An online guided tuning approach to run CNN pipelines on edge devices

Pirah Noor Soomro
Chalmers University of Technology
Gothenburg, Sweden
pirah@chalmers.se

Mustafa Abduljabbar
Chalmers University of Technology
Gothenburg, Sweden
musabdu@chalmers.se

Jeronimo Castrillon
Technical University of Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Miquel Pericàs
Chalmers University of Technology
Gothenburg, Sweden
miquelp@chalmers.se

Abstract

Modern edge and mobile devices are equipped with powerful computing resources. These are often organized as heterogeneous multicores, featuring performance-asymmetric core clusters. This raises the question on how to effectively execute the inference pass of convolutional neural networks (CNN) on such devices. Existing CNN implementations on edge devices leverage offline profiling data to determine a better schedule for CNN applications. This approach requires a time consuming phase of generating a performance profile for each type of representative kernel on various core configurations available on the device, coupled with a search space exploration. We propose an online tuning technique which utilizes compile time hints and online profiling data to generate high throughput CNN pipelines. We explore core heterogeneity and compatible core-layer configurations through an online guided search. Unlike exhaustive search, we adopt an evolutionary approach with a guided starting point in order to find the solution. We show that by pruning and navigating through the complex search space using compile time hints, 79% of the tested configurations turn out to be near-optimal candidates for a throughput maximizing pipeline on NVIDIA Jetson TX2 platform.

CCS Concepts

• **Computing methodologies → Parallel computing methodologies**; *Machine learning*.

Keywords

CNN pipelines, Online tuning, Design space exploration, Edge devices, Heterogeneous core clusters, Evolutionary algorithm, Task moldability, Task parallel runtimes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CF '21, May 11–13, 2021, Virtual Conference, Italy
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8404-9/21/05…$15.00
https://doi.org/10.1145/3457388.3458662

ACM Reference Format:
Pirah Noor Soomro, Mustafa Abduljabbar, Jeronimo Castrillon, and Miquel Pericàs. 2021. An online guided tuning approach to run CNN pipelines on edge devices. In *Computing Frontiers Conference (CF '21), May 11–13, 2021, Virtual Conference, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3457388.3458662

1 Introduction

Over the last decade, convolutional neural networks (CNN) have gained attention in many practical applications, such as image classification [1, 2] or natural language processing [3], among others. The training of neural networks is usually performed in the cloud while inference, which is a single forward pass of a neural network, is now often being executed on edge and mobile devices [4]. This is because offloading inference to the cloud often leads to unpredictable delays that are not acceptable for time-stringent IoT/mobile applications. Bringing streaming data analytics closer to the source of data not only reduces latency but also eliminates the communication cost [5, 6].

Modern edge devices are equipped with powerful and energy efficient compute resources which can be used to run CNNs on the device. This improves the real time performance of CNN applications and eliminates risks of communication delays due to poor network status [7]. Widely used DNN (Deep Neural Networks) frameworks such as Tensorflow [8], caffe [9], Torch [10], or Theano [11] are optimized for computing platforms with discrete GPUs coupled with high performance CPU clusters. Compute intensive kernels are optimized for GPUs while data preparation and communication is handled by the cores of the computing platform. On the other hand, the resource constraints of edge devices such as power, memory and compute capability are not directly addressed by existing server-side CNN implementations [12]. The inference performance on core clusters is comparable to GPUs in edge devices, therefore, many vendors prefer to use CPU clusters for inference. In fact, only 11% of Android smartphones contain a GPU which is at least 3x more powerful than the CPU cores [13]. Bringing inference to edge devices, however, comes with a new set of challenges. For instance, there is a wide diversity among SoCs for edge devices and not a single representative SoC architecture can be used to target for generalized optimizations [13, 14]. Many modern edge devices feature a combination of heterogeneous execution units packed on

the same chip, such as cores with different power-performance-area characteristics but share the same Instruction Set Architecture (ISA) [15]. Keeping the energy consumption in focus, some cores are energy efficient but slow, while others are high performance cores but consume more energy. Two examples are the NVIDIA Jetson TX2 [16] and the Apple A14 [17]. The TX2 platform contains a dual-core NVIDIA Denver 2 64-bit CPU and a quad-core ARM A57 cluster.

The common parallelization strategy in the above mentioned frameworks is a layer-wise data parallel implementation of CNNs [18, 19]. Furthermore, certain DNN libraries such as NNPACK [20], QNNPACK [21] and ARM-CL [22] provide CNN operations tailored for CPUs on edge devices but implement the same execution model i.e. layer-wise data parallelism. Since CNNs by nature consist of a sequence of data parallel layers, existing frameworks exploit the data level parallelism and apply optimizations at the level of kernel and/or network model, such as loop fusion, vectorization, compact image and weight representations, channel pruning and quantization among others [5]. However, running data parallel implementations on heterogeneous compute devices is challenging as it requires to perform an asymmetric partitioning that depends on the performance of each core. An alternative option is to run independent frames on different sets of cores, but this has several undesirable features: First, it results in variable latency across frames and, second, frames complete out of order.

A preferable approach to scale CNN inference on streaming data is to use model parallelism [23]. Consecutive CNN layers are grouped into pipeline stages, thus multiple input units can be processed at the same time by different pipeline stages, similar to processing multiple video frames. This approach avoids reordering and keeps inference latency similar across frames. Furthermore, it reduces the total amount of weights that need to be loaded by each core, which makes caches more effective. Frameworks exploiting pipeline parallelism include PipeIt [24], which targets heterogeneous core clusters, and graphi [25], which targets many-core platforms. These frameworks use offline analytical performance models to build efficient pipeline stages since the problem space becomes prohibitively complex with increased number of execution units and number of layers.

