

# A graph placement methodology for fast chip design

<https://doi.org/10.1038/s41586-021-03544-w>

Received: 3 November 2020

Accepted: 13 April 2021

Published online: 9 June 2021

 Check for updates

Azalia Mirhoseini<sup>1,4</sup>✉, Anna Goldie<sup>1,3,4</sup>✉, Mustafa Yazgan<sup>2</sup>, Joe Wenjie Jiang<sup>1</sup>, Ebrahim Songhori<sup>1</sup>, Shen Wang<sup>1</sup>, Young-Joon Lee<sup>2</sup>, Eric Johnson<sup>1</sup>, Omkar Pathak<sup>2</sup>, Azade Nazi<sup>1</sup>, Jiwoo Pak<sup>2</sup>, Andy Tong<sup>2</sup>, Kavya Srinivasa<sup>2</sup>, William Hang<sup>3</sup>, Emre Tuncer<sup>2</sup>, Quoc V. Le<sup>1</sup>, James Laudon<sup>1</sup>, Richard Ho<sup>2</sup>, Roger Carpenter<sup>2</sup> & Jeff Dean<sup>1</sup>

Chip floorplanning is the engineering task of designing the physical layout of a computer chip. Despite five decades of research<sup>1</sup>, chip floorplanning has defied automation, requiring months of intense effort by physical design engineers to produce manufacturable layouts. Here we present a deep reinforcement learning approach to chip floorplanning. In under six hours, our method automatically generates chip floorplans that are superior or comparable to those produced by humans in all key metrics, including power consumption, performance and chip area. To achieve this, we pose chip floorplanning as a reinforcement learning problem, and develop an edge-based graph convolutional neural network architecture capable of learning rich and transferable representations of the chip. As a result, our method utilizes past experience to become better and faster at solving new instances of the problem, allowing chip design to be performed by artificial agents with more experience than any human designer. Our method was used to design the next generation of Google's artificial intelligence (AI) accelerators, and has the potential to save thousands of hours of human effort for each new generation. Finally, we believe that more powerful AI-designed hardware will fuel advances in AI, creating a symbiotic relationship between the two fields.

In this work, we propose a new graph placement method based on reinforcement learning (RL), and demonstrate state-of-the-art results on chip floorplanning, a challenging problem<sup>2</sup> that has long defied automation, despite five decades of research. Our method generates manufacturable chip floorplans in under 6 h, compared to the strongest baseline, which requires months of intense effort by human experts.

A computer chip is divided into dozens of blocks, each of which is an individual module, such as a memory subsystem, compute unit or control logic system. These blocks can be described by a netlist, a hypergraph of circuit components, such as macros (memory components) and standard cells (logic gates such as NAND, NOR and XOR), all of which are connected by wires. Chip floorplanning involves placing netlists onto chip canvases (two-dimensional grids) so that performance metrics (for example, power consumption, timing, area and wirelength) are optimized, while adhering to hard constraints on density and routing congestion.

Since the 1960s, many approaches to chip floorplanning have been proposed, falling into three broad categories: partitioning-based methods<sup>3–5</sup>, stochastic/hill-climbing approaches<sup>6–8</sup> and analytic solvers<sup>9–14</sup>. However, none of these approaches could achieve human-level performance, and the exponential growth in chip complexity has rendered these techniques largely unusable on modern chips.

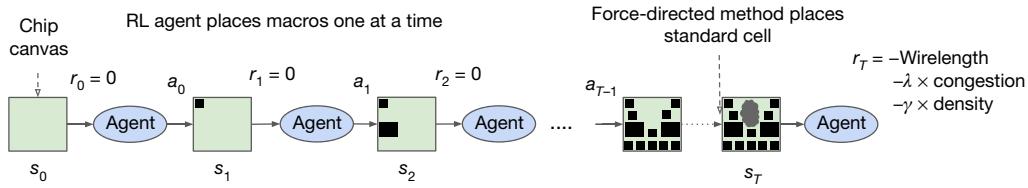
The limitations of these prior approaches are varied. For example, partitioning-based methods sacrifice the quality of the global solution

in order to scale to larger netlists, and a poor early partition may result in an unsalvageable final result. Hill-climbing approaches have low convergence rates and do not scale to modern chip netlists, which have millions or billions of nodes<sup>15</sup>. Prior to this work, analytic solvers were the leading approach, but they can only optimize for differentiable loss functions, meaning that they cannot effectively optimize for critical metrics, such as routing congestion or timing violations. Our method, on the other hand, can scale to netlists with millions of nodes, and optimizes directly for any mixture of differentiable or non-differentiable cost functions. Furthermore, our method improves in both speed and quality of result because it is exposed to more instances of the chip placement problem.

