HAPI: Hardware-Aware Progressive Inference

Stefanos Laskaridis[†]*, Stylianos I. Venieris[†]*, Hyeji Kim[†], Nicholas D. Lane[†,‡]
[†]Samsung AI Center, Cambridge [‡]University of Cambridge
* Indicates equal contribution.

ABSTRACT

Convolutional neural networks (CNNs) have recently become the state-of-the-art in a diversity of AI tasks. Despite their popularity, CNN inference still comes at a high computational cost. A growing body of work aims to alleviate this by exploiting the difference in the classification difficulty among samples and early-exiting at different stages of the network. Nevertheless, existing studies on early exiting have primarily focused on the training scheme, without considering the use-case requirements or the deployment platform. This work presents HAPI, a novel methodology for generating high-performance early-exit networks by co-optimising the placement of intermediate exits together with the early-exit strategy at inference time. Furthermore, we propose an efficient design space exploration algorithm which enables the faster traversal of a large number of alternative architectures and generates the highest-performing design, tailored to the use-case requirements and target hardware. Quantitative evaluation shows that our system consistently outperforms alternative search mechanisms and state-of-the-art early-exit schemes across various latency budgets. Moreover, it pushes further the performance of highly optimised hand-crafted early-exit CNNs, delivering up to 5.11× speedup over lightweight models on imposed latency-driven SLAs for embedded devices.

ACM Reference Format:
Stefanos Laskaridis, Stylianos I. Venieris, Hyeji Kim, Nicholas D. Lane. 2020. HAPI: Hardware-Aware Progressive Inference. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415698

1 INTRODUCTION

Recently, convolutional neural networks (CNNs) have become quintessential for modern intelligent systems; from mobile applications to autonomous robots, CNNs drive critical tasks including perception [7] and decision making [13]. With an increasing number of CNNs deployed on user-facing setups [36], latency optimisation emerges as a primary objective that can enable the end system to provide low response time. This is also of utmost significance for robotic platforms, to guarantee timely navigation decisions and improve safety, and smartphones to provide smooth user experience. Nevertheless, despite their unparalleled predictive power,

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415698

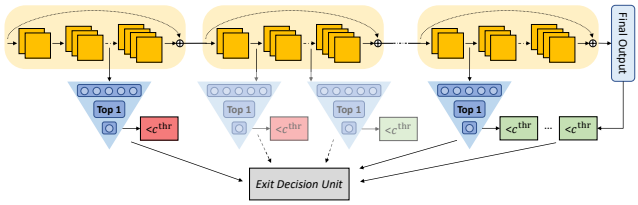


Figure 1: HAPI's early-exit network deployment architecture.

CNNs are also characterised by high inference time due to heavy computational demands, especially when deployed on embedded devices [2]. To this end, several methods have been proposed to reduce the complexity of CNNs and attain minimal latency [28].

Among the existing latency-oriented methods, one line of work focuses on the observation that not all inputs demonstrate the same classification difficulty, and hence each sample requires different amount of computation to obtain an accurate result. The conventional techniques that exploit this property typically involve classifier cascades [6, 9, 14, 15]. Despite their effectiveness under certain scenarios, these approaches come with the substantial overhead of deploying and maintaining multiple models. An alternative input-aware approach is grounded in the design of early-exit networks [10, 12, 17, 31, 43]. As illustrated in Fig. 1, early-exiting takes advantage of the fact that easy samples can be accurately classified using the low-level features that can be found in the earlier layers of a CNN. In this manner, each sample would ideally exit the network at the appropriate depth, saving time and computational resources.

So far, early-exit works have followed a hardware- and application-agnostic approach, focusing either on the hand-tuned design of early-exit CNN architectures [10, 43] or on optimising the corresponding training scheme [12, 31, 42]. Nonetheless, with CNN-based applications demonstrating diversity both in terms of performance requirements and target processing platforms, tailoring the early-exit network to the use-case needs has remained unexplored. Furthermore, due to the dynamic input-dependent execution and the large space of design choices, the tuning of the early-exit architecture poses a significant challenge that until now required prohibitively long development cycles.

In this paper, we propose **HAPI**, an automated framework that generates an optimised early-exit CNN tailored to the application demands and the target hardware capabilities. To generate a high-performance design, HAPI employs a novel accuracy- and hardware-aware design space exploration (DSE) methodology that enables the efficient traversal over a wide range of candidate designs and the effective customisation of the network to the given application-platform pair. The key contributions of this paper are the following:

- A *Synchronous Dataflow* (SDF) model for representing early-exit CNN workloads and their unique input-dependent dynamic execution. Our SDF model represents early-exit variants in a dual graph-matrix form that allows us to express

the hardware-aware design of an early-exit CNN as a mathematical optimisation problem. More importantly, it enables the previously unattainable fast traversal of the design space by means of algebraic operations that explore the accuracy-performance trade-off of the underlying early-exit network implementation.
- The HAPI framework for generating progressive inference networks customised for the target deployment platform. The developed framework takes as input a given CNN in PyTorch, performs fast design space exploration by manipulating the SDF model and yields an early-exit implementation customised to meet the user-specified latency target at the maximum accuracy. Through a multi-objective search algorithm, HAPI explores early-exit designs at both the architectural and exit-policy levels, enabling the rapid adaptation of the target CNN across heterogeneous hardware without the need for retraining, by means of a *train-once, deploy-anywhere* workflow.

2 BACKGROUND AND RELATED WORK

Several methods have been proposed for reducing the computational footprint of CNNs in order to speed up computation or fit the model into an embedded device. Diverse techniques such as *pruning* [19], *quantisation* [34] and *knowledge distillation* [41] all aim to reduce the size and latency of a model. Moreover, NetAdapt [40] also introduces hardware-awareness in its CNN pruning method. However, when a new platform is targeted, the pruned model needs to be fine-tuned through additional high-overhead training iterations. HAPI employs a single training round upfront, with the per-platform customisation taking place efficiently without training in the loop. All these methods are orthogonal to our approach and can be combined together to enable even lower inference cost.

Closer to our approach, cascade systems also exploit the difference in classification difficulty among inputs to accelerate CNN inference. A cascade of classifiers is typically organised in a multi-stage architecture. Depending on the prediction confidence, the input either exits at the current stage or passes to the next one. In this context, several optimisations have been proposed including domain-specific tuning [9], run-time model selection [6, 20, 30] and assigning different precision per stage [14, 15]. Although these techniques can be effective, the training and maintenance of multiple models add significant overhead to their deployment. In essence, multiple models have to be stored, with a scheduler implementing the model selection logic at inference time. Every time a different model is selected, the system pays the overhead of loading it.

In contrast to multi-model cascades, a few works have focused on introducing intermediate outputs to a single network. BranchyNet [31] is an network design with early exits "branching" out of the *backbone* architecture of the original network, aiming to speed up inference. While the technique is applicable to various backbone architectures, it was only evaluated on small models and datasets. Moreover, BranchyNet lacks an automated method for tuning the early-exit policy and setting the number and position of exits.

