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

Abstract

The Streaming Engine (SE) is a Coarse-Grained Reconfigurable Array developed by Micron Technology. The SE provides programming flexibility and high-performance with energy efficiency. A program is represented as a computation graph, where every instruction a node. Each node need to be mapped to the right slot and array in the SE to ensure the correct execution of the program. This creates an optimization problem with a vast and sparse search space. A manual mapping of the graph takes an infeasible amount of time by the programmer. such manual mapping is impractical because it requires an expertise and knowledge in the SE micro-architecture and reduces the search-space, thus, trading off optimization possibilities. In this work we propose an AI-based mapper to explore mappings and optimize them in an unsupervised manner using reinforcement learning 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, 7 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 Near-Data Computing (NDC) architecture that leverages the dramatic opportunities provided by the new CXL protocol [15]. NDC 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) that is composed of interconnected compute tiles. The compute tiles are interconnected with both a Synchronous Fabric (SF) and an Asynchronous Fabric (AF) as shown in Fig. 2. 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 (MS) unit and an Arithmetic/Logic (AL) SIMD units. The output of the MS unit is connected the one of the two inputs onf the AL unit. 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.

A simple example that illustrates a program mapped to the SE is shown in Fig. 1. The program in this example is a distance calculation function shown in Eq. 1.

$$D = \sqrt{(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2} \tag{1}$$

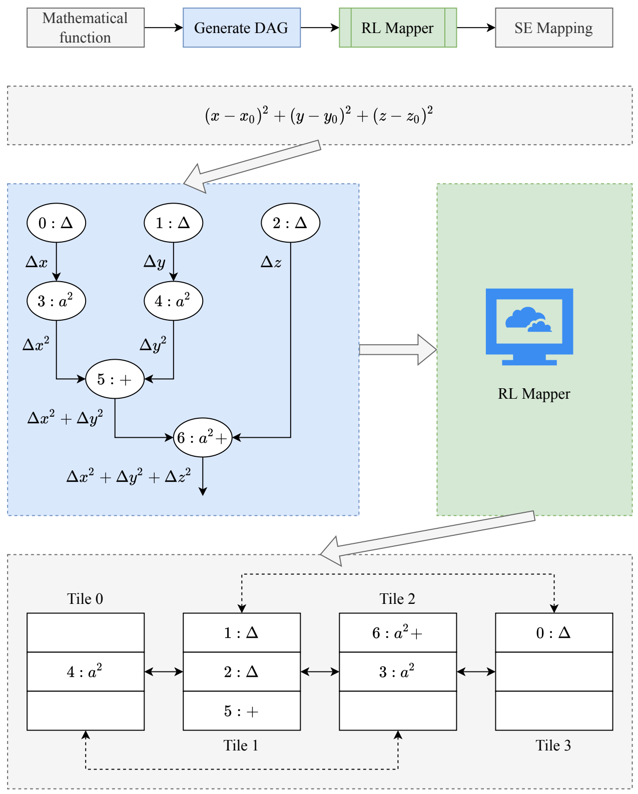

Figure 1. Example showing the high-level steps of mapping of distance calculation using the SE device. The solid lines connecting the tiles, represent an SF connection with one clock cycle latency. The dotted lines connecting the tiles, represent an SF connection with two clock cycles latency. The pipeline latency for an instruction is three clock cycles. For example, instruction #0 is scheduled at slot #0 of tile #3 and its output is sent to instruction #3. The result of instruction #0 is ready after three clock cycles, in addition to two clock cycles to move the result to tile #1, instruction #3 is correctly scheduled on on slot #1 of tile #1.

To keep this example simple, we ignore the square root part of the equation. The program is represented as a Directed Acyclic Graph (DAG) as shown in Fig. 1. Each operation in Eq. 1 is an instruction that is mapped to a slot at a tile on the SE. An instruction can include both a multiplication and an addition, as shown in Eq. 2.

$$MS/AL_{output} = (MS_{in1} \times MS_{in2}) + AL_{in2} \tag{2}$$

Hence, instruction 6 produces Δz^2 on the output on MS unit followed which is added to $(\Delta x^2 + \Delta y^2)$ at the AL unit.

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 is future work

The key contributions of this paper are as follows.

1. contribution 1
2. contribution 2
3. ... etc.

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. Each operation is an instruction that is mapped to a specific slot on a tile.

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.

The rest of this paper is organized as follows: Section 2 provides the necessary background on the SE device architecture. Section 3 presents the proposed RL approach for mapping applications to the SE. Finally, Section 4 discusses related work and Section 6 provides conclusions and future work.

