

Shisha: Online scheduling of CNN pipelines on heterogeneous architectures

Pirah Noor Soomro¹, Mustafa Abduljabbar²,
Jeronimo Castrillon³, and Miquel Pericàs¹

¹ Dept Computer Science and Engineering, Chalmers University of Technology
{pirah,miquelp}@chalmers.se

² Ohio State University abduljabbar.1@osu.edu

³ Chair for Compiler Construction, Technische Universität Dresden
jeronimo.castrillon@tu-dresden.de

Abstract. Many modern multicore processors integrate asymmetric core clusters. With the trend towards Multi-Chip-Modules (MCMs) and interposer-based packaging technologies, platforms will feature heterogeneity at the level of cores, memory subsystem and the interconnect. Due to their potential high memory throughput and energy efficient core modules, these platforms are prominent targets for emerging machine learning applications, such as Convolutional Neural Networks (CNNs). To exploit and adapt to the diversity of modern heterogeneous chips, CNNs need to be quickly optimized in terms of scheduling and workload distribution among computing resources. To address this we propose **Shisha**, an online approach to generate and schedule parallel CNN pipelines on heterogeneous MCM-based architectures. **Shisha** targets heterogeneity in compute performance and memory bandwidth and tunes the pipeline schedule through a fast online exploration technique. We compare **Shisha** with Simulated Annealing, Hill Climbing and Pipe-Search. On average, the convergence time is improved by $\sim 35\times$ in **Shisha** compared to other exploration algorithms. Despite the quick exploration, **Shisha**'s solution is often better than that of other heuristic exploration algorithms.

Keywords: CNN parallel pipelines · Online tuning · Design space exploration · Processing on heterogeneous computing units · Processing on chiplets

1 Introduction

Multicore processors are becoming more and more heterogeneous. Intel's Meteor Lake [2] features asymmetric multicore design containing high performance and power saving cores. Similarly, Apple's A14 Bionic [1] integrates high performance cores called Firestorm and power saving cores called Icestorm. The trend towards heterogeneity is complemented with the trend towards Multi-Chip-Module (MCM) integration, which enables lower cost during design and improves yield by reducing chip area (chiplets) [13]. When combined with interposer-based packaging technology, it enables lower latency and high bandwidth transmission to memory devices such as High Bandwidth Memory (HBM) [8]. Chip manufacturers are adopting a mix of these technologies in order to design high performance

processors, resulting in heterogeneity at the level of the cores, memory subsystem and the Network on Chip (NoC). In order to effectively exploit such architectures, applications must be optimized considering the impact of different levels of heterogeneity. Furthermore, to address the diversity of hardware platforms, the optimization process must be fast and preferably online.

Convolutional Neural Networks (CNNs) have high computational, bandwidth and memory capacity requirements owing to the large amount of weights and the increasing size of intermediate results that need to be transferred between layers. Parallel pipelining has the potential to address these requirements by partitioning the whole network across devices, and requiring only the inputs to be exchanged among stages. In chiplet architectures, CNNs could be efficiently pipelined by distributing layers across chiplets so as to reduce the amount of weights that need to be copied. Furthermore, pipelining makes the task of load balancing manageable among heterogeneous computing units.

In order to partition and schedule pipelines, current approaches rely on designing cost models to steer design space exploration [3, 5]. For instance, the auto-scheduler in [3] explores over ten thousand schedules for a single CNN-layer pipeline using Halide [22]. The effectiveness of these approaches depends on the accuracy of the cost model and the scalability of the exploration algorithm. Sophisticated cost models, some of them using ML-models themselves, have been proposed and used in [3, 4, 14, 18, 32–34]. These models, however, require extensive training for near-optimal solutions [5], are sensitive to changes in the execution environment (e.g., DVFS) and architectural parameters, need in-depth architectural knowledge for model updates, and do not consider the impact of heterogeneous multicore or chiplet architectures. As heterogeneity at different levels of processing (e.g. core performance, memory bandwidth and/or MCM organization) is expected to increase in future HPC platforms, static pipeline partitioning and scheduling become inflexible. Online auto-tuning of the pipeline schedule would help to ensure performance portability to future architectures. However, to make it practical, it is critical that online pipeline partitioning and scheduling finds an acceptable configuration with low overhead.