Shallow-Deep Network (SDN) [12] is a more recent work that emphasises the negative impact of always exiting at the last exit on accuracy – a term coined as "overthinking." SDN attaches early exits throughout the network and explores the joint training of the exits together with the backbone architecture. However, the placement of early exits is always equidistant and their number is fixed to six, without optimising for the task at hand or the device capabilities. Moreover, the degrading effect of early-exit placement to the accuracy of subsequent ones in joint training is not discussed. Last, although the approach is evaluated on various networks, they do not show any scalability potential to the full ImageNet dataset.

On the other hand, MSDNet [10] builds on top of the DenseNet architecture, with each layer working on multiple scales. At each layer, the network maintains multiple filter sizes of diminishing spatial dimensions, but growing depth. These characteristics make the network more robust to placing intermediate classifiers. However, this is a very computationally heavy network, which in turn makes it difficult to deploy on resource-constrained, latency-critical setups. Moreover, the placement of exits and their co-optimisation during training can hurt the performance of subsequent classifiers, or even lead to instability and non-convergence. Albeit this challenge has motivated HAPI's approach of decoupling the training of the early exits from the backbone network (see *Early-exit-only training* in Sec. 4.3), subsequent work from the same authors presents techniques for alleviating the limitations of early-exit networks training. Their proposed methodology remains orthogonal to our work [21].

We also note that in the existing early-exit approaches, the exit policy and the number and location of the early exits are determined *manually*. HAPI automates this process by tailoring the early-exit network to the performance requirements and target platform.

3 HAPI OVERVIEW

Fig. 2 shows an overview of HAPI's processing flow. The framework is supplied with a high-level description of a network, the task-specific dataset, the target hardware platform and the requirements in terms of accuracy, latency and memory. First, if the supplied CNN is not pre-trained, the *Trainer* component trains the network on the supplied training set. Next, the architecture is augmented by inserting intermediate classifiers at all candidate early-exit points, leading to an *overprovisioned* network. At this stage, HAPI freezes the main branch of the CNN and performs early-exit-only training (Sec. 4.3). As a next step, the trained overprovisioned network is passed to the *System Optimiser* to be customised for the target use-case (Sec. 6). At this stage, the *On-device Profiler* performs a number of runs on the target platform and measures the per-layer latency and memory of the overprovisioned CNN. Next, the *SDF Modelling* module converts the early-exit network to HAPI's internal representation (Sec. 5) and the optimiser traverses the design space following a hardware-aware strategy to generate the highest-performing design.

4 EARLY-EXIT NETWORK DESIGN

Given a CNN, the design space of early-exit variants is formed by the free parameters that would yield the resulting early-exit network (Fig. 1). These include 1) the number and 2) positions of exits, 3) the exit policy, 4) the training scheme and 5) the architecture of each exit. In this respect, HAPI adopts a training strategy that enables the co-optimisation of the number and positioning of early exits, the efficient exploration of various design points and the rapid customisation to the performance goals and target platform.

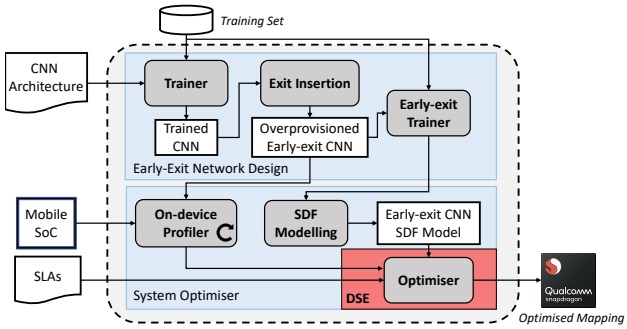


Figure 2: Overview of HAPI's processing flow.

4.1 Number and Placement of Early Exits

The number and positions of early exits have a direct impact on both the accuracy and latency of the end network [21]. Nevertheless, so far the conventional exit placement approach for early-exit networks [10, 12, 17, 31, 38, 43] is in a uniform, network- and platform-agnostic manner that severely constrains the optimisation opportunities for the target application. To enable fine-grained customisation and capture a wide range of designs, our placement scheme 1) allows early-exit positioning along the depth of the CNN, 2) operates at a fine granularity by allowing exits within building blocks[1] of the corresponding network family and 3) sets no *a priori* constraint to the number of exits. This approach differs to existing schemes [10, 12, 31, 38, 43] which keep a coarse granularity, with exits allowed only after a network's building blocks.

With this formulation, the structure of the early-exit model comprises a subset of the candidate early exits. The number and positions of the exits are selected during the DSE (described in Sec. 6). Given a subset of early exits, the latency of executing each subgraph on the target platform is known at design time, based on the on-board measurements from the *On-device Profiler* and the developed performance model (detailed in Sec. 6.2). As a result, the end-to-end latency of the early-exit network is estimated without the need for time-consuming on-device runs in the DSE loop.

4.2 Exit Policy

HAPI employs the confidence of each early classifier to identify potentially misclassified samples. At run time, low-confidence outputs are propagated to the next exit to maximise the probability of obtaining an accurate prediction. To estimate the confidence of a prediction, we employ the top-1 output of an exit, *i.e.* $\text{top1}(\boldsymbol{p})$ where $\boldsymbol{p}$ is the output of the softmax layer [5]. In this respect, a prediction is considered confident, and thus exits at the i-th classifier, when the condition $\text{top1}(\boldsymbol{p}_i) \geq c^{\text{thr}}$ is satisfied, where c^{thr} is the confidence threshold. If none of the instantiated classifiers exceeds the confidence threshold, the output of the most confident classifier is selected, leading to the following early-exit strategy: $\hat{y} = \max_{i \in [1, N_{\text{exit}}]} \text{top1}(\boldsymbol{p}_i)$, where $\hat{y}$ is the final output of the network for the current input. In our early-exiting scheme, we treat c^{thr} as a parameter that is shared across early exits and is autotuned by HAPI to meet the user-specified accuracy and latency. The selection of c^{thr} is exposed to the DSE (Sec. 6), and is co-optimised along with the number and positioning of the intermediate exits.

At run time, the *Exit Decision Unit* (see Fig. 1) considers the latency budget and, using the hardware-aware latency estimator, configures the network to execute up to the last classifier that does not violate the latency constraint. In contrast to existing progressive inference systems [12, 31, 43], whose exit strategy is design-time configurable, HAPI's approach enables the generated network to adapt upon deployment and change early-exit policy at run time, based on the device load or specific app requirements.

4.3 Early-Exit Training Scheme

There are two different training schemes that one can follow:

End-to-end training: Once the early-classifier positions have been fixed, the network can be trained from scratch, jointly optimising all the classifiers. However, this approach comes at a cost: a multi-objective cost function has to be defined so as to balance learning among all classifiers [10, 12, 31]; the classifiers can affect each other's accuracy based on their positioning in the CNN; the network needs new hyperparameter tuning for training; the network might not converge; high turnover time for exploring different exit positions, due to the required retraining and the associated long training time. On the contrary, a benefit of the end-to-end training is the higher accuracy if the classifiers are positioned correctly [12].

