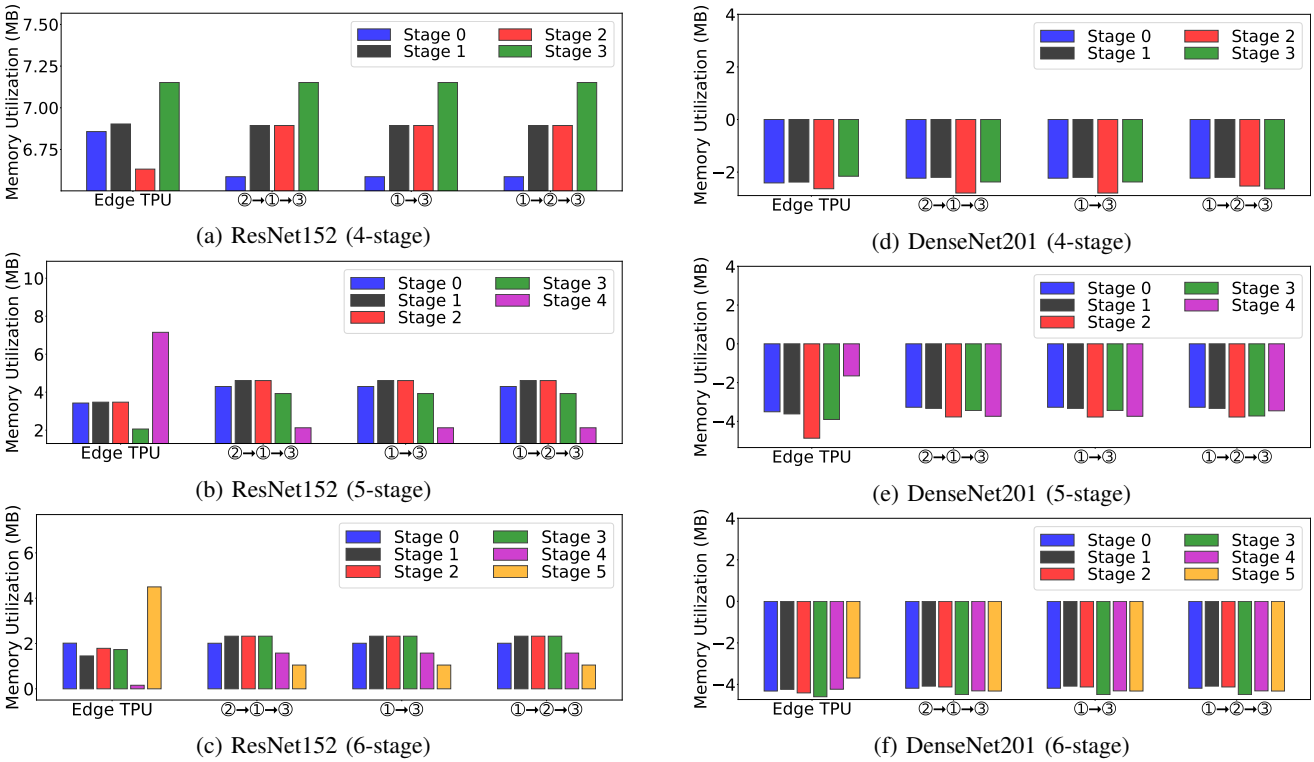


Fig. 6: Edge TPUs inference runtime comparisons between the proposed approaches and Edge TPU compiler (baseline scale=1).



