Reinforcement learning approach for mapping applications to dataflow-based Coarse-Grained Reconfigurable Array

Abstract

NDC architecture contains a Coarse Grain Reconfigurable Array (CGRA) Streaming Engine (SE). It is a custom hardware architecture that provides programming flexibility that enables high-performance and power savings. A program is represented as a computation graph and its operations need to be properly mapped into the SE device to ensure correct execution and dataflow. This creates an optimization problem with a vast and sparse search space. Creating mappings manually or using brute force algorithms mapping takes time and lots of effort. It also adds assumptions that reduce search-space trading off optimization possibilities. Programing SE manually requires expertise on how all the SE operates, which adds entry barrier to use SE device. In this work we propose a learning method to explore and optimize in an unsupervised manner using reinforcement learning (RL) framework. This provides an automated method that can produce programs quickly and searches for optimal mappings without programmers' interference. This tool also improves the usability of the SE device by encapsulating device configuration details. Proximal Policy Optimization (PPO) training a model to place operations into the SE tiles based on a reward function that models the SE device and its constraints. Graph neural networks are added to create embeddings to represent the computation graph. A transformer block is used to model sequential operation placement mode. The trained model was able to create valid mappings for SE. The implemented method is compared against evolutionary search and baseline.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Keywords: reinforcement learning, data-flow mapping, coarse grain reconfigurable array

ACM Reference Format:
. 2018. Reinforcement learning approach for mapping applications to dataflow-based Coarse-Grained Reconfigurable Array . In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

As Dennard scaling ends, big-data applications such as real-time image processing, graph analytics, and deep learning continue to push the boundaries of performance and energy efficiency requirements for compute system. One solution to this challenge is to move compute closer to memory or storage for substantial energy savings of data movement. We have developed an innovative Compute Near Memory (CNM) architecture that leverages the dramatic opportunities provided by the new CXL protocol. CNM incorporates heterogenous compute elements in the memory/storage subsystem to accelerate various computing tasks near data. One of these compute elements is the Streaming Engine (SE). The SE is a Coarse-Grained Reconfigurable Array (CGRA) of interconnected compute tiles.

The tiles in the CGRA are interconnected with both a synchronous fabric (SF) and an asynchronous fabric (AF) as shown in Fig. 1. SF connects each tile with neighboring tiles that are one or two clocks away. It also interconnects tile memory, multiplexers, and Single Instruction Multiple Data (SIMD) units within each tile. Tiles can be pipelined through SF to form a synchronous data flow (SDF) through multiply/shift or add/logical SIMD units. AF connects a tile with all other tiles, dispatch interface (DI), and memory interfaces (MIs). It bridges SDFs through asynchronous operations, which include SDF initiation, asynchronous data transfer from one SDF to another, system memory accesses, and branching and looping constructs. Together, SF and AF allow the tiles to efficiently execute high-level programming language constructs. Simulation results of hand-crafted SE kernels have shown orders-of-magnitude better performance per watt on data-intensive applications than existing computing platforms.

Mapping the instructions from the computation graph of a program onto the compute elements of the SE while adhering to architectural constraints is an NP-complete problem with a vast and sparse search space. Constraints related to tile memory and synchronous dataflow, use of delays to match timing requirements and more are necessary to ensure correct execution. Creating the mappings manually or using brute force algorithms takes time and lots of effort. This process also adds assumptions that reduce the search space, trading off optimization possibilities.

In this work we propose a Deep Reinforcement Learning (RL) method to explore and find optimal mappings in an unsupervised manner. Using the Proximal Policy Optimization (PPO) method we train a neural network model to place instructions onto the SE tiles guided by a reward function that models the SE device and its constraints. The trained model was able to create valid mappings for the SE by learning about the problem domain. By using a learning methodology, we are able to reuse what the neural network learned to obtain mappings for computation graphs that were not seen during training.

This work is motivated to provide improved tools set that lowers SE usability barrier. This also assist other tools or programmers in SE mapping creation by providing placement suggestions or tile configuration labels. The RL mapper performs unsupervised learning and optimization allowing it to search a wide and sparse search space.

