Efficient Batched Inference in Conditional Neural Networks

Surya Selvam, Amrit Nagarajan, and Anand Raghunathan, *Fellow, IEEE*

***Abstract*—Conditional neural networks (NNs) are networks in which the computations performed vary based on the input. Many NNs of interest (such as autoregressive transformers for sequence generation tasks) are inherently conditional since they process variable-length inputs or produce variable-length outputs. In addition, popular NN optimization techniques, such as early exit, result in the computational footprint varying across inputs. Computational irregularity across inputs presents a challenge to batching, a technique widely used to improve hardware utilization and throughput during NN inference. To address this challenge, we propose BatchCond, an optimized batching framework for Conditional NNs that consists of two key steps: 1) computational similarity-driven batching (SimBatch) and 2) adaptive batch reorganization (ABR). SimBatch utilizes a lightweight DNN predictor to create batches of inputs that are more likely to share similar computational patterns, thereby reducing computational irregularity. Further, ABR addresses residual irregularity by dynamically splitting batches into computationally similar sub-batches in a hardware-aware manner. Our experiments demonstrate that BatchCond improves the overall throughput of batched inference by up to 6.6× (mean of 2.5×) across a suite of diverse Conditional NNs, including early-exit networks, dynamic slimmable networks, and autoregressive transformers. Code is available at https://github.com/surya00060/BatchCond.**

***Index Terms*—Batching, conditional neural networks (NNs), early exit, hardware-aware inference, large language models (LLMs), NNs, transformers.**

I. Introduction

Neural networks (NNs) have achieved remarkable success in various domains, including computer vision [1], [2], [3], natural language processing [4], [5], [6], [7], [8], and audio processing [9], [10], and are used in many real-life applications, such as chatbots [11], [12], language translators [13], [14], photo editors [15], document processors [16], etc. As a result, NNs are executed on a wide spectrum of devices with varying computational and storage capabilities, ranging from resource-constrained devices (mobile phones, AR/VR headsets, smart watches, etc.) to large cloud servers. Across this entire spectrum, batching, which refers to the simultaneous processing of multiple inputs, is a technique commonly used to improve execution efficiency. When NNs are deployed for inference on cloud servers, they receive inputs simultaneously from multiple users. For instance, widely used services, such as voice search [17] and chatbots [12], receive thousands to tens of thousands of queries per second. These input queries are commonly batched together and processed concurrently, improving throughput by 1) increasing the utilization of highly parallel hardware platforms and 2) reducing data movement costs by increasing reuse of the NN's weights across inputs in a batch.

Batching is most effective when all inputs in a batch share the same computational pattern, thereby enabling fully parallel load-balanced execution across processing elements (PEs) in the underlying hardware platform. However, many popular NNs are inherently conditional, with different inputs activating different parts of the network and/or requiring different amounts of computational effort (Fig. 1). Transformers are a notable example of Conditional NNs, since the computational effort they expend directly varies based on the length of the input sequence (e.g., number of words or tokens). This variation is accentuated by the fact that the computational complexity of attention scales quadratically with input length. Similarly, autoregressive transformers used for sequence generation tasks like machine translation produce outputs in decoding steps, with different numbers of decoding steps executed for different inputs. Variable computational effort has also been shown to be a promising approach to reducing the processing requirements of NNs [18], [19]. Some notable examples include early-exit networks, which modulate network depth dynamically [20], [21], and slimmable networks, which modulate network width dynamically [22]. The computational irregularity present in Conditional NNs manifests as control flow divergence and load imbalance in the underlying hardware platform, degrading the efficiency of batched execution.

Due to the challenges of batching in Conditional NNs, prior works either use a batch size of one [20], [21] or perform ineffectual computations to maintain regularity [5], [23]. Each of these approaches has drawbacks. Executing inputs at a batch size of one leads to hardware underutilization and adversely impacts throughput. The alternative approach pads the data and/or computations to maintain regularity. For instance, data padding is performed in transformers by adding padding tokens to shorter sequences to equalize the lengths of all

Manuscript received 12 August 2024; accepted 13 August 2024. This work was supported in part by CoCoSys, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) Program sponsored by DARPA, and in part by the National Science Foundation under Award 2318101. This article was presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Surya Selvam.)*

Surya Selvam and Anand Raghunathan are with the Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA (e-mail: selvams@purdue.edu; raghunathan@purdue.edu).

Amrit Nagarajan was with the Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA. He is now with IBM TJ Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: Amrit.Nagarajan@ibm.com).

Digital Object Identifier 10.1109/TCAD.2024.3445263

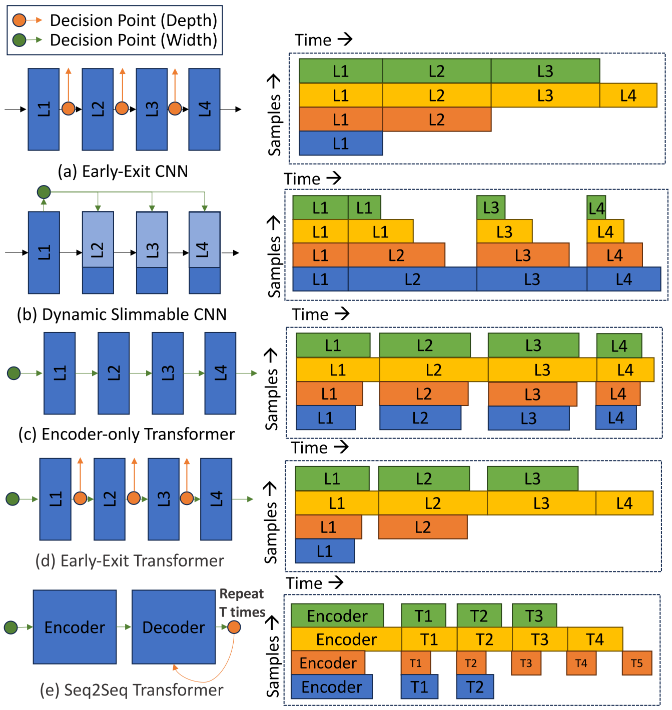

Fig. 1. Examples of conditional NNs and their execution traces depicting varying computations across inputs in a batch. Early-exit NNs and dynamic slimmable NNs selectively activate different parts of the model for each input, while the computational effort for transformers varies based on input and/or output length. Decision points are points in the network where control flow diverges for different inputs. (a) Early-exit CNN. (b) Dynamic slimmable CNN. (c) Encoder-only transformer. (d) Early-exit transformer. (e) Seq2Seq transformer.

sequences in a batch, resulting in fixed computational effort for all sequences. Padding ensures computational regularity, but the redundant computations lead to increased latency and energy consumption.

To overcome the aforementioned challenges, we propose BatchCond, a framework for optimized batched inference in Conditional NNs. BatchCond utilizes two complementary optimizations: 1) computational similarity-driven batching (SimBatch) and 2) adaptive batch reorganization (ABR). SimBatch identifies inputs that are likely to share similar computational patterns by using a lightweight DNN-based predictor, and groups them to form batches. Thus, SimBatch decreases computational irregularity among inputs in a batch, leading to improved hardware utilization with fewer redundant computations. ABR addresses any residual computational irregularity by dynamically splitting batches into sub-batches in a hardware-aware manner (i.e., when doing so is likely to result in improved throughput). We summarize our main contributions as follows.

1) We propose BatchCond, a framework for efficient batched inference in Conditional NNs. To the best of our knowledge, BatchCond is the first general framework for improving throughput during batched inference in all types of Conditional NNs.
2) We propose computational similarity-driven batching (SimBatch) to create batches of inputs that are likely to share similar computational patterns, thereby reducing intrabatch computational irregularity.
3) We introduce ABR to address residual computational irregularity by dynamically reorganizing batches into computationally similar sub-batches.
4) Across a suite of five diverse Conditional NNs, we demonstrate that BatchCond improves throughput by up to 6.6× (average of 2.5×) compared to existing methods.