Pipe-Search [29] adopts an online exploration approach for finding a pipeline configuration. It generates a database of pipeline configurations which is space-intensive and prohibitively slow for larger systems and deeper CNNs. In this paper, we propose a quick method to determine a meaningful starting point, or seed, for the exploration coupled with a simple navigation heuristic for efficient runtime auto-tuning. In **Shisha**, we leverage statically available information from the CNN and from the target platform to reduce the number of exploration points and find a near-optimal solution within reasonable time. A *configuration* explored by **Shisha** suggests grouping CNN layers into pipeline stages and mapping of pipeline stages onto available sets of processing units referred to as *Execution Places (EPs)*. When generating initial configurations, **Shisha** aims at balancing the load among pipeline stages while considering the allocation of stages to EPs. **Shisha** improves upon related work in two ways:

- **Shisha** achieves faster convergence by introducing two novel schemes: (i) the seed generation and (ii) the online tuning. We demonstrate that **Shisha** is able to converge faster than existing algorithms (Simulated Annealing, Hill Climbing and Pipe-Search) and that it is able to find a solution within practical time limits.
- We show that **Shisha** scales better with deeper CNNs and with larger amount of EPs per processing unit which is one of the limitations of prior online tuning approaches such as Pipe-Search [29].

Shisha maps pipeline stages to EPs, which could be of any type and number of processing units, such as multicores or manycores. To measure the quality of schedules explored by **Shisha** we compare our results to conventional search exploration algorithms such as Simulated Annealing (also used by TVM [34]), Hill Climbing, Exhaustive Search and Random Walk (executed for a longer period of time), and to Pipe-search, an earlier online tuning approach. We test **Shisha** on state of the art CNNs such as ResNet50 [11] and YOLOv3 [24]. The results show that, despite exploring only a tiny portion of the design space ($\sim 0.1\%$ of design space for ResNet50 and YOLOv3), **Shisha** finds a solution that is equivalent to exhaustive search. Moreover, due to the guided exploration, the convergence time is improved by $\sim 35\times$ in **Shisha** compared to the other representative exploration algorithms.

2 Motivation and problem definition

In a computing platform with different types of memories, the assignment of workload and data objects becomes crucial for better performance. To investigate the impact of different thread and data assignment strategies, we tested the STREAM Triad benchmark [16] with two data sizes, 19 GB and 31 GB on Intel’s Knights Landing (KNL) [28]. KNL has two types of memories, 16 GB of high bandwidth memory (HBM), also called MCDRAM, and 90 GB of DDR4 DRAM. The bandwidth of HBM is $4\times$ higher than that of DRAM [26]. This suggests that most of the application data should be placed in HBM. It also means that HBM should be able to handle more parallelism until the bandwidth is saturated. For each data size, 15 GB of data are placed in MCDRAM and the remainder of the data are placed in DRAM. In Figure 1, we show three cases, namely, 1) when all data are placed in DRAM (DDR only), 2) when MCDRAM is used as a cache (cache mode), and 3) when data is distributed across the two memories. As can be seen, with a sensible thread assignment, the case 3 yields the best performance. This shows that a clever data partitioning and thread assignment are key to achieve high performance in the presence of memory heterogeneity. Further analyzing case 3, Figure 2 shows the heatmap of the execution time of STREAM Triad with different thread assignments to MCDRAM [16, 32, 64, 128] and DRAM [2, 4, 8, 16]. The optimal number of threads is determined by a) the memory bandwidth of each memory type, b) the additional bandwidth consumed by each extra thread, and c) the amount of data to be processed. Results from the experiment show that for each data partitioning between HBM and DRAM there is a different optimal thread partitioning. An important observation from

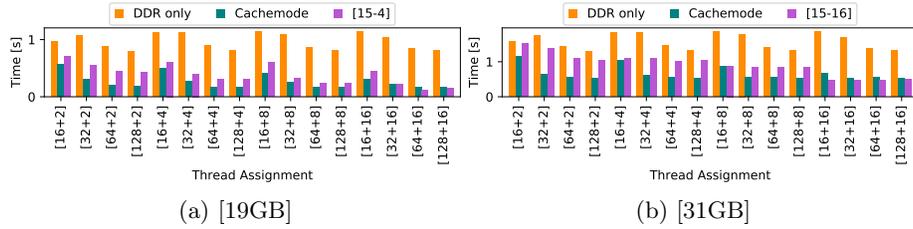


Fig. 1: Comparison of cases 1,2 and 3. X-axis $[X+Y]$ shows X = threads assigned to MCDRAM and Y = threads assigned to DRAM

Figures 2 is that better performance can be achieved by assigning fewer number of threads per memory type, rather than opting for assigning maximum number of threads.

