Fluid Batching: Exit-Aware Preemptive Serving of Early-Exit Neural Networks on Edge NPUs

Alexandros Kouris*, Stylianos I. Venieris*
Samsung AI Center, Cambridge, UK
{a.kouris, s.venieris}@samsung.com

Stefanos Laskaridis†
Brave Software
mail@stefanos.cc

Nicholas D. Lane†
University of Cambridge and *Flower Labs*
ndl32@cam.ac.uk

***Abstract*—With deep neural networks (DNNs) emerging as the backbone in a multitude of computer vision tasks, their adoption in real-world applications broadens continuously. Given the abundance and omnipresence of smart devices in the consumer landscape, "smart ecosystems" are being formed where sensing happens concurrently rather than standalone. This is shifting the on-device inference paradigm towards deploying centralised neural processing units (NPUs) at the edge, where multiple devices (*e.g.* in smart homes or autonomous vehicles) can stream their data for processing with dynamic rates. While this provides enhanced potential for input batching, naive solutions can lead to subpar performance and quality of experience, especially under spiking loads. At the same time, the deployment of dynamic DNNs, comprising stochastic computation graphs (*e.g.* early-exit (EE) models), introduces a new dimension of dynamic behaviour in such systems. In this work, we propose a novel early-exit-aware scheduling algorithm that allows sample preemption at run time, to account for the dynamicity introduced both by the arrival and early-exiting processes. At the same time, we introduce two novel dimensions to the design space of the NPU hardware architecture, namely Fluid Batching and Stackable Processing Elements, that enable run-time adaptability to different batch sizes and significantly improve the NPU utilisation even at small batches. Our evaluation shows that the proposed system achieves an average 1.97× and 6.7× improvement over state-of-the-art DNN streaming systems in terms of average latency and tail latency service-level objective (SLO) satisfaction, respectively.**

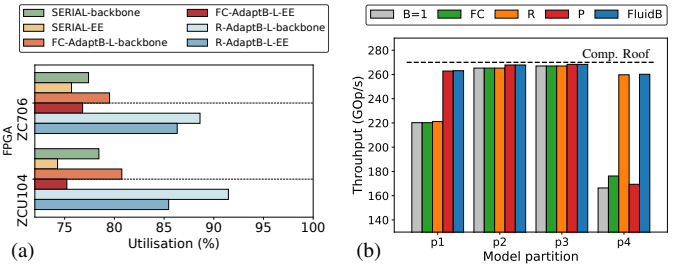


Fig. 1: Different batching strategies applied on a mainstream NPU design [2] that maps each convolution to a General Matrix Multiply operation, where inputs, weights and output activations are represented by $R{\times}P$, $P{\times}C$ and $R{\times}C$ Toeplitz matrices (see Fig. 2), respectively, assuming a batch of B samples: **SERIAL**: $B{=}1$ | **FC**: Batching only on FC layers | $\mathbf{R}/\mathbf{P}$: Appending samples across the row/col dimension of the activation matrices. **(a)** Impact of early exits (EE) on hardware utilisation with batch processing across two FPGA platforms, targeting a 4-exit ResNet-50 at an arrival rate of 25 samples/s. Although batching significantly improves the device utilisation for the original network, the stochasticity introduced by early exits radically attenuates this gain. Batches are formed using adaptive batching (AdaptB) (detailed in §II-A). **(b)** Achieved throughput on ZC706 for ResNet-50 (partitioned in 4 subnets) with different batching strategies adopted by the NPU (B=8). Dashed line denotes peak platform performance.

I. Introduction

The continued advancement of deep neural networks (DNNs) has led to their mainstream adoption in consumer applications, as a backbone for several computer vision tasks. This has led to the formation of smart ecosystems within the consumer environment, where numerous neighbouring devices are simultaneously collecting an abundance of data. The high concentration of sensing platforms in such local ecosystems has shifted the inference paradigm one step away from the device, at the edge, where data from several sources can be streamed to a more powerful shared compute platform for inference [1]. Smart homes and mobile robots constitute prime examples of this scenario, as in both cases several visual sensors are continuously capturing images that often need to be processed by the same model (*e.g.* person identification) under strict latency constraints.

This increased rate of inference requests on the same model from multiple sources, unlocks the potential of batch processing, which constitutes a promising approach for meeting the throughput demand with a given neural processing unit (NPU) at the edge. By grouping samples and dispatching them together to the NPU, parallelism and data-reuse are increased, leading to improved hardware utilisation, higher processing rate and potentially shorter waiting time for incoming samples. However, batching comes at a cost; the time overhead of assembling samples and the longer computation time induces increased latency which can impact the quality of experience (QoE). As such, the status-quo wisdom dictates that batch processing should be avoided in latency-critical applications. In this context, there is an emerging need for novel methods that leverage the throughput benefits of batching to serve a multitude of inference requests, while also meeting tight latency constraints imposed by the target application so that the QoE is not penalised.

However, the inherent dynamicity of smart ecosystems and user behaviour leads to varying inference request rates. For example, the dynamic adjustment of sampling rates –commonly used to reduce computational load and energy consumption– affects batch formation at the edge NPU where all samples are streamed. This calls for new hardware solutions that enable the NPU to sustain high performance across variable batch sizes.

At the same time, a growing body of work is developing multi-exit DNNs [3]. This family of dynamic models attaches intermediate exits throughout the architecture and saves computation by allowing easier samples to exit early. Despite the latency benefits, this mechanism introduces further dynamicity to the batch size, at a subnet level; as samples may exit early at intermediate exits, the batch size shrinks dynamically during inference. As shown in Fig. 1a, this leads to hardware underutilisation for deeper parts of the model and an inefficient use of the NPU, requiring a novel system design approach.

In order to deal with the enhanced dynamicity evident in the deployment of early-exit DNNs in streaming scenarios, based on the above observations we propose a novel SW/HW co-design framework for edge NPU-based serving that employs three core techniques: *i) Fluid Batching*, a hardware-based mechanism that finely adapts the batching strategy based on both the instantaneous batch size

*Equal Contribution.
†Work done while with Samsung AI.

and the characteristics of each layer, attaining high performance on the NPU across batch sizes (§III-A); *ii)* an exit- and deadline-aware *preemptive scheduler* that allows the preemption of processing at intermediate exit points and the subsequent merging into larger batches, alleviating the NPU underutilisation due to the reduced batch size in deeper parts of early-exit models (§III-B); *iii) Stackable PEs*, a type of run-time reconfigurable Processing Elements (PEs) that can be dynamically adapted to the dimensions of each layer, counteracting the hardware underutilisation caused by suboptimally mapped operations with minimal overhead (§III-A).