Owing to the limitations of these prior methods, human physical designers must iterate for months with commercial electronic design automation (EDA) tools, taking as input a register transfer level (RTL) description of the chip netlist, generating a manual placement of that netlist onto the chip canvas, and waiting up to 72 h for EDA tools to evaluate that placement. On the basis of this feedback, the human designer either concludes that the design criteria have been achieved, generates an updated floorplan for evaluation, or provides feedback to upstream RTL designers, who then modify the low-level code to make the placement task easier (for example, resolve timing violations).

To address the chip floorplanning problem, we developed an RL method capable of generalizing across chips—meaning that it can learn from experience to become both better and faster at placing

<sup>1</sup>Google Research, Brain Team, Google, Mountain View, CA, USA. <sup>2</sup>Google Chip Implementation and Infrastructure (CI2) Team, Google, Sunnyvale, CA, USA. <sup>3</sup>Computer Science Department, Stanford University, Stanford, CA, USA. <sup>4</sup>These authors contributed equally: Azalia Mirhoseini, Anna Goldie. ✉e-mail: azalia@google.com; agoldie@google.com



**Fig. 1 | Overview of our method and training regimen.** In each training iteration, the RL agent places macros one at a time (actions, states and rewards are denoted by  $a_i$ ,  $s_i$  and  $r_i$ , respectively). Once all macros are placed, the standard cells are placed using a force-directed method. The intermediate

rewards are zero. The reward at the end of each iteration is calculated as a linear combination of the approximate wirelength, congestion and density, and is provided as feedback to the agent to optimize its parameters for the next iteration.

new chips—allowing chip designers to be assisted by artificial agents with more experience than any human could ever gain.

Training placement policies that generalize across chips is extremely challenging, because it requires learning to optimize the placement of all possible chip netlists onto all possible canvases. Chip floorplanning is analogous to a game with varying pieces (for example, netlist topologies, macro counts, macro sizes and aspect ratios), boards (varying canvas sizes and aspect ratios) and win conditions (relative importance of different evaluation metrics or different density and routing congestion constraints). Even one instance of this game (placing a particular netlist onto a particular canvas) has an enormous state–action space. For example, the state space of placing 1,000 clusters of nodes on a grid with 1,000 cells is of the order of  $1,000!$  (greater than  $10^{2500}$ ), whereas Go has a state space of  $10^{360}$  (ref. <sup>15</sup>).

To enable generalization, we focused on learning transferable representations of chips, grounding representation learning in the supervised task of predicting placement quality. By designing a neural architecture that can accurately predict reward across a wide variety of netlists and their placements, we are able to generate rich feature embeddings of the input netlists. We then use this architecture as the encoder of our policy and value networks to enable transfer learning. In our experiments, we show that, as our agent is exposed to a greater volume and variety of chips, it becomes both faster and better at generating optimized placements for new chip blocks, bringing us closer to a future in which chip designers are assisted by artificial agents with vast chip placement experience.

In addition to the immediate impact on chip floorplanning, the ability of our method to generalize and quickly generate high-quality solutions has major implications, unlocking opportunities for co-optimization with earlier stages of the chip design process. Large-scale architectural explorations were previously impossible, because it took months of human effort to accurately evaluate a given architectural candidate. However, modifying the architectural design can have an outsized impact on performance, and would facilitate full automation of the chip design process. Automating and accelerating the chip design process can also enable co-design of AI and hardware, yielding high-performance chips customized to important workloads, such as autonomous vehicles, medical devices and data centres.

At an abstract level, our method learns to map the nodes of a hypergraph onto a limited set of resources, subject to constraints. Placement optimizations of this form appear in a wide range of science and engineering applications, including hardware design<sup>1</sup>, city planning<sup>16</sup>, vaccine testing and distribution<sup>17</sup>, and cerebral cortex layout<sup>18</sup>. Therefore, we believe that our placement optimization methodology can be applied to impactful placement problems beyond chip design.

Beyond the experimental results reported here, our method is already having real-world impact, and our floorplan solutions are in the product tapeout of a recent-generation Google tensor processing unit (TPU) accelerator.

### Chip floorplanning as a learning problem

The underlying problem is a high-dimensional contextual bandits problem<sup>19</sup> but, as in prior work, such as refs. <sup>20–23</sup>, we have chosen to

reformulate it as a sequential Markov decision process (MDP), because this allows us to more easily incorporate the problem constraints as described below. Our MDP consists of four key elements:

(1) States encode information about the partial placement, including the netlist (adjacency matrix), node features (width, height, type), edge features (number of connections), current node (macro) to be placed, and metadata of the netlist graph (routing allocations, total number of wires, macros and standard cell clusters).