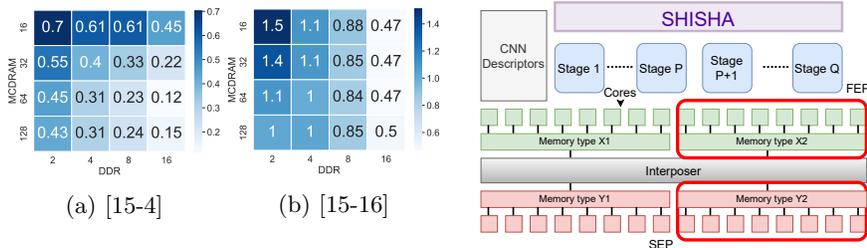


Fig. 2: (a) & (b) Execution time [s] of STREAM Triad with data distribution $[X-Y]$, where X = GBs placed in MCDRAM and Y = GBs placed in DRAM

Fig. 3: System targeted in this paper. Memory type X and Y represent different memory bandwidths.

Problem definition and general approach of the solution:

This work considers a computing platform which is composed of a set of nodes consisting of high performance cores attached to a high-bandwidth memory (referred to as Fast Execution Place – FEP) and clusters of relatively slower cores attached to a low-bandwidth memory (referred to as Slow Execution Place – SEP). This MCM based scenario is expected for chiplet architectures with heterogeneous integration and is shown in Figure 3. Our goal is to run throughput maximizing CNN inference pipelines on such an architecture.

3 Background and related work

There are various schemes for parallelizing CNNs. In data parallelism the work of a minibatch (a set of inputs) is partitioned among multiple computational resources. In model parallelism the work is divided according to neurons in each layer which corresponds to the tensor dimensions in each layer. In layer pipelining [6] the work is partitioned by distributing network layers among computational resources. Model parallelism within the layer is combined with layer pipelining by arranging computational resources into multiple teams of workers.

This hybrid parallelism has following benefits: 1) there is no need to replicate weight and input tensors on all devices, 2) the communication volume and points are reduced, and 3) the weights can remain cached, thus decreasing memory round-trips. In the rest of the paper we will refer to CNN pipelines in which network layers are grouped into pipeline stages. Each pipeline stage is assigned a unique set of computational resources, referred to as EPs.

Finding out the right schedule and mapping of CNN pipelines on mentioned architectures is a design space exploration problem, where we are interested in the configuration that achieves the highest throughput. The configuration consists of the number of pipeline stages, CNN layers per pipeline stage and a mapping of pipeline stages to EPs. In the literature, various meta heuristic and machine learning algorithms have been used such as Simulated Annealing [34], evolutionary algorithms [3, 29], reinforcement learning [4, 21] and deep neural network techniques [5]. The design space under consideration is large and complex, requiring tens of thousands of trials in order to reach a near optimum with current search schemes. Exploring in such a complex space is NP-hard. Parallel pipelines for CNN training have been applied in practice [9, 12, 19, 20]. Recently, Chimera [14] generates a schedule for bi-directional pipelines by using complex cost models that represent the execution time of one network pass and calculate the depth and parallelism per pipeline stage. In Halide, [3] the pipeline scheduling approach uses a cost model that considers 66 platform and application specific features. For the cost model, 26 out of 66 feature values are predicted by a neural network trained on random representative programs. According to the specifications, one training point takes at most 320 minutes to train the neural network using different schedule configurations. To predict a schedule for Halide pipelines of a single CNN layer, the scheduler considers 10k configurations. In comparison, we show that for a large YOLOv3 network of 52 layer, **Shisha** considers only 18 configurations.

4 Shisha exploration approach

A pipeline configuration consists of two components: 1) the number of CNN layers assigned to each pipeline stage, and 2) the assignment of each pipeline stage to an EP. An EP can be a single or multiple cores attached to a memory module. Therefore, we classify the EPs according to the type of memory. For example, in Figure 3 EPs are colored in green or red. We use this classification in **Shisha** to provide hints about the characteristics of the computing platforms with heterogeneous modules.

Shisha is a two-step approach. The first step is the “seed generation”, in which we use a simplified cost-model to come up with an initial solution. This initial solution is used in the second step, “online tuning” for faster convergence.

4.1 Seed generation

The goal of the seed generation is to determine a sensible starting configuration using only static information.