The remainder of this article is organized as follows. Section II provides an overview of Conditional NNs, and outlines the challenges they present to batched inference. Section III introduces the BatchCond framework and describes the constituent steps in detail. Our experimental setup is described in Section IV, and the results of our experiments are presented in Section V. Section VI describes existing efforts closely related to our work, and Section VII concludes this article.

II. Preliminaries

This section provides a brief overview of Conditional NNs and outlines the challenges of performing batched inference therein.

A. Conditional Neural Networks

1) Definition and Taxonomy: In this work, we define Conditional NNs as NNs that satisfy one or more of the following criteria.

1) *CondNN.1:* The computations performed are not the same for all possible inputs.
2) *CondNN.2:* The same set of weights and biases are not used to process all possible inputs.

We broadly categorize Conditional NNs into three types based on the attribute of the NN that is modulated: 1) Conditional-Depth NNs; 2) Conditional-Width NNs; and 3) Conditional-Depth+Width NNs. At a high level, Conditional-Depth NNs use different numbers of layers to process each input, while Conditional-Width NNs use different activation sizes and/or numbers of weights in each layer (but use the same number of layers) for different inputs. Conditional-Depth+Width NNs modulate both the number and sizes of layers for different inputs. Conditional NNs adjust the computational effort expended on each input based on outcomes at *decision points*. We define decision points as locations in the computational graph where control flow diverges across different inputs.

2) Examples of Conditional NNs: Table I provides representative examples of Conditional NNs from the literature, along with their types and decision points. Decision points are also illustrated using examples in Fig. 1. For instance, early-exit NNs [20] can process *easy* samples without having to execute all layers of the NN by using side-branch classifiers. As a result, fewer computations are performed on *easy* samples compared to *difficult* samples by activating only a subset of all weights and biases in the NN (satisfying conditions CondNN.1 and CondNN.2). On the other hand, inputs to large language models (LLMs) [7], [8] can have different lengths, since real-world text inputs can be arbitrarily long. As a result, more computations are performed on longer sequences (since more tokens need to be processed) compared to shorter sequences.

TABLE I
TAXONOMY OF CONDITIONAL NNS

Axes of Conditionality	Examples	Conditionality Criteria	Decision Points
Depth	Early Exit NNs [20], [24]	CondNN.1,CondNN.2	Side-branch predictor at the end of each layer
	Layer Skipping NNs [25]–[27]	CondNN.1,CondNN.2	Classifiers that determine whether to skip each layer
Width	Dynamic Slimmable NNs [22], [28]–[30]	CondNN.1,CondNN.2	Classifiers that determine the number and/or sizes of filters used in each layer
	Encoder-only Transformers [6], [31]	CondNN.1	Sequence length of the input
	Mixture of Experts [32], [33]	CondNN.2	Classifiers that determine the expert chosen at each layer
Depth + Width	Channel and Layer Skipping NNs [34]–[36]	CondNN.1,CondNN.2	Classifiers that determine whether to skip each layer, and the widths of non-skipped layers
	Seq2Seq NNs [4], [5], [37], [38]	CondNN.1	Sequence length of input, and check for EOS token at output
	Early Exit Transformers [21], [39], [40]	CondNN.1,CondNN.2	Sequence length of input and side-branch predictors at the end of each encoder/decoder layer
	Large Language Models [7], [8]	CondNN.1	Sequence length of input, and check for EOS token at output

Therefore, even though the same weights and biases are used for sequences of different lengths, LLMs are conditional since they satisfy CondNN.1.

B. Batched Inference and Its challenges in Conditional NNs

1) Batched Inference: During batched inference, multiple inputs are processed in parallel in order to better utilize the available hardware resources. Batched inference in parallel hardware systems involves the following steps.

1) When NNs are deployed for inference, they receive inputs simultaneously from multiple sources, which are then concatenated to form batches. Inputs that are each of shape (h, w) are combined along a *batching axis* to form a batched input of shape (b, h, w). Here, b is the batch size, and b is chosen such that all weights and activations fit in device memory.
2) At the start of execution, weights and biases of the first layer of the NN are loaded from off-chip device memory to on-chip scratchpad memory, where the input activations reside.
3) PEs perform the necessary computations by reading weights and activations from the scratchpad, and writing outputs back into the scratchpad.
4) Then, weights of the second layer of the NN are loaded into the scratchpad (commonly pipelined by overlapping memory transfers with computations on the first layer), and this process is repeated for all layers.

As multiple inputs in a batch are processed in parallel in step 3, the cost of data movement for weights and biases is amortized across all samples in a batch, instead of being repeated for each input as is done when $b = 1$. In addition, modern parallel engines [41], [42], [43], [44] contain large numbers of PEs to allow for massively parallel matrix multiplications. Consequently, the number of computations when $b = 1$ is not large enough to fully utilize all the available PEs, leading to underutilization. Batched inference takes advantage of this underutilization to process multiple inputs in parallel, thereby improving throughput.

2) Challenges in Conditional NNs: Massively parallel hardware accelerators, such as GPUs [41], [42] and TPUs [43], [44], are designed to exploit the implicit parallelism present in NN workloads for maximum performance. Unlike traditional CPUs with branch predictors and reorder buffers, parallel accelerators have orders-of-magnitude more compute units (PEs). However, each compute unit has a much simpler control path. For instance, in GPUs, the control flow is shared among a group of threads (called warps), which perform computations together in lockstep. Similarly, matrix multiplications can be realized on systolic arrays in TPUs by orchestrating the inputs and weights using a predefined dataflow with no explicit control. The regularity of the computations involved enables *parallel vector/matrix operations to be executed with a scalar control input*, thereby maximizing compute throughput. In summary, since modern parallel systems tradeoff complicated control logic for more compute units, workloads that require fine-grained control are executed inefficiently. For instance, when a simple `if then else` code block is executed on GPUs, certain PEs stall and wait for other PEs to complete execution (since GPUs always execute in lockstep fashion), leading to poor hardware utilization.

When the exact same computations are performed on all samples in a batch, they can be efficiently executed in parallel due to the regularity of computations. However, the computations performed on different samples in a batch are different in Conditional NNs as illustrated in Fig. 1, making them ill-suited to batched inference. For instance, in early-exit NNs, different samples in the batch exit at different layers. Consequently, if some inputs exit at layer i while other inputs exit only at layer $i+j$, then PEs assigned to the exited samples remain idle during execution of layers $i+1$ to $i+j$ for the late-terminating samples. In slimmable NNs, samples executed at smaller width are processed faster than samples requiring larger widths, leading to underutilization in PEs processing samples at smaller width. In transformers, shorter sequences in a batch finish execution earlier than longer sequences, leaving PEs assigned to shorter sequences idle while waiting for longer sequences to finish execution. Similarly, during machine translation, words are generated one at a time. Therefore, inputs leading to longer translated outputs require more decoding steps, leaving PEs assigned to inputs with shorter translated outputs idle. In summary, batched-inference in Conditional NNs introduces control flow divergence among samples in a batch due to the varying outcomes at each decision point, leading to hardware underutilization and hence, reduced throughput.