There are two limitations to offline solutions based on performance prediction models. Firstly, the performance is modeled using prediction models which only utilize workload and profiled execution time of representative kernels, as in *PipeIt* [24] and *AUGUR* [26]. These models do not take real-time performance degrading factors into account, such as resource contention that occurs when multiple tasks are scheduled to run in parallel. In fact, this discrepancy is already reported by PipeIt [24] when comparing the pipeline configurations picked by the algorithm based on predicted execution time versus actual execution time of the CNN layers. Hence, such offline empirical prediction schemes can lead to choosing sub-optimal configurations, resulting in performance loss. As platforms become increasingly heterogeneous and hierarchical (i.e. more cache levels and NUMA domains), shortcomings of analytical model are only expected to increase. Secondly, performance sampling and throughput maximization need to be repeated whenever the platform configuration is changed, which requires additional efforts. An online approach that relies on real-time performance measurements can potentially eliminate both these limitations. The main challenge of the online approach is the complex design space. To the best of our knowledge, there is no online solution that can effectively prune the design space and quickly converge to a near-optimal solution, while adapting to the performance asymmetry present at runtime.

In this paper, we introduce an online tuning algorithm that uses an evolutionary approach for design space exploration. In evolutionary algorithms, the set of initial search configurations can have a big impact on the quality of the final solution. In our work, we utilize compile-time hints generated from CNN descriptors to accelerate the convergence of the algorithm. Network layer descriptors are an effective source of information for determining the computational intensity of each layer. This approach has also been explored by PipeIt [24], Graphi [25], AUGUR [26] and S. Minakova et al [18]. We leverage this information coupled with online performance profiling to efficiently navigate the complex multidimensional design space in order to find a near-optimal configuration point. Online profiling enables the detection of performance issues such as contention due to memory systems and inter-cluster communication [27]. The goal of this work is to find a near-optimal throughput maximizing pipeline configuration which defines layer distribution for pipeline stages and core assignment to each stage. If more than one execution unit is assigned to a pipeline stage, we utilize inherent data parallelism of the layers.

To demonstrate our approach we introduce a multi-layer solution with language, compiler and runtime support. For programmability, we develop a simple tensor template language to implement CNNs which generates compile time hints. The language is akin to many other tensor languages used in machine learning frameworks. To enable adaptive pipelines for CNNs, we exploit moldable tasks in the state-of-the-art XiTAO task-based runtime [28]. In this setup, the number of processors assigned cannot be changed while the task is in execution, but it can be changed across tasks.

Our evaluation shows that the use of computational hints increases the chances of finding near optimal solutions. For example, provided an exhaustive search results with VGG16 CNN on TX2 platform, 79% of the configurations tested by our ***Pipe-search*** algorithm are good candidates for a near-optimal case and the configuration suggested by *Pipe-search* is close to the one chosen by the exhaustive exploration. We observe that *Pipe-search* converges 70× faster than exhaustive search and 11× faster than random walk. We also demonstrate that the solution explored by *Pipe-search* yields a balanced pipeline for state of the art CNNs. This paper makes the following contributions:

- We propose a tuning algorithm which leverages task moldability and online performance measurement to find an optimal configuration for throughput maximizing CNN pipelines while adapting to performance asymmetries in the underlying computing platform.
- We show how to use seeds to effectively shortcut the exploration in a complex multi-dimensional design space.
- We demonstrate a multi-layer solution, integrating a tensor template language interface to describe the CNN descriptors and generate the seeds, and the XiTAO runtime to provide low-overhead, locality-aware and moldable execution.

The rest of the paper is organized in the following way: Section 2 discusses the essentials of CNNs and pipeline parallelism required for establishing CNN pipelines for a heterogeneous computing platform. Section 3 details the design challenges for CNN pipelines. Section 4 provides details of the proposed approach along with the discussion on proposed online tuning algorithm; *Pipe-search*. Section 5 provides the implementation details of the framework. Section 6 presents the evaluation of the proposed scheme. The related work is discussed in Section 7. The work is concluded in Section 8.

2 Background

To understand the foundation of CNN pipelines we first elaborate the computational breakdown of CNNs in Section 2.1, then we discuss the essentials of modeling CNN pipelines in Section 2.2.

2.1 Convolutional neural networks

The forward pass of CNNs mainly consists of convolutional and fully-connected layers. The most time-consuming part of CNNs is comprised of convolutional layers. There are a set of filters called weights which are learned during the training phase. Weights are convolved across the height, width and depth of the input tensor. The main operation is a dot product between the weight tensor and the local input region, which can be formulated as matrix-multiplication. The computational complexity of convolutional layers is given by equation 1. Here, $[H, W, C]$ refer to height, width and depth of the input tensor, respectively, and $[R, S, K]$ refer to height, width and depth of convolutional kernel, respectively.

$$W_C = H \times W \times C \times R \times S \times K \tag{1}$$

Fully connected layers, in turn, appear towards the end of CNN architectures. Each neuron is connected to all activations of the previous layer. These layers contain a huge number of parameters due to full connectivity resulting into dense computations and high memory usage. Fully connected layers are the second most computationally heavy category of layers in CNNs. The computational complexity is given by equation 2. Here, F refers to the number of output categories.

$$W_{fc} = H \times W \times C \times F \tag{2}$$

Since there are various algorithms for implementing convolution, the parameters we selected for representing computational intensity are generic and have been used also in prior works for modeling computational intensity [18, 24–26]. The intermediate pooling layer is for downsampling the spatial size (height and width) of the forwarded input tensor. There are no learned parameters in the pooling layer, therefore computations are simple. We use input tensor dimension as a computational weight for pooling layer.