Firstly, Equation 1 is used to calculate the weights of the CNN layers [15, 17, 31, 32]. For each layer, H, W, C denote the height, width and depth of the

Algorithm 1 Seed Generation	Algorithm 2 Online Tuning
Require: W_l, H_e, N, L, C 1: $seed[N]$ 2: $E[N]$ 3: for $passes$ in $[0.. L - N]$ do 4: $min_w \leftarrow \min(W_l)$ 5: $n \leftarrow \min(min_w - 1, min_w + 1)$ 6: $W_l \leftarrow merge(min_w, n)$ 7: $seed \leftarrow merge_layers(min_w, n)$ 8: end for 9: $Rank \leftarrow rank(seed, W_l, C)$ 10: for i in $[0..N]$ do 11: $E[Rank_i] \leftarrow assign(Rank_i, H_{ei})$ 12: end for 13: return $seed, E$	Require: $seed, E, H_e, \alpha$ 1: $conf \leftarrow seed$ 2: $throughput = execute(conf)$ 3: $\gamma \leftarrow 0$ 4: while $\gamma < \alpha$ do 5: $stage \leftarrow slowest_stage(conf)$ 6: $t_stage \leftarrow nearestFEP(E)$ 7: $conf \leftarrow move(conf, t_stage)$ 8: $Tp = execute(conf)$ 9: if $Tp \leq throughput$ then 10: $\gamma ++$ 11: else 12: $\gamma \leftarrow 0$ 13: $throughput \leftarrow Tp$ 14: end if 15: end while 16: return $conf$

input tensor. R, S represent the height and width of the underlying convolutional kernel and K is the number of filters of the convolutional kernel. Note that, we are considering conventional CNNs in this paper, other type of layers can be incorporated in the context of this work by replacing Equation 1 with a model for the estimation of computational intensity of the layers.

$$W = H \times W \times C \times R \times S \times K \quad (1)$$

Secondly, we capture the heterogeneity of the system to support the seed generation. This is used to guide the mapping of pipeline stages to EPs together with the total weight of each pipeline stage. We rank the EPs in a decreasing order of performance, for example, from Figure 3 green EPs have rank 1 (FEP) and red ones have rank 2 (SEP). This is a hint to **Shisha** to balance the workload considering static knowledge about the heterogeneity of the system.

The seed generation process is described in Algorithm 1. $W_l = [w_{l1}, w_{l2}, \dots, w_{lL}]$ is the weight list, where a layer weight w_{li} is calculated using Equation 1. $H_e = [e_1, e_2, \dots, e_N]$ is a list of EPs sorted in descending order w.r.t. performance. For example, for Figure 3 $H_e = [G_1, G_2, \dots, G_p, R_1, R_2, \dots, R_q]$ represents the p EPs that belong to memory types X (green) and q Y (red) EPs. L is the total number of layers in a given CNN. N is the total number of pipeline stages in final pipeline ($N \leq L$) and C is assignment choice which is discussed later in this section. The output of Algorithm 1 is a pipeline configuration $Seed = [PS_1, PS_2, \dots, PS_N]$, where PS_i represents the number of CNN layers assigned to i_{th} pipeline stage. Output $E = [e_1, e_2, \dots, e_N]$ is a list of EPs from H_e and the corresponding assignment to pipeline stages. Algorithm 1 comprises two phases. In phase 1 (Lines from 3-8) we generate pipeline stages by combining CNN layers. The goal of this phase is to merge layers into groups in order to balance out the cumulative weight of groups. These groups eventually become pipeline stages. The idea is to look for the layer with lowest weight (Line 4) and merge it with the immediate

neighbour with the smallest weight (Line 5,6). Typically, the weight distribution in CNN layers does not follow any order, i.e. a light weight layer can be found between two layers with heavy weights. The second phase of Algorithm 1 (Lines 9-11) assigns the pipeline stages output by phase 1 to EPs. In principle, heavy pipeline stages should be assigned to high performance EPs, however, the assignment is not trivial in practice and requires to examine the impact of a few heuristics. Eventually, this will help in balancing execution time per pipeline stage, thus achieving a balanced pipeline.

Stage-to-EP assignment heuristics: Once CNN layers are grouped into pipeline stages, we then assign an EP to each pipeline stage. Since we have information about performance heterogeneity among EPs, we can make different choices, such as; 1) Rank pipeline stages w.r.t. number of *layers* assigned to each pipeline stage ($Rank_l$). While merging layers into stages, it is sometimes inevitable to have pipeline stages which are heavy in terms of aggregated weight with many light weight layers as opposed to a pipeline stage with one heavy layer. The highest rank corresponds to the pipeline stage with highest number of layers. We assign higher ranks to SEPs. This facilitates the online tuning phase later to greedily move the layers among pipeline stages to reach a solution. 2) Rank pipeline stages w.r.t. aggregated *weight* of each pipeline stage ($Rank_w$) Here, we assign the pipeline stages with heavy weights to fast EPs to balance the load. Line 9 controls this choice in Algorithm 1.