II. Background & Related Work

A. Batch Processing in Inference Systems

Batch processing has been adopted as a key technique towards increasing the inference throughput. Nonetheless, contrary to the training stage, forming batched inputs during inference is challenging as the processing platform receives DNN inference requests at rates that vary significantly based on the time of day and the number of applications or users that share the same model. Importantly, waiting to form a large-enough batch often has a prohibitive impact on latency.

Existing batching approaches commonly integrate two techniques, *i) model-level* and *ii) adaptive batching*. *Model-level batching* dictates that batching is performed at the entire model granularity, *i.e.* the same batch size B is used for all layers of the DNN. *Adaptive batching* (AdaptB), which is employed by modern inference systems [4]–[8] to adapt the batch size to the changing arrival rate, introduces another parameter, the batch-forming timeout window T_{timeout}, resulting in the batching strategy $\langle B_{\max}, T_{\text{timeout}} \rangle$. Concretely, AdaptB dispatches incomplete batches when T_{timeout} is exceeded, otherwise it issues the platform- or model-dependent maximum batch size $B_{\max}$. Closer to our approach, LazyBatching [9] allows the selective preemption of inference at the layer level. Nonetheless, the use of a coarse latency estimator, together with its exit-unaware design, lead to conservative batching decisions, leaving untapped optimisation opportunities.

Edge- vs. Cloud-based Inference Systems. Contrary to their cloud-residing counterparts, edge-based inference systems are constrained in three ways [1]: *1)* edge NPUs have lower resource and energy budget and hence provide lower processing throughput. Thus, in spite of any throughput benefits, the large batch sizes that are common on the cloud (*e.g.* typically >8 and up to 64 [8]–[10]) directly violate the latency service-level objective (SLO) of interactive applications when executing high-accuracy -but costly- models; *2)* the number of served devices -and in turn the queries per second- is one to two orders of magnitude lower (*e.g.* 10^2-10^3 queries/s on the cloud vs. tens at the edge [11]). Forming large batch sizes would require prohibitively long waiting times to assemble enough samples; *3)* although lightweight models could be employed to increase throughput, this would degrade the accuracy of the target task and would defeat the purpose of using a server. Fluid Batching alleviates these constraints by maximising the performance even at smaller batch sizes (see *NPU Ablation* in §IV) and by dynamically filling gaps in the active batch as old samples early-exit and new ones arrive.

Batching in Hardware. From a hardware perspective, existing NPU designs [8] adopt a *uniform* solution for implementing batching across all layers, which can lead to suboptimal mapping and underutilisation in certain layers. Instead, Fluid Batching (§III-A) introduces a flexible mechanism that adapts the batching strategy on-the-fly at a per-layer basis to maximise resource utilisation and, thus, inference efficiency.

B. Early-Exit Neural Networks

Dynamic DNNs come in many shapes and forms [12]–[15]. A successful variant of such networks comes in the form of early-exit networks [3], DNNs with intermediate heads where samples can "exit early" based on shallow (and more coarse) features of the original network – the backbone. This mechanism yields computation savings due to the early termination of inference, while also providing early actionable results during inference and allowing for the progressive refinement of the output prediction. Early-exit networks have successfully been deployed in a multitude of tasks, spanning from image classification [16]–[18] to semantic segmentation [19] or even various NLP tasks [20]–[22]. On the system side, existing work has so far focused on the hardware-aware design of the exit policy, and the placement and architecture of the exits along the depth of the backbone network [16], [17], [19], the distributed execution of such models across device and server [23]–[25] with [26] also studying the implications of their deployment in streaming scenarios, and the co-design of an early-exit model and its hardware accelerator [27].

All of the above works, however, have focused on single-sample execution and do not consider the hardware utilisation impact of such a decision at inference time. In contrast, we embrace larger batch sizes in the streaming setting and investigate an efficient way of scheduling such stochastic workloads on NPUs under tight latency budgets.

C. NPU Architecture Design

General matrix multiply (GEMM) comprises the most widely optimised computational kernel that lies at the heart of most NPU accelerators due to its ability to support both the convolutional (CONV) and fully-connected (FC) layers [2] of CNNs, as well as the various matrix multiplications of Transformer models [28]. This paper considers a generic NPU design [29]–[31] that represents a large potion of actual deployed NPUs, where each layer's input activation maps, weights and output activation maps are formed into $R \times P$, $P \times C$ and $R \times C$ Toeplitz matrices, respectively, which are stored in the off-chip memory. The execution of each layer is then reduced to a tiled GEMM operation, as depicted in Fig. 2. Typically, tile sizes across each dimension (T_R, T_P, T_C) are tightly coupled with architectural characteristics of the accelerator (concerning both the compute engine and the on-chip memory structure) and are tuned based on the target workload through design space exploration (DSE). In this work, the adopted NPU consists of T_C processing elements (PEs), each comprising a Multiply-Accumulate (MAC) tree of width T_P (Fig. 4b). Parameter T_R controls the pipeline depth, while three on-chip memory buffers that match the input, weight and output tile dimensionalities are instantiated, each allowing for double-buffering.

Under this accelerator design, it is common that not all layers of the target model can be efficiently mapped, due to a mismatch between layer dimensions and the accelerator's configuration. As a result, the resources of the NPU are not fully utilised throughout the inference process, leading to suboptimal performance. To remedy this inefficiency, in this paper we propose Stackable PEs (§III-A), a dynamically adjustable PE structure that allows the on-the-fly re-purposing of the underutilised resources in order to exploit different parallelism dimensions with minimal hardware overhead.

III. Methodology

Fig. 3 depicts an overview of the proposed system. Inference requests associated with a common model are coming into a *queue* (❶) to be processed by a multi-exit model resident in the *NPU* (§III-A). The *scheduler* (§III-B), running on CPU, reads from the queue, forms a batch into a *buffer* and submits inference jobs to the *NPU* (❷). Upon meeting a preemption point, *i.e.* an intermediate classifier where some of the samples are expected to exit early, an early-exit event is triggered (❸) and the number of exiting samples, B_{exit}, is communicated to the scheduler (❹). At this point, the scheduler

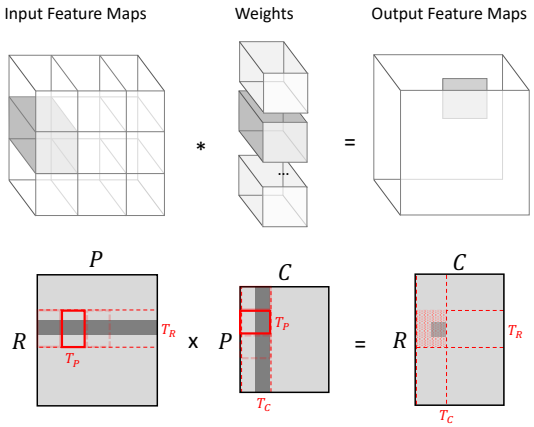


Fig. 2: The computation of a convolutional layer, mapped as GEMM.

