Thesis for The Degree of Licentiate of Engineering

Automated CNN pipeline generation for heterogeneous architectures

Pirah Noor Soomro

Division of Computer Science Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2022

Automated CNN pipeline generation for heterogeneous architectures

Pirah Noor Soomro

Department of Computer Science & Engineering
Division of Computer Science Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2022.

"Calmness is a sign of Intelligence"
- Imam Ali

Abstract

Heterogeneity is a vital feature in emerging processor chip designing. Asymmetric multicore-clusters such as high-performance cluster and power efficient cluster are common in modern edge devices. One example is Intel's Alder Lake featuring Golden Cove high-performance cores and Gracemont power-efficient cores [1]. Chiplet-based technology allows organization of multi cores in form of multi-chip-modules, thus housing large number of cores in a processor. Interposer based packaging has enabled embedding High Bandwidth Memory (HBM) on chip and reduced transmission latency and energy consumption of chiplet-chiplet interconnect. For Instance Intel's XeHPC Ponte Vecchio package integrates multi-chip GPU organization along with HBM modules. Since new devices feature heterogeneity at the level of cores, memory and on-chip interconnect, it has become important to steer optimization at application level in order to leverage the new heterogeneous, high-performing and power-efficient features of underlying computing platforms. An important high-performance application paradigm is Convolution Neural Networks (CNN). CNNs are widely used in many practical applications. The pipelined parallel implementation of CNN is favored for inference on edge devices.

In this Licentiate thesis we present a novel scheme for automatic scheduling of CNN pipelines on heterogeneous devices. A pipeline schedule is a configuration that provides information on depth of pipeline, grouping of CNN layers into pipeline stages and mapping of pipeline stages onto computing units. We utilize simple compile-time hints which consists of workload information of individual CNN layers and performance hints of computing units. The proposed approach provides near optimal solution for a throughput maximizing pipeline. We model the problem as a design space exploration technique. We developed a time-efficient design space navigation through heuristics extracted from the knowledge of CNN structure and underlying computing platform. The proposed search scheme converges faster and utilizes real-time performance measurements as fitness values. The results demonstrate that the proposed scheme converges faster and can scale when used with larger networks and computing platforms. Since the scheme utilizes online performance measurements, one of the challenges is to avoid expensive configurations during online tuning. The results demonstrate that on average, $\sim 80\%$ of the tested configurations are sub-optimal solutions. Another challenge is to reduce convergence time. The experiments show that proposed approach is $35\times$ faster than stochastic optimization algorithms. Since the design space is large and complex, We show that the proposed scheme explores only $\sim 0.1\%$ of the total design space in case of large CNNs (having 50+ layers) and results in near-optimal solution.

Keywords

CNN parallel pipelines, Online tuning, Design space exploration, Heterogeneous computing units, Processing on chiplets

Acknowledgment

Verily every milestone is achieved by the will of God. This journey has been long and challenging but it taught me many things. I would like to thank my supervisor, Associate Professor Miquel Pericàs for his continuous guidance. With his great knowledge in research and keen insight, this journey became much more easier. I am also thankful to my co-supervisor Dr. Mustafa Abduljabbar. His keen observation and problem-solving skills helped me in many ways to continue my research. I am grateful to Prof. Jeronimo Castrillon from Technische Universität Dresden for hosting me during my research visit to his group and providing me insightful feedback on my work. I am also thankful to my research group: Dr. Madhavan Manivannan, Bhavishiya Goel, Jing, Sonia, Nadja, Nufail and Mohammad Eljamali for always being there to discuss ideas and help in every possible way. I would like to extend my gratitude to my friends; Wardah, Muhaddisa, Shehrbano and Zainab, thank you for creating a home like environment in a far far land.

Finally I would like to thank my family for always being a strong back of mine, thanks to my Mom and Dad for praying a lot for me and remembering all technical terms of my work despite not knowing what they even mean. Special thanks to my fiancé Ali Raza for listening to my rants and being encouraging and supportive throughout.

This research is partly funded by the European Union Horizon 2020 research and innovation programme under LEGaTO [1] with grant agreement No.780681, PRIDE: Principles for Computing Memory Devices [2] with grant agreement No. CHI 19-0048 funded by Swedish Foundation for Strategic Research and Eurolab4HPC [3] with grant agreement No.800962. 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 [4] grant agreement No.2018-05973.

[1] `https://legato-project.eu/`
[2] `https://www.pride-project.se/`
[3] `https://www.eurolab4hpc.eu/`
[4] `https://www.vr.se/`

List of Publications

Appended publications

This thesis is based on the following publications:

I: Pirah Noor Soomro, Mustafa Abduljabbar, Jeronimo Castrillon, and Miquel Pericàs. "An online guided tuning approach to run cnn pipelines on edge devices"
Published in Proceedings of the 18th ACM International Conference on Computing Frontiers, pp. 45-53. 2021..

II: Pirah Noor Soomro, Mustafa Abduljabbar, Jeronimo Castrillon, and Miquel Pericàs. "Sisha: Online scheduling of CNN pipelines on heterogeneous architectures"
In submission to Euro-Par 2022: Parallel Processing: 28th International Conference on Parallel and Distributed Computing.