4.2 Online tuning

For the exploration phase, we strive to reduce the exploration time so that it is still practical to carry out an online exploration without causing a significant overhead on execution time. This is particularly challenging given the size of the multidimensional pipeline configuration space, which often includes an overwhelming majority of slow configurations. We avoid visiting such configurations by starting from the seed configuration and incrementally adjusting load distribution by moving layers from one pipeline stage to an adjacent lighter stage. In Algorithm 2, we describe the auto-tuning scheme of **Shisha**. The required input is a pipeline configuration generated as a seed. A list of EPs E which represent a mapping of pipeline stages to the computing platform. The α parameter controls how many configurations are attempted after a configuration that outperforms the seed and recently found solution has been detected. The rationale behind Algorithm 2 is to gradually reduce the load of the slowest pipeline stage in order to improve the overall throughput of the pipeline. Hence, **Shisha** finds the slowest stage (Line 5) and remaps one layer at a time to the nearest faster EPs (Line 6-7). The layer could be popped from front or back end of the stage depending on the location of new EP. Once a better configuration is found than any previous one, we try α more times to search for a better configuration. In Line 6 we balance the workload by moving layers to a nearest fast EP ($nFEP$) in pipeline i.e a closer stage which is running on an FEP. However, this is not the only choice that can be made. The nearest lightest fast EP ($nlFEP$) is also a good target to move layers as well. Therefore we keep both options open for the

user to select. The complexity of **Shisha** is negligible therefore it does not cause much work to test different choices for a given CNN and computing platform.

5 Experimental setup

Shisha targets systems that are heterogeneous in core performance and memory bandwidth. As discussed in Section 2, the system under consideration consists of different types of cores attached to different memory modules. Chiplets such as Nvidia’s Simba [27] and Intel Meteor Lake [2] resemble such types of architectures. We used the gem5 simulator [7] to simulate heterogeneous cores and memory bandwidth. The simulator provides flexibility in modeling different architectures. To simulate different core performances, we used ARM’s bigLittle cores [10] models in gem5 and to simulate different memory types, we tried different memory bandwidth values using a simple memory model connected to core cluster in gem5. Inter-EP latency is set to *20ns* [27]. However, the execution time of pipeline stages is orders of magnitude higher than inter-EP latency, thus it does not impact the performance of pipeline.

A GEMM-based implementation [23] consists of two operators; 1) Im2Col and 2) GEneralized Matrix Multiplication (GEMM). We include both operators to simulate execution time for CNN layers of ResNet50, YOLOv3 and AlexNet.

6 Evaluation

As highlighted previously, **Shisha** includes a seed generation component and an online tuning heuristic. In this section, we evaluate the quality of the seed and the final solution generated by **Shisha** and analyze the convergence of the online auto-tuning phase.

Pipe-Search [29] is an online approach that uses a database of pipeline configurations sorted w.r.t. the distribution of workload among pipeline stages. It tests pipeline configurations of various depths and converges to a solution when no better solution is found by a time limit set by the user. This approach incurs a high overhead when generating the database of pipeline configurations which also limits its scalability. We compare **Shisha**’s auto-tuning module with a set of exploration algorithms commonly used in literature, such as Hill Climbing (HC) with proximity equal to the number of layers in the network, Simulated Annealing (SA) with cooling factor values ranging from 0.9^{-5} – 0.01, Random walk (RW) and in selected cases, Exhaustive Search (ES). For a fair comparison we test SA and HC with seeds produced by **Shisha** referred to as SA_s and HC_s . For randomized algorithms, we run 200 times and picked the solution which is closer to near optimal value

We use three CNNs in our experiments. ResNet50 [11] and YOLOv3 [24] are widely used image classification CNNs. There are 50 compute intensive layers in ResNet50 and 52 compute intensive layers in YOLOv3. The generation of sorted configurations, as required by Pipe-Search and ES, incurs an impractical time overhead when running ResNet50 and YOLOv3 for more than 4-stage pipelines. Therefore, we extend our benchmark set with a synthetic network (SynthNet) consisting of 18 convolutional layers. These layers are taken from the AlexNet

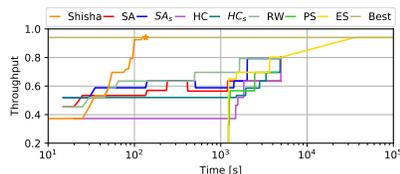


Fig. 4: Convergence of exploration algorithms for SynthNet on 8 EPs.