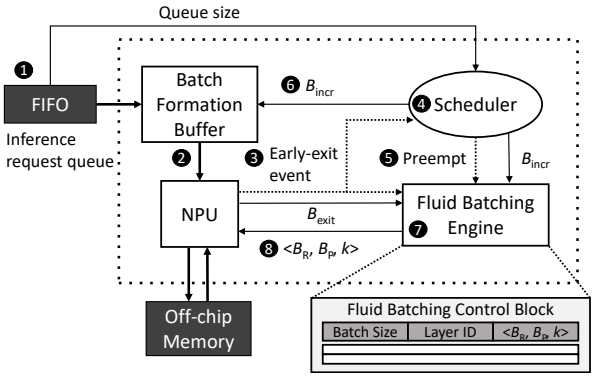


Fig. 3: Overview of the proposed system.

selectively preempts execution (❺) and a new batch of size B_{incr} is formed and dispatched to run until the previous preemption point (❻). Subsequently, the halted and the new batch are merged and execution is resumed, benefiting from the higher efficiency of the increased batch size, until the next exit is met. The whole process is orchestrated by the scheduler and the *Fluid Batching Engine* (❼) which tailor the adaptable batching and processing components of the NPU (❽) based on the current batch size, the model at hand, the latency SLO and the arrival rate of the incoming samples, aiming to maximise efficiency. Next, we delve into the details of each component.

A. Edge NPU Design with Fluid Batching

Under the GEMM formulation (§II-C), existing systems typically realise batching of B samples by appending activation matrices uniformly along the *row* dimension for all layers (see Fig. 4a (①)), so that $\hat{R}{=}R{\cdot}B$, allowing for better reuse of the weight matrix. This has been proven particularly effective in counteracting the low computation-to-communication ratio of the memory-bound fully-connected layers, where $R{=}1$ when batching is not used [30]. Adopting the same approach in convolutional layers, across either the R (①) or P (②) dimension to facilitate further parallelism between the samples of a batch, has also shown considerable performance gains [8], [32].

Nonetheless, in all cases, batching and parallelism are *statically* applied across all layers, whereas NPUs are typically optimised for a *fixed* batch size. Fig. 1b reports the achieved throughput of different static batching strategies for a ResNet-50 backbone on the examined NPU, with its layers partitioned into groups of equivalent workload. Evidently, different parts of the model benefit the most from different batching implementations, as shallow and deep layers demonstrate fundamental variability in matrix dimensionalities, leading to distinct mapping inefficiencies.

Flexible Batch Processing Mechanism. In order to remedy the inefficient mapping of static batching approaches, Fluid Batching generalises existing strategies by dynamically selecting the breakdown of samples that are appended in each matrix dimension (R, P), based on the running layer l and the active batch size B_{act}. As such the NPU is able to exploit parallelism across different samples of a batch (previously being pipelined), in cases where other parallelism dimensions lead to an inefficient mapping on the available resources. In practice, Fluid Batching is realised by modifying the Direct Memory Access (DMA) to the off-chip memory, to affect the Toeplitz matrix formation process for each batch. Formally:

$$\begin{aligned}\hat{R}^{(l)} &= B_R^{(l,B_{\text{act}})} \cdot R^{(l)} \quad \text{and} \\ \hat{P}^{(l)} &= (B_{\text{act}} - B_R^{(l,B_{\text{act}})} + 1) \cdot (P^{(l)} + P^{(l)} \% T_P)\end{aligned} \tag{1}$$

where $B_R{=}f(l, B_{\text{act}}) \in [1, B_{\max}]$ is provided by the Fluid Batching engine (described below) that controls the batching strategy for each layer at run time through a fine-grained mechanism, aiming to improve efficiency by eliminating resource underutilisation; and $\%$ denotes the modulo operation, used to add zero "guard" elements across the P dimension (Fig. 4a) to prevent interference between different samples of the batch. As also illustrated in Fig. 4a, R- and P-batching form special cases of Fluid Batching (③) for $B_R{=}B_{\text{act}}$ and 1, respectively, along with other hybrid schemes. Notably, through DMA handling, the result of Fluid Batching's computation remains a $B_{\text{act}} \cdot R \times C$ matrix, as in the case of R-batching, and facilitates the adoption of a different batching scheme at the subsequent layers.

Stackable Processing Elements. Since all L layers $\langle R^{(l)}, P^{(l)}, C^{(l)} \rangle$ of a model are mapped to the same NPU $\langle T_R, T_P, T_C \rangle$, naturally some compute or processing elements will remain underutilised during the execution of layers with deviating feature volume shape (where the globally optimised tile sizes exceed or cannot perfectly divide the corresponding matrix dimension). The proposed batching strategy is able to remedy the inefficient mapping of CONV and FC layers to hardware, by taking advantage of the increased dimensionality offered by batching in a flexible manner. In essence, Fluid Batching configures the input matrices so that different samples can be processed concurrently by the NPU, in layers where some processing elements remain underutilised, *e.g.* when $C^{(l)}{<}T_C$. Expectedly, this flexibility is decreased when the batch size remains small, which can often occur during low-traffic periods in streaming scenarios. Additionally, in some layers, mapping inefficiencies caused by other factors can remain after this optimisation is applied, *e.g.* when $P^{(l)}{<}T_P$, leading to a persistent device underutilisation.

To further improve the inference efficiency across the spectrum, in this section we introduce a mechanism, termed Stackable PEs, that empowers the NPU with further flexibility on the allocation of its resources. Stackable PEs allow partial on-the-fly restructuring of the NPU's processing elements, by re-purposing their MAC units and on-chip memory buffers to better fit the shape of the input and weight matrices. Through the design depicted in Fig. 4c, a $\langle T_R, T_P, T_C \rangle$ NPU can be transformed on-the-fly to an alternative $\langle \lfloor T_R/2 \rfloor, k \cdot T_P, \lfloor T_C/k \rfloor \rangle$ design, where $k{\in}\{1/2, 2\}$, with minimum hardware overhead. This approach is essentially trading coarse- to fine-grained parallelism, while ensuring adequacy of the instantiated memory buffers, in order to maintain as many PEs as possible active during the execution of each layer. The most efficient PE configuration

for each layer of the target workload is identified during DSE and controlled by the Fluid Batching Engine during inference through its $k^{(l)}$ signal.