Early-exit-only training: A more modular approach to training early-exit networks is to first train the original network and then the intermediate exits. Specifically, the network is initially trained with only the last classifier attached. Then, intermediate exits are added at all candidate points and trained with the main backbone of the network frozen.[2] Last, only the most relevant classifiers can remain attached to the network, depending on their accuracy, exit rate and position in the network.

We select the latter approach as our training method in HAPI, due to its high flexibility in post-training customisation with respect to use-case requirements and target hardware. The first approach of joint training as a strict prerequisite to assess a design's performance not only limits the tractability of evaluating many alternative early-exit designs, but also imposes a maintenance cost for deploying such a model in the wild, where it will run on heterogeneous hardware [2, 36]. In this case, the overhead of retraining a network variant whenever a different platform is targeted can be prohibitive.

4.4 Early-Exit Architecture

In this work, we treat the exit's architecture as an invariant across the exits, borrowing the structure of MSDNet classifiers [10].

5 MODELLING FRAMEWORK

Several deep learning systems [27, 33, 39] and frameworks [1, 3, 32] model CNNs as computation graphs. Typically, the primary goal of this approach is to capture the dependencies between operations and expose their computational and memory requirements in order to apply compiler or hardware optimisations. While this approach suits the execution predictability [27, 35, 37, 40] of typical CNN workloads where the *exact same* computation graph is executed for all inputs, early-exit networks pose a unique challenge: due to their input-dependent early-exit mechanism, samples processed

[1] For residual and Inception networks, the building blocks are the residual and Inception modules respectively. For networks without skip or multi-path connections (*e.g.* VGG), classifiers can be placed after conv and pool layers.

[2] Weights of "frozen" layers do not get updated during the backpropagation phase.

by early-exit models can exit at different points along the network based on their complexity, leading to non-deterministic execution.

To analyse and optimise the deployment of early-exit networks, an *execution-rate*-driven modelling paradigm is introduced. The proposed modelling framework builds upon synchronous dataflow (SDF) [18] and enhances it to capture the unique properties of early-exit CNN workloads. HAPI represents *design points* as SDF graphs (SDFGs) that correspond to different early-exit variants (Fig. 3). Given a CNN's overprovisioned architecture, an SDFG, $G = (V, E)$, is formed by assigning one SDF node $v \in V$ to each layer. Its edges $e \in E$ represent data flow between the network's layers. The SDFG can be represented compactly by a *topology matrix*, $\mathbf{\Gamma}$. Each column of $\mathbf{\Gamma}$ corresponds to a node and each row to an edge of the SDFG. Each element γ_{ij} is an integer value that captures the *production/consumption rate* of node j on edge i and its sign indicates the direction of the data flow.

The proposed framework enhances the SDF model with two extensions: 1) The decomposition of the topology matrix ($\mathbf{\Gamma}$) into two matrices ($\mathbf{C}$ and $\mathbf{R}$, Eq. (1)). Each of the two matrices allows us to analyse a design point based on the distinct components that affect its performance; 2) A method for propagating the effects of local tunings to the overall performance of the design. The proposed approach automatically propagates the effect of a local change to the rest of the SDF graph and calculates the execution rates of different parts of the early-exit network.

Topology Matrix Structural Decomposition. To expose the factors that shape the performance of a design point, we decompose the topology matrix into the Hadamard product[3] between two matrices. The first matrix is the *connectivity matrix*, denoted by $\mathbf{C}$. Each element $c_{ij} \in \{-1, 0, 1\}$ indicates whether node j is connected to another node via edge i, with 1 and -1 signifying data production and consumption respectively, and 0 no connection. The second matrix is the *rates matrix*, denoted by $\mathbf{R}$. Each element $r_{ij} \in [0, 1]$ captures the expected normalised rate of data production or consumption of node j on edge i. A value of 0 indicates no data flow and 1 indicates that data are produced or consumed by node j on edge i at *every* input sample. Following this decomposition, for a network with N_b backbone layers and N candidate exit positions, the topology matrix of the SDFG is expressed as follows:

$$\mathbf{\Gamma} = \mathbf{C} \odot \mathbf{R} \tag{1}$$

where $\mathbf{\Gamma} \in \mathbb{R}^{|E|\times|V|}$, $\mathbf{C} \in \{-1, 0, 1\}^{|E|\times|V|}$, $\mathbf{R} \in [0, 1]^{|E|\times|V|}$ with $|V| = N_b + N$ nodes and $|E| = N_b + N - 1$ edges. To accommodate the real-valued rates matrix $\mathbf{R}$, we extend the conventional SDF and allow the topology matrix to contain real values. The two-matrix representation allows us to decouple the architecture of the early-exit network, *i.e.* the number and position of exits, through matrix $\mathbf{C}$, and the impact of the early-exit policy and the inter-exit dynamics on execution rates through matrix $\mathbf{R}$.

Fig. 3 shows the translation of an example early-exit network to the corresponding SDF graph. In this scenario, the early-exit network consists of seven layers, five in the backbone architecture (N_b=5), two potential early-exit positions (N=2) and one selected early-exit (N_{exit}=1). A sample early-exits at the first exit (layer 7) if the prediction confidence exceeds the threshold c^{thr}=0.85 of

[3]The Hadamard product, denoted by $\odot$, is defined as the elementwise multiplication between two matrices.

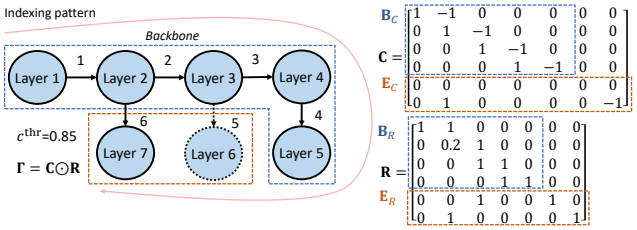


Figure 3: HAPI design point as an SDF graph.

the early-exit policy. In this example, we assume that the selected confidence threshold leads to 80% of the inputs to stop at exit 1. The 2nd column of matrix $\mathbf{C}$ corresponds to layer 2 and has two edges (2 and 6) to layer 3 and 7 respectively. With the exit rate at exit 1 being 80%, only 20% of the inputs carry on from layer 2 to 3 and hence the associated element $r_{2,2}$ of $\mathbf{R}$ is set to 0.2. Finally, the 5-th row of $\mathbf{C}$ is set to zero as the second exit (layer 6) is not instantiated.

Following our indexing scheme for nodes and edges (red arrow in Fig. 3), the topology matrix $\mathbf{\Gamma}$ and both its constituent matrices $\mathbf{C}$ and $\mathbf{R}$ have a profound structure (Eq. (2)): i) the first N_b-1 rows capture the backbone CNN architecture forming the *backbone submatrix* $\mathbf{B}$. The submatrix is upper bidiagonal with nonzero elements only along the main diagonal and the diagonal above it;[4] ii) the rest of the rows are equal to the number of candidate early exits, forming the *exits submatrix* $\mathbf{E}$. Given a selection of number and position of exits, only the corresponding entries of $\mathbf{E}$ are nonzero. Eq. (2) shows the partitioned structure of $\mathbf{\Gamma}$.

$$\mathbf{\Gamma} = \left[\begin{array}{c|c} \mathbf{B} & \mathbf{O}_{(N_b-1)\times N} \\ \hline \mathbf{E} & \end{array}\right] \tag{2}$$

with $\mathbf{B} \in \mathbb{R}^{(N_b-1)\times N_b}$, $\mathbf{E} \in \mathbb{R}^{N\times(N_b+N)}$ and $\mathbf{O}$ is the zero matrix. The same structure is present in $\mathbf{C}$ and $\mathbf{R}$, consisting of the respective submatrices $\mathbf{B}_C$, $\mathbf{E}_C$, $\mathbf{B}_R$ and $\mathbf{E}_R$. This partitioned structure enables the efficient manipulation of the SDF model by operating only on specific submatrices, as detailed in Sec. 6.1.