This line of research is inspired by recent work that used RL for chip placement. Our problem requirement of mapping nodes of a computation graph to available compute elements is similar to the problem of placing nodes of a chip netlist on a chip canvas.

2 Method

2.1 Streaming Engine environment

In the computation graph to be mapped, each node represents an instruction that needs to be placed onto a SE tile. The edges of the graph represent the data dependencies of the instructions. The nodes also contain information about the variables that need to be present in tile memories during instruction execution. All data needed for the instruction must be present at the tile for the instruction to be executed. The tiles can pass data to their neighbors and each tile can be configured with a different number of initiation intervals (II). Each II can be allocated to run a single instruction during program execution. After one cycle, the tiles will move on to the next II, with execution returning to first II after the last. All tiles run the instruction in the same II in parallel. Thus, whenever possible, independent instructions should be mapped to different tiles and same II to exploit parallelism. The first operation of each disconnected component of the computation graph (one component representing a synchronous flow) needs to be placed on different tiles. The SE device configuration and its constraints are modeled in a simulation environment and the reward function. Violating a constraint or placing instructions sub-optimally leads to a reduced reward whereas placements that optimally reduce total execution time of the graph are rewarded. In figure 1, the tiles connections that enables various possible instruction placements.

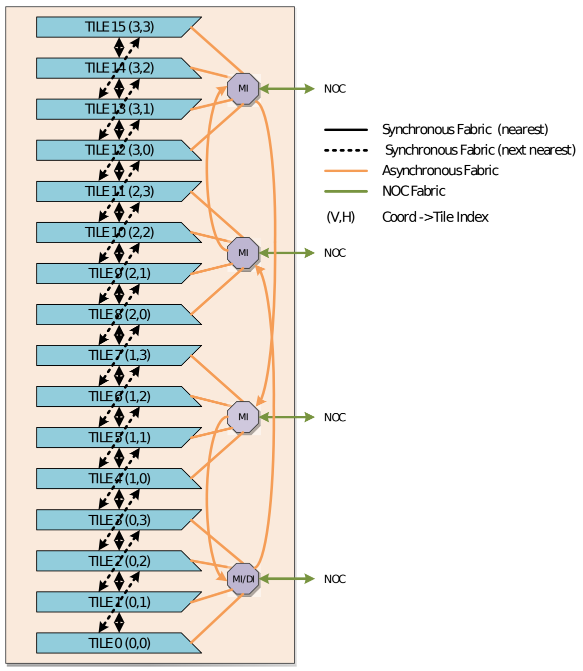

Figure 1. Diagram of the Streaming Engine.

2.2 Reinforcement learning

Reinforcement learning (RL) framework was developed to explore and optimize the mapping of operations to SE in an unsupervised manner. Proximal Policy Optimization (PPO) trains a neural network model to place operations into the SE tiles based on a reward function that models the SE device and its constraints. The SE mapping task is formulated as a discrete action problem. In PPO, samples of state, reward and action are collected by running inference on the latest copy of actor model and getting the outcome from the SE reward function. These samples are stored in a buffer that will be used as data to train the models. The state is a concatenation of an array of placed nodes, a selected node to be placed next and an embedding of the whole computation graph. An action is a tile location and a II count. The reward is the number of cycles taken to execute all nodes. After each action, a node is placed and its state, action, reward is saved as sample. The buffer can choose to keep or discard certain number unsuccessful placements samples to balance number of successful and unsuccessful samples for the training phase. After sampling, the actor model is trained to produce actions from the sample states and the critic model is trained to match the sampled rewards from the actions produced by the actor using a surrogate loss function. This sample and train process is repeated over various iterations. Figure 2 shows a diagram of the sample and training phase. After node placements, routing info and configurations for programming each tile are saved for final output.