Design Space Exploration (DSE). In this context, we enhance the conventional DSE mechanism for edge NPUs in a two-fold manner. First, we consider the attainable performance of each candidate design point *across different batch sizes*, effectively co-optimising the hardware architecture for various batching scenarios. Second, for each visited NPU design point $d{=}\langle T_R, T_P, T_C\rangle$, we expose the design spaces of Fluid Batching and Stackable PEs to the optimisation, leading to $\langle d, \boldsymbol{k}, \boldsymbol{B}_R\rangle$. Parameters $\boldsymbol{k}{\in}\{1/2, 1, 2\}^{L\times B_{\max}}$ ($k{=}1$ is the default PE configuration) and $\boldsymbol{B}_R{\in}[1, B_{\max}]^{L\times B_{\max}}$ hold the Stackable PEs and Fluid Batching configuration for each layer and batch size, and are used to populate the Fluid Batching Engine (Fig. 3). Hence, we cast DSE as a formal optimisation problem:

$$\langle d^*, \boldsymbol{k}, \boldsymbol{B}_R\rangle = \arg\max_{d} \underbrace{\sum_{b=1}^{B_{\max}} w_b}_{\text{for each batch size}} \max_{\boldsymbol{k},\boldsymbol{B}_R} T\Big(\langle d, \boldsymbol{k}, \boldsymbol{B}_R\rangle, W, b\Big) \tag{2}$$

where weight w_b is used to control the contribution of each batch size. We estimate the performance, $T(\cdot)$, in GOp/s of each examined design point on a given workload W with a combination of analytical and roofline modelling [30]. Finally, the highest performing design is obtained through exhaustive search.

Microarchitecture. Fig. 5 shows the hardware design of the Fluid Batching Engine, comprising the Fluid Batching Control Block (FBCB) and a Control Unit (CU). The FBCB constitutes a look-up table that stores the highest performing batching policy and Stackable PEs configuration $\langle B_R, B_P, k\rangle$ for each layer of the given DNN and for different batch sizes. As such, the FBCB contains $L{\times}B_{\max}$ entries. We store only k and B_R, and derive B_P as $B_{\text{act}}{-}B_P$. With B_R bounded by $B_{\max}$, we represent each layer's B_R entry with $\lceil\log_2(B_{\max})\rceil$ bits and encode k's three states with 2 bits.

The CU uses B_{exit} and B_{incr} (§III-B) to keep track of the active batch size (B_{act}). When a new layer is to be processed, the CU uses B_{act} and the layer index l to address the FBCB and fetch the correct batching policy. Finally, the batching policy is used to configure the NPU's DMA controllers. This affects how the NPU reads and writes the input and output activation matrices from and to the off-chip memory.

B. Exit-Aware Preemptive Scheduler for Early-Exit DNNs

Conventional inference systems [4]–[8] utilise the same batch size for the whole DNN. In particular, once a batch of inputs has been dispatched to the accelerator, future arriving samples have to wait until the processing of the whole batch has completed. The limitations of this approach become especially evident when processing early-exit DNNs, where the active batch size can change dynamically through the depth of the network. In this case, the status-quo model-level batching constrains the DNN to execute until the end with a reduced batch size, even if it severely underutilises the accelerator's resources.

We propose an exit-aware preemptive scheduler that considers both the active batch size and the SLO to balance latency, throughput and SLO satisfaction. In contrast to existing systems, we introduce a scheduling granularity at the subnet level, with boundaries at the intermediate exits. As such, the already running *active* batch can be preempted and new samples can be processed in an interleaved manner. Concretely, the scheduler selectively preempts execution at the exit points and dispatches a new batch so that it can catch up. Next, the new samples are merged with the preempted samples to form a larger batch and resume execution with increased hardware utilisation. Formally, we parametrise this policy as $\langle B_{\max}, T_{\text{SLO}}\rangle$, where T_{SLO} is the latency SLO. Our scheduler launches execution as soon as the first sample enters the request queue (Fig. 3). Algorithm 1 details our scheduling method. Next, we point to the related lines as we describe our method.

Preemptible Points. Given an L-layer DNN with a set of early exits $\mathcal{E}$, our scheduler allows for preemption only at the early-exit points $i{\in}\mathcal{E}$. This design decision is based on two key insights. First, preemption is beneficial only when there are dynamic changes in the batch size and hence evaluating the preemption criterion elsewhere leads to redundant computation. Second, treating every layer as preemptible would introduce prohibitively high overhead [9]; the scheduler would be invoked too regularly, inducing excessive computations and interruption of the NPU's operation; *e.g.* our approach yields 16.6× fewer scheduler invocations for a 4-exit ResNet-50 compared to LazyBatching's layerwise approach [9].

Preemption Mechanism. Fig. 6 illustrates the operation of our scheduler. Upon preemption at exit i, the intermediate results of the *remaining* active batch of size $B_{\text{rem}}{=}B_{\text{act}}{-}B_{\text{exit}}$ are written back to the off-chip memory (line 7) and a new batch of samples is issued (lines 9-11). The new batch size is determined as $B_{\text{incr}}{=}\min(N_Q, B_{\text{slack}})$, where $B_{\text{slack}}{=}B_{\max}{-}B_{\text{rem}}$ and N_Q is the instantaneous queue size. When the new samples reach the preemption point (line 14), B_{incr} might have been reduced as some samples might have exited at the earlier exits. Batch backfilling is accomplished per exit and non-recursively, *i.e.* no nested preemption for intermediate exits. This allows us to have bounded stalling time for the preempted samples while maximising hardware utilisation. Finally, the rest of the DNN is executed with a merged batch size of $B_{\text{merged}}{=}B_{\text{rem}}{+}B_{\text{incr}}$ (line 17), until the next preemption point (line 6).

Preemption Criterion. We introduce an SLO-aware preemption criterion that aims to minimise SLO violations. As a first step, the scheduler estimates the remaining time T_{slack} until the latency SLO is reached for the oldest sample in the active batch:

$$T_{\text{slack}} = T_{\text{SLO}} - \Big(T_{\text{wait}} + T_{\text{exec}}^{\text{so-far}}\Big) \tag{3}$$

where T_{wait} is the queue waiting time of the oldest sample in the active batch and $T_{\text{exec}}^{\text{so-far}}$ is the execution time passed until the preemption point was reached. The final criterion (line 12) is:

$$T_{\text{overhead}} < T_{\text{slack}} \quad \text{with} \quad T_{\text{overhead}} = T_{0:i}^{B_{\text{incr}}} + T_{i+1:L-1}^{B_{\text{merged}}} \tag{4}$$

where $T_{i:j}^{B}$ is the execution time from exit i to j (inclusive) with batch size B, capturing the time for the *new* batch to reach the i-th exit and for the *merged* batch to complete the inference. As B_{incr} might become smaller due to its own early-exiting samples, the actual latency can be smaller and hence this approach can lead to a slight overestimation of the overhead. Nonetheless, it constitutes an upper bound and hence prevents the scheduler from introducing additional SLO violations.

Exit-Level Latency Prediction. Leveraging the deterministic dataflows of modern NPUs, we developed a performance model to estimate the per-exit latency. Alternatively, if the NPU architecture is unknown, we can profile the average per-exit latency. At design time, we measure the per-exit latency by varying the batch size from 1 to $B_{\max}$ and we bookkeep the results. Upon deployment, the scheduler loads the results in a $N_{\text{exits}}{\times}B_{\max}$ look-up table and uses it at run time to evaluate the preemption criterion and guide preemption decisions.