2.2 Pipeline parallel implementation of CNNs

The computations in CNNs are orchestrated in the form of layers, where the output of one layer is fed into the following layer. The task graph of a CNN can be represented as a linear task chain where the input of a task is the output of the previous task, thus creating a dependency. CNN inference can be viewed as an application which processes streaming input data on a persistent, chain-like task graph. This task DAG can be split into sub-DAGs making a pipeline stage, where a single node represents a layer.

A pipeline achieves highest efficiency when the execution time of all pipeline stages is balanced, ie. all stages take almost the same time to complete. In addition, the end-to-end latency should be minimized. The latency gap between the pipeline stages is commonly referred to as the *bubble* [29]. The smaller the bubble size, the higher will be the throughput. The performance of the pipeline is defined by the slowest stage, also called the *bottleneck*. In conclusion, a high throughput CNN pipeline is the one in which the bottleneck is smallest possible and the bubble size is also smallest. Hence, the search goal is to find a layer distribution in a pipeline such that it minimizes the latency of the bottleneck.

3 Problem Definition

On platforms like NVIDIA Jeston TX2, there are more than one type of core clusters with different properties in terms of performance and efficiency, a feature commonly called core asymmetry [30]. On such devices, the performance of kernels varies from one type of core cluster to another. Performance asymmetry can also arise due to other reasons. For example, the OS power governor policy impacts DVFS settings, which by itself influences core performance and can introduce asymmetry even on homogeneous platforms. Figure 1 shows the execution time of the slowest stage in a two-stage VGG16 pipeline, when tested with two different governor settings on an NVIDIA Jetson TX2. The governor settings are listed in Table 1, the terminology will be used int the rest of the paper. Ten different configurations have been tested, sorted by most balanced weight assignment to least balanced. The main observation is that the best configuration is different depending on the governor setting. This shows how core performance variations impact the adding or dropping layers among stages. Hence, computational hints are not enough to determine the optimal pipeline, but as we will see, they can be used to achieve a balanced state in a shorter period of time.

Given the CNN layers structure and variable performance of the underlying computing platform, we can formally define the problem of finding a near-optimal pipeline configuration. CNN layers are characterized by the number of computations performed, we refer to it as the weight associated with layers. The initial idea is to divide the layers into pipeline stages such that the number of computations are balanced. Note that the order of layers needs to be preserved because CNNs have a chain-like dependency task-DAG. In an ideal scenario, if there was no performance variation among core clusters, an equal division of layers based on weights would be the best configuration. However, in practice, due to effects such as core asymmetry or resource contention, it is practically difficult to find the best configuration by an analytical method based solely on computational weights.

4 Framework overview

The proposed approach comprises two parts: 1) A Pre-Processing to generate seed heuristics and 2) an Online Tuning (using the generated seeds) followed by pipelining. Figure 2 presents an overview of both modules. The CNN network is implemented using a tensor template language, described in Section 4.1 along with details on designing and launching CNN pipelines as moldable tasks. The next

Table 1: Governor settings of core clusters in NVIDIA Jetson TX2 board. "Performance" refers to highest frequency setting, while "Powersave" refers to the lowest frequency setting

Governor Setting	*A57 Cluster*	*Denver Cluster*
1	Performance	Performance
2	Performance	Powersave
3	Powersave	Performance
4	Powersave	Powersave

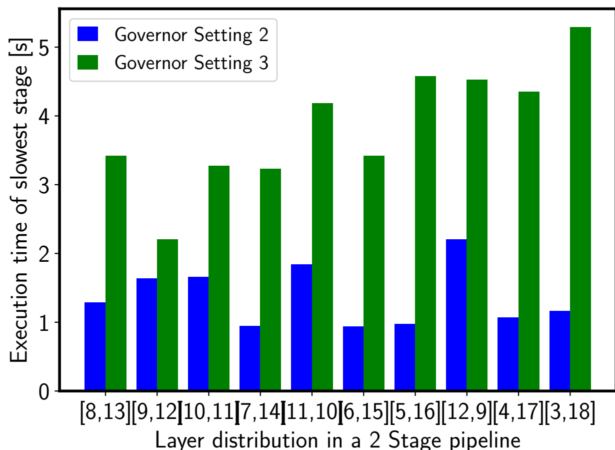

Figure 1: Execution time of the slowest stage with top 10 configurations generated based on computational hints, tested on governor setting 2 and 3. [X,Y] represent X layers assigned to pipeline stage 1 and Y layers assigned to pipeline stage 2.

step consists in generating a search space for the *Pipe-search* algorithm. Section 4.3 describes how the search space is generated and the criteria for selecting configurations. The online tuning phase implements the *Pipe-search* algorithm which is explained in Section 4.4.

4.1 Tensor programming interface for Pipe-search

The computational hints adopted by *Pipe-search* are derived from network layer descriptors. To facilitate programmability, we design a simple tensor template language embedded in C++ [31] that is used to define the CNN layout. This could be added to any other DNN descriptors based on NNEF or ONNX interoperability standards. A sample program is shown in Figure 2. The network descriptors are then analyzed to generate computational hints according to Equations 1 and 2. Figure 2 depicts the conversion of a CNN into a 2-staged pipeline on an arbitrary 4-core cluster. The interface compiles down to a task DAG, where each layer is converted into a moldable task. Using moldable tasks contributes to our goals, as the online search for an optimal pipeline configuration must be able to dynamically map tasks to resources (e.g. cores).