Other publications

I: Jing Chen, Pirah Noor Soomro, Mustafa Abduljabbar, Madhavan Manivannan, and Miquel Pericàs "Scheduling Task-parallel Applications in Dynamically Asymmetric Environments"
Published in 49th International Conference on Parallel Processing-ICPP: Workshops. 2020

Contents

Chapter 1

Introduction

Convolutional Neural Networks (CNNs) have gained popularity in many practical applications, such as image classification and natural language processing. The computational, bandwidth and memory capacity requirements of CNNs is high due to the large amount of weights (up to 1.6 trillion parameters for large switch transformers [2]). Existing frameworks for implementing CNNs, such as TensorFlow [3], Cafe [4], Torch [5] and Theano [6] provide a well optimized implementations for GPU based computing platforms. However, these implementations are not optimized for heterogeneous processors, which are common in embedded devices.

Modern processors are equipped with powerful and energy efficient compute resources. Edge devices feature a combination of different execution units packed on the same chip. For instance, cores with different power-performance area characteristics but share the same Instruction Set Architecture (ISA) [7], one such example is NVIDIA Jetson TX2 [8]. Recent chip design technologies, such as 3D stacking, chiplet and interposer based packaging have added heterogeneity at the level of cores, memory subsystems and Network on Chip (NoC). Advancement in 3D stacking enabled Processing In Memory (PIM) [9,10], while interposer-based packaging technology enabled low latency and high bandwidth transmission to memory devices such as HBM. Chiplet architecture, considered as future processor design [11] feature Multi-Chip-Module (MCM) technology. Chip manufacturers adopt a mix of these technologies to produce high performance processors. For example, Nvidia's Simba [12] is packaged with several chiplets with heterogeneous interconnect bandwidth and Intel's XeHPC Ponte Vecchio [13] consists of multi-chiplet GPUs with HBM. Moreover, owing to the wide application domain of CNNs, there is a wide variety of heterogeneous computing architectures being used for running CNNs. There is no single standard architecture that can represent such a wide variety of systems in order to optimize CNN implementation [14].

CNNs consists of a sequence of compute intensive layers. The common parallelization strategy in above mention frameworks is layer-wise model parallelism. The individual layers in CNNs are parallelised over available computing resources. This approach is well suited for homogeneous computing resources. In heterogeneous computing platforms, this implementation ends up under/over-utilizing available resources. Layer pipelining coupled with model parallelism is

used for streaming CNN applications. This approach reduces communication volume and points across CNN layers and eliminate the need for copying weights on all computing resources, thus reducing memory capacity requirement [15]. CNN pipelines are implemented by various platforms, such as Pipe-It [16], PipeDream [17], Chimera [18] and Gpipe [19]. Scheduling of CNN pipelines on a computing platform which is heterogeneous at many levels (core performance, memory capacity and interconnect bandwidth) is a challenging task. A pipeline schedule comprises of packaging CNN layers into pipeline stages and mapping pipeline stages to computing units in order to generate a throughput maximising pipeline. In order to group CNN layers into pipeline stages, the scheduler must be augmented with the information regarding computational load of CNN layers and performance critical knowledge of computing platform such as memory requirement, bandwidth and core performance. Existing solutions utilize offline profiling of the representative workloads of CNNs coupled with cost models to predict layer performance along with design space exploration. This approach has following downfalls:

1. It is not portable. The offline profiling and schedule explorations need to be repeated for every computing platform and CNN.
2. In some cases, as reported in [16], prediction error leads to less effective pipeline configuration.
3. In terms of usability, these approaches require expert knowledge of computing hardware and CNN architecture to augment schedulers with performance related information.

In this thesis I investigate the requirements for automatizing the generation of pipeline schedule and propose a fast method of formulating a throughput maximizing and balanced CNN pipelines on a heterogeneous computing platform.

1.1 Problem Statements

Prior works present an offline approach for generating a throughput maximizing pipeline schedule. In this thesis we propose an online and automatic way of producing pipeline schedule for CNNs targeting heterogeneity at the level of core performance and memory bandwidth. Following are the problem statements investigated in this thesis:

Problem 1

There is no online search scheme for finding out a CNN pipeline schedule for on-chip heterogeneous cores.
Producing a schedule is a design space exploration problem. In online setting, exploring a huge space without heuristics is impractical. We divide the problem into following research questions:

Q1. The computational complexity of CNN layers vary across the network. Therefore grouping CNN layers into pipeline stages such that the workload

is balanced is NP-complete task. Moreover, the computational complexity of CNN layers is extracted from network descriptors. Therefore, our first challenge is, ***grouping layers into stages such that the work load is balanced up-to maximum possibility by leveraging the meta data of CNN description.***

Q2. The design space of pipeline schedule is complex and large. It requires tens of thousands of trials by classical search algorithms to explore a near optimal solution. Moreover many configurations are expensive in terms of time and resources, ultimately slowing down the exploration. Our second challenge is, ***to navigate through the search space such that solution could be found in limited number of trials and that the individual trials should not be expensive ones.***

Q3. Modeling design space i.e a database of all possible pipeline schedules is a challenge. As discussed earlier, design space is large and complex and it scales with large networks and computing platforms. Our third challenge is, ***to model design space and arrange the schedules for smart navigation at the time of exploration.***

Problem 2

A fully online and scalable CNN pipeline scheduling approach is required, that targets heterogeneity in memory bandwidth in addition to core heterogeneity. In this problem we mainly target chiplet architectures that are available in wide variety of processor designs adding heterogeneity in on-chip memory and core clusters.