Moreover, existing methods perform compute and/or data padding to execute batches with divergence by introducing ineffectual computations. Conditional-Depth NNs use compute padding, whereas Conditional-Width NNs use data padding. For instance, in early-exit networks, where the execution

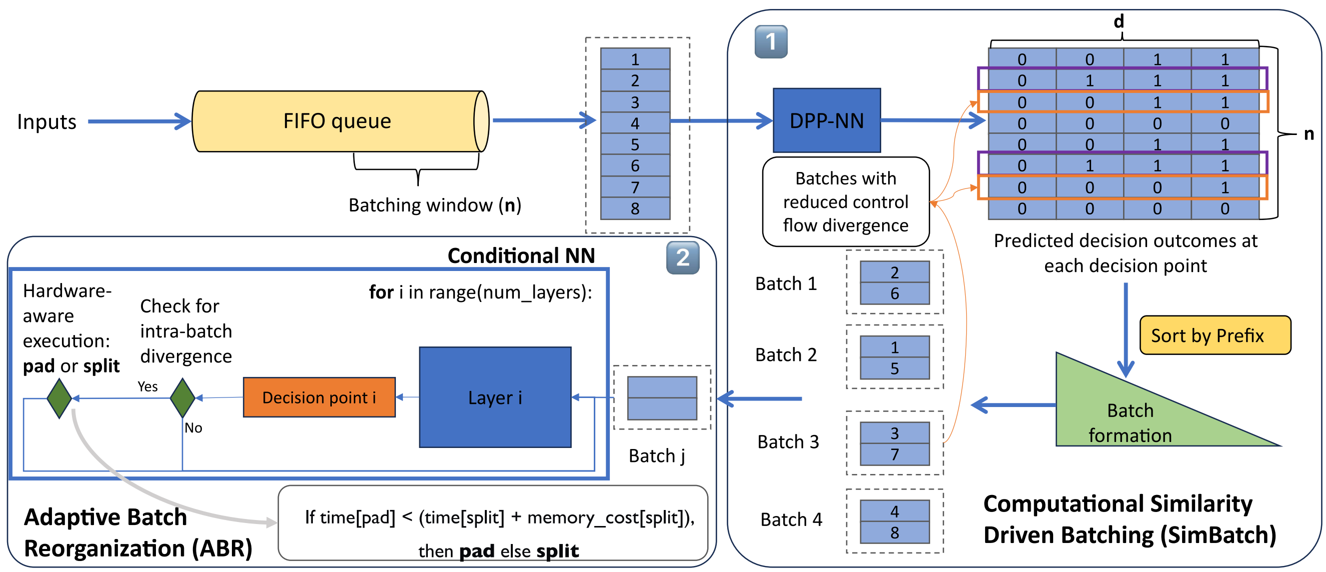

Fig. 2. Overview of the BatchCond framework, which consists of two key components—SimBatch and ABR. SimBatch forms batches of samples that are likely to share similar computational patterns. ABR optimizes the execution of batches in the presence of residual computational irregularity by dynamically choosing between padding and sub-batch splitting.

of each input is terminated at a different layer, compute padding is performed to ensure all samples in a batch are executed up to the maximum depth required by samples in the batch. For instance, in a batch with two samples, if the first sample exits at layer i, and the second sample exits at layer $i+j$, both samples are executed up to layer $i+j$ to maintain regularity. Consequently, unnecessary computations are performed on the first sample, thereby increasing latency of the first sample, while also preventing PEs from doing useful work. On the other hand, in transformers, each input requires a variable amount of computational effort based on the length of the input. However, all samples in a batch are padded to the length of the longest sequence in the batch, thereby introducing ineffectual computations and adversely impacting latency of shorter sequences in each batch. In summary, batched inference in Conditional NNs presents a distinct challenge due to varying computational requirements across inputs in a batch.

III. BatchCond Framework

BatchCond is a framework that optimizes batched inference in Conditional NNs using two complementary techniques. The first technique, computational similarity-driven batching (SimBatch), batches samples that are likely to lead to the same outcomes at each decision point in the Conditional NN, thereby minimizing computational irregularity. The second technique, ABR, optimizes execution in the presence of residual computational irregularities that remain after SimBatch. Fig. 2 provides an overview of the BatchCond framework. We explain SimBatch and ABR in greater detail in the following sections.

A. Computational Similarity-Driven Batching

The overall goal of SimBatch is to create batches of samples that are likely to share decision point outcomes, and hence, require the same computations. However, decision point outcomes are made during runtime and are unknown prior to sample execution. We address this challenge by creating a decision point predictor NN (DPP-NN), which is a lightweight NN that predicts the outcomes of different decision points in the Conditional NN for a given input sample prior to execution. The outputs of DPP-NN on a set of input samples are used to create batches of samples that are predicted to share computational patterns. Our procedure for designing the DPP-NN for a given Conditional NN is as follows.

1) *Data Collection:* The training dataset for the DPP-NN is generated by collecting decision outcomes at all decision points in the Conditional NN for each sample in the training dataset. This is done by performing inference on the training dataset using the trained Conditional NN.
2) *Model Initialization:* The DPP-NN shares the same architecture as the first layer of the Conditional NN, and its weights are initialized from the same. Then, a fully connected regression head is added to the model. The size of the output produced by the regression head is equal to the number of decision points in the Conditional NN, with each entry in the output predicting the outcome of the corresponding decision point. Since the DPP-NN uses only one layer of the Conditional NN, its runtime is only a small fraction of the Conditional NN's runtime, thereby limiting the overheads
3) *Model Training:* The DPP-NN is trained till convergence on the collected dataset.

During inference, all samples are passed through the DPP-NN to predict the outcomes at different decision points. Then, the samples with similar decision outcomes are batched together and fed to the Conditional NN for processing. In effect, SimBatch reduces the intrabatch control flow divergence, leading to higher utilization and hence, enhanced throughput. We note that we do not use the predicted outcomes to control execution, i.e., if the predicted outcome does not

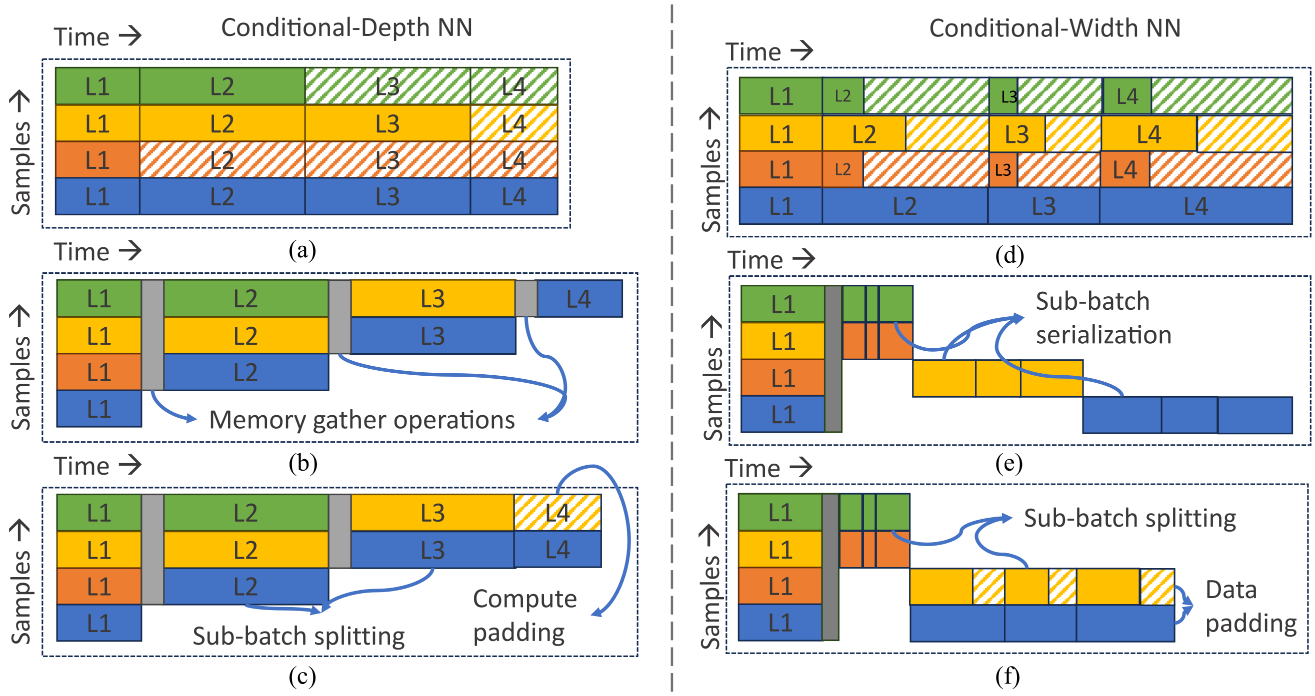

Fig. 3. Execution strategies for batches with control flow divergence in conditional-depth NNs [left, (a)–(c)] and conditional-width NNs [right, (d)–(f)]. (a) Compute padding. (b) and (e) Sub-batch splitting. (c) and (f) ABR. (d) Data padding.

match the actual outcome at a decision point, execution flow is decided based on the actual outcome and not the predicted outcome. Thus, there is no impact on accuracy.