(a) ResNet152 (4-stage)

(b) ResNet152 (5-stage)

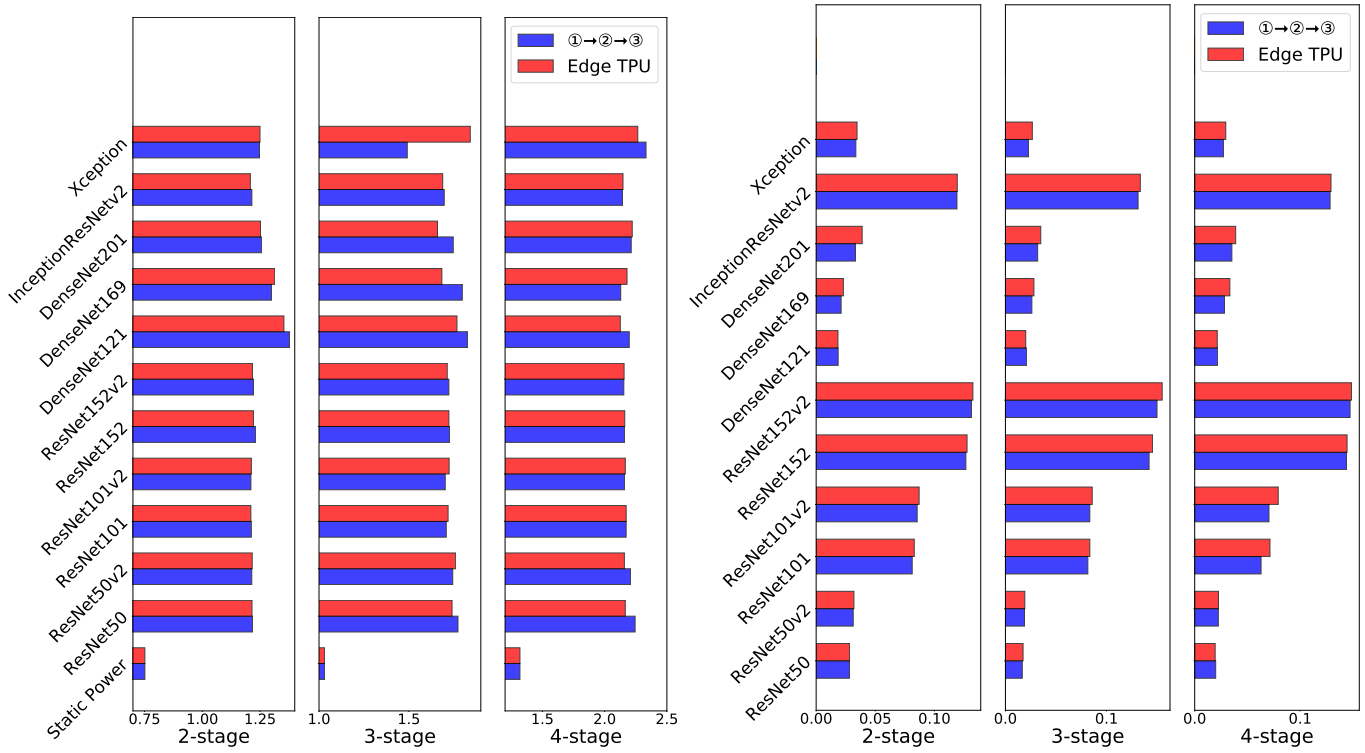
(c) ResNet152 (6-stage)

(d) DenseNet201 (4-stage)

(e) DenseNet201 (5-stage)

(f) DenseNet201 (6-stage)

Fig. 7: **Cache/DRAM usage comparisons between four proposed strategies and Edge TPU compiler uisng ResNet152 and DenseNet201, where vertical-axis represents $m_{\text{cache}} - m_{s_i}$. Therefore, positive values represent parameter caches fully on on-chip cache, and negative values represent the DRAM usage.**



(a) Power in Watt

(b) Energy efficiency in Joule/(frame/second)

Fig. 8: Power and energy cost comparison between our methods (strategy ① → ② → ③) and Edge TPU compiler.

differences. For example, ResNet152 scheduling generated by the SDC+ILP scheduler has much lower memory upper bound than Edge TPU scheduling in 5-stage and 6-stage pipelining. Hence, compared with Edge TPU compiler, we have observed a runtime speedup of 1.06× and 4.07× (① → ② → ③), respectively. On the other hand, DenseNet201 scheduling generated with SDC+ILP has similar memory distribution but improved slightly on the memory upper bound. Compared to Edge TPU compiler, the speedup reduced to 1.12 × in 6-stage pipelining.**(2) Evenly distributed memory and lower upper bound memory usage across all pipeline Edge TPU stages lead to better runtime performance.** For example, the Edge TPU compiler clearly shows unbalanced and higher upper bound memory usage, in which the runtime speedups confirm the domain-specific knowledge, e.g., 5-stage ResNet152 and DenseNet201.