Automatic Execution Rate Propagation. Given its intrinsic input-dependent data flow, a key characteristic of an early-exit network is the varying execution rate of different parts of the architecture due to its conditional execution. In our modelling framework, we introduce a method to automatically obtain the execution rates of different parts of the network, while propagating them along the architecture. In conventional SDF theory [18], by solving $\mathbf{\Gamma}\boldsymbol{q} = \mathbf{O}$, we can derive an admissible execution schedule for the topology matrix. In this case, vector $\boldsymbol{q} \in \mathbb{R}^{|V|}$ indicates how many times each node should be executed in one schedule period in order to avoid deadlocks and unbounded buffering.

In HAPI, we introduce an alternative view, tailored to early-exit networks. Under this view, we interpret q_i as the expected normalised *execution rate* of the i-th node, *i.e.* the probability of executing the i-th layer when processing a sample. To enable this interpretation, we set a constraint on the range of $\boldsymbol{q}$'s elements so that $\boldsymbol{q} \in [0, 1]^{|V|}$ and proceed to obtain $\boldsymbol{q}$ by solving $\mathbf{\Gamma}\boldsymbol{q} = \mathbf{O}$. Following this approach, in the example of Fig. 3, the vector would be $\boldsymbol{q} = [1, 1, 0.2, 0.2, 0.2, 0, 1]^T$. Our method enables two key functions. First, it provides the reinterpretation of the values of $\boldsymbol{q}$ as the execution rates of the network's layers. For Fig. 3, these values

[4]In the case of multi-branch modules such as residual, Inception and depthwise-separable blocks, the same partitioned structure exists, but $\mathbf{B}$ is not upper bidiagonal.

indicate that the first two layers and the early exit (last element of $\boldsymbol{q}$) would process all inputs, while layers 3 to 5 are expected to process 20% of the inputs, due to the 80% that would exit early. Second, the effect of a local tuning (*e.g.* a selection of c^{thr} that led to 80% exit rate in the first exit) is automatically propagated along the SDFG through the calculation of $\boldsymbol{q}$ (*i.e.* the 20% production rate of layer 2 is propagated to the execution rate of layers 3-5 through the 0.2 value in the corresponding elements of $\boldsymbol{q}$). As a result, the overall impact of a local change on the design's performance is automatically propagated and calculated rapidly through $\boldsymbol{q}$.

6 DESIGN SPACE EXPLORATION

In HAPI, the basic mapping of an SDF graph (SDFG) is the early-exit network implementation as illustrated in Fig. 1. The network architecture is first constructed by mapping each SDFG node to a layer and connecting them according to the connectivity matrix $\mathbf{C}$. Furthermore, the *Exit Decision Unit* is configured using the selected confidence threshold c^{thr}. While HAPI's highly parametrised early-exit network design provides fine-grained control over customisation, it also leads to an excessively large design space.

In this context, we exploit the analytical power of our modelling framework to efficiently navigate the design space. We visit alternative designs by tuning HAPI's design parameters through a set of graph *transformations* that can be directly applied over the SDF model (Sec. 6.1). To assess the quality of a design point without continuously accessing the target platform, we build analytical models that provide rapid estimates of the attainable latency and memory footprint (Sec. 6.2). Overall, these exploration and design evaluation techniques are integrated into HAPI's optimiser which solves a multi-objective optimisation formulation of the DSE task (Sec. 6.3).

6.1 Early-Exit Engine Search Space

Based on its early-exit network parametrisation (Sec. 3), HAPI defines a particular design space formed by 1) the number of early exits, 2) their positions along the network and 3) the early-exit policy. In this respect, we model the configuration of an early-exit design with a tuple of the form $\langle N_{\text{exit}}, \boldsymbol{p}_{\text{exit}}, c^{\text{thr}}, \boldsymbol{\Gamma}\rangle$, where N_{exit} is the number of selected exits, $\boldsymbol{p}_{\text{exit}} \in \{0,1\}^N$ the positioning vector with the i-th element set to 1 if an exit is placed at position i, c^{thr} the threshold of the early-exit policy, and $\boldsymbol{\Gamma}$ the topology matrix.

Our SDF-based modelling allows us to express the complete design space captured by HAPI by defining graph transformations for the manipulation of SDFGs. In this way, any design tuning that transforms the SDFG can be applied directly to the topology matrix $\boldsymbol{\Gamma}$ by means of efficient algebraic operations. HAPI employs the following set of transformations:

(1) **Early-Exit Repositioning** $exitrepos(N_{\text{exit}}, \boldsymbol{p}_{\text{exit}})$: The first transformation changes the number and position of early exits along the network by adding and removing early-exit nodes on the SDF graph. Early-exit repositioning modifies both the structure of the SDF graph by altering the architecture of the CNN and the exiting rates as different combinations of early exits have varying early-exit dynamics. As a result, this transformation affects both the connectivity matrix $\mathbf{C}$ and rates matrix $\mathbf{R}$.

(2) **Confidence-Threshold Tuning** $conftune(c^{\text{thr}})$: The second transformation modifies the early-exit policy by tuning the confidence threshold c^{thr}. In particular, low values lead to a less restrictive policy with more samples exiting at the earlier stages of the CNN, while higher values form a more conservative policy with more samples exiting deeper in the network. As a result, a change in c^{thr} has an impact on the exit rate of each exit and hence affects only the rates matrix $\mathbf{R}$. Since the network architecture remains unchanged, matrix $\mathbf{C}$ is not modified.

Algorithm 1: Design tuning as algebraic operations

Input: Topology matrix $\boldsymbol{\Gamma} = \mathbf{C} \odot \mathbf{R}$
Transformation $t \in \mathcal{T}$
Output: Updated topology matrix $\boldsymbol{\Gamma}'$

1 /* - - - *update connectivity matrix* $\mathbf{C}$ - - - */
2 **if** t **is** $exitrepos(N_{\text{exit}}, \boldsymbol{p}_{\text{exit}})$ **then**
3 $\quad \mathbf{E}_{\text{sel}} \leftarrow diag(\boldsymbol{p}_{\text{exit}})$ // *Form the positioning matrix*
4 $\quad \mathbf{E}'_C \leftarrow \mathbf{E}_{\text{sel}}\mathbf{E}^{\text{all}}_C$ // *Update early-exit submatrix* $\mathbf{E}_C$
5 $\quad \mathbf{C}' \leftarrow \text{UpdateMatrix}(\mathbf{B}_C, \mathbf{E}'_C)$
6 **end**

Backbone submatrix $\mathbf{B}_C$ is not affected by changes in the no. and position of exits.