(2) Actions are all possible locations (grid cells of the chip canvas) onto which the current macro can be placed without violating any hard constraints on density or blockages.

(3) State transitions define the probability distribution over next states, given a state and an action.

(4) Rewards are 0 for all actions except the last action, where the reward is a negative weighted sum of proxy wirelength, congestion and density, as described below.

We train a policy (an RL agent) modelled by a neural network that, through repeated episodes (sequences of states, actions and rewards), learns to take actions that will maximize cumulative reward (see Fig. 1). We use proximal policy optimization (PPO)<sup>24</sup> to update the parameters of the policy network, given the cumulative reward for each placement.

We can formally define the objective function as follows:

$$J(\theta, G) = \frac{1}{K} \sum_{g \in G} E_{g,p=\pi_\theta}[R_{p,g}]. \quad (1)$$

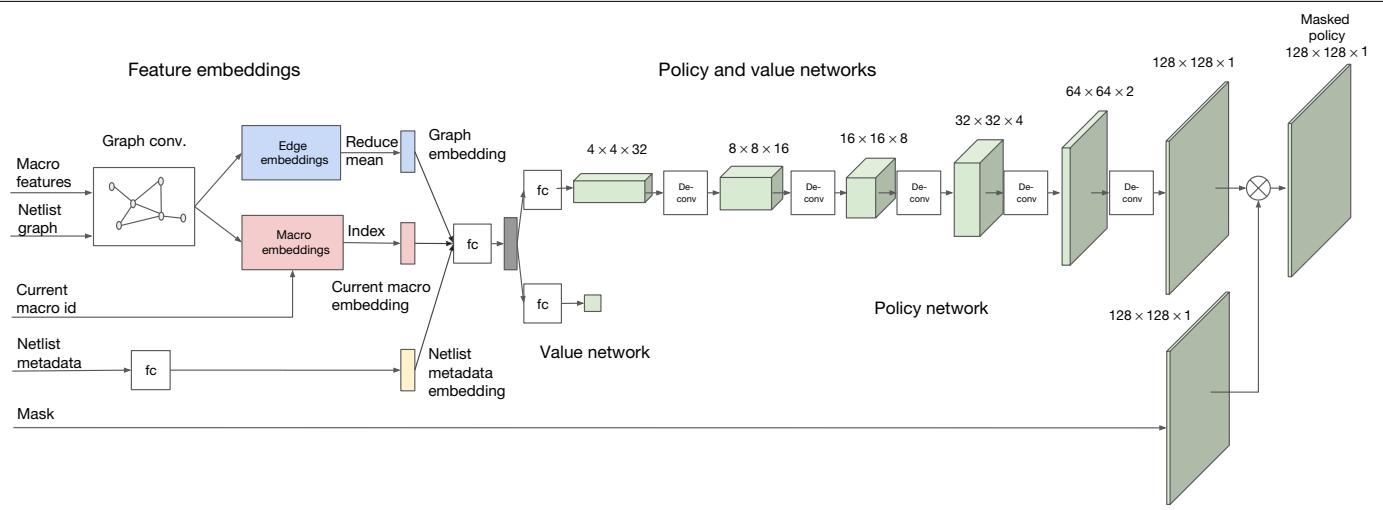
Here  $J(\theta, G)$  is the cost function. The agent is parameterized by  $\theta$ . The dataset of netlists of size  $K$  is denoted by  $G$ , with each individual netlist in the dataset written as  $g$ .  $R_{p,g}$  is the episode reward of a placement  $p$  drawn from the policy network applied to netlist  $g$ .  $E_{g,p}[R_{p,g}]$  is the expected reward, given a netlist  $g$  and placement  $p$  drawn from the policy distribution  $\pi_\theta$ .

$$R_{p,g} = -\text{Wirelength}(p,g) - \lambda \text{Congestion}(p,g) - \gamma \text{Density}(p,g). \quad (2)$$

In each iteration, the RL agent (policy network) sequentially places the macros. Once all macros are placed, we use a force-directed method<sup>11,25–27</sup> to approximately place clusters of standard cells. The reward at the end of each iteration is calculated as a linear combination of the approximate wirelength, congestion and density (equation (2)). In our experiments, the congestion weight  $\lambda$  is set to 0.01, the density weight  $\gamma$  is set to 0.01 and the maximum density threshold is set to 0.6.

### Designing domain-adaptive policies

As mentioned earlier, developing domain-adaptive policies for the chip floorplanning problem is extremely challenging, because this problem is analogous to a game with varying pieces, boards and win conditions, and has an enormous state–action space. To address this challenge, we first focused on learning rich representations of the state space. Our intuition was that a policy capable of the general task of chip placement should also be able to encode the state associated with a new unseen chip into a meaningful signal at inference time.