B. *Adaptive Batch Reorganization*

SimBatch forms batches with reduced control flow divergence. However, since predictions from the DPP-NN are used to batch samples, it is unlikely that all batches will have zero divergence. In addition, input samples may have strict latency constraints (deadlines), and hence, cannot remain in the queue until other computationally similar inputs arrive. Either of these factors may result in the execution of computationally irregular batches. Therefore, BatchCond incorporates optimizations to deal batches with control flow divergence.

Existing approaches deal with computational irregularity within a batch by padding data and/or computation, as described in Section II-B, leading to considerable overheads. In order to address the shortcomings of existing padding-based approaches, we propose ABR. The overarching idea in ABR is to dynamically select between batch splitting and data or compute padding. This is performed in a hardware-aware manner by precharacterizing conditions under which each of these alternatives is beneficial. The specifics of ABR are different for Conditional-Depth and Conditional-Width NNs, hence we describe it in each context in the following sections.

1) Adaptive Batch Reorganization for Conditional-Depth NNs: We observe that compute padding used in Conditional-Depth NNs [Fig. 3(a)] leads to ineffectual computations, thereby adversely impacting throughput. To address this challenge, we propose sub-batch splitting, an optimized execution strategy [Fig. 3(b)]. Sub-batch splitting splits the batch into two sub-batches at each decision point, with one sub-batch containing all samples that terminated at the decision point, and the other sub-batch containing samples that did not terminate.[1] Then, sub-batch splitting continues execution of only the sub-batch with nonterminated samples, thereby eliminating the need for compute padding and the resulting ineffectual computations.

While sub-batch splitting eliminates ineffectual computations, it adds memory copy overheads during execution at each decision point. Splitting batches into sub-batches of terminated and nonterminated samples involves the following steps: 1) *indexing:* positions of nonterminated samples in the batch are obtained based on decision outcomes and 2) *tensor gathering:* a new sub-batch is created by gathering nonterminated samples from the original batch. Modern parallel systems require tensors to be in contiguous memory locations (Fig. 4) to maximally exploit parallelism. Consequently, when nonterminated samples reside in noncontiguous memory locations, they need to be gathered and copied to contiguous locations in memory, resulting in overheads. As a result, sub-batch splitting does not always improve throughput over compute padding [Fig. 5(a)]. In particular, we observe that the memory overheads of sub-batch splitting outweigh the impact of performing fewer computations in two scenarios.

[1] We use the term *terminated samples* to refer to those samples whose execution is halted between the current decision point and the next decision point. For instance, in layer-skipping NNs, if layer l is skipped for a sample, then it is placed in the batch of terminated samples at the decision point immediately before layer l, and the sample is re-evaluated at the following decision point. On the other hand, in early-exit NNs, if a sample is terminated at layer l, it is retained in the batch of terminated samples till the end of execution.

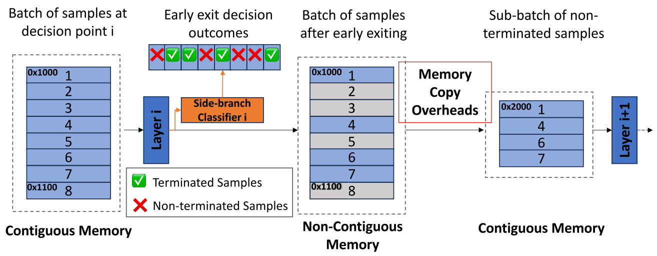

Fig. 4. Memory overheads introduced while splitting a batch into two sub-batches of terminated and nonterminated samples in early-exit NNs.

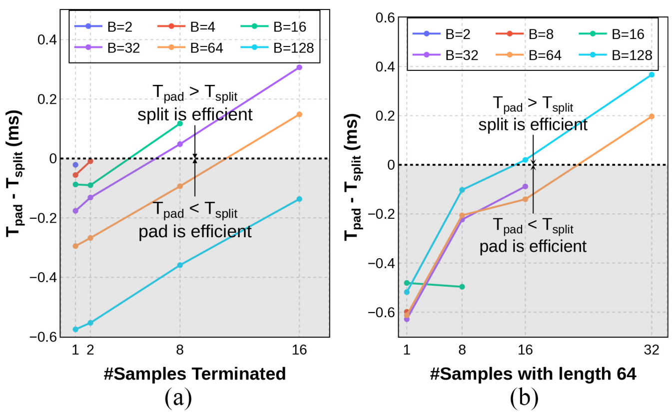

Fig. 5. Difference between compute/data padding T_{pad} and sub-batch splitting T_{split} execution times. (a) Execution time for the final residual block of ResNet-34 with early exit (Conditional-Depth) for different numbers of samples in a batch exiting after the prefinal block. (b) Execution time for the first encoder layer of BERT-base (Conditional-Width) for different numbers of samples with a length of 64 (remaining samples in the batch are of length 96). B indicates the batch size.

1) When the size of sub-batch containing terminated samples is much smaller than the size of the sub-batch containing nonterminated samples, the computational savings from sub-batch splitting are small. On the other hand, the memory overheads are large tensors containing activations corresponding to nonterminated samples need to be copied into the new sub-batch. Since memory cost scales linearly with number of copies performed, the overheads outweigh the computational savings from using sub-batch splitting, making compute padding more efficient than sub-batch splitting.
2) When the batch sizes are low, the hardware is underutilized. As a result, the ineffectual computations from compute padding do not have any impact on throughput, since these computations are performed by PEs that would otherwise be idle. On the other hand, sub-batch splitting incurs overheads due to memory copies, but the computational savings from sub-batch splitting are not beneficial in any way. As a result, execution with sub-batch splitting is slower than execution with padding.

Based on these observations, we propose ABR that combines the best of both worlds. In particular, ABR finds the best combination of padding and sub-batch splitting to maximize throughput [Fig. 3(c)]. At each decision point i, we check if the time taken for executing layers between decision points i and $i+1$ with padding (T_{pad}) is less than the time taken for execution with sub-batch splitting (T_{split}). If $T_{\text{pad}} > T_{\text{split}}$, we execute the layers between decision points with sub-batch splitting, and vice-versa.

When executing a batch of size B, assume E samples are terminated at decision point i. Then, T_{pad} is equal to the time taken to execute layers between i and $i+1$ for a batch of size B. On the other hand, T_{split} is equal to the time taken to execute layers between i and $i+1$ for a batch of size $B-E$ plus the time taken to create the new sub-batch with $B-E$ nonterminated samples. Let $T_i[B]$ be the time taken to execute layers between decision points i and $i+1$ for a batch size of B. Then

$$\begin{aligned} T_{\text{pad}} &= T_i[B] \\ T_{\text{split}} &= T_i[B-E] + T_{\text{gather}}[B-E] \end{aligned}$$

$$\text{Execution Strategy} = \begin{cases} \text{pad} & \text{if } T_{\text{pad}} < T_{\text{split}} \\ \text{split} & \text{if } T_{\text{pad}} > T_{\text{split}}. \end{cases} \quad (1)$$

2) Adaptive Batch Reorganization for Conditional-Width NNs: We find that data padding used in Conditional-Width NNs [Fig. 3(d)] leads to ineffectual computations on padding data, thereby adversely affecting throughput. Similar to the Conditional-Depth case, we propose a compute-optimal sub-batch splitting strategy that eliminates the need for padding [Fig. 3(e)]. Sub-batch splitting splits the batch into multiple sub-batches at each decision point, with each sub-batch containing all samples that need to be executed at the same width. Consequently, the number of sub-batches generated at a decision point is equal to the number of possible width value outcomes at the decision point. Then, sub-batch splitting executes each sub-batch sequentially, thereby eliminating the need for data padding and the resulting ineffectual computations.