7 /* - - - *update rates matrix* $\mathbf{R}$ - - - */
8 **if** t **is** $exitrepos(N_{\text{exit}}, \boldsymbol{p}_{\text{exit}})$ **or** $conftune(c^{\text{thr}})$ **then**
9 $\quad \boldsymbol{r}_{\text{exit}} \leftarrow \text{MemoisedData}(N_{\text{exit}}, \boldsymbol{p}_{\text{exit}}, c^{\text{thr}})$ // *Obtain exit rates through memoisation (Sec. 6.3)*
10 $\quad \boldsymbol{r}_{\text{layer}} = \mathbf{E}'_C(:, 1 : N_b)^T \boldsymbol{r}_{\text{exit}}$ // *Map exit rates to their layer positions*
11 $\quad \mathbf{B}'_R = \mathbf{B}_R \odot diag(\boldsymbol{r}_{\text{exit}})$ // *Update the backbone submatrix*
12 $\quad \mathbf{R}' \leftarrow \text{UpdateMatrix}(\mathbf{B}'_R, \mathbf{E}_R)$
13 **end**
14 $\boldsymbol{\Gamma}' = \mathbf{C}' \odot \mathbf{R}'$ // *reconstruct topology matrix*

Given these transformations, we define the transformation set as $\mathcal{T} = \{exitrepos(N_{\text{exit}}, \boldsymbol{p}_{\text{exit}}), conftune(c^{\text{thr}})\}$. To generate a new design point, we apply one or multiple transformations from $\mathcal{T}$ over the current design point s: $s' \xleftarrow{t} s,\ t \in \mathcal{T}$. Formally, the overall search space defined by HAPI is captured by means of a set $\mathcal{S}$ that contains all reachable alternative designs:

$$\mathcal{S} = \left\{ s \mid s = \left\langle s_{\text{overprv}}, \mathcal{T}^* \right\rangle \right\},\ \mathcal{T}^* \subset \mathcal{T} \tag{3}$$

where s_{overprv} is the overprovisioned variant of the CNN, $\mathcal{T}^*$ is the subset of transformations that are applied on s_{overprv} to obtain s.

Our SDF-based framework allows us to express these transformations through algebraic operations directly applied on the topology matrix as described by Algorithm 1. The algorithm takes as inputs the $\boldsymbol{\Gamma}$ matrix of the given SDFG and the transformation, t, to be applied. The connectivity matrix $\mathbf{C}$ is affected by the early-exit repositioning (lines 1-6), while the rates matrix $\mathbf{R}$ is affected by both transformations (lines 7-13). On line 3, a positioning matrix is constructed with $\boldsymbol{p}_{\text{exit}}$ along its diagonal and it is used to left-multiply matrix $\mathbf{E}^{\text{all}}_C \in \mathbb{R}^{N \times (N_b+N)}$, which holds *all* the candidate exits. With this operation, only the rows of $\mathbf{E}^{\text{all}}_C$ that map to the edges between the *selected* exits and the backbone network are selected, with the rest set to zero. As changes in the number and position of exits do not affect the backbone architecture, submatrix $\mathbf{B}_C$ is not altered and the updated connectivity matrix $\mathbf{C}'$ is produced following Eq. (2) (line 5). A similar procedure is followed for $\mathbf{R}'$ on lines 7-13. First, the exit rate of each exit is calculated using an efficient memoisation scheme (line 9), detailed in Sec. 6.3. Next, the exit rates are projected to the associated layer position (line 10). Finally, the production rates of nodes that are connected to exits are updated (line 11) and $\mathbf{R}'$ is formed. As a final step, the updated topology matrix $\boldsymbol{\Gamma}'$ is constructed (line 14).

6.2 Performance and Memory Footprint Model

To estimate the latency and memory footprint of each design point, we developed an analytical performance model that leverages HAPI's modelling framework. As a first step, after the given CNN is augmented with exits at all candidate positions, the *On-device Profiler* executes a number of on-board benchmark runs to measure the per-layer execution time of the overprovisioned early-exit CNN, denoted by l_i for $\forall i \in [1, |V|]$. The execution time measurements are integrated into vector $\boldsymbol{l} = [l_1, l_2, ..., l_{|V|}]^T$. This phase takes place only *once* upfront and hence the DSE task does not require access to the target platform. Given the topology matrix $\boldsymbol{\Gamma}$ of a design point $s = \langle N_{\text{exit}}, \boldsymbol{p}_{\text{exit}}, c^{\text{thr}}, \boldsymbol{\Gamma} \rangle$, the execution rate vector $\boldsymbol{q}$ is calculated using the automatic execution rate propagation scheme (Sec. 5). With each element of $\boldsymbol{q}$ giving the expected execution rate of each layer in design point s, the hardware-specific average latency of processing an input I can be estimated as $L_{\text{hw}}(I, s) = \boldsymbol{q}^T \boldsymbol{l}$. For memory consumption, due to the typically small batch size of the inference stage, the model size (*i.e.* the CNN's weights) dominate the run-time memory. In this respect, we define the memory footprint vector $\boldsymbol{m} \in \{0\} \cup \mathbb{Z}^{+|V|}$ with the i-th element holding the footprint of the i-th node's weights. Given vector $\mathbb{1}(\boldsymbol{q}{>}0) \in \{0, 1\}^{|V|}$ masking only the nodes that are used in design point s, the memory consumption of s is estimated as $m(s) = \mathbb{1}(\boldsymbol{q}{>}1)^T \boldsymbol{m}$.

6.3 System Optimisation

To evaluate the quality of the design points that lie within the search space and select the highest-performing ones, we cast the problem as multi-objective optimisation (MOO) and design an objective function that reflects the key requirements of the use-case. With respect to latency, the majority of existing early-exit works [10, 12, 17, 31] rely on the theoretical FLOPs as a proxy to its real processing speed. Such an approach ignores essential platform-specific characteristics including caching, I/O and hardware-level features, leading to the FLOP count not accurately capturing the actual attainable performance of executing a CNN on a particular processing platform [2, 11]. In contrast, we employ a hardware-aware approach that utilises real device latency, alongside memory footprint and accuracy, as metrics to assess the quality of each design and drive HAPI's search towards high-performance designs.

In our MOO setup, we employ two objective functions (Eq. (4, 5)) that reduce the multi-objective problem to a single objective by means of the weighted sum and ϵ-constraint methods [22] respectively. In the weighted sum formulation, the modelling of the interplay between quality metrics plays a decisive role in shaping the trade-offs to be explored [22]; in HAPI the dynamics between accuracy and latency determine how much additional latency cost we allow to pay for each percentage point (pp) of accuracy gain.