Overhead. As we are targeting a streaming setting, our scheduler picks B_{incr} samples from the top of the queue when forming new batches, so the scheduling complexity is $O(1)$. Samples can terminate out-of-order, with the reordering happening afterwards using the sample ID. The required input/output buffers in the off-chip memory are

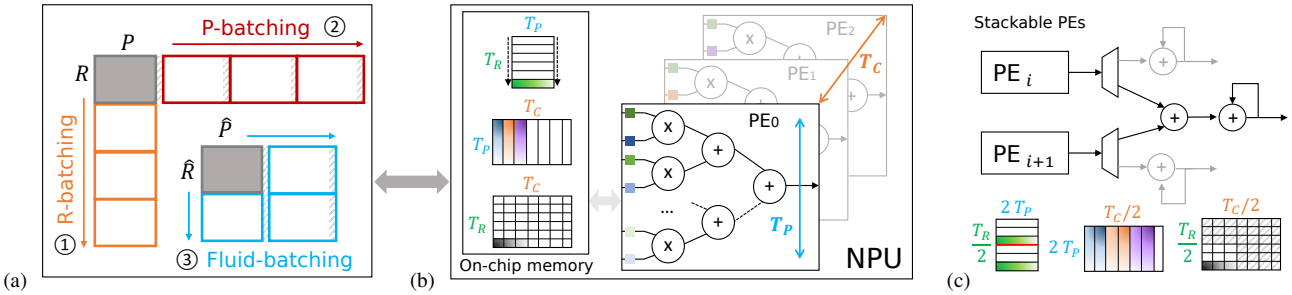


Fig. 4: **(a)** Input matrix formation with different batching strategies (B=4), **(b)** NPU architecture, **(c)** Stackable PEs (k=2).

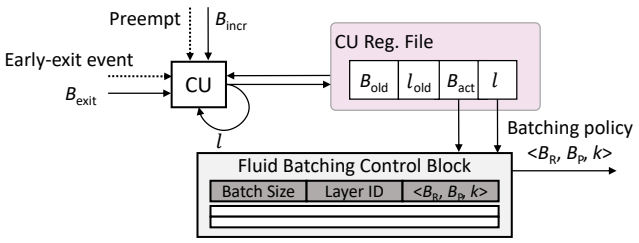


Fig. 5: Microarchitecture of the Fluid Batching Engine.

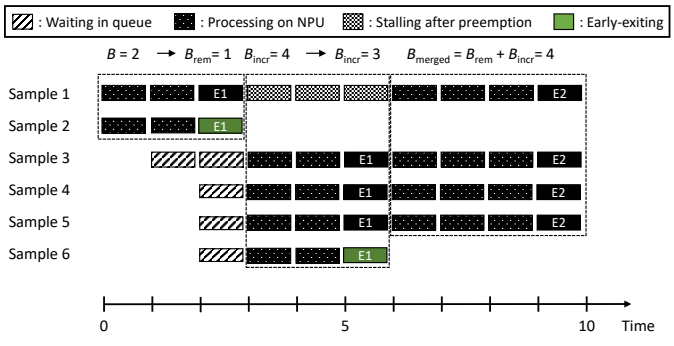


Fig. 6: Scheduling timeline for a 2-exit (E1, E2) DNN with $B_{\max}$=8. After reaching E1, sample 2 exits early. The scheduler evaluates the SLO-aware preemption criterion and issues a preemption with B_{incr}=3. Sample 1 is preempted and samples 3 to 6 are processed up to E1. Sample 6 exits early. The scheduler determines that further preemptions would introduce SLO violations and merges the remaining samples, leading to a batch size of 4 up to exit E2.

allocated upon initialisation to be large enough to accommodate the largest activation matrix for $B_{\max}$, hence avoiding run-time memory management and the associated latency. Finally, as we preempt active batches at the end of an exit's last layer's execution, the output activations are already stored in the off-chip memory, avoiding the need for explicit checkpointing operations. The main overhead is that the memory transfer of the new input samples is not overlapped with computation. However, our empirical evaluations showed that this contributed at most 0.05% in latency across instances.

IV. Evaluation

Setup. We target Xilinx ZC706 hosting the mid-range Z7045 FPGA and Xilinx ZCU104 with the larger ZU7EV. All hardware designs were developed in Vivado HLS and clocked at 150 and 200 MHz for ZC706 and ZCU104, respectively. All designs were synthesised and placed-and-routed using Vivado Design Suite (v2019.2) and run on both boards. The Arm CPU was used to set up the AXI interfaces to the

Algorithm 1: Exit-Aware Preemptive Scheduling Method

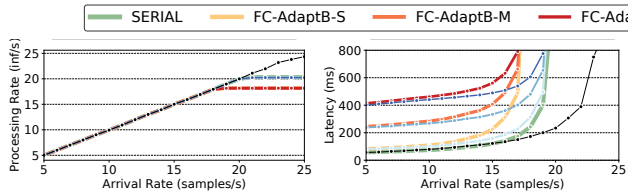
Input: Model m with exits $\mathcal{E}$
Latency SLO T_{SLO}
Maximum batch size $B_{\max}$

```
/* — Initialise batch size and exit index — */
B_act ← min(N_Q, B_max)
i ← 1

/* — Process until the end of the network — */
while i < |E| do                          ▷ Until having reached the last exit
    ProcUntilExit(m, B_act, i)            ▷ Process up to the i-th exit
    B_rem ← B_act − B_exit
    B_act ← B_rem

    while B_rem < B_max do                ▷ If there is room in the batch
        B_slack ← B_max − B_rem
        B_incr ← min(N_Q, B_slack)

        if T_overhead < T_slack then      ▷ SLO-aware preemption
            PreemptActiveBatch(·)
            ProcFromStartUntilExit(m, B_incr, i)
            B_merged ← B_rem + B_incr − B_exit

            B_rem ← B_merged
            B_act ← B_merged
        else
            break
        end
    end
    i ← i + 1                             ▷ Move to next exit
end
```

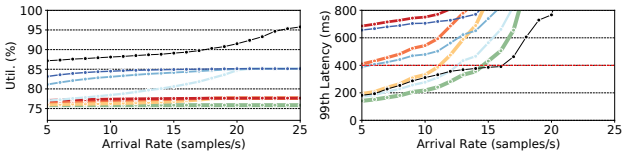
off-chip memory and run our scheduler. We measured performance via hardware counters and used 16-bit fixed-point across all experiments.