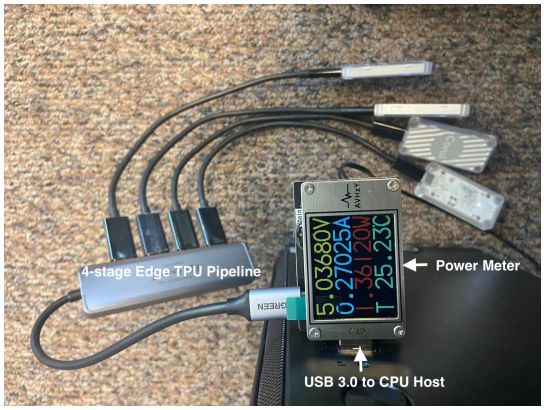


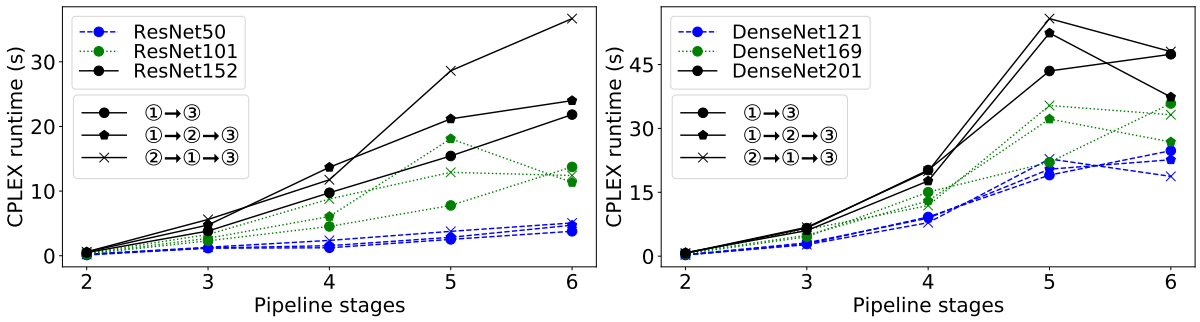
Fig. 9: Power measurement setups with an illustrating example of a 4-stage pipelined Edge TPU system.

D. Power and Energy Evaluation

First, we can see that ILP-based scheduling offers better computation resource utilization due to the exact memory allocation and communication optimization, which is reflected in slightly dynamic power (Figure 8a). Note that edge devices aim to offer high energy efficiency in executing DNNs models, which is discussed in Figure 8b. We can see that our proposed scheduling approach consistently improve the efficiency across all the models tested. For example, in 4-stage ResNet101v2 pipelining, the energy efficiency is improved over Edge TPU compiler, which reduces the cost from 7.92×10^{-2} J/fps to 7.04×10^{-2} J/fps (11.4% reduction).

E. Optimization Complexity and Scalability

To demonstrate the scalability of the proposed approaches, we measure the complexity of solving the exact pipeline scheduling formulation using two different neural architectures (ResNet and DenseNet) with various graph sizes and pipeline stages. While exact optimization schemes mostly require long runtime optimization, the results shown in Figure 10 demonstrate the CPLEX solving runtime is reasonable for



(a) ImageNet ResNet models

(b) ImageNet DenseNet models

Fig. 10: Runtime of the proposed multi-objective scheduling solved using CPLEX.

large DNNs models, and most importantly, is almost linear complexity to the graph size and the number of pipeline stages.