2 Background on the SE

The SE is a coarse-grained fabric composed of compute tiles, which are interconnected with an SF, allowing data to traverse from one tile to another without queuing. This SF allows many tiles to be pipelined together to produce a continuous data flow through SIMD arithmetic operations. The tiles are also interconnected with an AF that allows synchronous domains of compute to be bridged by asynchronous operations. These asynchronous operations include initiating synchronous domain operations, transferring data from one synchronous domain to another, accessing system memory (read and write), and performing branching and looping constructs. Together, the SF nad AF allow the tiles to efficiently execute high level language constructs.

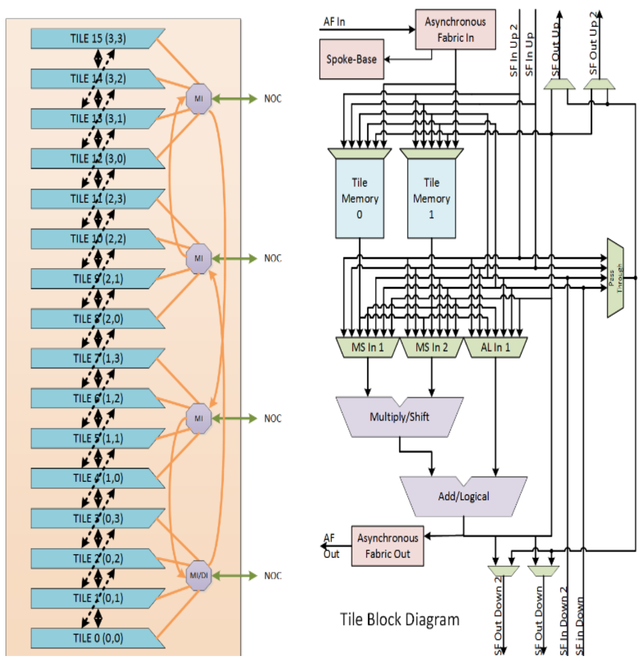

Figure 2. Diagram of the Streaming Engine to the right and diagram of a tile to the left. In the SE device diagram to the left, the solid black lines connecting the tiles, represent an SF connection with one clock cycle latency. The dotted black lines connecting the tiles, represent an SF connection with two clock cycles latency. The orange solid lines represent AF connections to the four MIs and the DI. The green solid lines represent connections to the Network-On-Chip (NOC) interfaces. The MIs and DI communicate with memory and the Command Manager using these NOC interfaces. In the Tile diagram to the right, SF Out Up1 and SF Out Down1 represent outputs from the tile to the tiles are one-hop above and below the tile respectively. SF Out U2 and SF Out Down2 represent outputs from the tile to the tiles are two-hops above and below the tile respectively. SF In Up1 and SF In Down1 represent inputs from the tile to the tiles are one-hop above and below the tile respectively. SF In U2 and SF In Down2 represent inputs from the tile to the tiles are two-hops above and below the tile respectively.

As shown in the Fig. 2, tiles are interconnected in a one by sixteen configuration. The SF is used to connect a tile to another tile one hop above it, a tile two hops above it, a tile one hop below it and a tile two hops below it. Information is transferred over the SF with a deterministic latency. Each tile acts independently, streaming data through internal memory and MS/AL units, to other tiles over the SF and AF. The tiles use the AF to communicate between synchronous domains, send load and stores to memory through the MI, and receive commands from the host through the DI to initiate work on the SE. Information is transferred over the AF with a non-deterministic latency.

2.1 Synchronous Fabric Interface

All tiles that participate in a synchronous domain act as a single pipelined data path. The Base Tile (BT) is define as the first tile of a synchronous domain initiates a thread of work through the pipelined tiles. The BT is responsible for starting work on a predefined cadence referred to as the Spoke Count or Initiation Interval (II). E.g. if $II = 3$ as in the example in Fig. 1, then the BT can initiate work every third clock. The synchronous fabric provides both data and control information.

2.2 Asynchronous Fabric interface

The Asynchronous Fabric is used to perform operations that occur asynchronous to a synchronous domain. Each tile contains an interface to the Asynchronous Fabric. Asynchronous Fabric messages can be classified as either data messages or control. Data messages contain a SIMD width data value that is written to one of the two tile memories. Control messages are for controlling thread creation, freeing resources, or issuing external memory requests.

2.3 Tile Base

The tile base contains data structures that are used to initiate a synchronous flow. A new thread is launched by the tile base of the BT every II. The II allows the base logic to launch new threads or continue previously launched threads when all resources needed for execution are available.

2.4 Tile Memory

Each tile contains two memories, each are the width of the data path (512-bits), and the depth will be in the range of 512 to 1024 elements. The tile memories are used to store data required to support data path operations. The stored data can be constants loaded as part of the program's arguments, or variables calculated as part of the data flow. The tile memory can be written from the AF as either a data transfer from another synchronous domain, or the result of a load operation initiated by another synchronous domain.