Despite being compute-optimal, sub-batch splitting incurs batch-splitting overheads, similar to the Conditional-Depth case. In addition, sub-batch splitting serializes the execution of different sub-batches in Conditional-Width NNs. (In contrast, only one sub-batch is executed in Conditional-Depth NNs, since the other sub-batch contains only terminated samples.) However, we also note that sub-batches requiring smaller width can be executed substantially faster than data padded batches that must be executed at the largest width required by all samples in the batch. As a result, the serialization overheads always scale sublinearly with number of sub-batches formed. Consequently, we find that sub-batch splitting is faster than data padding only when the hardware is compute-bound (i.e., all PEs are fully utilized) as shown in Fig. 5(b). In addition, we also find that the serialization overheads of sub-batch splitting can be reduced by merging sub-batches that are not large enough to fully utilize the hardware into larger batches. At a finer granularity, some samples from sub-batches requiring smaller widths can be moved into sub-batches requiring larger widths (using data padding) to ensure that all sub-batches fully utilize the hardware, thereby maximizing throughput.

Based on these observations, we propose ABR to find the best combination of padding and sub-batch splitting to maximize throughput [Fig. 3(f)]. At each decision point i, we first find the time taken for execution with sub-batch splitting

Algorithm 1 Hardware-Aware Batch Splitting at Each Decision Point

Require: *sub_batches* Sub-batches of samples requiring same widths; the number of sub-batches is equal to the number of possible width outcomes

Require: *ideal_batch_sizes* smallest batch size that fully utilizes the hardware for each possible width

```
sub_batches.sort_by_decreasing_width()
for i = 1 to num(sub_batches) do
    if size(sub_batches[i]) ≥ ideal_batch_sizes[i] then
        continue
    end if
    for j = i to num(sub_batches) do
        num_samples_to_add = ideal_batch_sizes[i] - size(sub_batches[i])
        Move num_samples_to_add samples from sub_batches[j] to sub_batches[i]
        if size(sub_batches[i]) ≥ ideal_batch_sizes[i] then
            break
        end if
    end for
end for
return sub_batches
```

(T_{split}) using the best arrangement of samples into sub-batches. We find this best arrangement using the procedure described in Algorithm 1. In particular, we identify sub-batches that are not large enough to fully utilize the hardware, and move as many samples as needed from sub-batches requiring smaller widths to enable full utilization. In effect, our hardware-aware batch-splitting method reduces serialization overheads by ensuring that all sub-batches are large enough to fully utilize the hardware, while also minimizing the amount of padding introduced while merging sub-batches. Subsequently, we check if the time taken for executing layers between decision points i and $i+1$ with padding (T_{pad}) is less than the time taken for execution with sub-batch splitting (T_{split}). If $T_{\text{pad}} > T_{\text{split}}$, we execute the layers between decision points with sub-batch splitting, and vice-versa.

When executing a batch of size B, assume there are k outcomes at decision point i, resulting in $b_1, b_2, .., b_k$ samples that require execution at width $w_1, w_2, .., w_k$, respectively, such that $b_1 + b_2 + .. + b_k = B$ and $w_1 < w_2 < .. < w_k$. Then, T_{pad} is equal to the time taken to execute layers between i and $i+1$ for a batch of size B at the maximum width w_k. On the other hand, T_{split} is equal to the time taken to execute layers between i and $i+1$ for each sub-batch serially at their respective widths. We obtain the best arrangement of samples $s_1, s_2, .., s_k$ that require execution at width $w_1, w_2, .., w_k$, respectively, using Algorithm 1, such that $s_1 + s_2 + \cdots + s_k = B$, by moving samples requiring smaller width to larger width to make sub-batches compute bound. Let $T_i[B, W]$ be the time taken to execute layers between decision points i and $i+1$ for a batch size of B at maximum width of $W = w_k$. Then

$$T_{\text{pad}} = T_i[b_1 + b_2 .. + b_k, w_k]$$
$$T_{\text{split}} = T_i[s_1, w_1] + T_i[s_2, w_2] + \cdots + T_i[s_k, w_k]$$

$$\text{Execution Strategy} = \begin{cases} \text{pad} & \text{if } T_{\text{pad}} < T_{\text{split}} \\ \text{split} & \text{if } T_{\text{pad}} > T_{\text{split}}. \end{cases} \quad (2)$$

TABLE II
CONDITIONAL NN BENCHMARKS

Model	DNN Type	Dataset	Conditional NN Axes	#Decision Points
ResNet-34 [46]	Early Exit CNN	CIFAR-10	Depth	16
MobileNet-V1 [22]	Dynamic Slimmable CNN	ILSVRC 2012	Width	1
BERT-base [6]	Encoder-only Transformer	MNLI	Width	1
BERT-base [21]	Early Exit Encoder-only Transformer	MNLI	Depth+ Width	12 (Depth) + 1 (Width)
Transformer [38]	Seq2Seq Transformer	WMT'19 En-De	Depth+ Width	103 (Depth) + 1 (Width)

IV. Experimental Methodology

Performance Evaluation: We implement BatchCond using PyTorch [45] and evaluate its performance on three different hardware platforms: 1) NVIDIA Jetson AGX Xavier; 2) NVIDIA GeForce RTX 2080Ti; and 3) NVIDIA A40. Jetson AGX Xavier is an edge platform that features an edge GPU with 32 GB of unified memory. RTX 2080Ti is a desktop GPU with 11 GB of memory. A40 is a data center GPU with 48 GB of memory. Due to limited space, we present overall improvements on all platforms, while supplementary results are reported only on the RTX 2080 Ti GPU. We use the largest batch size that fits on the GPU for all experiments, unless specified otherwise.

Application Benchmarks: We benchmark BatchCond on five diverse Conditional NNs (Table II) with different axes of conditionality. Early-exit networks represent Conditional-Depth NNs, while dynamic slimmable networks and encoder-only transformers are Conditional-Width NNs. Transformers with early exits and Seq2Seq transformers are conditional in both width and depth.

Hardware Precharacterization: We precharacterize our hardware platform to obtain the following numbers for a given Conditional NN: 1) *Conditional-Depth NNs:* $T_i[B]$ and $T_{Gather}[B]$ for each layer (i) using different batch sizes (B) for finding whether to pad or perform reorganization and 2) *Conditional-Width NNs:* $ideal\_batch\_size[W]$ for different widths (W) for finding the best arrangement of samples into sub-batches, and $T_i[B, W]$ and $T_{Gather}[B]$ for each layer (i) using different batch sizes (B) and widths (W) for finding whether to pad or perform sub-batch splitting.

V. Results

We first present the overall inference throughput improvements achieved by BatchCond after incorporating all runtime overheads. Subsequently, we present an ablation study to evaluate the contribution of SimBatch and ABR to the overall improvement. We also analyze the efficacy of SimBatch in reducing computational irregularity and evaluate how BatchCond performs in a deadline-aware inference setting.