Xaxis is time in log scale

architecture as AlexNet has only five convolutional layers and our testing platform consists of 8 EPs. This is to analyze CNNs that can be run on a higher number of EPs (i.e. $EP > 8$) and have a compute complexity matching widely used CNNs.

6.1 Comparison of Shisha with exploration algorithms

Figure 4 shows the convergence behavior of all exploration algorithms. The solution found by **Shisha** is equal to the best solution found by ES. For a fair comparison we run SA and HC using the same seed (SA_s , HC_s) generated by **Shisha** as a starting configuration. HC tries configurations in close proximity; both versions of HC and SA managed to find a better solution ($throughput = 0.80$) compared to the best solution ($throughput = 0.94$). However, the time of convergence of representative exploration approaches is high, this is because of using many configurations out of which some are very slow. ES and PS, on the other hand, incur the overhead of generating a database of configuration. As shown in Figure 4, it took 1200s, after that ES and PS started exploring. **Shisha** explores 0.12% of the total design space as compared to Pipe-search which explores 2.03% of the design space. this is because **Shisha** attempts configurations which leads towards the solution faster. On average, the convergence time is improved by $\sim 35\times$ in **Shisha** compared to other search algorithms. In our approach, the stopping condition is controlled by α as mentioned in Section 4.2. We used $\alpha = 10$ in our experiments.

6.2 Analysis of optimality

To quantify the confidence on **Shisha** solutions, we compared against ES using larger CNNs. In this experiment we configured a system of four EPs as it takes a lot of time for ResNet50 and YOLOv3 to run ES for higher number of EPs. Figure 5 shows the throughput ($= 1/(ExecutionTime\ of\ slowest\ stage)$) of the solution found by **Shisha** and other algorithms normalized to best solution found by ES. In case of ResNet50 and YOLOv3, **Shisha** found the best solution by exploring 0.1% of the design space. In case of SynthNet, **Shisha** explored 2.5% of the design space to find the best solution. This is due to the fact that design space of SynthNet (18 layers) is smaller than ResNet50 (50 layers) and **Shisha** on average tries 25 – 35 exploration points with $\alpha = 10$.

6.3 Importance of seed in the auto-tuning phase of Shisha

The seed generated by **Shisha** contains the mapping of pipeline stages to EPs. Figure 6 represents the throughput and convergence time of **Shisha** when initiated with the seed generated by Algorithm 1, represented as **Shisha** mark (red),

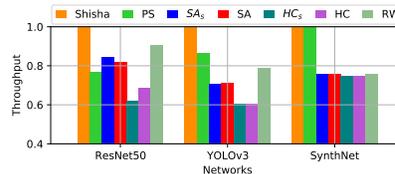


Fig. 5: Throughput of search schemes normalized to ES

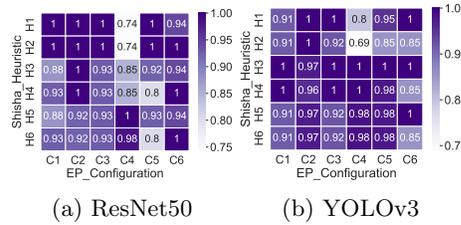
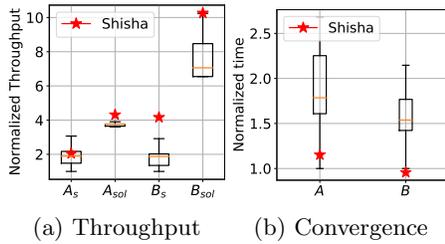


Fig. 6: Comparison of **Shisha** seed against a set of 100 random seeds. s = seed, sol = solution, A = YOLOv3 and B = ResNet50.

compared to a set of 100 random seeds and solutions obtained with random seeds. In case of ResNet50, the solution quality in both cases is similar but convergence time is increased by 35% when started with a random seed. In case of YOLOv3, the throughput of the solution found using **Shisha** seed is 16% better and the convergence time is always better than a solution found using a set of 100 random seeds.