Q4. In a chiplet architecture, execution units are organized into MCMs. Assuming that there is a heterogeneity between the chip-modules, we need to leverage this information for generating pipeline schedules. Our fourth challenge is ***to leverage the knowledge of heterogeneity for designing a heuristic based exploration of CNN pipeline schedules.***

Q5. Online exploration is aimed to be portable for any scale-length of CNN and computing platforms. Both layers in CNNs and execution units in targeted platforms are scaling higher in modern CNNs and processor designs. Our fifth challenge is, ***to design a fully online and scalable exploration scheme.***

1.2 Contributions

This thesis is based on two papers. The aim of this works is to automatically schedule CNN pipelines on heterogeneous platforms. First paper is the first ever attempt of developing an online search scheme for CNNs. Paper I answers question 1, 2 and 3. The main contributions are:

- A full-stack framework for generation and online scheduling of CNN pipelines. We designed a tuning algorithm which leverages task moldability and online performance measurements to find a near-optimal schedule

for throughput maximizing pipeline. The scheme adapts to performance asymmetries between the core clusters. We developed a tensor template language interface to describe CNN descriptors which are utilized to formulate initial schedule (referred to as seeds) for online design space exploration.

In Paper II we target chiplet based heterogeneous architectures. We applied an approach that utilizes easily available information of computing platform and CNN structure without human intervention. In this paper we answer question 4 and 5 by proposing a fully online and scalable pipeline schedule exploration for chiplets. Following are the main contributions in Paper II:

- We propose a faster way of generating seed by leveraging platform knowledge . The approach is compared to a set of representative exploration algorithms including scheme in Paper I.
- Paper I has a requirement of generation and pre-processing of design space before online phase. This is limitation when larger platforms and deeper CNNs are used as use cases. In Paper II the need for pre-processing phase is eliminated.
- We show that scheme proposed in Paper II scales better with CNNs which have 50+ compute intensive layers.

Chapter 2

Summary of the papers

2.1 Summary of Paper I

This paper aims at producing throughput maximizing pipelines for CNNs on platforms that feature performance asymmetry among core clusters. Widely used CNN frameworks [3, 4] are well optimized implementation for platforms with discrete GPUs couples with high performance CPU clusters. These frameworks commonly parallelize CNNs in data parallel fashion. The heterogeneous devices also called edge devices targeted in this paper have resource constraints in terms of power, memory and compute capability. These constraints are not directly addressed in existing frameworks. Moreover, prior work [16] that proposes CNN pipelines for heterogeneous devices relies on empirical cost model and offline profiling.

In this paper we introduce an online tuning algorithm termed as ***Pipe-Search***. It is based on evolutionary search with a pre-processed design space. We leverage the compile-time hints computed from CNN descriptors to pre-process design space such that the navigation during online tuning is guided and leads to faster convergence. The online performance profiling aids in guided navigation and enables the detection of performance issues such as contention due to memory system and inter-cluster communication [20]. We propose adaptive CNN pipelines using task moldability, a concept borrowed from job scheduling. Moldable tasks are the ones in which resource assignment (number of cores) is decided when it is first activated [21]. Figure 2.1 represents the full-stack overview of the proposed framework. We developed tensor template language interface to describe CNN architectures. Computational hits are extracted and layers are converted into moldable tasks. The next step is to generate design space for pipeline schedules and sort it with respect to the degree of work load balance with in a single schedule. The configuration which balances the computational load among pipeline stages the most, is taken as the best available schedule in the pre-processed database. The next phase is online tuning, when CNN is executed and real time performance measurements are used to navigate through the search space.

In experiments, we tested pipelines of VGG16 and compared the solution quality against exhaustive search. The navigational heuristics of *Pipe-Search* are effective, on average, 80% of the tested configurations are sub-optimal

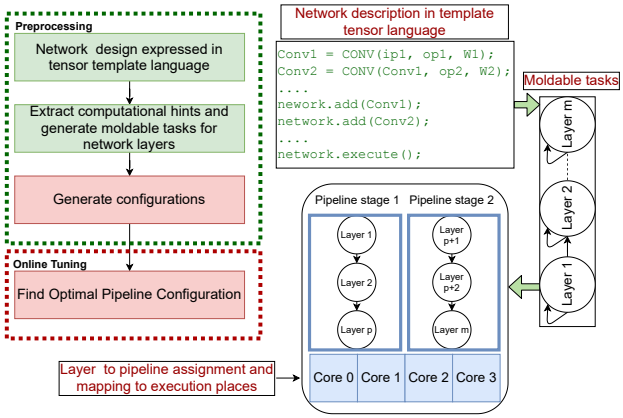


Figure 2.1: An overview of framework for generation and schedule exploration for moldable CNN pipeline

solutions. In the paper I we further show that *Pipe-Search* is 11× faster than Random Walk and 70× faster than exhaustive search.

2.2 Summary of Paper II

Chiplet based processor design is a major trend nowadays. Recent advancements in 3D stacking, interposer based packaging and chiplet based core organization has added heterogeneity at many levels. Such as memory capacity, core performance and bandwidth of on-chip network. In paper II we propose scheduling of CNN pipelines for chiplet architectures. Finding out a schedule for CNN pipelines comprises of two modules. One is exploration strategy and other modules provide fitness value. Prior work [18, 21], except *Pipe-search*, does offline performance modeling and exploration of schedules. We propose a strategy in Paper II named as ***Shisha***. Instead of offline performance modeling we utilize compile-time hints to generate initial configuration. We incorporate simple hints from hardware to generate initial seed. The online tuning is then steered based on the seed generated. This eliminated the need for pre-processing the design space.

Figure 2.2 shows the overview of *Shisha*. We utilize compile-time hits and hardware categorization information to generate seed for exploration. The hardware categorization is done based on performance. A heterogeneous system can comprise of Fast Execution Places (FEP) and Slow Execution Places (SEP). This information helps in assigning pipeline stages to execution places initially. Our approach does not require exact information on performance difference for mapping pipeline stages. The initial assignment is a sub-optimal solution but in online phase, the solution is further improved to maximize throughput of the pipeline. During online tuning phase, We again leverage the knowledge of SEP and FEP to move layers from one to another pipeline stage in order to balance execution time of all pipeline stages while maximizing overall throughput.