2.5 Instructions RAM

Each tile has an instruction RAM has multiple entries to allow a tile to be time sliced, performing multiple, different operations in a pipelined synchronous domain.

2.6 Spoke RAM

The Spoke RAM entries are used to configure the tile at each time slice. The number of active entries spoke RAM in a tile is equal to the II of that tile. Some of the relevant configurations per an entry are:

- Which of the four SF inputs, feedback from the output of the tile's MS/AL unit, or the tile base is the master input.
- Which of the four SF outputs is used to send the output of the MS/AL unit to another tile using SF.

The Spoke RAM iterates over its entries comes using a counter that modulo counts from zero to II minus one and back to zero. The proposed RL mapper provides the II and the configuration of each spoke RAM entry for each tile, as well as which entry from the instruction RAM can be active on that spoke entry.

A program is broken down into a set of one or more synchronous data-flows. A synchronous data-flows is represented as a computation graph, where every instruction a node. Each node represents an instruction that needs to be placed onto a SE tile at he proper time-slice such that when all nodes are placed, the program executes correctly. In Fig. 1, 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. The instruction intended to execute at a specific time-slice will only execute when the master input configured in the corresponding Spoke RAM entry receives a valid control message.

2.7 SE Mapping Constraints

The SE hardware imposes constraints that the RL mapper must adhere to in order to for it to produce a valid mapping. These constraints are:

1. Instructions that share one or more tile memory variables must be placed on the same tile.
2. No two or more instructions that start a synchronous data-flow can share a tile.
3. No two or more instructions that are siblings can share a tile.

The SE device configuration and its constraints are modeled in a simulation environment and the reward function. In order to enforce these constraints, we explore two methods: the first method is to give a negative reward when a constraint is violated and the second is creation of a mask on the invalid actions so that the network only outputs valid actions. In both cases, placing instructions sub-optimally leads to a reduced reward whereas placements that optimally reduce total execution time of the graph are rewarded.

3 Method

3.1 Reinforcement learning

We present a Reinforcement learning (RL) based framework to explore and optimize the mapping of instructions to SE in an unsupervised manner, guided by a reward function that informs the mapping algorithm about the quality of the mapping at each step.

3.2 Overview

Proximal Policy Optimization (PPO) is an RL method widely used for continuous and discrete action problems. It trains actor and critic models (represented as a neural networks). The actor model outputs actions (node placements) given an observation of the current state of the SE. The critic model is trained to predict the expected reward given an action in the current state. Our SE mapping task is formulated as a discrete action problem where samples of SE state, actions performed, and rewards obtained at each time step are collected after executing the actions provided by the actor model in simulation. These samples are stored in a buffer that is used as data to train the models. The key components of our RL method are as follows:

- States: The state is represented by a concatenation of an array of features of placed nodes, a selected node to be placed next and an embedding of the whole computation graph.
- Action: The action consists of the node to be placed along with the tile and spoke location it is to be placed at.
- Reward function: The reward obtained is based on the number of cycles taken to execute all nodes.
- Transition function: The transition function gives the probability distribution over next states given the current state and the action to be performed.

The actor model is trained to produce actions from sample states obtained during simulation 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. In this manner, the problem search space is explored using the reward function as a heuristic. Figure 3 shows the overall RL framework. After node placements, routing info and configurations for programming each tile are saved for final output.

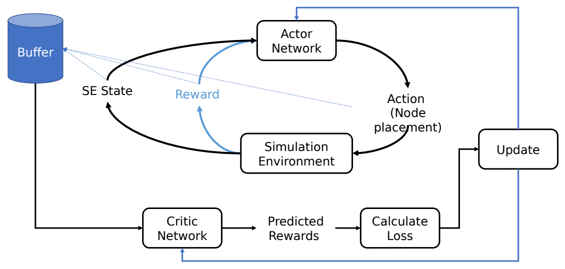

Figure 3. Diagram of the RL framework showing the role of actor and critic networks during RL training.

3.3 State Representation

The state is a vector of size $TS + 1$ where TS is the number of tile slices in the Streaming Engine with $TS = T \times I$. T and I are the total number of tiles and initiation intervals per tile in the SE.

3.4 Action Representation

Ensure mathematical notations are consistent throughout paper

The action at each step is a tuple (n, t, i) where $n \in N$ is the node to be placed and $t \in T$ and $i \in I$ are the tile and initiation interval at which the node n is to be placed. Selecting the node to be placed is not a part of the learning problem. We place nodes in a topological order which ensures that all of the nodes predecessors have been placed before the node.