Specifically, Figure 10 includes the CPLEX solving time of the three multi-objective optimization formulations using ResNet50/101/152 and DenseNet121/169/201 models, where the y-axis represents the CPLEX runtime and the x-axis represents the target number of pipeline stages in the Edge TPU system. There are three important takeaways: **(1)** the pipeline schedules of all six models can be optimized within 60 seconds under all three multi-objective optimization constraints, which is efficient for static ahead-of-time compilation. Particularly, the input graph like DenseNet201 includes more than 709 nodes and 708 depth is much more complex than traditional datapath scheduling but still can be solved very effectively. It is believed that the iterative multi-objective ILP solving strategy benefits the solution space pruning, which improves the solving runtime significantly. In addition, in most industrial applications, the deployment process is a one-shot effort. Therefore, for the context of deploying real-world models on edge devices, we believe that the solving runtime in about 60 seconds is acceptable in generating "exact-optimal" solutions. **(2)** runtime increases almost linearly w.r.t to the number of pipeline stages, regardless of the size of the DNNs computational graphs; **(3)** given similarly neural architectures, runtime increases also in linear scale w.r.t to the size of the graphs. As shown in Table III, $|V|$ of ResNet50/101/152 equal to 177, 357, 517, respectively, in which the solving runtime increases linearly. Besides, in Figure 10(b), we can see that the runtime for 6-stage decreases slightly compared to 5-stage optimization. This is because the optimization solution space is relaxed while more cache memory is available for optimizing objectives ① and ②.

F. Enabling Multi-model Co-deployment

The proposed SDC+ILP scheduling framework enables multi-model co-deployment for real-time inference on Edge TPUs. Multi-model co-deployment has a wide range of applications in real-world scenarios, where a single computing platform is usually supporting multiple tasks. For example, Edge TPU compiler supports co-compilation to speed up performance when continuously run multiple models on the same Edge TPUs. However, the Edge TPU compiler does not support co-deployment in pipeline settings. According to Section IV, the proposed versatile formulation is directly applicable to co-deployment, where we build a multi-model DAG by merging the two inputs as a source node and two outputs as a sink model. Figure 11 includes the mean runtime of ResNet101 and DenseNet169 co-deployment with 10 rounds of inference tests. The results demonstrate the multi-model runtime performance can be linear improved as we increased the stages from two to five, which is the ideal case in pipelining on Edge TPU devices.

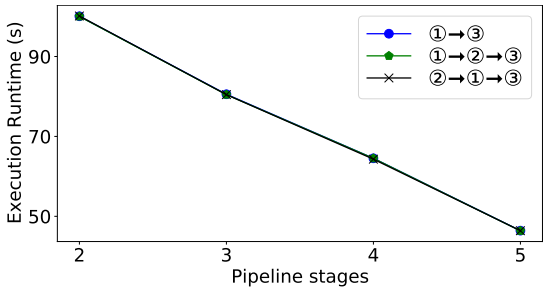


Fig. 11: Execution runtime of co-deployment of RestNet101 and DenseNet169 inference using the proposed framework.

VI. Conclusion

We propose a novel exact scheduling framework that enables versatile multi-objective optimization, including parameter caching, data dependency, and device-to-device communication, in the domain of neural networks. With the proposed novel versatile SDC+ILP formulations that support both propositional logic and non-equality constraints, the proposed framework is capable to perform exact scheduling for a wide range of customized edge devices. Our approaches are evaluated with 2,3,4,5, and 6-stage physical pipeline Edge TPU setups, using eleven popular ImageNet models, with commercial Edge TPU Compiler as the baseline. The proposed approaches demonstrate consistent inference runtime speedups across all pipeline setups, with up to 4× against Edge TPU Compiler, as well as real-world power consumption and energy efficiency (Joules/fps) improvements.

Acknowledgement This work is funded by National Science Foundation (NSF) under NSF-2007832, NSF-2008144, NSF-2019306, and NSF-2019336.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale Machine Learning. *Symp. on Operating System Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[3] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. *arXiv preprint arXiv:2003.02369*, 2020.