To measure the quality of solution explored by *Shisha* we modeled the same problem for stochastic optimization algorithms such as Simulated Annealing

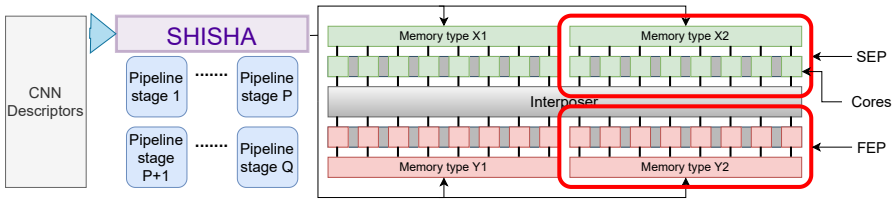


Figure 2.2: System overview in Paper II

and Hill Climbing. We also show the comparison against exhaustive search in the cases where exhaustive search finished in practical time limits. The results show that in case of large networks such as YOLOv3 and ResNet50 and larger computing platforms, *Shisha* is able to explore near-optimal solution by just exploring $\sim 0.1\%$ of the total design space. We further demonstrate that the convergence time of *Shisha* is improved by $35\times$ compared to *pipe-search* and other exploration algorithms.

Chapter 3

Conclusion and Future Directions

CNN pipelines running on heterogeneous architectures can increase the throughput by exploiting high performance features of the underlying systems. This Licentiate thesis presents an in-depth analysis on the implementation, seed generation and auto-tuning of CNN pipelines leading to better scalability and requiring less human intervention. Our approach does not require expert knowledge of CNNs or performance critical knowledge of the computing platform such as memory bandwidth, SIMD vector length, etc. Moreover, the approach should be applicable to other types of accelerators such as GPUs and FPGAs, as it considers execution places for mapping pipeline stages and the execution places is an abstract representation of a set of processing elements . With the advancement in processor design and CNN architectures, scalability of the approach is crucial to enable portability. Paper II shows that the scheme proposed is scalable by testing CNNs with 50+ layers on eight execution places. Both algorithms, *Pipe-Search* and *Shisha*, converge faster and their solution is near-optimal in all the tested cases. We present a comparison of our exploration strategies, quality of seeds and solution found against a representative set of exploration algorithms.

The importance of CNNs is widespread and it is backbone of many automated applications. Moreover an interesting shift in processor design has opened several challenges for optimizing existing applications. Processing In Memory (PIM) [9, 10], for instance, is an emerging processing technology. This opens several research directions, such as:

- To adapt CNN pipelines on such an architecture requires an in depth analysis of performance gains for CNN operators when computed in the memory.
- The placement of large CNN parameters on PIM based chiplets also needs to be investigated.
- Energy and power efficiency is a crucial requirement. Therefore, it is important to investigate pipeline schedules with the goal of reducing energy consumption while maximizing throughput of the pipeline.

Bibliography

[1] E. Rotem, Y. Mandelblat, V. Basin, E. Weissmann, A. Gihon, R. Chabukswar, R. Fenger, and M. Gupta, "Alder lake architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–23.

[2] Fedus *et al.*, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *arXiv preprint arXiv:2101.03961*, 2021.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous systems," 2015.

[4] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[5] "Torch," http://torch.ch, accessed: 2021-01-20.

[6] "Theano," http://deeplearning.net/software/theano/, accessed: 2021-01-20.

[7] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. 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.* IEEE, 2003, pp. 81–92.

[8] "Nvidia jetson tx2," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/, accessed: 2021-01-20.

[9] J. Torrellas, "Flexram: Toward an advanced intelligent memory system: A retrospective paper," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 3–4.

[10] Zhang *et al.*, "Top-pim: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on HPDC*. New York, NY, USA: Association for Computing Machinery, 2014, p. 85–98.

[11] Cho *et al.*, "Design optimization of high bandwidth memory (hbm) interposer considering signal integrity," in *2015 IEEE EDAPS*, 2015, pp. 15–18.

[12] 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*, 2019, pp. 14–27.

[13] D. Blythe, "Xehpc ponte vecchio," in *2021 IEEE HCS*. IEEE Computer Society, 2021, pp. 1–34.

[14] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[15] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[16] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. 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.

[17] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[18] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[19] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[20] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta, "Modeling the resource requirements of convolutional neural networks on mobile devices," in *Proceedings of the 25th ACM international conference on Multimedia*, 2017, pp. 1663–1671.

[21] D. Skinner and W. Kramer, "Understanding the causes of performance variability in hpc workloads," in *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* IEEE, 2005, pp. 137–149.

Paper I

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

Pirah Noor Soomro, Mustafa Abduljabbar, Jeronimo Castrillon and Miquel Pericàs

Published in
Proceedings of the 18th ACM International Conference on Computing Frontiers, pp. 45-53. 2021.

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

CF '21, May 11–13, 2021, Virtual Conference, Italy

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

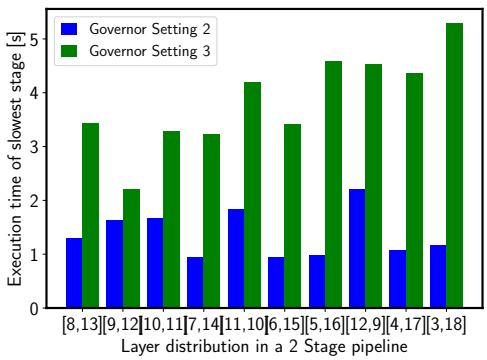


Figure 1: Execution time of the slowest stage with top 10 configurations generated based on computational hints, tested on governor setting 2 and 3. [X,Y] represent X layers assigned to pipeline stage 1 and Y layers assigned to pipeline stage 2.