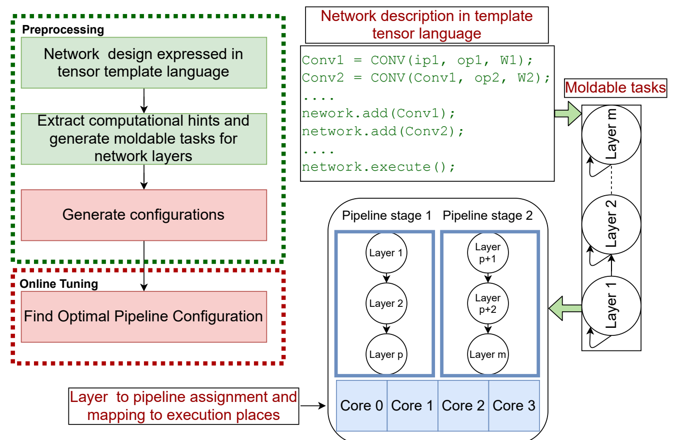

Figure 2: Left: An overview of the proposed approach, Right: Implementation of a CNN pipeline using the tensor template language, a set of moldable tasks, and two pipeline stages executing on a 4-core device.

4.2 Problem formalization

We now formally define the problem addressed by the Pipe-search algorithm. Let L be a set containing the weight corresponding to each layer $L = \{LW_1, LW_1, .., LW_{M-1}, LW_M\}$, where M is the number of layers in the network. Let P_c be a set defining a possible pipeline configuration that groups layers into pipeline stages, i.e. $P_c = \{P_0, P_1, .., P_N\}, N \leq C$, such that C is the number of available cores/threads in the system and a pipeline stage P_n represents number of layers assigned to a stage. Finally, let PS_{count} be the number of stages in a given P_c ($PS_{count} \in \{2, .., C\}$). The objective of Pipe-search is to find P_c that maximizes throughput (layers/s) by minimizing the execution time of the slowest stage within a given P_c.

4.3 Generation of the initial population

Our hypothesis is that an optimal pipeline configuration P_c lies near those with the most balanced weights. However, we do not know in advance which PS_{count} will yield an optimal configuration. For example, for each possible PS_{count}, we may have an arrangement that results in nearly equal weights, but this does not guarantee that all such arrangements will have an optimal solution (e.g. due to core performance asymmetry). Therefore, we initially generate all possible configurations for each PS_{count}.

4.3.1 Selecting candidates for Pipe-search: Trying all possible configurations is impractical since the number of search points increases exponentially with the dimensionality of the search space. Consider a network with M layers, the search space consists of the permutations of all possible P_c and PS_{count} and M. This is shown in equation 4. Since we will not try all configurations during our online tuning phase, we select a subset that is likely to be near the optimal point. To validate our hypothesis on the importance of balancing the weights, we consider the coefficient of variation (CV) [32] as an indicator of the degree of weight balance. Hence, we calculate CV of the weights distribution among the pipeline stages,

given by Eq. 3. Higher CV means higher imbalance of weights among pipeline stages, which may cause computational imbalance among the pipeline stages. We order the configurations based by the CV value. For each PS_{count}, configurations are explored in an increasing order of CV in Pipe-search. The first configuration in the sorted list serves as a seed for a given PS_{count}.

$$CV = \frac{\sigma(W)}{\mu(W)} \quad (3)$$

Where W is the set of weights calculated for each pipeline stage for a given PS_{count} and layer distribution, σ stands for standard deviation and μ stands for average.

The pre-processing step is applied once for each CNN architecture for a specific set of PS_{count}. Algorithm 1 lists the steps of generating initial population. For each pipeline stage count, a set of layer distribution is generated using Eq. 4.

$$\begin{aligned} &L = \{1, 2, 3, ...M\}, P = \{2, 3, ...C\} \\ &S_c = \forall p \in P,\ C(L, p),\ \text{if} \sum C(L, p) = M \\ &S = \forall s \in S_c, P(s, s_{max}) \\ &\text{where, } C = \text{Combinations } \textit{and } P = \text{Permutations} \end{aligned} \quad (4)$$

4.4 *Pipe-search* Algorithm

Pipe-search requires three inputs: the sorted configurations (S), the maximum number of stages (C) and a tunable α parameter, which serves as an upper limit for the number of points to explore around a particular configuration.

To explain the algorithm we consider the following example. We want to run a network that consists of 7 layers with a weight distribution of $W = \{1, 4, 8, 4, 8, 8, 4\}$ (normalized to the smallest value) on a 4-core platform. Possible values for PS_{count} are $\{2, 3, 4\}$. *Pipe-search* has two phases of exploration. The first phase (Lines 2 - 14) tests at least α search points around the seeds for each value of PS_{count}. The global minimum, which is the configuration that minimizes the execution time of the slowest stage, is updated and saved in S_{min} during exploration. For each PS_{count} value we test the top α configurations from S (Lines 3 - 13). If any configuration results in a better performance than S_{min}, the global minimum is updated and the confidence variable γ is reset (Lines 9 - 10). The purpose of α is to limit the exploration around the found minimum. Note that S_{min} is initialized to the seed that yields the best performance among all seeds. Although the exploration phase is limited by α, the number of search points per PS_{count} can vary if the global minimum is updated. After phase one, the algorithm is able to find the PS_{count} value around which the optimal solution lies. At this stage, Pipe-search has managed to reduce the dimensionality of the search space by 1. The second phase (depicted by Lines 16 - 28) explores the configurations that have the number of stages (PS_{count}) that achieves the best performance during the start-up phase. The extent of exploration is still controlled by α, which is the accepted limit at which the algorithm ceases to attempt further search points after a new minimum is found. In the best case, one of the seeds could be the optimal solution. Hence, the total number of trials is a function of α and the size of C (Eq. 5).