As a first step, for the weights to closely capture the importance of each metric in the target application [23], the accuracy and latency of each design point s are divided by the accuracy and latency of the original CNN respectively, to obtain a non-dimensional objective function. Next, we model the dynamics of the accuracy-latency trade-off through a non-linear logarithmic function (Fig. 4). The selected function reflects the fact that the accuracy-latency trade-off is more prominent in the beginning of the network compared to the end, where the accuracy typically plateaus and we obtain diminishing returns on the computation time. In this respect, we

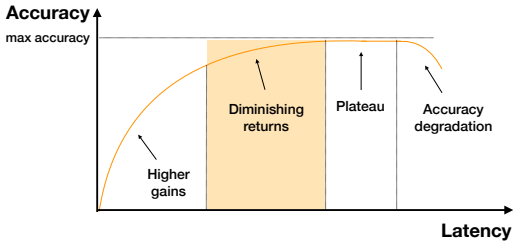


Figure 4: Accuracy-latency trade-off.

set the accuracy weight to 1 and tune the latency weight w_{lat} via grid search to obtain the most beneficial trade-off in the DSE phase, aiming for a solution in the highlighted area of Fig. 4. Overall, we pose the following MOO problems:

$$\max_s \ \frac{A(s)}{A^{\max}} - \text{w}_{\text{lat}} \cdot \log\left(\frac{L_{\text{hw}}(I, s)}{L_{\text{hw}}^{\max}} + 1\right) \tag{4}$$

$$\text{s.t. } L_{\text{hw}}(I, s) \leq \epsilon \quad \& \quad m(s) \leq m_{\max} \tag{5}$$

where $A(s)$ is the average accuracy of the current early-exit design s, $L_{\text{hw}}(I, s)$ and $m(s)$ are the latency and memory footprint of s on the target platform respectively, and ϵ and $m_{\max}$ are the user-specified upper bound on latency and maximum memory capacity of the platform respectively. The objective functions aim to either: 1) co-optimise accuracy and latency (Eq. (4)) or 2) also impose latency and memory constraints (Eq. (4,5)).

Efficient Evaluation through Memoisation. Given a CNN, the optimisation problems are defined over the set of all points $\mathcal{S}$ in the presented design space (Sec. 6.1). For the objective functions to be evaluated, the exit rate of each exit is required to construct $\boldsymbol{\Gamma}$ (see $\boldsymbol{r}_{\text{exit}}$ in Algorithm 1) and then calculate $L_{\text{hw}}(I, s)$ (Sec. 6.2), together with the accuracy $A(s)$. Typically, to obtain these values, the design point s would have to be materialised in the form of a CNN and run over the calibration set, monitoring how many samples stopped at each exit together with whether they were classified correctly. This process leads to the excessive overhead of running inference over the calibration set for each examined design point.

To alleviate this high cost, we exploit the key observation that by processing each sample of the calibration set *once* using the overprovisioned CNN and storing only 1) the top-1 value and 2) whether the sample was correctly classified at each exit, we can evaluate the accuracy and exit rates of any design point. For a calibration set of size $|D|$, N candidate exit positions and N_{conf} candidate confidence thresholds, the memoised evaluation would require $2|D| \cdot N \cdot N_{\text{conf}}$ elements to be stored, which can be used to evaluate the objective function of any s. As an example of the required space, for the validation set of ImageNet ($|D|$=50,000), ResNet-56 (N=58) and three confidence thresholds (N_{conf}=3), the storage requirement is 66 MB. With this approach, given the selection of exits and the confidence threshold of an examined design point, the expensive inference process is replaced with a fast lookup of the associated values from the memoised data and applying the rule of HAPI's exit strategy (Sec. 4.2). This process takes place offline at design time and hence places no burden on the end device upon deployment.

Optimiser. Given a CNN, the objective functions of the defined optimisation problems can be evaluated for all design points given the introduced memoisation scheme and the performance model of Sec. 6.2. To jointly optimise the number and positioning of early

Table 1: Target Platforms

Platform	Processor	Memory	GPU	TDP
Server	Intel i7-7820X (8 cores, HT)	128GB DDR4 @ 2133MHz	Nvidia GTX 1080Ti	400W
Jetson Xavier	8-core ARM-Karmel v8.2	16GB LPDDR4x	512-core Volta	30W, (u)10W

exits, we cast them as a search problem where we aim to select adequate early-exit positions that optimise the objective function. In this respect, for a CNN with N possible exit positions, we seek the value of the binary positioning vector $\boldsymbol{p}_{\text{exit}} \in \{0,1\}^N$ that optimises the target objective function.

In theory, the optimal early-exit design could be obtained by means of exhaustive enumeration. Given the different number of exits, exit positions and early-exit policies, the overall number of candidate designs to be examined can be calculated as $N_{\text{conf}} \cdot 2^N - 1$ where N_{conf} is the number of distinct examined values for the confidence threshold (*e.g.* {0.4, 0.6, 0.8}). With an increase in the network's depth, N increases accordingly, and brute-force enumeration quickly becomes intractable. To this end, a heuristic optimiser is adopted to obtain a solution in the non-convex space.

In this work, Simulated Annealing (SA) [24] has been selected as the basis of the developed optimiser. Given the set of SDF transformations $\mathcal{T}$ defined in Sec. 6.1, the neighbourhood of a design s is defined as the set of design points that can be reached from s by applying one of the operations $t \in \mathcal{T}$. Overall, the optimiser navigates the design space by considering the described SDF transformations and converges to a solution of the target objective function. To prune the exponential space, we introduce a prior by initially not allowing exits to be in adjacent positions. After the optimiser has selected the highest-performing design, `HAPI` explores adjacent positions of the already chosen exits, as a refinement step.

7 EVALUATION

In this section, we evaluate `HAPI`'s performance against a random search optimiser, the improvement over the state-of-the-art early-exit methods under varying latency budgets and the performance gains over hand-crafted CNN models.

7.1 Experimental Setup

In our experiments, we target two platforms with different resource characteristics (Table 1): a server-grade desktop computer and an Nvidia Jetson Xavier AGX. For the latter, we evaluate on two different power profiles (30W, underclocked 10W) by adjusting the thermal design power (TDP) and clock rate of the CPU and GPU. We build our framework on top of `PyTorch` (v1.1.0) and `torchvision` (v0.3.0) compiled with Nvidia cuDNN.

Benchmarks. We show the generalisability of our system across different benchmark networks which vary in terms of depth, computational load and architecture. Specifically, we include VGG-16 [26] as a large and computationally intensive network that has conventional single-layer connectivity; ResNet [8] and Inception-v3 [29] as representative mainstream networks from the residual and Inception-based network families, that include non-trivial connectivity via the residual and Inception blocks respectively. We also compare `HAPI` with two hand-optimised networks: the state-of-the-art early-exit network MSDNet [10], and MobileNetV2 [25], a highly-optimised architecture for resource-constrained devices.

Datasets and Training Scheme. We evaluate the effectiveness of our approach on the CIFAR-100 [16] and ImageNet [4] image classification datasets. We use the process described in each model's implementation for data augmentation and preprocessing, such as scaling and cropping the input, stochastic horizontal flipping and channel colour normalisation. In `HAPI`'s early-exit-only training policy (Sec. 4.3), for the initial step of training the main network, we train our own networks for CIFAR-100 using the authors' guidelines for hyperparameter selection. For ImageNet, we use the pretrained networks distributed by `torchvision`, while for MSDNet, we train the ImageNet variant from [10]. To train the early exits in the second step, we continue for an additional 300 and 90 epochs for CIFAR-100 and ImageNet respectively, using the same batch size and an Adam optimiser, with momentum 0.9 and weight decay 10^{-4}.