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).

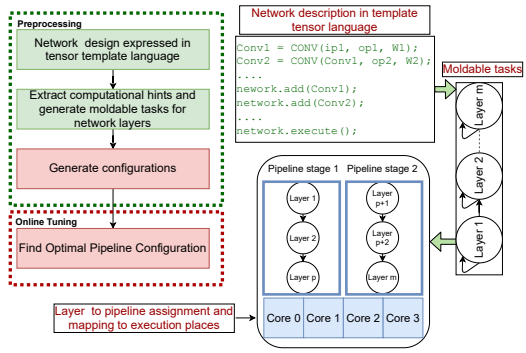


Figure 2: Left: An overview of the proposed approach, Right: Implementation of a CNN pipeline using the tensor template language, a set of moldable tasks, and two pipeline stages executing on a 4-core device.

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}.

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

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.

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

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).

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

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: γ + +
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: γ + +
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

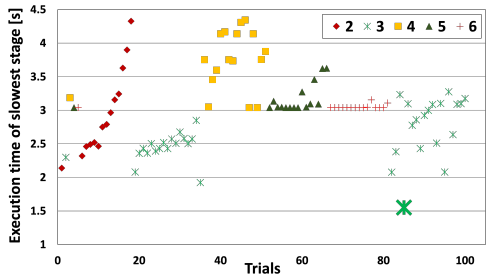


Figure 3: *Pipe-search* exploration timeline for Synth2 under governor setting 3

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

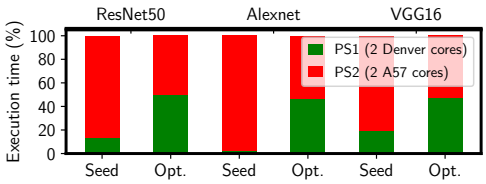


Figure 4: Percentage execution time of CNN pipelines with seeds and optimal configurations

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.

Sisha: Online scheduling of CNN pipelines on heterogeneous architectures

Pirah Noor Soomro, Mustafa Abduljabbar, Jeronimo Castrillon and Miquel Pericàs

Under rview

Shisha: Online scheduling of CNN pipelines on heterogeneous architectures

Pirah Noor Soomro[1], Mustafa Abduljabbar[1],
Jeronimo Castrillon[2], and Miquel Pericàs[1]

[1] Dept Computer Science and Engineering, Chalmers University of Technology
{pirah,musabdu,miquelp}@chalmers.se
[2] Chair for Compiler Construction, Technische Universität Dresden
jeronimo.castrillon@tu-dresden.de

Abstract. Chiplets have become a common methodology in modern chip design. Chiplets improve yield and enable heterogeneity at the level of cores, memory subsystem and the interconnect. Convolutional Neural Networks (CNNs) have high computational, bandwidth and memory capacity requirements owing to the increasingly large amount of weights. Thus to exploit chiplet-based architectures, CNNs must be optimized in terms of scheduling and workload distribution among computing resources. We propose Shisha, an online approach to generate and schedule parallel CNN pipelines on chiplet 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 · Heterogeneous computing units · Processing on chiplets

1 Introduction

Chiplet-based heterogeneous integration has been touted as the future of processor design [15]. Chiplet technology has evolved from Multi-Chip-Module (MCM) technology [34], which enables low cost during design and improves yield (i.e. by reducing chip area) [14]. Furthermore, when combined with interposer-based packaging technology, it enables lower latency and high bandwidth transmission to memory devices such as High Bandwidth Memory (HBM) [7]. Recent advancements in 3D stacking have also opened ways for enabling Processing-In-Memory (PIM) technology [37], which can be combined with chiplets to provide a solution for workloads that require large data processing. 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). For example, Nvidia's Simba research

prototype [28] features several chiplets with heterogeneous interconnect bandwidth, which in this case means that the intra-chiplet bandwidth is significantly higher than inter-chiplet bandwidth. Another example, Intel's XeHPC Ponte Vecchio [6] package, integrates a multi-chiplet GPU organization along with high bandwidth memory (HBM) modules.In order to effectively exploit such architectures, applications must be optimized considering the impact of different levels of heterogeneity present in the computing platform.

Due to their high potential memory throughput, chiplet and 3D stacked architectures have become a prominent target for emerging machine learning applications [13]. Deep Neural Networks (DNNs) have high computational, bandwidth and memory capacity requirements owing to the large amount of weights (up to 1.6 trillion parameters for large switch transformers [9]) and the increasing size of inputs 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, DNNs could be efficiently pipelined by distributing DNN layers across chiplets so as to reduce inter-chiplet communication.

In order to partition and schedule pipelines, current approaches rely on designing *cost models* to steer *design space exploration* [1, 3]. For instance, the auto-scheduler in [1] explores over ten thousand schedules for a single CNN-layer Halide [24] pipelines. 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 [1, 2, 16, 20, 33, 35, 38]. These models, however, require extensive training for near-optimal solutions [3], 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 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 time overhead.