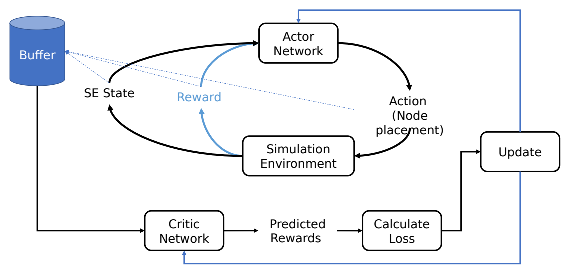

Figure 2. Diagram of the RL framework showing the role of actor and critic networks during RL training.

2.3 Model design

The actor and critic models architecture is shown in figure 3. The input is separated in two categories: static and dynamic data. Static data is infomation that doesn't change as nodes are being placed, such as: computation graph and tile memory constraint. Device state, node to be placed and placed node latencies are dynamic data that changes during placement.

Tile memory variables need to be placed in a tile so that operation can use that variable. This memory constraint is captured as memory dependency array. Tile memory constraints are incorporated into nodes in the computation graph. The computation graph has each node representing an instruction. The node features are tile memory dependencies. A Graph Neural Network (GNN) is used to process node dependencies and create an embedding for each node. An attention module is applied to the embedding matrix to select which dependency nodes are relevant to the current node to be placed. The dynamic data is fed into a MLP model to create another embedding to represent current state. The two embeddings are combined and fed into another MLP model to create actions. Invalid actions are masked before being sent to the reward function. Masking was shown to be effective in RL setting [4].

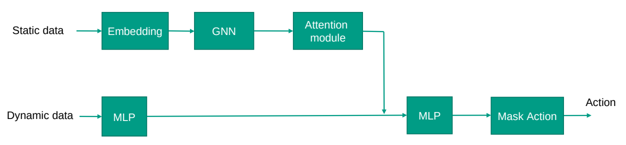

Figure 3. Actor and critic model architecture. GNN is used to process the computation graph (static data). Attention module gives importance to relevant nodes. The embedding created from dynamic data is combined with static data embedding. A final MLP model is used to generate actions. Actions are masked to ensure only valid actions are produced.

3 Related work

Reinforcement Learning (RL) is widely used to tackle unsupervised optimization problems. It has been applied in chip placement [7], workload distribution [1, 8, 11], compiler optimizations [10] and other decision based tasks [5, 12]. Alternative optimization algorithms includes evolutionary strategies [6] and bayesian optimization [9]. An advantage of RL approach is learning from a collection of programs and reuse previous data for new programs by training a Deep Learning model [11].

Recurrent neural networks (RNN) [3] were popular approach to process sequence of nodes from a graph representation. Graph neural networks (GNN) [2] showed success in processing structured data without preprocessing required for RNNs. GNNs are widely used task involves processing graphs [10, 11]. In addition, attention modules are used to supplement the embeddings created by GNNs to further improve results [1].

In [7], a RL method for chip placement is presented. A graph neural network to create embedding from a netlist graph and passed through an actor model via PPO. The produced output is the whole chip placement. Their actor is composed of deconvolution layers which are more computationally intensive. [11] presents a combination of graph neural network and transformer-XL model to place operations on devices.

In our case, the input is a combination of computation graph, SE device state and node to be placed. Instead of only feeding our actor model with embedding from graph neural network, we combine the graph neural network embedding with information from the SE device and a representation of node that is going to be placed.

A configuration is also added to guide the model to optimize different goals.

Another difference from [11] is that our strategy is to place one computation node at a time, instead of generating a whole assignment per iteration. This allows the model to break down the placement problem into sub-problems. This also allows the framework to start from a different start point. For example, if some nodes are already placed by some other algorithm, the RL mapper can place the remaining nodes. This approach also allows us to sample more data during sampling phase. It can save per node placement, instead of only saving a sample for an entire sequence.

4 Results

The implemented RL approach was able to successfully map the computation graphs for vector addition and multiply-add operations. Fig. 3 shows the increase in the value of the rewards obtained as the number of training iterations increase. As the reward increases, we obtain mappings with lesser total execution times.