6.4 Assignment and balancing schemes in Shisha

Section 4.1 and 4.2 discuss various choices that **Shisha** makes while assigning EPs and balancing workload among pipeline stages. We investigate the impact of each of these choices, with results shown in Figure 7. Table 1 lists the heuristics to be configured in **Shisha**. Assignment of EPs in H5 and H6 is random, in order to study the impact on convergence when no heuristic is used. Table 2 lists various configurations of the computing platform used to run this sensitivity analysis. The balancing scheme *lightest FEP* is effective in all cases as **Shisha** tries to move workload to an FEP which takes least time to execute assigned pipeline stage. This helps in balancing the pipeline as well as maximizing the throughput of the pipeline. In 80% of the cases, H1 and H3 yield better results. We investigated the convergence time of both schemes in order to determine the effectiveness of H1 and H3. Figure 8 Shows that the convergence time of H3 is less than H1 in 90% of the cases. This is due to the fact that in H3 assignment is done w.r.t. weights which means the configurations tested during exploration take reasonably less time than in H1. We recommend to use H3 because it converges faster and yields a near optimal solution.

Heuristic #	Assignment of EPs	Balancing
H1	$Rank_l$	$nlFEP$
H2	$Rank_l$	$nFEP$
H3	$Rank_w$	$nlFEP$
H4	$Rank_w$	$nFEP$
H5	random	$nlFEP$
H6	random	$nFEP$

Table 1: Heuristics of **Shisha**

Conf.	FEPs	SEPs
C1	1 8-core	1 8-core
C2	2 8-core	2 8-core
C3	4 4-core	2 8-core
C4	2 8-core	4 4-core
C5	4 4-core	4 4-core
C6	8 4-core	NULL

Table 2: EPs

6.5 Sensitivity analysis of α

The extent of exploration of **Shisha** is controlled by α . The value of α should be chosen such that it allows tuning according to the performance heterogeneity

among FEPs and SEPs while keeping a sensible convergence time. A higher value of α also means a longer tuning phase. Figure 9 shows the quality of solution (normalized to throughput obtained when $\alpha = 100$) for the YOLOv3 pipeline tested on three platform configurations with the SEPs $[3\times, 6\times, 12\times]$ slower than the FEPs. In our experiments, the performance difference between ARM’s Big and Little cores is three folds on average, which is the first case in the figure. It is shown that with the higher heterogeneity between EPs, higher α yields a better solution. We use the same starting seed for the same CNN in all cases, therefore, for lower values of α , throughput behavior is similar, irrespective to the performance difference between EPs, but in the case of a higher performance difference, throughput is improved with a higher value of α .

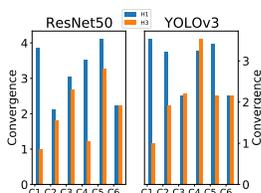


Fig. 8: Convergence time normalized to minimum value in each group for H1 and H3.

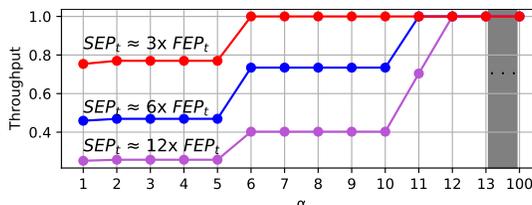


Fig. 9: Impact of α on the quality of solution in presence of heterogeneity

7 Conclusion

In this work we demonstrate a fast approach to scheduling CNN pipelines on heterogeneous computing platforms consisting of fast and slow cores. The proposed approach is generic and can be used on platforms featuring GPUs or FPGAs, in addition to asymmetric multicores and chiplets. We utilize compile time information in combination with a brief and guided online search for auto-tuning the CNN layers into parallel pipelines. Our experimental evaluation shows that the solution found by *Shisha* is as good as one produced by an exhaustive search of the design space. The results also show that *Shisha* scales well with larger networks and computing platforms. In future work, we will look at more generic tensor expressions [25] and the effect on seed parameters of high-level algebraic transformations [30].

Acknowledgment

This work has received funding from the EU Horizon 2020 Programme under grant agreement No 957269 (EVEREST), from the AI competence center ScaDS.AI Dresden/Leipzig (01IS18026A-D), PRIDE from Swedish Foundation for Strategic Research with reference number CHI19-0048 and eProcessor from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 956702. 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 <https://www.vr.se/> under grant agreement No 2018-05973.