Existing works rely on predicted fitness and offline exploration. In this context, trying and testing hundreds of schedules is acceptable, including schedules that are too expensive to test in an online setting [1, 2]. Pipe-Search [30] adopts an online exploration approach for finding a pipeline configuration. Pipe-Search 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 autotuning. 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

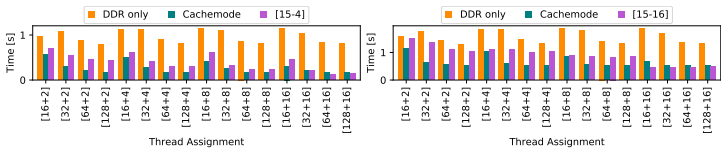
by `Shisha` suggests grouping CNN layers into pipeline stages and mapping of pipeline stages onto available sets of processing units referred to as *Execution Places (EPs)*. When generating initial configurations, `Shisha` aims at balancing the load among pipeline stages while considering the allocation 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 [30].

`Shisha` maps pipeline stages to EPs, which could be 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 [38]), 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 [26]. 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.

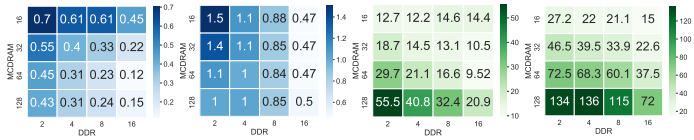
2 Motivation and problem definition

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 test the STREAM Triad benchmark [18] on Intel's Knights Landing (KNL) [29].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 DDR [27]. 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. We tested two sizes of data objects, 19 GB and 31 GB, to run STREAM Triad on KNL. For each data size, 15 GB of data are placed in MCDRAM and the remainder of the data are placed in DDR. In Figure 1, we show three cases, namely, 1) when all data is placed in DDR (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 latter yields the best performance. This shows that a clever data partitioning and thread assignment are key to achieve high



(a) [19GB] (b) [31GB]

Fig. 1: Comparison of case 1,2 and 3



(a) [15-4] (b) [15-16] (c) [15 -4] (d) [15 -16]

Fig. 2: (a) & (b) Execution time [s], (c) & (d) Parallel cost (*Core count* × *execution time*) of STREAM Triad with data distribution [X-Y], where X = GBs placed in MCDRAM and Y = GBs placed in DDR

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 DDR $[2, 4, 8, 16]$. The optimal number of threads is determined by the a) 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 DDR there is a different optimal thread partitioning. Figure 2 represents the parallel cost of the same experiment. We observed that an optimal distribution does not always lead to a minimal parallel cost. A suboptimal distribution can, in turn, reduce the parallel cost. An important observation from Figures 2b–2d is that better performance can be achieved by assigning less number of threads per memory type, rather than opting for assigning maximum number of threads. Therefore finding a distribution that results into better performance and parallel cost is challenging.

Problem definition and general approach of the solution:
This work considers a computing platform which is composed of a set of clusters 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. In this paper we demonstrate a dynamic approach to map pipeline stages, while considering heterogeneity. `Shisha` aims at grouping layers into pipeline stages such that it balances out the workload among all the stages. However a fully balanced distribution is not possible since the distribution of weights among layers is variable.

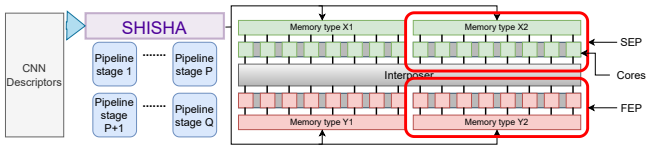


Fig. 3: System targeted in this paper. Memory type X and Y represent different memory bandwidths.

On the hardware side, the EPs are also heterogeneous in performance. `Shisha` maps pipeline stages of higher weights to FEPs. `Shisha` investigates chiplet architectures but the approach is not restricted to such platforms only. Computing platforms that are composed of processing elements each having own memory space, for instance, PIM based chiplets, can be targeted by `Shisha`. Moreover `Shisha` does not require additional human effort to adapt to changes in the computing units which makes it portable and adaptive.

3 Background

In layer pipelining [4] the work is partitioned by distributing network layers among computational resources. Model parallelism 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 roundtrips. In the rest of the paper we will refer CNN pipeline 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 chiplet 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 stochastic optimization and machine learning algorithms have been used such as Simulated Annealing [38], evolutionary algorithms [1,30], reinforcement learning [2,23] and deep neural network techniques [3]. 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. In this paper we introduce an exploration algorithm which starts with an acceptable configuration and later on is guided by heuristics that avoid trying bad configurations and taking longer time to converge.

4 Related work

Pipeline parallelism is an effective tool to fully control workload assignment to various processing elements. Pipeline parallelism is at the core of Halide [24], a language originally designed to program imaging pipelines, but which can also

be used to implement DNN inference [1,36]. Parallel pipelines for CNN training have been applied in practice [8,12,21,22]. Recently, Chimera [16] 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, [1] 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 configuration. To predict a schedule for Halide pipelines of a single CNN layer, the scheduler considers 10k configurations. In comparison, we show that for a 52 layer large YOLOv3 `Shisha` takes 24 minutes to converge and considers only 18 configurations.

Schedule space exploration has also been studied extensively. State of the art schedulers such as TVM's auto-scheduler, Ansor [38] use a trained DNN to predict fitness values for an evolutionary search approach. Chameleon [2] also adopts an adaptive sampling approach over exploration techniques such as simulated annealing and reinforcement learning to reduce the exploration time.

5 `Shisha` exploration approach

As mentioned, 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 EPs. 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 "online tuning" step for faster convergence.

5.1 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 [17, 19, 32, 33]. For each layer, H, W, C denote the height, width and depth of the 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.