Benchmarks. We evaluate on two mainstream DNNs: ResNet-50 and Inception-v3, which constitute widely used backbones across multiple downstream vision tasks. In the multi-exit setup, we adopt the design methodology [16] of state-of-the-art hardware-aware early-exit models. The examined instances comprise three intermediate exits, placed equidistantly in terms of FLOPs on the corresponding frozen ImageNet-pretrained backbone. Similarly to relevant literature [16], [33], softmax top-1 is employed as a metric for confidence, and the exit policy is tuned to minimise the workload while maintaining accuracy within 1.5 percentage points from the original backbone [14], [17]. This optimisation led to a uniform confidence threshold of 0.8 across exits, yielding exit rates of $\langle 5.1\%, 16.9\%, 9.0\%, 69.0\% \rangle$ and $\langle 14.5\%, 18.6\%, 22.2\%, 44.7\% \rangle$, and accuracies of 75.6% and 75.8%, for ResNet-50 and Inception-v3.

NPU DSE and Resource Usage. Table I shows the designs generated by our DSE method (with w_b=1), together with their resource consumption. In addition to the processing engine, the Fluid Batching Engine's CU requires less than 0.5% of LUTs. The FBCB consumes

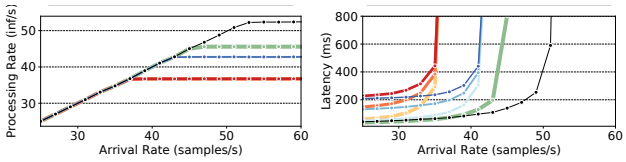


(a) Processing Rate vs Arrival Rate. (b) Avg. Latency vs Arrival Rate.

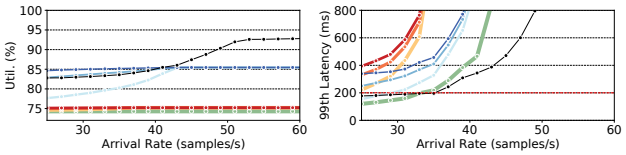


(c) Utilisation vs Arrival Rate. (d) Tail Latency vs Arrival Rate.

Fig. 7: Comparison on 4-exit ResNet-50 on ZC706 under 400ms SLO.



(a) Processing Rate vs Arrival Rate. (b) Avg. Latency vs Arrival Rate.



(c) Utilisation vs Arrival Rate. (d) Tail Latency vs Arrival Rate.

Fig. 8: Comparison on 4-exit Inception-v3 on ZCU104 under 200ms SLO.

0.57% and 0.54% of registers on ZC706 and ZCU104, while the larger FCBC of the deeper Inception-v3 uses 0.95% and 0.90%. The Stackable PEs instantiate demultiplexers and one LUT-based adder per two PEs, consuming less than 5.5% and 6% of the available LUTs for ResNet-50 and Inception-v3, respectively, on both devices.

Baselines. We compare against state-of-the-art (SOTA) batching approaches: *i)* single-sample execution (SERIAL), *ii)* FC-only batching (FC-AdaptB), *iii)* R dim. batching (R-AdaptB), and *iv)* LazyBatching [9]. Baselines *ii)* and *iii)* correspond to SOTA adaptive batching (AdaptB) systems [4]–[8]. Given the resource constraints of the target edge-grade platforms and DNN workloads, to support latency SLOs that make the system deployable, we need to limit batch size to 8. Indicatively, executing ResNet-50 on ZCU104 with B=16 yields an average latency of 277 ms that exceeds the 200-ms SLO examined below. As such, we use B_{max}=8 across all baselines. For FC and R, we set AdaptB's T_{timeout} parameter to three different values: small (S), medium (M) and large (L) corresponding to a batch-forming waiting time equal to 5%, 45% and 95% of the 99th percentile latency SLO. For the SLO-aware LazyBatching, we configure its scheduler with the respective SLO in each experiment. For each baseline, we perform DSE using a batch size of 8 and select the highest performing NPU design for the target DNN-device pair.

Performance Comparison. To evaluate the performance and adaptability of our approach across traffic levels, we vary the request arrival rate between 5 and 25 samples/s and 20 and 60 samples/s for ResNet-50 (Fig. 7) and Inception-v3 (Fig. 8), respectively. Following the MLPerf standard [11], the arrival times of the input samples were generated following a Poisson distribution, with an expected rate equal to the specified samples/s. We set the 99th percentile latency SLO to 400 ms for ZC706 and to 200 ms for ZCU104. All experiments were run three times with different seeds and the average is reported.

- Low-to-mid traffic. For slow-arriving samples, the large waiting windows of AdaptB-M and -L lead to excessive latency for batch forming, despite the higher utilisation due to large-batch processing. The small waiting windows of AdaptB-S, on the other hand, provide a better balance between utilisation and latency. SERIAL yields the lowest latency, but also the lowest utilisation. In contrast, our approach (FluidB) combines the merits of both approaches; it provides the user with SERIAL's QoE by meeting the SLO and achieving similar average latency, but also yields significantly higher utilisation (20.4% average gain across the range 5 to 18 samples/s on ZC706 and 13.5% average gain across the range 25 to 35 samples/s on ZCU104), by means of the flexible opportunistic batching of its scheduler.

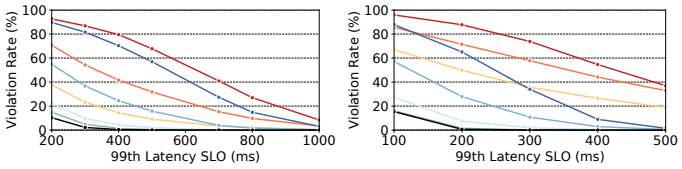
TABLE I: Design Points and Resource Consumption.

Model	Platform	Design Point $\langle T_R, T_P, T_C \rangle$	Resource Utilisation [DSPs, BRAM, LUTs]
ResNet-50	ZC706	$\langle 4652,\ 7, 128 \rangle$	[99.56, 99.96, 77]%
	ZCU104	$\langle 6832, 10, 172 \rangle$	[99.53, 99.99, 78]%
Inception-v3	ZC706	$\langle 2742,\ 4, 225 \rangle$	[100.0, 99.94, 75]%
	ZCU104	$\langle 6832, 10, 172 \rangle$	[100.0, 99.99, 79]%

- Mid-to-high traffic. For higher traffic, FC reaches a plateau in processing rate as it solely enables FC layers to benefit from batching. Similarly, R reaches its limit, because batching uniformly only along the R dimension cannot extract any additional performance. At the same time, all FC and R variants gradually lead to excessive average and tail latency and constant utilisation. This can be attributed to the impact of early-exiting on batch size and the model-level batching of these approaches; as samples exit early, the effective batch size through the DNN is dynamically decreased. As FC and R variants are not exit-aware and do not allow preemptions, they execute the rest of the DNN with smaller batch size and hence lower utilisation. This leaves input samples unnecessarily waiting in the queue, despite the extra batch room in the system.