3.4.1 Masked Actions. In order to enure that the network only outputs valid actions, we determine a binary mask over all possible actions and set the value of logits corresponding to invalid actions to $-\infty$. This in turn sets the probability of sampling invalid actions to 0, ensuring that we never take an invalid action. Finally, we only calculate entropy on valid actions, so that our algorithm maximizes exploration only on valid actions.

3.5 Reward Function

Our goal in this work is to get optimal mappings in terms of total clock cycles taken and for this purpose, the reward that is given at each time step is the difference between the clock cycle at which the current node to be placed is ready (its ready time) and the ready time of its predecessor. R_n is the reward obtained for placing node n and t_n is its ready time. The function $p(n)$ gives the predecessor of n. If the current node can't be placed because of the constraints (all values in the mask vector m are zero), then a high negative reward $-\lambda$ is given.

$$R_n = \begin{cases} -\lambda, & m_i = 0, \forall\, i \in T \times I \\ t_n - t_{p(n)}, & \text{otherwise} \end{cases}$$

3.6 Model design

The actor and critic models architecture is shown in figure 4. The input is separated in two categories: static and dynamic data. Static data is information 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 [6].

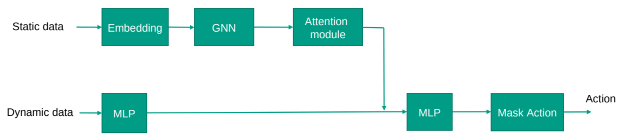

Figure 4. Actor and critic model architecture. GNN is used to process the computation graph (static data). Attention module gives importance to relevant nodes. The embedding created from dynamic data is combined with static data embedding. A final MLP model is used to generate actions. Actions are masked to ensure only valid actions are produced.

4 Related work

4.1 CGRA

CGRAs are a heavily researched architectural paradigm with a long history and can provide an excellent balance of high-performance compute, memory bandwidth, and area and energy efficiency [17]. Due to such advantages, CGRAs are currently enjoying a resurgence in interest—not just in the research realm [14], but also commercially [12, 13, 18]. SE differentiates from others as the first CGRA in a near-data computing architecture. SE also provides asynchronous messaging as a first-class programming construct along with synchronous data flow.

A key challenge with SE applications is efficient compilation of high-level language code (for example, in C/C++) to executable program. Research compilers targeting CGRAs are available (for example, [2, 3, 9, 14]) but they are limited in quality and code coverage. Industry-strength compilers such as Clang/LLVM do not provide official support for CGRA-like architectures.

4.2 Reinforcement Learning

Reinforcement Learning (RL) is widely used to tackle unsupervised optimization problems. It has been applied in chip placement [10], workload distribution [1, 11, 20], compiler optimizations [19] and other decision based tasks [7, 21]. Alternative optimization algorithms includes evolutionary strategies [8] and bayesian optimization [16]. 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 [20].

Recurrent neural networks (RNN) [5] were popular approach to process sequence of nodes from a graph representation. Graph neural networks (GNN) [4] showed success in processing structured data without preprocessing required for RNNs. GNNs are widely used task involves processing graphs [19, 20]. In addition, attention modules are used to supplement the embeddings created by GNNs to further improve results [1].

In [10], 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. [20] 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 [20] 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.

5 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.

5.1 Experimental Setup

5.2 Abrasion study

6 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

Placeholder.

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] Michaël Adriaansen, Mark Wijtvliet, Roel Jordans, Luc Waeijen, and Henk Corporaal. 2016. Code generation for reconfigurable explicit datapath architectures with llvm. In *2016 Euromicro Conference on Digital System Design (DSD)*. IEEE, 30–37.

[3] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 184–189.

[4] 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.

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

[6] 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

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

[8] 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

[9] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings-Computers and Digital Techniques* 150, 5 (2003), 255.

[10] 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).

[11] 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

[12] Timothy Prickett Morgan. 2018. Intel's Exascale Dataflow Engine Drops X86 and von Neumann.

[13] Chris Nicol. 2017. A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing. *Wave computing white paper* (2017).

[14] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Plasticine: a reconfigurable accelerator for parallel patterns. *IEEE Micro* 38, 3 (2018), 20–31.

[15] Debendra Das Sharma and Siamak Tavallaei. 2020. Compute express link 2.0 white paper. *Tech. Rep.* (2020).

[16] 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).

[17] George Theodoridis, Dimitrios Soudris, and Stamatis Vassiliadis. 2007. A survey of coarse-grain reconfigurable architectures and cad tools. In *Fine-and Coarse-Grain Reconfigurable Computing*. Springer, 89–149.

[18] Kees Vissers. 2019. Versal: The xilinx adaptive compute acceleration platform (acap). In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 83–83.

[19] 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

[20] 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).

[21] 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