TABLE III
THROUGHPUT GAINS ACHIEVED BY THE BATCHCOND FRAMEWORK

DNN Type	Jetson AGX Xavier		RTX 2080Ti		A40	
	Throughput gain over batch size 1	**Throughput gain over random batching with padding**	**Throughput gain over batch size 1**	**Throughput gain over random batching with padding**	**Throughput gain over batch size 1**	**Throughput gain over random batching with padding**
Early Exit CNN	6.7×	1.4×	14.5×	1.5×	20.0×	1.4×
Dynamic Slimmable CNN	7.4×	1.2×	19.9×	1.2×	34.5×	1.2×
Encoder-only Transformer	4.9×	2.1×	26.2×	3.6×	61.3×	4.1×
Early Exit Transformer	7.6×	3.4×	44.2×	6.6×	75.5×	5.3×
Seq2Seq Transformer	9.7 ×	1.8×	20.1 ×	1.9×	39.9 ×	2.4×
Geometric Mean	**7.1**×	**1.8**×	**23.2**×	**2.4**×	**41.8**×	**2.5**×

Additionally, we examine the impact of batch size on throughput gains. Finally, we analyze the preinference (one-time) and inference-time overheads of the BatchCond framework.

A. Overall Throughput

Table III presents the throughput gains resulting from using BatchCond for batched inference on diverse Conditional NN benchmarks using all three hardware platforms. We compare BatchCond with the two baseline techniques currently used for Conditional NNs—inference with a batch size of 1 and random batching with padding. BatchCond improves throughput by up to 6.6× (geometric mean of 2.5×) compared to inference with random batching with padding. BatchCond also improves throughput by up to 75.5× (geometric mean of 41.8×) compared to inference with a batch size of 1.

For transformers that are conditional in both depth and width, we empirically compare the two possible predictive batching strategies—batching samples that are likely to require the same network depth, and batching samples that have similar widths (in the case of text inputs, sequences that have similar numbers of words). We find that batching samples based on similarity in width leads to 1.6× higher average throughput compared to batching based on depth. This is because different samples exhibit substantially higher variance in width values compared to depth values (for instance, with BERT-base on the MNLI dataset, variance in width values is 75× higher than depth values). Hence, batching based on width leads to greater reduction in computational irregularity, and thereby higher throughput gains.

In addition, we evaluate the impact of BatchCond on per-batch latency, using the example of Seq2Seq transformers (Conditional-Depth+Width) in Fig. 6. We find that BatchCond reduces the average latency by 1.9× compared to random batching with padding. The reduced latency is a direct consequence of the reduction in ineffectual computations performed. In particular, batches of sequences with shorter inputs and outputs are executed with substantially lower latency using BatchCond compared to random batching with padding. The maximum latency seen for a single batch with BatchCond is also lower because ABR drops terminated samples (once all output tokens have been generated), thereby speeding up subsequent decoding iterations.

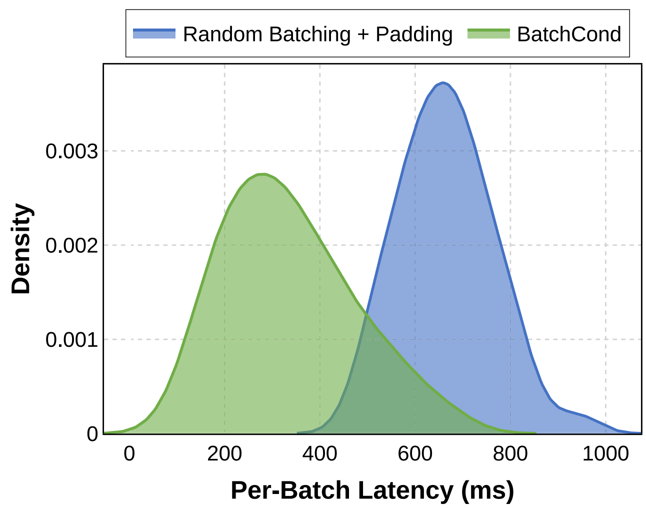

Fig. 6. Improvement in per-batch latency distribution from BatchCond in Seq2Seq transformers.

B. Ablation: Breakdown of Benefits From Each Technique in the BatchCond Framework

We analyze the impact of each BatchCond optimization on end-to-end performance in Fig. 7. We observe that SimBatch reduces intrabatch computational irregularity, resulting in 1.8× higher average throughput. We note that the throughput gain from using SimBatch takes the runtime overheads of the DPP-NN into account. We also find that ABR optimizes execution in the presence of residual computational irregularity, resulting in an additional 1.4× average increase in throughput. In summary, SimBatch and ABR are synergistic optimizations that can be combined to increase throughput during batched inference of Conditional NNs.

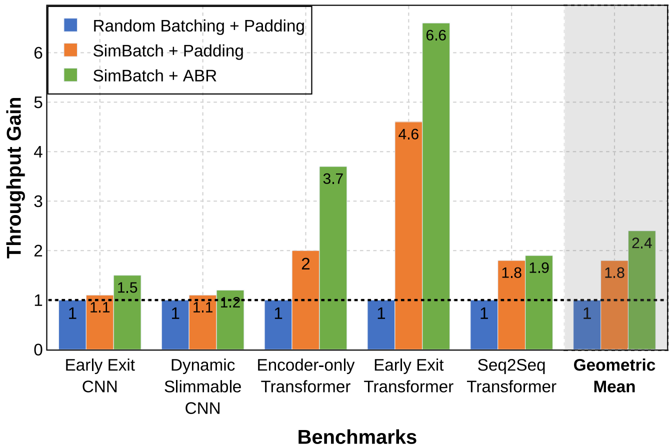

Fig. 7. Ablation Study: Breakdown of benefits from each technique in the BatchCond framework.

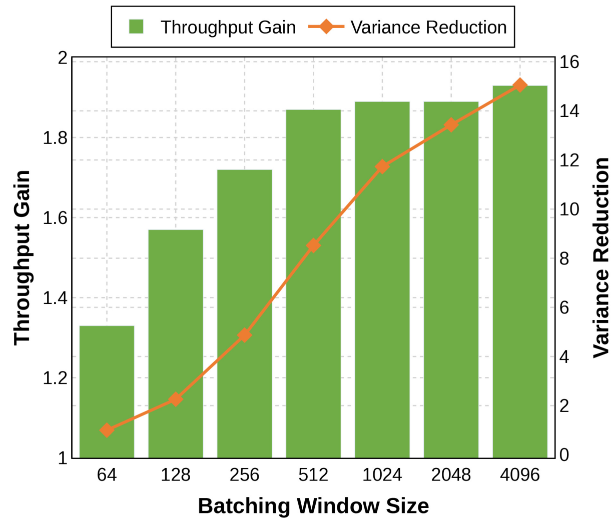

Fig. 8. Impact of input window size on throughput gains and intrabatch variance reduction for Seq2Seq transformer. For a window of size k, we assume that only k inputs are available in the inference queue at any time, and all k inputs in one window must be processed before moving on to samples in the next window, thereby simulating bursty input rates and deadline constraints that are likely to arise in practical scenarios.

TABLE IV
REDUCTION IN AVERAGE INTRABATCH VARIANCE FROM USING THE BATCHCOND FRAMEWORK

DNN Type	Reduction in variance w.r.t. random batching
Early Exit CNN	2×
Dynamic Slimmable CNN	1.2×
Encoder-only Transformer	37.5×
Early Exit Transformer	37.5×
Seq2Seq Transformer	15.9 ×
Geometric Mean	**8.8×**

C. Impact of SimBatch on Computational Irregularity

In order to quantify the effectiveness of SimBatch in reducing computational irregularity, we measure the reduction in intrabatch variance of decision point outcomes (effective depth/width of the network) when SimBatch is used. The results are reported in Table IV. We find that SimBatch reduces variance by up to 37.5× (geometric mean of 8.8×). This results in the device utilization increasing by an average of 23.6% over random batching. The utilization improvement is a direct consequence of two factors: 1) control flow divergence is reduced, thereby reducing the amount of time for which some PEs are idle while waiting for others to finish and 2) the amount of ineffectual computations arising from the use of padding is reduced, thereby freeing up more PEs to perform useful work.