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

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

Algorithm 1 Seed Generation

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: $R \leftarrow rank(seed, W_l, C)$
10: **for** i in $[0...N]$ **do**
11: $E[R_i] \leftarrow assign(R_i, H_{ei})$
12: **end for**
13: **return** $seed, E$

Algorithm 2 Online Tuning

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** $conf$

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 all 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. $W_l = [w_{l1}, w_{l2}, ...w_{lL}]$ is the weight list, where a layer weight w_{li} is calculated using Equation 1. $H_e = [e_1, e_2, ...e_N]$ is a list of EPs sorted in descending order w.r.t. performance. For example, for Figure 3 $H_e = [G_1, G_2, ..G_p, R1, R2, ..R_q]$ represents the EPs that belong to memory types X (green) and Y (red). 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 in Section 5.1 . The output of Algorithm 1 is a pipeline configuration $Seed = [PS_1, PS_2, ...PS_N]$, where PS_i represents the number of CNN layers assigned to i_{th} pipeline stage. Output $E = [e_1, e_2, ...e_N]$ 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. Since CNN layers make a chain like directed acyclic graph, therefore we can only merge consecutive layers to respect the input/output relationship across the layers within a pipeline stage.

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-chiplet 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;

- Rank pipeline stags 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.
- Rank pipeline stags w.r.t. aggregated *weight* of each pipeline stage ($Rank_w$) Here, we assign the pipeline stages with large weights to fast EPs to balance the load.

Line 9 controls this choice in Algorithm 1.

5.2 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). 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 ($nFEP$). 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.

Table 1: Gem5 System configuration

Conf #	Memory bandwidth	Core type	# of cores
1	40 GB/s	arm Big	4
2	40 GB/s	arm Big	8
3	20 GB/s	arm Little	4
4	20 GB/s	arm Little	8

6 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. This is common in chiplet architectures such as Nvidia's Simba [28], Intel XeHPC ponte Vecchio [6] and AMD Zen 2 [31]. We simulated such an architecture using the gem5 simulator [5]. To simulate different core performances we used ARM's Big and Little cores' [10] and to test different memory types, we used the simple memory design in gem5 and changed the bandwidth value to get performance numbers. Table 1 summarizes various configurations simulated for evaluating `Shisha`. Figure 3 shows a configuration for EPs that can be used with the database generated using gem5. We will use these configurations to test `Shisha` in Section 7

To obtain performance number for CNNs, we simulated convolutional kernels of widely used representative CNNs such as Resnet50 and YOLOv3. A GEMM-based implementation [25] consists of two operators; 1) Im2Col and 2) GEneralized Matrix Multiplication (GEMM). We include both operators to simulate performance numbers for CNN layers of ResNet50, YOLOv3 and AlexNet. A fixed fraction of each layer is simulated for every configuration from Table 1, which is then scaled to the full size of the layer. In our experiments we use database to query execution time of layers which is used to calculate execution time of pipeline stages. All exploration algorithms use this database which, on actual machine, is a runtime performance value.

7 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.

7.1 Baseline and test applications

Pipe-Search [30] 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 depth 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. Another limitation is that the algorithm does not consider

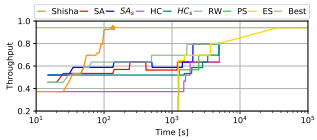


Fig. 4: Convergence of exploration algorithms for SynthNet on 8 EPs. Xaxis is time in log scale

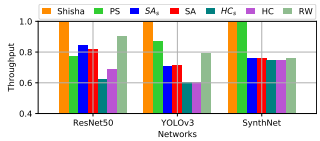


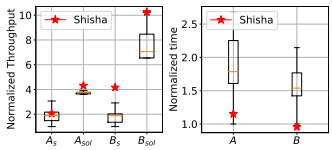
Fig. 5: Throughput of search schemes normalized to ES

heterogeneity of the platform and thus converges before trying configurations with a higher variance in computational workload among pipeline stages. We compare `Shisha`'s auto-tuning module with a set of exploration algorithms commonly used in literature, such as Hill Climbing (HC), Simulated Annealing (SA) Random walk (RW) and in selected cases, Exhaustive Search (ES).

We use three CNNs in our experiments. ResNet50 [11] and YOLOv3 [26] 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 $pipline\_depth > 4$. Therefore, we extend our benchmark set with a synthetic network (SynthNet) consisting of 18 convolutional layers. SynthNet consists of the a replication of AlexNet convolutional layers. 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.

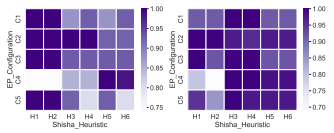
7.2 Comparison of `Shisha` with exploration algorithms

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. 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 5.2. We used $\alpha = 10$ in our experiments.



(a) Throughput (b) Convergence

Fig. 6: Comparison of `Shisha` seed with set of 100 random seeds. A_s = random seed throughput for YOLOv3 and B_{sol} = solution throughput for ResNet50



(a) ResNet50 (b) YOLOv3

Fig. 7: Throughput using different heuristics 2 and configurations of EPs 3

7.3 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 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 design space to find 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 $\alpha = 10$.

7.4 Importance of seed in auto-tuning phase of Shisha

The seed generated by `Shisha` contains the mapping of pipeline stages to EPs. The goal of online tuning in `Shisha` is to adjust the layer distribution among pipeline stages such that a final solution yields a balanced pipeline for improved throughput. `Shisha`'s seed is significant for fast convergence with a high quality solution. Figure 6 represent the throughput and convergence time of `Shisha` when initiated with the seed generated by Algorithm 1, represented as `Shisha` mark 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.