$$trials = \alpha(PS_{countMax} + 1) \quad (5)$$

Algorithm 1 Generate Configurations

Require: L, C
1: **for** PS_{count} in $[2..C]$ **do**
2: $p_c \leftarrow$ layer_distributions(PS_{count}, L)
3: **for** c in p_c **do**
4: $CV \leftarrow$ calculate_CV(c)
5: **end for**
6: $(p_c, CV) \leftarrow$ sort(CV)
7: $S[p] \leftarrow p_c$, add first k samples in Samples, about dimension
8: **end for**
9: **return** S , initial population

Algorithm 2 Pipe-search

Require: S, C, α
1: $S_{min} = S[0]$, seed which yielded minimum execution time for slowest stage.
2: **for** PS_{count} in $[2..C]$ **do**
3: $p \leftarrow PS_{count}$
4: $c \leftarrow 0$
5: **while** $\gamma < \alpha$ **do**
6: $t \leftarrow execute(S[p][c])$, execute network using configuration s
7: $T_s \leftarrow max(t)$, Time of slowest stage corresponding to configuration $S[p][c]$
8: **if** $T_s > T_s[S_{min}]$ **then**
9: $\gamma + +$
10: **else**
11: $S_{min} \leftarrow S[p][c]$, found a new minimum
12: $\gamma \leftarrow 0$
13: **end if**
14: $visited[p] \leftarrow c + +$;
15: **end while**
16: **end for**
17: $p \leftarrow PS_{count}(S_{min})$
18: $c \leftarrow visited[p]$
19: $\gamma \leftarrow 0$
20: **while** $\gamma < \alpha$ **do**
21: $t \leftarrow execute(S[p][c])$
22: $T_s \leftarrow max(t)$
23: **if** $T_s > T_s[[S_{min}]$ **then**
24: $\gamma + +$
25: **else**
26: $S_{min} \leftarrow S[p][c]$, found a new minimum
27: $\gamma \leftarrow 0$
28: **end if**
29: $c + +$
30: **end while**
31: **return** S_{min}

5 Implementation

This section describes how Pipe-search can be implemented on a task parallel runtime by using the XiTAO runtime [33] as a case study. We then conclude with a description of the experimental setup for the evaluation of the proposed scheme.

Table 2: Description of symbols used in *Pipe-search* algorithm

Symbols	*Description*
L	Weights per layer, derived from computational hints.
PS_{count}	pipeline stage count
C	maximum number of pipeline stages
CV	Co-efficient of variation of weights distribution of a given pipeline configuration. Calculated by 3
S	A data structure that contains all configuration sorted by the corresponding CV value.
T_s	Execution time of the slowest stage
S_{min}	pipeline configuration with least T_s
α	The confidence on found S_{min}, this parameter is tunable

5.1 Moldable pipelines using XiTAO

XiTAO [33] is a runtime for executing mixed-mode computations in which the individual tasks of a task-DAG are themselves parallel computations. These parallel computations are usually data-parallel, but any sort of parallel structure is possible. XiTAO supports the aforementioned task moldability, that is, the ability to assign n tasks to m resources (n-to-m mapping). Such decision can be made dynamically. This includes the choice of the task width, which is the number of resources (e.g. cores or threads) to assign to a certain task, and the location (place) where to execute the task [28]. Thus, dynamically selecting the task's location and width facilitates the online tuning of pipeline stage configurations. To accomplish this, each layer is encapsulated into a task. We further define dependencies between XiTAO tasks to create pipeline stages. Hence, a single pipeline stage consists of a task-DAG, in which all layers(tasks) share the same location and width. While handling the dependencies across the stages, the runtime executes multiple task-DAGs in parallel (one DAG per pipeline stage). This enables pipeline parallel execution, (Figure 2 also depicts the XiTAO task-DAGS for two staged pipeline). The task-DAGs are adjusted according to given pipeline configuration during online tuning phase. Note that we do not execute layers in pipeline fashion during tuning phase, the pipeline is launched once a configuration is selected by the *Pipe-search* algorithm.

5.2 Testbed

For evaluation, we use an NVIDIA Jetson TX2 development board, featuring a dual-core NVIDIA Denver 2 64-bit CPU, a quad-core ARM A57 Complex (each with 2 MB L2 cache) and an NVIDIA Pascal Architecture GPU with 256 CUDA cores. Both the Denver 2 and the A57 cores implement the ARMv8 64-bit instruction set and are cache coherent. For the purpose of this work, we consider only the two ARMv8 cores, and leave GPU scheduling as future work.

5.3 Benchmarks

This work is mainly focused on inference pass of CNN networks. Our framework does not use any neural network library, instead, we implemented our own library which is compatible with underlying XiTAO runtime. To evaluate the contribution of this work, we ported both, widely used CNNs and synthetic neural networks. Among widely used networks, we implemented VGG16 [34], AlexNet [35] and ResNet50 [36]. VGG16 is composed of 21 layers, out which 16 are compute intensive. AlexNet is composed of 11 layers, out which 8 are compute intensive. ResNet50 is comprised of 52 layers, out of which 50 are compute-intensive. We designed synthetic networks which not only represent usual CNNs but consists of interesting weight distributions particularly to stress test the capabilities of the *Pipe-search* algorithm. The synthetic networks are further discussed in section 6.3.