D. Deadline-Aware Batched Inference With BatchCond

The results presented in earlier sections are obtained under the assumptions that 1) all the samples in the test dataset are available at the start of inference and 2) none of the samples have any deadlines (latency constraints) that require them to processed before others. However, in practical deployment scenarios, not all samples may be available at the same time, and samples are likely to have deadlines. To demonstrate the effectiveness of BatchCond in this scenario, we consider a scenario where inputs arrive in windows, and all samples in one window must be processed before processing samples in the next window. We present the results of using BatchCond with different window sizes in Fig. 8. When small window sizes are used, i.e., when only few inputs are available for batching, it is impossible to create batches composed entirely of computationally similar samples. In other words, the flexibility available to BatchCond is reduced. As a result, we find that throughput improvements from BatchCond are smaller for smaller window sizes such as 64. However, we note that even with small window sizes, the use of ABR leads to substantial throughput improvement over both inference with a batch size of 1 and batched inference with padding, indicating that ABR is highly impactful even under strict latency constraints (where SimBatch is not as effective due to reduced options for batching). The throughput improvements increase with window size, but largely saturate at a window size of 512.

E. Impact of Batch Size on Throughput Improvements

We evaluate the effectiveness of BatchCond when different batch sizes are used (Fig. 9). We observe that throughput gains are typically higher at larger batch sizes. When very small batch sizes are used, the hardware is often underutilized, and hence, the ineffectual computations introduced by data padding do not have a significant impact on throughput in Conditional-Width NNs. For instance, in encoder-only transformers, padding all sequences to the maximum length in the batch does not impact throughput, since all sequences must be processed by all Transformer layers irrespective of length, and padding tokens are processed by PEs that would otherwise have been idle. Consequently, BatchCond does not provide significant improvements over random batching with padding (Fig. 9). However, in Conditional-Depth NNs, batches of samples that terminate early can be processed at lower latency

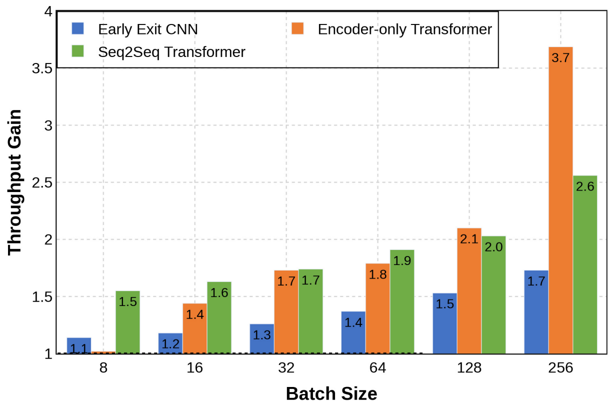

Fig. 9. Impact of batch size on throughput gain.

compared to batches of late-terminating samples. As a result, even though ABR always chooses padding over splitting (and hence, ABR does not directly improve throughput), the use of SimBatch leads to substantial throughput gains over prior methods even when very small batch sizes are used (Fig. 9). We also note that at very small batch sizes, throughput gains arise solely from SimBatch, since ABR always chooses to pad compute and/or data due to hardware underutilization. When batch sizes are large enough to fully utilize the hardware, both ABR and SimBatch contribute toward throughput gains by reducing ineffectual computations and control flow divergence, leading to substantial throughput gains over prior methods in all types of Conditional NNs.

F. Discussion of Overheads

We discuss and quantify the overheads associated with each technique in the BatchCond framework. We reiterate that the results presented in prior sections are inclusive of all overheads.

1) SimBatch: The use of DPP-NN to estimate outcomes at all decision points introduces two types of overheads.

1) The training of this predictor incurs a one-time cost and is completed offline prior to deployment for inference. In our experiments, the training duration was less than 1 h on a single Nvidia GeForce RTX 2080 Ti GPU for all our studied tasks.
2) During inference, the DPP-NN processes each input to determine decision outcomes, adding runtime overhead. However, since the DPP-NN is composed of only the first layer of the Conditional NN, we find that the latency increase due to the DPP-NN is very small. In fact, the DPP-NN leads to $<4\%$ increase in latency in all our studied tasks. As reported earlier, the net improvement in throughput from SimBatch alone is $1.8\times$ after considering this overhead.

2) ABR: While ABR does not introduce any additional inference-time overheads, it incurs a one-time cost to precharacterize the Conditional NN of interest on a given hardware platform (performed offline prior to deployment for inference). In particular, the precharacterization involves executing representative inputs to measure all quantities mentioned in Section IV under *Hardware precharacterization*. We repeat all experiments 30 times, and average the measured times in order to eliminate potential sources of noise and obtain stable results. We found that precharacterizing a RTX 2080 Ti GPU for executing all our studied benchmarks takes approximately 20 min.

TABLE V
COMPARISON WITH OTHER BATCHING FRAMEWORKS. NOTES: 1) AN ENTRY OF "N/A" DENOTES THAT THE TECHNIQUE IS NOT APPLICABLE TO THAT BENCHMARK. 2) NUMBERS FOR RELATED WORKS WERE OBTAINED THROUGH OUR BEST-EFFORT REPRODUCTION OF THE PROPOSED METHODS, AS NO OPEN-SOURCE CODE WAS AVAILABLE

DNN Type	**Throughput gain over LazyBatching [47]**	**Throughput gain over [48]**
Early Exit CNN	1.3×	N/A
Dynamic Slimmable CNN	N/A	N/A
Encoder-only Transformer	N/A	1.8×
Early Exit Transformer	N/A	1.4×
Seq2Seq Transformer	1.5×	1.2×

VI. RELATED WORK

The vast majority of prior works on Conditional NNs either perform inference with a batch size of one [20], [21], [22], or use padding to ensure computational regularity [5], [23], thereby adversely affecting throughput. LazyBatching [47] and FluidBatching [49] are the only notable exceptions for Conditional-Depth NNs, wherein samples are stalled at decision points by caching intermediate activations. Execution of a stalled sample is continued only when sufficient numbers of other samples with the same outcome arrive at the decision point, or if samples are close to their deadlines. However, these methods incur substantial storage overheads for storing the large intermediate activations of stalled samples (thereby limiting batch sizes that can be used), as well as high data movement costs, both of which increase with number of decision points in the network. We quantitatively compare BatchCond with LazyBatching on an RTX 2080 Ti GPU (Table V) and find that BatchCond achieves $1.3\times$ and $1.5\times$ higher throughput on the early-exit CNN and the Seq2Seq transformer, respectively. Gonzalez et al. [48] proposed sorting and bucketing variable-length inputs based on their lengths to reduce the amount of padding tokens. However, this method is not applicable to Conditional NNs where input sizes are fixed (e.g., early-exit CNNs, where easy inputs terminate early). In addition, bucketing is challenging during inference, since inputs arrive in windows. As a result, [48] is not guaranteed to produce computationally similar batches, and [48] does not provide any mechanism to accelerate batches where padding becomes necessary. On the other hand, the ABR component of BatchCond also accelerates the processing of computationally irregular batches, leading to an average throughput gain of

1.4× over [48] on benchmarks with variable size inputs (Table V).

Prior works have also attempted to predict the outcomes of decision points in Conditional NNs. For instance, EdgeBERT [50] and Predictive Exit [51] design exit point predictors for early-exit networks to dynamically scale the voltage and frequency of the underlying hardware based on the exit point, enabling energy-efficient inference. However, these works do not focus on improving throughput, and in fact, evaluate only at batch sizes of 1. There have also been recent efforts in compiler research [52] to optimize program execution in the presence of control flow divergence through compiler optimizations, such as fusing memory gather operations and end-to-end kernel generation. These techniques are complementary to our optimizations.