Instead, FluidB exploits this extra room through its exit-aware scheduler that selectively preempts execution and merges new samples to form larger batches, thus processing the rest of the DNN with higher utilisation, while also benefiting from the enhanced mapping efficiency of all layers to the NPU. In addition, as the scheduler considers the SLO when making decisions, the average and tail latency are also kept under control. In particular, in traffic levels where there is enough slack from the SLO to perform a preemption, *e.g.* between 5 and 15 samples/s in Fig. 7d, the scheduler trades off a slightly higher tail latency -still without introducing violations- for a 20% increase in NPU utilisation. Above 15 samples/s, FluidB provides significant gains in both average and tail latency, even over SERIAL. This can be attributed to the fact that under high traffic, incoming samples experience large waiting times. FluidB is able to better fill any space in the active batch through its flexible batching. As such, it cuts the waiting time and boosts the utilisation of the NPU, reaching close to 95% for high traffic, far exceeding all alternatives. Finally, it sustains higher processing rate than other approaches, with above 20 and 55 inf/s on ZC706 and ZCU104, respectively.

Comparison to LazyBatching. Table II shows a comparison with LazyBatching [9] as measured on ZC706 and ZCU104, with stringent tail latency SLOs and mid-to-high arrival rates. Targeting ResNet-50, our system achieves 1.43× and 2.5× lower average latency (1.89× geo. mean) while providing a significant SLO violation reduction of 13.2 percentage points (pp) and 27.1 pp (20.15 pp avg.), respectively.



(a) 4-exit ResNet-50 - ZC706. (b) 4-exit Inception-v3 - ZCU104.

Fig. 9: Violation rate as a function of tail latency SLO (x-axis) at 15 samples/s (left) and 35 samples/s (right). See Fig. 7 for legend.

TABLE II: Comparison with LazyBatching.

Model	Platform	Arrival Rate	SLO	LazyBatching Avg. Lat.	LazyBatching Viol. Rate	Fluid Batching Avg. Lat.	Fluid Batching Viol. Rate
ResNet-50	ZC706	15 samples/s	200 ms	173 ms	23.53%	121 ms	10.33%
	ZCU104	40 samples/s	100 ms	143 ms	35.48%	57 ms	8.38%
Inception-v3	ZC706	15 samples/s	400 ms	287 ms	15.94%	213 ms	7.93%
	ZCU104	40 samples/s	200 ms	216 ms	14.20%	97 ms	6.21%

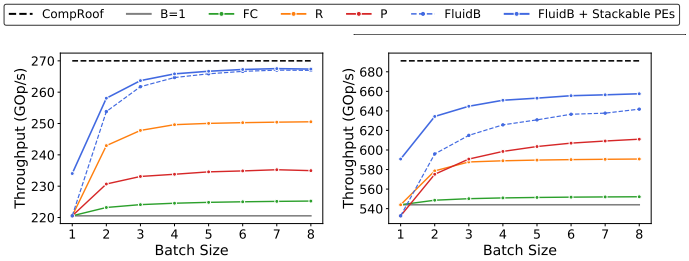
LazyBatching introduces three sources of latency overhead. First, as it is exit-unaware, it invokes the preemption logic on every layer. As such, the associated latency is not fully amortised, affecting the average latency. Second, when the maximum batch size is reached, preemption is no longer considered, despite the fact that samples may exit early. Third, LazyBatching adopts a coarse and conservative method of estimating the latency of preemption, *i.e.* instead of considering the actual latency-throughput trade-off of batched execution, it approximates batched latency as the product of batch size and single-sample latency [9]. This leads to an overestimation of the preemption overhead, with the system often deciding not to perform a preemption, even in cases where there is enough SLO slack and room in the batch due to early exiting. The overall effect is unnecessarily higher waiting time for many samples. Instead, our approach shows that accurately estimating the preemption overhead and incorporating exit-awareness into the scheduler are critical especially under high load and tight tail latency requirements, as they lead to well-amortised preemptions and significantly reduced SLO violations.

Sensitivity to SLO. To assess robustness to different latency SLOs, we sweep the tail latency SLO and measure the violation rate of FluidB and the alternatives for the same arrival rate of 15 samples/s for ZC706 and 35 samples/s for ZCU104. As shown in Fig. 9, AdaptB variants experience significant violations even when the SLO is relaxed, *i.e.* to the right of the x-axis. FluidB achieve no violations unless the SLO is set to excessively low latencies considering the workload at hand, demonstrating its effectiveness even under stringent constraints. Compared to SERIAL, FluidB provides similar or fewer violations, at the added value of substantially improved NPU utilisation.

NPU Ablation. To evaluate the benefits of Fluid Batching and Stackable PEs on the NPU, we obtain several baselines by running DSE targeting the ResNet-50 and Inception-v3 backbones without early exits, considering multiple uniform batching strategies across layers and a static batch size during inference. Fig. 10 shows the performance of the resulting designs across batch sizes. We observe that Fluid Batching converges to significantly higher performance than all alternatives, even approaching the theoretical peak of the device in the case of ResNet-50 for $B>3$. Stackable PEs provide an additional performance boost that is more prominent for smaller batch sizes (incl. $B=1$). As such, the proposed methods act complementarily, offering pronounced gains across the spectrum.

V. Discussion, Limitations & Future Work

Although our evaluation includes two diverse processing platforms and a broad range of request traffic and latency constraints, we



(a) ResNet-50 - ZC706. (b) Inception-v3 - ZCU104.

Fig. 10: Comparison of Fluid Batching with status-quo batching techniques. The impact of Stackable PEs on performance is also ablated.

have mainly evaluated the performance of the proposed system on convolutional neural networks. As early-exit mechanisms are being increasingly integrated into emerging Transformer architectures [20]–[22], [27], an investigation on the applicability of the proposed techniques in this family of models constitutes a natural continuation of our work. Notably, the attention mechanism of Vision Transformers [34] comprises a mainly compute-bound and more regular workload [35] better fitting the GEMM structure of the examined accelerators, while demonstrating limited weight reuse between samples which leads to significantly less pronounced benefits from batching. However, exploiting different realisations of the architectural flexibility of the proposed NPU design (*e.g.* to dynamically trade intra- and inter-attention head parallelism at a layer granularity at run time to deal with varying input-sequence length), as well as the introduced exit-aware scheduler to make more informed preemption decisions and improve inference efficiency under such dynamic workloads, remains an open research question to be investigated in future work. Nonetheless, many Transformer architectures to date feature a mix of convolutional and attention layers [36]–[38] mapped on the same NPU. In such cases, Fluid Batching can equip the dimensionality of convolutional layers with further flexibility, leading potentially to a more efficient mapping to the hardware accelerator.

Furthermore, in our prototype we have not allowed changes that significantly alter the accuracy of the original model. Towards introducing further flexibility, a future avenue would investigate the tuning of the confidence threshold -and thus the exit policy- at run time, as another means of optimising the execution under varying traffic.