This method also presents an advantage compared to baselines random search and evolutionary search (ES) methods which do not learn. The RL approach can also learn from a collection of computation graphs and reuse the learning for mapping previously unseen computation graphs. The other methods on the other hand will have to perform a new search and start from scratch for every new computation graph. PPO with graph embeddings showed that we can obtain more optimized placements by finding higher rewards than other RL approaches.

4.1 Experimental Setup

4.2 Abrasion study

5 Conclusion

Our RL mapper improves and increments the capabilities of the toolset used to program the SE device. It can search for optimal mappings while using the learning to map previously unseen workloads. It improves on the total time required to get a mapping as compared to the existing brute force search approach and allows for automated search of mappings with different optimizations under different trade-offs. It also reduces the manual labor required to find mappings. The techniques presented could be applied to other applications such as chip placement, which is worth trying in future work. Our future work also includes:

- Optimizing training methods to obtain mappings for problems like IFFT which have a bigger search space than currently used computation graphs.
- Increasing sample efficiency of learning methods and improving the simulation environment for the SE by adding more constraints.
- Integration of RL mapper into the SE toolset

Acknowledgments

To Robert, for the bagels and explaining CMYK and color spaces.

References

[1] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv preprint arXiv:1906.08879* (2019).

[2] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE international joint conference on neural networks*, Vol. 2. 729–734.

[3] Sepp Hochreiter and Jürgen Schmidhuber. 1996. LSTM can solve hard long time lag problems. *Advances in neural information processing systems* 9 (1996).

[4] Shengyi Huang and Santiago Ontañón. 2020. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *CoRR* abs/2006.14171 (2020). arXiv:2006.14171 https://arxiv.org/abs/2006.14171

[5] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. 2013. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics* 2, 3 (2013), 122–148.

[6] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh D. Dhebar, Kalyanmoy Deb, Erik D. Goodman, and Wolfgang Banzhaf. 2018. NSGA-NET: A Multi-Objective Genetic Algorithm for Neural Architecture Search. *CoRR* abs/1810.03522 (2018). arXiv:1810.03522 http://arxiv.org/abs/1810.03522

[7] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Sungmin Bae, et al. 2020. Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746* (2020).

[8] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. *CoRR* abs/1706.04972 (2017). arXiv:1706.04972 http://arxiv.org/abs/1706.04972

[9] Zhan Shi, Chirag Sakhuja, Milad Hashemi, Kevin Swersky, and Calvin Lin. 2020. Learned Hardware/Software Co-Design of Neural Accelerators. *arXiv preprint arXiv:2010.02075* (2020).

[10] Yanqi Zhou, Sudip Roy, AmirAli Abdolrashidi, Daniel Wong, Peter C. Ma, Qiumin Xu, Hanxiao Liu, Mangpo Phitchaya Phothilimtha, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. 2020. Transferable Graph Optimizers for ML Compilers. *CoRR* abs/2010.12438 (2020). arXiv:2010.12438 https://arxiv.org/abs/2010.12438

[11] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. 2019. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint arXiv:1910.01578* (2019).

[12] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. *CoRR* abs/1611.01578 (2016). arXiv:1611.01578 http://arxiv.org/abs/1611.01578

A Research Methods

A.1 Part One

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi malesuada, quam in pulvinar varius, metus nunc fermentum urna, id sollicitudin purus odio sit amet enim. Aliquam ullamcorper eu ipsum vel mollis. Curabitur quis dictum nisl. Phasellus vel semper risus, et lacinia dolor. Integer ultricies commodo sem nec semper.

A.2 Part Two

Etiam commodo feugiat nisl pulvinar pellentesque. Etiam auctor sodales ligula, non varius nibh pulvinar semper. Suspendisse nec lectus non ipsum convallis congue hendrerit vitae sapien. Donec at laoreet eros. Vivamus non purus placerat, scelerisque diam eu, cursus ante. Etiam aliquam tortor auctor efficitur mattis.

B Online Resources

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam interdum magna at lectus dignissim, ac dignissim lorem rhoncus. Maecenas eu arcu ac neque placerat aliquam. Nunc pulvinar massa et mattis lacinia.