VII. Conclusion

Batched inference is challenging in Conditional NNs due to irregularity in computational patterns across inputs. We address this problem by proposing BatchCond, an optimized batching framework for Conditional NNs. BatchCond is composed of two complementary techniques. Computational similarity-driven batching (SimBatch) batches samples that are likely to share similar computational patterns, thus reducing intrabatch divergence. ABR addresses the residual computational irregularity by dynamically reorganizing batches into computationally similar sub-batches in a hardware-aware manner. Our evaluations on diverse hardware platforms reveal that BatchCond improves throughput of batched inference by up to 6.6× across diverse Conditional NN benchmarks.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1–9.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[3] A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," 2020, *arXiv:2010.11929*.

[4] M. Johnson et al., "Google's multilingual neural machine translation system: Enabling zero-shot translation," *Trans. Assoc. Comput. Linguist.*, vol. 5, pp. 339–351, Oct. 2017.

[5] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1–11.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[7] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 1877–1901.

[8] H. Touvron et al., "LLaMA: Open and efficient foundation language models," 2023, *arXiv:2302.13971*.

[9] A. v. d. Oord et al., "WaveNet: A generative model for raw audio," 2016, *arXiv:1609.03499*.

[10] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," in *Proc. Int. Conf. Mach. Learn.*, 2023, pp. 28492–28518.

[11] "Siri." Apple. Accessed: Aug. 24, 2024. [Online]. Available: https://www.apple.com/siri/

[12] "ChatGPT." OpenAI. 2022. Accessed: Aug. 24, 2024. [Online]. Available: https://openai.com/chatgpt/

[13] "Google translate." Google. Accessed: Aug. 24, 2024. [Online]. Available: https://translate.google.com/

[14] "DeepL translator." DeepL. Accessed: Aug. 24, 2024. [Online]. Available: https://www.deepl.com/en/translator

[15] "Google photos." Google. Accessed: Aug. 24, 2024. [Online]. Available: https://www.google.com/photos/about/

[16] "Recognizing text in images." Apple. Accessed: Aug. 24, 2024. [Online]. Available: https://developer.apple.com/documentation/vision/recognizing-text-in-images

[17] (Vixen Labs, London, U.K.). *Voice Consumer Index*. 2022. Accessed: Aug. 24, 2024. [Online]. Available: https://vixenlabs.co/wp-content/uploads/2022/06/VixenLabs_VoiceConsumerIndex2022.pdf

[18] S. Ganapathy, S. Venkataramani, G. Sriraman, B. Ravindran, and A. Raghunathan, "DyVEDeep: Dynamic variable effort deep neural networks," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 3, pp. 1–24, 2020.

[19] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 11, pp. 7436–7456, Nov. 2022.

[20] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *Proc. 23rd Int. Conf. Pattern Recognit. (ICPR)*, 2016, pp. 2464–2469.

[21] J. Xin, R. Tang, J. Lee, Y. Yu, and J. Lin, "DeeBERT: Dynamic early exiting for accelerating BERT inference," 2020, *arXiv:2004.12993*.

[22] C. Li, G. Wang, B. Wang, X. Liang, Z. Li, and X. Chang, "Dynamic Slimmable network," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2021, pp. 8607–8617.

[23] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empir. Methods Natural Lang. Process., System Demonstrat.*, 2020, pp. 38–45.

[24] E. Park et al., "Big/little deep neural network for ultra low power inference," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ ISSS)*, 2015, pp. 124–132.

[25] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez, "SkipNet: Learning dynamic routing in convolutional networks," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 409–424.

[26] A. Veit and S. Belongie, "Convolutional networks with adaptive inference graphs," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 3–18.

[27] Z. Wu et al., "BlockDrop: Dynamic inference paths in residual networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 8817–8826.

[28] Z. Jiang, C. Li, X. Chang, L. Chen, J. Zhu, and Y. Yang, "Dynamic slimmable denoising network," *IEEE Trans. Image Process.*, vol. 32, pp. 1583–1598, 2023.

[29] W. Hua, Y. Zhou, C. M. De Sa, Z. Zhang, and G. E. Suh, "Channel gating neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 1886–1896.

[30] Z. Chen, Y. Li, S. Bengio, and S. Si, "You look twice: GaterNet for dynamic filter selection in CNNs," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 9172–9180.

[31] P. He, X. Liu, J. Gao, and W. Chen, "DeBERTa: Decoding-enhanced BERT with disentangled attention," 2020, *arXiv:2006.03654*.

[32] N. Shazeer et al., "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," 2017, *arXiv:1701.06538*.

[33] R. T. Mullapudi, W. R. Mark, N. Shazeer, and K. Fatahalian, "Hydranets: Specialized dynamic architectures for efficient inference," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 8080–8089.

[34] Y. Wang et al., "Dual dynamic inference: Enabling more efficient, adaptive, and controllable deep inference," *IEEE J. Sel. Topics Signal Process.*, vol. 14, no. 4, pp. 623–633, May 2020.

[35] A. Ehteshami Bejnordi and R. Krestel, "Dynamic channel and layer gating in convolutional neural networks," in *Proc. 43rd German Conf. Artif. Intell. AI*, Bamberg, Germany, 2020, pp. 33–45.

[36] W. Xia, H. Yin, X. Dai, and N. K. Jha, "Fully dynamic inference with deep neural networks," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 2, pp. 962–972, Jun. 2022.

[37] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 1–9.

[38] H. Wang et al., "HAT: Hardware-aware transformers for efficient natural language processing," 2020, *arXiv:2005.14187*.

[39] T. Schuster et al., "Confident adaptive language modeling," in *Proc. Adv. Neural Inf. Process. Syst.*, 2022, pp. 17456–17472.

[40] W. Zhou, C. Xu, T. Ge, J. McAuley, K. Xu, and F. Wei, "BERT loses patience: Fast and robust inference with early exit," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 18330–18341.

[41] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Apr. 2021.

[42] J. Choquette, "NVIDIA hopper H100 GPU: Scaling performance," *IEEE Micro*, vol. 43, no. 3, pp. 9–17, Jun. 2023.

[43] N. P. Jouppi et al., “In-datacenter performance analysis of a tensor processing unit,” in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.

[44] N. Jouppi et al., “TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, 2023, pp. 1–14.

[45] A. Paszke et al., “PyTorch: An imperative style, high-performance deep learning library,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.

[46] E. Demir, “Early-exit convolutional neural networks,” M.S. thesis, Dept. Natural Appl. Sci., Middle East Tech. Univ., Ankara, Türkiye, 2019.

[47] Y. Choi, Y. Kim, and M. Rhu, “LazyBatching: An SLA-aware Batching system for cloud machine learning inference,” in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2021, pp. 493–506.

[48] P. Gonzalez, T. S. Alstrøm, and T. May, “On Batching variable size inputs for training end-to-end speech enhancement systems,” in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, 2023, pp. 1–5.

[49] A. Kouris, S. I. Venieris, S. Laskaridis, and N. D. Lane, “Fluid Batching: Exit-aware preemptive serving of early-exit neural networks on edge NPUs,” 2022, *arXiv:2209.13443*.

[50] T. Tambe et al., “EdgeBERT: Sentence-level energy Optimizations for latency-aware multi-task NLP inference,” in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2021, pp. 830–844.

[51] X. Li et al., “Predictive exit: Prediction of fine-grained early exits for computation-and energy-efficient inference,” in *Proc. AAAI Conf. Artif. Intell.*, 2023, pp. 8657–8665.

[52] P. Fegade, T. Chen, P. Gibbons, and T. Mowry, “ACRoBat: Optimizing auto-batching of dynamic deep learning at compile time,” in *Proc. Mach. Learn. Syst.*, 2024, pp. 14–30.