References

1. Apple a14 bionic: Specs and benchmarks, <https://nanoreview.net/en/soc/apple-a14-bionic>
2. Intel technology roadmaps and milestones (Feb 2022), <https://www.intel.com/content/www/us/en/newsroom/news/intel-technology-roadmaps-milestones.html#gs.z471iy>
3. Adams, et al.: Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* **38**(4), 1–12 (2019)
4. Ahn, B.H., et al.: Chameleon: Adaptive code optimization for expedited deep neural network compilation. 8th International Conference on Learning Representations, ICLR 2020 (2020)
5. Anderson, et al.: Efficient automatic scheduling of imaging and vision pipelines for the gpu. *Proceedings of the ACM on Programming Languages* **5**(OOPSLA) (2021)
6. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* **52**(4) (2019)
7. Binkert, et al.: The gem5 simulator. *ACM SIGARCH computer architecture news* **39**(2) (2011)
8. Cho, et al.: Design optimization of high bandwidth memory (hbm) interposer considering signal integrity. In: 2015 IEEE EDAPS. pp. 15–18 (2015)
9. Fan, et al.: Dapple: A pipelined data parallel approach for training large models. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. pp. 431–445 (2021)
10. Greenhalgh, P.: Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper* **17** (2011)
11. He, et al.: Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. pp. 770–778 (2016)
12. Huang, et al.: Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* **32**, 103–112 (2019)
13. Kannan, et al.: Enabling interposer-based disintegration of multi-core processors. In: 2015 48th Annual IEEE/ACM MICRO. pp. 546–558. IEEE (2015)
14. Li, S., Hoefler, T.: Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. pp. 1–14 (2021)
15. Lu, et al.: Modeling the resource requirements of convolutional neural networks on mobile devices. In: *Proceedings of the 25th ACM international conference on Multimedia*. pp. 1663–1671 (2017)
16. McCalpin, J.D.: Stream benchmark. Link: www.cs.virginia.edu/stream/ref
17. Minakova, et al.: Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsoes. In: SAMOS. Springer (2020)
18. Mullapudi, et al.: Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* **35**(4), 1–11 (2016)
19. Narayanan, et al.: Pipedream: generalized pipeline parallelism for dnn training. In: *Proceedings of the 27th ACM SOSP*. pp. 1–15 (2019)
20. Narayanan, et al.: Memory-efficient pipeline-parallel dnn training. In: *International Conference on Machine Learning*. pp. 7937–7947. PMLR (2021)

21. Oren, et al.: Solo: Search online, learn offline for combinatorial optimization problems. In: Proceedings of the International Symposium on Combinatorial Search. vol. 12, pp. 97–105 (2021)
22. Ragan-Kelley, et al.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* **48**(6), 519–530 (2013)
23. Redmon, J.: Darknet: Open source neural networks in c. <http://pjreddie.com/darknet/> (2013–2016)
24. Redmon, J., Farhadi, A.: Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767 (2018)
25. Rink, N.A., Castrillon, J.: 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 (ARRAY). pp. 57–68. ARRAY 2019, ACM, New York, NY, USA (Jun 2019). <https://doi.org/10.1145/3315454.3329959>, <http://doi.acm.org/10.1145/3315454.3329959>
26. Salehian, S., Yan, Y.: Evaluation of knight landing high bandwidth memory for hpc workloads. In: Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms. pp. 1–4 (2017)
27. Shao, et al.: Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 14–27 (2019)
28. Sodani, A.: Knights landing (knl): 2nd generation intel® xeon phi processor. In: 2015 IEEE HCS'27. pp. 1–24. IEEE (2015)
29. Soomro, et al.: An online guided tuning approach to run cnn pipelines on edge devices. In: Proceedings of the 18th ACM International Conference on Computing Frontiers. pp. 45–53 (2021)
30. Susungi, A., Rink, N.A., Cohen, A., Castrillon, J., Tadonki, C.: Meta-programming for cross-domain tensor optimizations. In: Proceedings of 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'18). pp. 79–92. GPCE 2018, ACM, New York, NY, USA (Nov 2018). <https://doi.org/10.1145/3278122.3278131>, <http://doi.acm.org/10.1145/3278122.3278131>
31. Tang, et al.: Scheduling computation graphs of deep learning models on manycore cpus. arXiv preprint arXiv:1807.09667 (2018)
32. Wan, et al.: High-throughput cnn inference on embedded arm big. little multi-core processors. *IEEE TCAD* (2019)
33. Wu, et al.: A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems. In: 2020 2nd IEEE International Conference on AICAS. pp. 46–49. IEEE (2020)
34. Zheng, et al.: Ansor: Generating high-performance tensor programs for deep learning. In: 14th {USENIX} Symposium on {OSDI} 20. pp. 863–879 (2020)