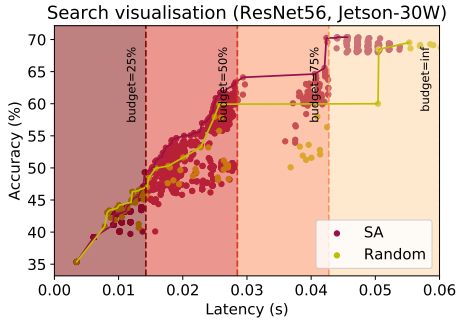


Figure 5: Visualisation of explored design space.

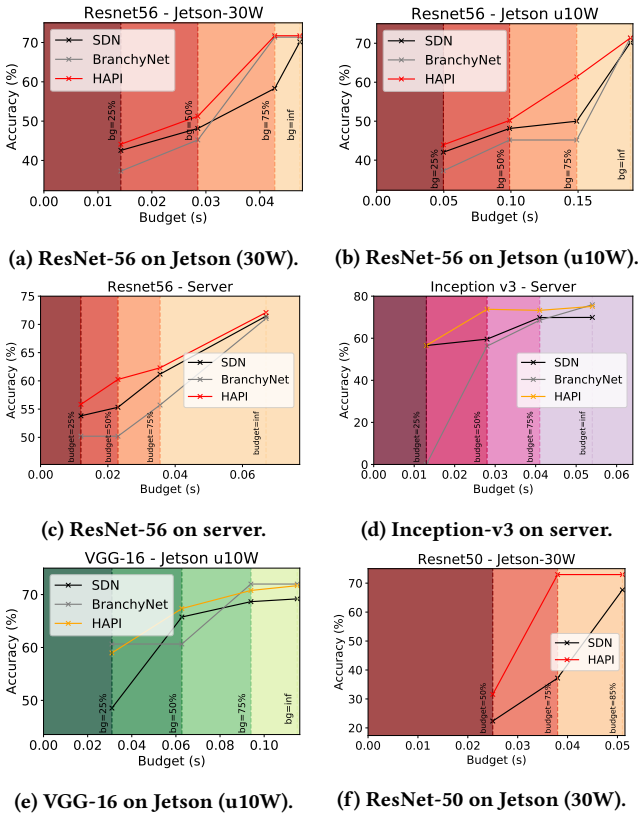
7.2 Evaluation of Proposed Optimiser

To evaluate our DSE, we compare our algorithm with a random search (RS) baseline. Specifically, we compare each exploration of the search space *under the same runtime budget*. We employ ResNet-56 on CIFAR-100 targeting the 30-watt AGX, across four settings by varying the latency SLA. Fig. 5 visualises the points visited by each search, clustered by SLA deadline. Across latency budgets, our SA-based optimiser yields a Pareto front with designs that dominate the RS Pareto points, achieving 3.39 and 10.32 percentage points (pp) higher accuracy under 28- and 42-ms SLAs respectively. We also observe that RS tends to *revisit* already examined designs due to remembering nothing but the best examined design, leading to inefficient utilisation of the available runtime with fewer distinct design points examined. We note that RS has found marginally better designs in the beginning of the 25% SLA due to the small acceptable search space caused by the latter exits becoming infeasible as they violate the tight latency deadline.

7.3 Evaluation against Early-Exit Frameworks

In this section, we evaluate `HAPI` against the state-of-the-art early-exit frameworks, namely BranchyNet and SDN. BranchyNet [31] uses two manually placed early exits and an entropy-based exit policy. We place two early exits at 33% and 66% of FLOPs and perform a sweep over entropy thresholds to tune the value for each experiment. For SDN [12], we place 6 early exits equidistantly with respect to FLOPs and perform a sweep over confidence thresholds to adjust the exit policy for each experiment.

Fig. 6a-6e show the respective early-exit designs under various maximum latency SLAs, represented by the different colour gradients, on CIFAR-100. `HAPI` generates *consistently* more accurate designs for a variety of given latency budgets, when compared to the other strategies not explicitly optimising for the hardware platform or the SLA deadline. Specifically, for ResNet-56 (Fig. 6a-6c),



(a) ResNet-56 on Jetson (30W).

(b) ResNet-56 on Jetson (u10W).

(c) ResNet-56 on server.

(d) Inception-v3 on server.

(e) VGG-16 on Jetson (u10W).

(f) ResNet-50 on Jetson (30W).

Figure 6: Comparison of HAPI with SDN and BranchyNet.

our search yields higher-accuracy designs across devices and budgets, ranging from 0.5 to 6 pp gain over SDN and up to 55 pp over BranchyNet. In particular, this situation manifests when the BranchyNet's first exit (statically positioned at 33% of the network's FLOPs) violates the SLA. We further evaluate HAPI on different architectures, such as Inception-v3 (Fig. 6d) on the server and VGG-16 (Fig. 6e) on the u10W-profile AGX. We observe the same behaviour for BranchyNet in the low-latency SLAs, while HAPI delivers up to 14.2 pp higher accuracy over SDN for a budget of 30 ms.

We showcase HAPI's scalability by selectively training and optimising ResNet-50 on ImageNet (Fig. 6f). HAPI dominates SDN's solutions across budgets on the 30W AGX, with accuracy gains of 4.1-35.7 pp (avg. 16.36 pp). At a 38-ms budget, we observe a significant accuracy improvement of 35.7 pp. This is due to the substantial latency overhead of executing early classifiers on the larger-scale ImageNet. Thus, with HAPI generating a design with fewer exits than SDN's static 6-exit scheme, the CNN can reach deeper layers, without latency violations, and hence achieve higher accuracy.

7.4 Comparison with Hand-Crafted Networks

In this section, the quality of HAPI designs is assessed with respect to two state-of-the-art hand-optimised models: i) the early-exit MSDNet and ii) the lightweight MobileNetV2.

Hand-tuned Early-exit Network. This is investigated on CIFAR-100 by comparing the achieved performance in the accuracy-latency space. Our MSDNet model comprises 10 exits, each positioned after a block. We treat MSDNet as a network pre-populated with all candidate exits and for each latency budget we let HAPI generate the highest-performing subset of exits and the associated c^{thr} value.

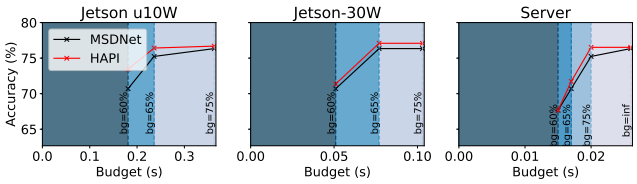


Figure 7: Comparison of HAPI with MSDNet-CIFAR.