6 Experiments

This section evaluates the impact of the different components that constitue *Pipe-search*. We start by studying the convergence speed and the quality of the explored configurations in Section 6.1. Section 6.2 evaluates the importance of using computational hints in *Pipe-search*. Section 6.3 demonstrates the capability of *Pipe-search* in adapting to various levels of core heterogeneity while searching for a balanced pipeline configuration. It calibrates the capability of *Pipe-search* in the situation when optimal configuration is farther away from the seeds. we study the overall impact of using the online tuning in *Pipe-search* versus using only the pre-processed seeds (offline), in the presence of performance asymmetry due to cluster-level DVFS settings.

6.1 Quality of solution and convergence of *Pipe-search*

The search space for a CNN with M layers on a platform with $\{2, 3, .., C-1, C\}$ possible PS_{count} values is represented by Equation 4. The size and dimension of the search space grow exponentially with increased number of layers and PS_{count}. We, therefore, design an experiment for a rather small search space to compare exhaustive search and *Pipe-search* algorithm. This is done to understand the convergence and quality of the solutions found by *Pipe-search*. We use 4 cores from our testing platform to run VGG16 with $L_{max} = 21$ and $P = \{2, 3, 4\}$ under governor setting 1 (Table 1). The exhaustive search algorithm prunes 1970 pipeline configurations compared to 34 in the case of *Pipe-search*. The results from *Pipe-search* are reported in Table 3. Only 2% of the total search space points are visited by *Pipe-search*. For the sake of higher expectancy of finding an optimal configuration, we set $\alpha = 10$. *Pipe-search* successfully found the best configuration in much less number of trials. We further investigate the quality of pipeline configurations tested by *Pipe-search*. Table 4 shows that 79% configurations lie in the best range (1s - 1.5s) of high throughput pipeline configuration. We further observe that non of the trials from *Pipe-search* lie in the lowest throughput region visited by the exhaustive search (2.0s - 5.0s). *Pipe-search* favors the high-throughput configurations during the search because it prioritizes those with the least CV values. This speeds up convergence to an optimal solution and reduces the number of steps of the search to a factor of Equation 5, which is a 70x reduction in convergence time in this case.

6.2 Impact of using computational hints

Pipe-search traverses the possible pipeline stage lengths. For example, if $PS_{count} \in \{2, 3, 4\}$, then the different configurations

Table 3: Comparison between exhaustive search and *Pipe-search* using Vgg16 on NVIDIA Jetson TX2

Algorithm	*Trials*	*Opt. Conf.*	*Seed*
Pipe-search	38	[7,4,10]	[6,5,10]
Exhaustive Search	1970	[7,4,10]	N/A

Table 4: Distribution of all pipeline configuration based on throughput for VGG16 on 4 cores.

Algorithm	*1.0 - 1.5 (sec)*	*1.5 - 2.0 (sec)*	*2.0 - 5.0 (sec)*
Pipe-search	79%	21%	**0%**
Exhaustive Search	11%	18.3%	70.7%

Table 5: Tuning time and throughput(frames/sec) of pipe search with and without hints, compared to Random search.

	With hints	*Without hints*	*Random*
Throughput [f/s]	1.2	0.6	0.9
Training time [s]	0.1	0.1	1.1

pertaining to each PS_{count} are explored starting from the respective seeds. The seeds are calculated based on the best CV value of weights for each configuration with PS_{count} stages, and are provided as input to *Pipe-search* algorithm. We investigate the impact of using computational hints by executing the algorithm with and without the knowledge of weight based seeds. In random walk, we set the stopping condition to a throughput value (0.9 frames/s) to reduce search time. Also, in *Pipe-search* (no hints), we just balance the number of layers per pipeline stage. Results are shown in Table 5. The training time in both *Pipe-search* variants is 90% less than random walk. However, we observe that the throughput of the resulting pipeline configuration with *Pipe-search* is 50% better than the version with no computational hints.

6.3 Impact of performance asymmetry on the solution and convergence

Our testing platform can be configured in four different governor settings which can determine the performance of the two clusters. The configurations are listed in Table 1. Note that the cases with clusters on different frequency levels are the most performance asymmetric. Hence, each setting exhibits a different level of heterogeneity in the platform. This means that a pipeline configuration that is effective in one governor setting cannot be as effective in another. Table 6 shows the optimal configurations reached by *Pipe-search* for the VGG16 network under different governor settings. We observe that not only does asymmetry affect layer partitioning, but it also impacts the PS_{count}. This shows that *Pipe-search* tends to adapt to the heterogeneity while finding the optimal configuration. Section 6.4 discusses the quality of the found optimal in more detail.

Now we compare the number of convergence steps (rank) and the solutions in the case of symmetric and asymmetric governor settings (1 and 3) using synthetic networks. The synthetic networks have 1 or 2 perfect seeds ($CV \approx 0$) with different stage counts. The weight distribution of these synthetic networks are listed in Table 7.