Last, we have focused on edge-based settings, where the target platforms have limited computational capabilities and deployable configurations in terms of the maximum batch size supported. Future work would encompass scaling the proposed methods on cloud-grade setups, where batch sizes can be larger and traffic rates higher, but also offloading can be partial [23]–[25]. We also envision evaluating our approach on real-world traces of commercially available edge NPUs.

VI. Conclusion

We have presented a framework for efficiently scheduling and serving multiple DNN inference requests of multi-exit models on edge NPUs. Despite the common belief that batch processing can be prohibitive for low-latency applications, we showed that dynamicity-aware preemptive scheduling yields the best of both worlds; the high utilisation of batching and the low latency of single-sample execution. Moreover, through hardware support for Fluid Batching, the attainable NPU utilisation is pushed beyond what was previously possible. Last, exit-awareness and accurate latency estimation leads to well-amortised preemptions, even under high load, and significantly fewer SLO violations. We envision that this can pave the way towards research in hardware and dynamic DNN co-design.

REFERENCES

[1] S. Laskaridis, S. I. Venieris, A. Kouris, R. Li, and N. D. Lane, "The Future of Consumer Edge-AI Computing," in *arXiv*, 2022.

[2] N. P. Jouppi *et al.*, "Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product," in *ISCA*, 2021.

[3] S. Laskaridis, A. Kouris, and N. D. Lane, "Adaptive Inference through Early-Exit Networks: Design, Challenges & Directions," in *EMDL*, 2021.

[4] D. Crankshaw, X. Wang, G. Zhou *et al.*, "Clipper: A Low-Latency Online Prediction Serving System," in *NSDI*, 2017.

[5] C. Zhang *et al.*, "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving," in *ATC*, 2019.

[6] H. Shen *et al.*, "Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis," in *SOSP*, 2019.

[7] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching," in *SC*, 2020.

[8] M. Drumond, L. Coulon, A. Pourhabibi, A. C. Yüzügüler, B. Falsafi, and M. Jaggi, "Equinox: Training (for Free) on a Custom Inference Accelerator," in *MICRO*, 2021.

[9] Y. Choi, Y. Kim, and M. Rhu, "Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference," in *HPCA*, 2021.

[10] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated Model-less Inference Serving," in *ATC*, 2021.

[11] V. J. Reddi *et al.*, "MLPerf Inference Benchmark," in *ISCA*, 2020.

[12] Z. Wu, T. Nagarajan, A. Kumar, S. Rennie, L. S. Davis, K. Grauman, and R. Feris, "BlockDrop: Dynamic Inference Paths in Residual Networks," in *CVPR*, 2018.

[13] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez, "SkipNet: Learning Dynamic Routing in Convolutional Networks," in *ECCV*, 2018.

[14] S. Teerapittayanon, B. McDanel, and H. Kung, "BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks," in *ICPR*, 2016.

[15] S. Horvath, , S. Laskaridis, M. Almeida, I. Leontiadis, S. I. Venieris, and N. D. Lane, "FjORD: Fair and Accurate Federated Learning under heterogeneous targets with Ordered Dropout," in *NeurIPS*, 2021.

[16] S. Laskaridis, S. I. Venieris, H. Kim, and N. D. Lane, "HAPI: Hardware-Aware Progressive Inference," in *ICCAD*, 2020.

[17] B. Fang, X. Zeng, F. Zhang, H. Xu, and M. Zhang, "FlexDNN: Input-Adaptive On-Device Deep Learning for Efficient Mobile Vision," in *SEC*, 2020.

[18] A. Ghodrati, B. E. Bejnordi, and A. Habibian, "FrameExit: Conditional Early Exiting for Efficient Video Recognition," in *CVPR*, 2021.

[19] A. Kouris, S. I. Venieris, S. Laskaridis, and N. D. Lane, "Multi-Exit Semantic Segmentation Networks," in *ECCV*, 2022.

[20] W. Liu, P. Zhou, Z. Wang, Z. Zhao, H. Deng, and Q. Ju, "FastBERT: a Self-distilling BERT with Adaptive Inference Time," in *ACL*, 2020.

[21] J. Xin, R. Tang, J. Lee, Y. Yu, and J. Lin, "DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference," in *ACL*, 2020.

[22] W. Zhou, C. Xu, T. Ge, J. McAuley, K. Xu, and F. Wei, "BERT Loses Patience: Fast and Robust Inference with Early Exit," in *NeurIPS*, 2020.

[23] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed Deep Neural Networks over the Cloud, the Edge and End Devices," in *ICDCS*, 2017.

[24] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud," in *MobiCom*, 2020.

[25] Z. Liu, J. Song, C. Qiu, X. Wang, X. Chen, Q. He, and H. Sheng, "Hastening Stream Offloading of Inference via Multi-exit DNNs in Mobile Edge Computing," *TMC*, 2022.

[26] Z. Wang, W. Bao, D. Yuan, L. Ge, N. H. Tran, and A. Y. Zomaya, "Accelerating On-Device DNN Inference during Service Outage through Scheduling Early Exit," *Computer Communications (CC)*, 2020.

[27] T. Tambe *et al.*, "EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference," in *MICRO*, 2021.

[28] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training," in *HPCA*, 2020.

[29] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *ISCA*, 2015.

[30] A. Kouris, S. I. Venieris, and C. S. Bouganis, "CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks," in *FPL*, 2018.

[31] S. I. Venieris, J. F. Marques, and N. D. Lane, "unzipFPGA: Enhancing FPGA-based CNN Engines with On-the-Fly Weights Generation," in *FCCM*, 2021.

[32] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré, "Caffe con Troll: Shallow Ideas to Speed Up Deep Learning," in *Workshop on Data analytics in the Cloud*, 2015.

[33] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Weinberger, "Multi-Scale Dense Networks for Resource Efficient Image Classification," in *ICLR*, 2018.

[34] Y. Liu, Y. Zhang, Y. Wang, F. Hou, J. Yuan, J. Tian, Y. Zhang, Z. Shi, J. Fan, and Z. He, "A Survey of Visual Transformers," *TNNLS*, 2023.

[35] S. Kim, C. Hooper, T. Wattanawong, M. Kang, R. Yan, H. Genc, G. Dinh, Q. Huang, K. Keutzer, M. W. Mahoney *et al.*, "Full Stack Optimization of Transformer Inference: a Survey," *arXiv*, 2023.

[36] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-End Object Detection with Transformers," in *ECCV*, 2020.

[37] S. Mehta and M. Rastegari, "MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer," in *ICLR*, 2022.

[38] Y. Li, G. Yuan, Y. Wen, J. Hu, G. Evangelidis, S. Tulyakov, Y. Wang, and J. Ren, "EfficientFormer: Vision Transformers at MobileNet Speed," in *NeurIPS*, 2022.