7.5 Assignment and balancing schemes in `Shisha`

Section 5.1 and 5.2 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 2 lists the heuristics

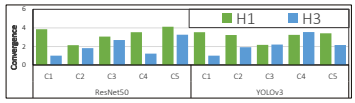


Fig. 8: Convergence time of H1 and H3 for ResNet50 and YOLOv3, normalized to minimum value in each group.

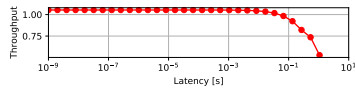


Fig. 9: Impact of inter-chiplet latency on CNN pipelines

to be configured in `Shisha`. Assignment of EPs in H5 and H6 is random, this is done to study the impact on convergence when no heuristic is used. Table 3 lists various configurations of 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. The likelihood of H1 and H3 is similar, 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 2: 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

Table 3: EPs

7.6 Impact of inter-chiplet latency on pipeline stages

To study the impact of communication latency among pipeline stages on chiplet architecture, we executed SynthNet using best solution found after exploration. We added extra latency ranging between $1ns$ to $1s$ in all chip-to-chip data transfers. Figure 9 shows that the pipeline throughput is not impacted by inter-chiplet latency unless one considers very large latencies above $1ms$. This is because the latency of pipeline stage execution is orders of magnitude higher than inter-chiplet latency. In cases when interconnect latency $> 1ms$, pipeline throughput is impacted but `Shisha` still finds near optimal solutions. However, such a high value of interconnect latency is unlikely to appear in chiplet architectures.

8 Conclusion

In this work we demonstrate a fast approach to scheduling CNN pipelines on heterogeneous computing platforms consisting of fast and slow cores. In principle, the approach is generic and can be used also on platforms featuring GPUs or FPGAs, in addition to CPUs. 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.

References

1. Adams, et al.: Learning to optimize halide with tree search and random programs. ACM Transactions on Graphics (TOG) **38**(4), 1–12 (2019)
2. Ahn, B.H., et al.: Chameleon: Adaptive code optimization for expedited deep neural network compilation. 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020 (2020)
3. Anderson, et al.: Efficient automatic scheduling of imaging and vision pipelines for the gpu. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 1–28 (2021)
4. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys (CSUR) **52**(4), 1–43 (2019)
5. Binkert, et al.: The gem5 simulator. ACM SIGARCH computer architecture news **39**(2), 1–7 (2011)
6. Blythe, D.: Xehpc ponte vecchio. In: 2021 IEEE HCS. pp. 1–34. IEEE Computer Society (2021)
7. Cho, et al.: Design optimization of high bandwidth memory (hbm) interposer considering signal integrity. In: 2015 IEEE EDAPS. pp. 15–18 (2015)
8. 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)
9. Fedus, et al.: Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. arXiv preprint arXiv:2101.03961 (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. Jiao, et al.: Computing Utilization Enhancement for Chiplet-Based Homogeneous Processing-in-Memory Deep Learning Processors, p. 241–246. Association for Computing Machinery, New York, NY, USA (2021)
14. Kannan, et al.: Enabling interposer-based disintegration of multi-core processors. In: 2015 48th Annual IEEE/ACM MICRO. pp. 546–558. IEEE (2015)
15. Li, et al.: Chiplet heterogeneous integration technology—status and challenges. Electronics **9**(4), 670 (2020)

16. 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)
17. 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)
18. McCalpin, J.D.: Stream benchmark. Link: www. cs. virginia. edu/stream/ref. html# what **22**, 7 (1995)
19. Minakova, et al.: Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In: SAMOS. pp. 18–35. Springer (2020)
20. Mullapudi, et al.: Automatically scheduling halide image processing pipelines. ACM Transactions on Graphics (TOG) **35**(4), 1–11 (2016)
21. Narayanan, et al.: Pipedream: generalized pipeline parallelism for dnn training. In: Proceedings of the 27th ACM SOSP. pp. 1–15 (2019)
22. Narayanan, et al.: Memory-efficient pipeline-parallel dnn training. In: International Conference on Machine Learning. pp. 7937–7947. PMLR (2021)
23. 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)
24. 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)
25. Redmon, J.: Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/` (2013–2016)
26. Redmon, J., Farhadi, A.: Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767 (2018)
27. 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)
28. 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)
29. Sodani, A.: Knights landing (knl): 2nd generation intel® xeon phi processor. In: 2015 IEEE HCS'27. pp. 1–24. IEEE (2015)
30. 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)
31. Suggs, D., Subramony, M., Bouvier, D.: The amd "zen 2" processor. IEEE Micro **40**(2), 45–52 (2020)
32. Tang, et al.: Scheduling computation graphs of deep learning models on manycore cpus. arXiv preprint arXiv:1807.09667 (2018)
33. Wan, et al.: High-throughput cnn inference on embedded arm big. little multi-core processors. IEEE TCAD (2019)
34. Wong, C., Wong, M.: Recent advances in plastic packaging of flip-chip and multi-chip modules (mcm) of microelectronics. IEEE Transactions on Components and Packaging Technologies **22**(1), 21–25 (1999)
35. 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)

36. Yang, et al.: Interstellar: Using halide's scheduling language to analyze dnn accelerators. In: Proceedings of the ASPLOS. pp. 369–383 (2020)
37. Zhang, et al.: Top-pim: Throughput-oriented programmable processing in memory. In: Proceedings of the 23rd International Symposium on HPDC. p. 85–98. Association for Computing Machinery, New York, NY, USA (2014)
38. Zheng, et al.: Ansor: Generating high-performance tensor programs for deep learning. In: 14th {USENIX} Symposium on {OSDI} 20. pp. 863–879 (2020)