Table 6: VGG-16 executed with different governor settings

Governor Setting	*Opt. Conf*	*Ranks*	*Throughput*
1	[6,5,10]	1	1.22 Frames/s
2	[5,16]	7	0.36 Frames/s
3	[9,12]	2	0.21 Frames/s
4	[7,4,10]	3	0.17 Frames/s

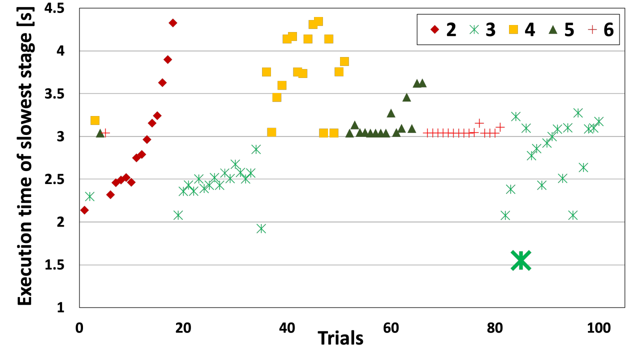

Figure 3: *Pipe-search* exploration timeline for Synth2 under governor setting 3

Note that we consider only convolutional layers for these networks in order to be the representative of state of the art CNNs. We observe that in different governor settings, different optimal configurations were selected away from seed, and in some cases, with different stage counts (PS_{count}) from the perfect seeds. The rank represents the location of the solution in the search space sorted by CV values. In the case of governor setting 3, there is high asymmetry in the core performance so a configuration with higher rank is selected, which means that, unlike the symmetric case, a higher CV value can be selected in such cases. Therefore, *Pipe-search* adapts to both low and high performance asymmetry. Figure 3 shows the exploration timeline of *Pipe-search* for "Synth 2" under governor setting 3. Each point represents execution time of slowest stage achieved by the configuration tested by *Pipe-search*. We observe that the optimal configuration lies in $PS_{count} = 3$. The algorithm walks through all PS_{count}, and later on focuses the search on $PS_{count} = 3$ region. The exploration seizes after reaching a confidence limit of $\alpha = 15$.

6.4 *Pipe-search* on common CNN networks

In this study, we aim at showing the effectiveness of adopting the dynamic online approach (using *Pipe-search*) compared to only using computational hints. To investigate this, we execute ResNet, AlexNet and VGG16 under governor setting 3 as this entails the highest level of performance asymmetry between the core clusters of TX2. We use 2 Denver cores on "highperformance" and 2 A57 cores on "powersave" mode (at lowest frequency). *Pipe-search* concludes that a pipeline of two stages would yield higher throughput based on the online search. This is because of the fact that for $PS_{count} = 3$ or 4, the pipeline stages will be mapped across the core cluster which causes performance degradation due to inter-cluster communication overhead. Figure 4 shows the execution time of two pipeline stages when tested using seeds from preprocessing stage

Table 7: *Pipe-search* exploration for synthetic networks

Network Design					*Governor setting 1*			*Governor setting 2*		
Networks	*layers*	*Weight distributions*	*Best seed*	*CV*	*Optimal*	*Rank*	*CV*	*Optimal*	*Rank*	*CV*
Synth 1	7	{1,4,8,4,8,8,4}	[3,2,2]	3.8	[4,1,1,1]	6	51.4	[5,1,1]	10	73.8
Synth 2	15	{1,9,4,8,5,4,8,5,7,1,1,1,4,8,22}	[8,7] or [4,4,6,1]	0	[4,7,4]	8	18.5	[8,6,1]	22	35
Synth 3	13	{1,9,4,8,20,2,22,3,4,8,7,11,11}	[4,2,1,4,2]	0	[3,1,2,1,4,2]	9	29.8	[7,4,2]	22	56.5

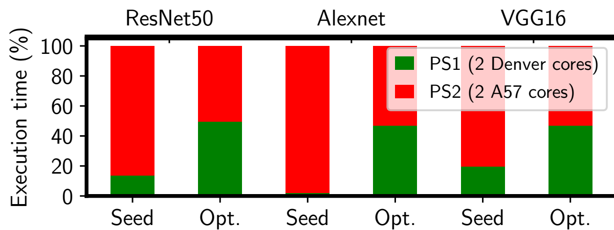

Figure 4: Percentage execution time of CNN pipelines with seeds and optimal configurations

compared to the optimal configuration. It is evident form the results that the offline approach produces unbalanced configurations especially during high performance asymmetry but with exploration, *Pipe-search* is able to find a configuration which balances out the execution time for two pipeline stages. In these experiments, we have used α in the range of $[3, 15]$.

7 Related Work

Common neural network frameworks such as TenserFlow [8], Caffe [9] and ARMCL [22] provide efficient implementation of CNNs by leveraging, for example, optimized GEMMs and fused and vectorized MAC (Multiply and Accumulate) operations, among others. These implementations do not provide platform specific optimizations, especially when edge devices are considered as computing platforms for CNN inference. It is shown in literature that the most suitable implementation for CNN inference on edge devices is a task-based parallel implementation [18, 24]. Therefore, developers need to extend the existing frameworks to enable task level parallelism to exploit the benefits of the heterogeneous computing environments present in edge devices. Efforts have been made for CNN inference on edge devices as well. Minakova et al. [18] convert CNN models into Synchronous DataFlow (SDF) model to represent computational and communication cost of CNN layers. Annotated SDF models are then used by a genetic algorithm to find a mapping of tasks on embedded CPUs and GPUs. The main focus of their work is to balance the workload among the cores and GPUs in the embedded system utilizing task and data level parallelism. Their approach suggests to assign the heaviest SDF node to the core that accompanies the GPU so that dense layers can exploit data level parallelism while assigning the rest of the nodes to the remaining cores in order to balance the workload. The heterogeneity of embedded CPUs is not explicitly highlighted in this approach. Additionally, the SDF to core mapping is agnostic to dynamic system changes (e.g. DVFS). To construct a balanced pipeline targeting heterogeneous computing platforms, we require the performance estimation of each type of core. Two closely related works that propose a prediction model to provide an estimation of CNN performance on a given architecture are AUGUR [26] and Pipe-It [24]. AUGUR is a tool that provides performance prediction of CNNs on CPUs and GPUs using CNN layer descriptors. Pipe-It also utilizes CNN layer descriptors and a regression model to approximate the performance of different types of cores in an embedded device. The prediction model in [24] is an enhanced version of AUGUR's prediction model [26], which greatly reduces the prediction error. The average prediction error reported by the authors is 13% on big cores and 11% on little cores in a big.LITTLE architecture. Since the prediction error leads to a throughput degradation in a pipeline, it is desirable to eliminate the effects caused by prediction error. This shows that scheduling decisions taken from prediction models may lead to performance degradation, therefore, we propose an online tuning approach that can reduce the chances of choosing the wrong layer to pipeline stage distribution.