[4] Jeff Barnes. Azure machine learning. *Microsoft Azure Essentials. 1st ed, Microsoft*, 2015.

[5] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. *Int'l Conf. on Computer Aided Verification (CAV)*, 2011.

[6] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. Mitigating edge machine learning inference bottlenecks: An empirical study on accelerating google edge models. *arXiv preprint arXiv:2103.00768*, 2021.

[7] Hongzheng Chen and Minghua Shen. A deep-reinforcement-learning-based scheduler for fpga hls. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[8] 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. *Symp. on Operating System Design and Implementation (OSDI)*, pages 578–594, 2018.

[9] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[10] Jason Cong and Zhiru Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. *Design Automation Conf. (DAC)*, 2006.

[11] Steve Dai, Gai Liu, and Zhiru Zhang. A Scalable Approach to Exact Resource-constrained Scheduling based on a Joint SDC and SAT Formulation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[13] Leonardo De Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 2011.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] Bruno Dutertre. Yices 2.2. *Int'l Conf. on Computer Aided Verification (CAV)*, 2014.

[16] Christodoulos A Floudas and Xiaoxia Lin. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research*, 139(1):131–162, 2005.

[17] Thomas James Fox, Eric R Kern, and Michael Scott Rollins. Autonomous Fail-over to Hot-spare Processor using SMI, 2007. US Patent 7,251,746.

[18] Rida M Hamza, Michael E Bazakos, and Murray J Cooper. Face Identification Verification using 3 Dimensional Modeling. 2008. US Patent 7,421,097.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[21] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[23] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, pages 1–12, 2017.

[24] Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(3):355–383, 2003.

[25] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[26] Charles E Leiserson and James B Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[27] Christophe Lenté, Mathieu Liedloff, Ameur Soukhal, and Vincent t'Kindt. Exponential algorithms for scheduling problems. 2014.

[28] TensorFlow Lite. Ml for mobile and edge devices, 2020.

[29] Sharad Malik and Lintao Zhang. Boolean Satisfiability from Theoretical Hardness to Practical Success. *Communications of the ACM*, 2009.

[30] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[31] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems (NIPS)*, pages 8024–8035, 2019.

[33] Pierre G Paulin and John P Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[34] Alexander Ratner, Dan Alistarh, Gustavo Alonso, Peter Bailis, Sarah Bird, Nicholas Carlini, Bryan Catanzaro, Eric Chung, Bill Dally, Jeff Dean, et al. SysML: The New Frontier of Machine Learning Systems. *arXiv preprint arXiv:1904.03257*, 2019.

[35] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv preprint arXiv:1805.00907*, 2018.

[36] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[37] Shuran Sheng, Peng Chen, Zhimin Chen, Lenan Wu, and Yuxuan Yao. Deep reinforcement learning-based task scheduling in iot edge computing. *Sensors*, 21(5):1666, 2021.

[38] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[39] Chao Wang, Franjo Ivančić, Malay Ganai, and Aarti Gupta. Deciding Separation Logic Formulae by SAT and Incremental Negative Cycle Elimination. *Logic for Programming, Artificial Intelligence, and Reasoning*, 2005.

[40] Tao Yang and Apostolos Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[41] Amir Yazdanbakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. An evaluation of edge tpu accelerators for convolutional neural networks. *arXiv preprint arXiv:2102.10423*, 2021.

[42] Bo Yu, Lu Yang, and Fang Chen. Semantic segmentation for high spatial resolution remote sensing images based on convolution neural network and pyramid pooling module. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(9):3252–3261, 2018.

[43] Cunxi Yu, Chau-Chin Huang, Gi-Joon Nam, Mihir Choudhury, Victor N Kravets, Andrew Sullivan, Maciej Ciesielski, and Giovanni De Micheli. End-to-end industrial study of retiming. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 203–208. IEEE, 2018.

[44] Zhiru Zhang and Bin Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2013.