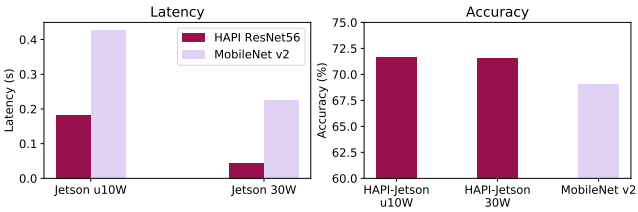


Figure 8: Comparison of HAPI-ResNet with MobileNetV2.

As shown in Fig. 7, our framework is able to sustain the performance of MSDNet across all settings, while achieving higher accuracy under certain cases. On a severely power-constrained device (Fig. 7 left), HAPI yields up to 2.74 pp of accuracy improvement with a latency constraint of less than 200 ms, with an average gain of 1.42 pp across the different latency budgets. In the 30W mode of Jetson AGX (Fig. 7 middle), HAPI achieves up to 0.75 pp under a 80-ms latency constraint. Finally, in the case of the server-grade platform (Fig. 7 right), HAPI yields up to 1.28 pp over MSDNet.

In the case of ImageNet, HAPI selected the fully populated network. This can be attributed to most of the computations of MSDNet for ImageNet being located in the model's backbone. Thus, selecting a subset of exits does not significantly benefit latency, but has a non-negligible impact on accuracy. With respect to deployability, MSDNet's computationally heavy architecture struggles to meet stringent requirements on resource-constrained platforms. On 30W AGX, HAPI's ResNet-56 achieves similar accuracy to MSDNet at 41 ms, yielding 20% speedup over MSDNet's 50 ms. For even tighter constraints, MSDNet does not contain any viable exit.

Hand-tuned Lightweight Network. Although we pose HAPI as an orthogonal, model-agnostic methodology to architecture-specific techniques, we compare with the state-of-the-art lightweight MobileNetV2, taking its end latency as our budget for optimisation. As shown in Fig. 8, HAPI outperforms MobileNetV2 on Jetson with an accuracy gain of 2.53 and 2.45 pp and a speedup of 2.33× and 5.11× under the u10- and 30-watt profiles respectively.

8 CONCLUSION

This paper presents a framework for generating optimised progressive inference networks on heterogeneous hardware. By parametrising early-exit networks in a highly customisable manner, the proposed system tailors the number and placement of early exits together with the exit policy to the user-specified performance requirements and target platform. Evaluation shows that HAPI consistently outperforms all baselines by a significant margin, demonstrating that i) the design choices are critical in the resulting performance and ii) HAPI effectively explores the design space and yields a high-performing early-exit network for the target platform.

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and*

Implementation (OSDI). 265–283.

[2] Mario Almeida, Stefanos Laskaridis, Ilias Leontiadis, Stylianos I. Venieris, and Nicholas D. Lane. 2019. EmBench: Quantifying Performance Variations of Deep Neural Networks Across Modern Commodity Devices. In *International Workshop on Embedded and Mobile Deep Learning (EMDL)*.

[3] Tianqi Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[4] L. Fei-Fei, J. Deng, and K. Li. 2010. ImageNet: Constructing a Large-Scale Image Database. *Journal of Vision* (2010).

[5] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. On Calibration of Modern Neural Networks. In *International Conference on Machine Learning*.

[6] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[7] K. He et al. 2018. Mask R-CNN. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (2018).

[8] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] Kevin Hsieh et al. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[10] Gao Huang et al. 2018. Multi-Scale Dense Networks for Resource Efficient Image Classification. In *International Conference on Learning Representations (ICLR)*.

[11] J. Huang et al. 2017. Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[12] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. 2019. Shallow-Deep Networks: Understanding and Mitigating Network Overthinking. In *International Conference on Machine Learning (ICML)*.

[13] A. Kouris and C. Bouganis. 2018. Learning to Fly by MySelf: A Self-Supervised CNN-Based Approach for Autonomous Navigation. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[14] A. Kouris, S. I. Venieris, and C. Bouganis. 2020. A Throughput-Latency Co-Optimised Cascade of Convolutional Neural Network Classifiers. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1656–1661.

[15] A. Kouris, S. I. Venieris, and C. S. Bouganis. 2018. CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks. In *28th International Conference on Field Programmable Logic and Applications (FPL)*.

[16] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[17] Stefanos Laskaridis, Stylianos I. Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D. Lane. 2020. SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud. In *The 26th Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[18] Edward A Lee and David G Messerschmitt. 1987. Synchronous Data Flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[19] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. 2019. SNIP: Single-Shot Network Pruning based on Connection Sensitivity. In *International Conference on Learning Representations (ICLR)*.

[20] Royson Lee, Stylianos I. Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D. Lane. 2019. MobiSR: Efficient On-Device Super-Resolution Through Heterogeneous Mobile Processors. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[21] Hao Li, Hong Zhang, Xiaojuan Qi, Ruigang Yang, and Gao Huang. 2019. Improved Techniques for Training Adaptive Deep Networks. In *IEEE International Conference on Computer Vision (ICCV)*.

[22] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* (2004).

[23] R Timothy Marler and Jasbir S Arora. 2010. The weighted sum method for multi-objective optimization: new insights. *Structural and multidisciplinary optimization* 41, 6 (2010), 853–862.

[24] Colin R. Reeves (Ed.). 1993. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, Inc.

[25] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[26] K. Simonyan and A. Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR)*.

[27] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[28] V. Sze, Y. Chen, T. Yang, and J. S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. of the IEEE* (2017).

[29] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *AAAI*.

[30] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive Deep Learning Model Selection on Embedded Systems. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 31–43.

[31] Surat Teerapittayanon, Bradley McDanel, and HT Kung. 2016. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks. In *International Conference on Pattern Recognition (ICPR)*.

[32] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[33] S. I. Venieris and C. Bouganis. 2019. fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* 30, 2 (2019), 326–342.

[34] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *CVPR*.

[35] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra. 2019. High-Throughput CNN Inference on Embedded ARM big.LITTLE Multi-Core Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[36] C. Wu et al. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[37] Wencong Xiao et al. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[38] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2246–2251.

[39] Y. Xing, S. Liang, L. Sui, X. Jia, J. Qiu, X. Liu, Y. Wang, Y. Shan, and Y. Wang. 2019. DNNVM: End-to-End Compiler Leveraging Heterogeneous Optimizations on FPGA-based CNN Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[40] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. 2018. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In *European Conference on Computer Vision (ECCV)*.

[41] Sergey Zagoruyko and Nikos Komodakis. 2017. Paying more Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer. In *International Conference on Learning Representations (ICLR)*.

[42] Linfeng Zhang, Jiebo Song, Anni Gao, Jingwei Chen, Chenglong Bao, and Kaisheng Ma. 2019. Be Your Own Teacher: Improve the Performance of Convolutional Neural Networks via Self Distillation. In *IEEE International Conference on Computer Vision (ICCV)*.

[43] Linfeng Zhang, Zhanhong Tan, Jiebo Song, Jingwei Chen, Chenglong Bao, and Kaisheng Ma. 2019. SCAN: A Scalable Neural Networks Framework Towards Compact and Efficient Models. In *Advances in Neural Information Processing Systems (NeurIPS)*.