8 Conclusion

This paper presents a novel online tuning approach for throughput maximizing CNN inference pipelines that adapts to performance asymmetry in core clusters. We leverage compile-time hints to generate seeds for faster design space exploration via our novel evolutionary algorithm called *Pipe-search*. We evaluate *Pipe-search* on a set of three state of the art CNNs and three synthetic CNNs. Our results show that our approach effectively prunes the design space, and that guided navigation results in faster convergence making it a feasible approach for processing streaming data on edge devices.

Acknowledgments

The authors would like to thank Norman Alexander Rink for his support and assistance. We also thank the reviewers for their valuable comments and feedback. This work has received funding from the European Union Horizon 2020 research and innovation programme under LEGaTO with grant agreement No. 780681 (https://legato-project.eu/), and Eurolab4HPC with grant agreement No. 800962 (https://www.eurolab4hpc.eu/). Some of the computations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at Chalmers Centre for Computational Science and Engineering (C3SE) partially funded by Swedish Research Council 2018-05973 (https://www.vr.se/).

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[3] Marc Moreno Lopez and Jugal Kalita. Deep learning applied to nlp. *arXiv preprint arXiv:1703.03091*, 2017.
[4] Nicholas D Lane and Petko Georgiev. Can deep learning revolutionize mobile sensing? In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 117–122, 2015.
[5] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960, 2018.
[6] Ziheng Jiang, Tianqi Chen, and Mu Li. Efficient deep learning inference on edge devices. *ACM SysML*, 2018.
[7] Xiaofei Wang, Yiwen Han, Victor CM Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(2):869–904, 2020.
[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015.
[9] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.
[10] Torch. http://torch.ch. Accessed: 2021-01-20.
[11] Theano. http://deeplearning.net/software/theano/. Accessed: 2021-01-20.
[12] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.
[13] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.
[14] Cao Gao, Anthony Gutierrez, Madhav Rajan, Ronald G Dreslinski, Trevor Mudge, and Carole-Jean Wu. A study of mobile device utilization. In *2015 ieee international symposium on performance analysis of systems and software (ispass)*, pages 225–234. IEEE, 2015.
[15] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 81–92. IEEE, 2003.
[16] Nvidia jetson tx2. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/. Accessed: 2021-01-20.
[17] The apple a14 soc: Firestorm & icestorm. https://www.anandtech.com/show/16192/the-iphone-12-review/2. Accessed: 2021-03-24.
[18] Svetlana Minakova, Erqian Tang, and Todor Stefanov. Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In *International Conference on Embedded Computer Systems*, pages 18–35. Springer, 2020.
[19] Aniruddha Parvat, Jai Chavan, Siddhesh Kadam, Souradeep Dev, and Vidhi Pathak. A survey of deep-learning frameworks. In *2017 International Conference on Inventive Systems and Control (ICISC)*, pages 1–7. IEEE, 2017.
[20] M Dukhan. Acceleration package for neural networks on multi-core cpus: Maratyszcza. *NNPACK, Oct*, 2018.
[21] Marat Dukhan, Yiming Wu, and Hao Lu. Qnnpack: open source library for optimized mobile deep learning, 2018.
[22] Compute library: A software library for computer vision and machine learning. https://developer.arm.com/ip-products/processors/machine-learning/compute-library. Accessed: 2021-01-20.
[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.
[24] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. High-throughput cnn inference on embedded arm big. little multi-core processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
[25] Linpeng Tang, Yida Wang, Theodore L Willke, and Kai Li. Scheduling computation graphs of deep learning models on manycore cpus. *arXiv preprint arXiv:1807.09667*, 2018.
[26] Zongqing Lu, Swati Rallapalli, Kevin Chan, and Thomas La Porta. Modeling the resource requirements of convolutional neural networks on mobile devices. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1663–1671, 2017.
[27] David Skinner and William Kramer. Understanding the causes of performance variability in hpc workloads. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 137–149. IEEE, 2005.
[28] Miquel Pericàs. Elastic places: An adaptive resource manager for scalable and portable performance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(2):1–26, 2018.
[29] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*, 2018.
[30] S. Balakrishnan, Ravi Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 506–517, 2005.
[31] Norman A Rink and Jeronimo Castrillon. Teil: a type-safe imperative tensor intermediate language. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 57–68, 2019.
[32] Charles E Brown. Coefficient of variation. In *Applied multivariate statistics in geohydrology and related sciences*, pages 155–157. Springer, 1998.
[33] Xitao. https://github.com/CHART-Team/xitao. Accessed: 2021-02-02.
[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.
[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.