**Fig. 2 | Policy and value network architecture.** An embedding layer encodes information about the netlist adjacency, node features and the current macro to be placed. The policy and value networks then output a probability

distribution over available grid cells and an estimate of the expected reward for the current placement, respectively. id, identification number; fc, fully connected layer; de-conv, deconvolution layer.

We therefore trained a neural network architecture capable of predicting reward on placements of new netlists, with the ultimate goal of using this architecture as the encoder layer of our policy.

To train this supervised model, we needed a large dataset of chip placements and their corresponding reward labels. We therefore created a dataset of 10,000 chip placements where the input is the state associated with a given placement and the label is the reward for that placement.

To accurately predict the reward labels and generalize to unseen data, we developed an edge-based graph neural network architecture, which we call Edge-GNN (Edge-Based Graph Neural Network). The role of this network is to embed the netlist, distilling information about the type and connectivity of nodes into a low-dimensional vector representation that can be used in downstream tasks. The impact of our edge-based neural architecture on generalization is shown in Extended Data Fig. 2.

In Edge-GNN, we create an initial representation of each node by concatenating its features—including node type, width, height, x and

y coordinates, and its connectivity to other nodes. We then iteratively perform the following updates: (1) each edge updates its representation by applying a fully connected network to a concatenation of the two nodes that it connects, and (2) each node updates its representation by passing the mean of all in- and outgoing edges into another fully connected network. The node and edge updates are shown in equation (3).

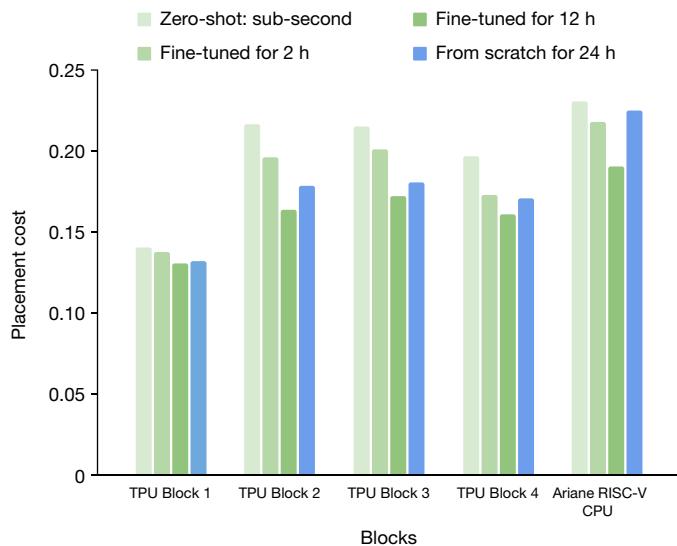
$$e_{ij} = \text{fc}_e(\text{concat}(v_i|v_j|w_{ij}^e)), \\ v_i = \text{mean}_{j \in \text{Neighbours}(v_i)}(e_{ij}). \quad (3)$$

Node embeddings are denoted by  $v_i$  for  $1 \leq i \leq N$ , where  $N$  is the total number of macros and standard cell clusters. Vector representations of edges connecting nodes  $v_i$  and  $v_j$  are denoted as  $e_{ij}$ .  $\text{fc}_e$  indicates a fully connected layer and  $w_{ij}^e$  corresponds to the learnable weight applied to edge  $e_{ij}$  (which connects nodes  $i$  and  $j$ ). The outputs of the algorithm are the node and edge embeddings.

**Table 1 | Comparisons against baselines**

Name	Method	Timing		Total area ( $\mu\text{m}^2$ )	Total power (W)	Wirelength (m)	Congestion	
		WNS (ps)	TNS (ns)				H (%)	V (%)
Block 1	RePLAce	374	233.7	1,693,139	3.70	52.14	1.82	0.06
	Manual	136	47.6	1,680,790	3.74	51.12	0.13	0.03
	Our method	84	23.3	1,681,767	3.59	51.29	0.34	0.03
Block 2	RePLAce	97	6.6	785,655	3.52	61.07	1.58	0.06
	Manual	75	98.1	830,470	3.56	62.92	0.23	0.04
	Our method	59	170	694,757	3.13	59.11	0.45	0.03
Block 3	RePLAce	193	3.9	867,390	1.36	18.84	0.19	0.05
	Manual	18	0.2	869,779	1.42	20.74	0.22	0.07
	Our method	11	2.2	868,101	1.38	20.80	0.04	0.04
Block 4	RePLAce	58	11.2	944,211	2.21	27.37	0.03	0.03
	Manual	58	17.9	947,766	2.17	29.16	0.00	0.01
	Our method	52	0.7	942,867	2.21	28.50	0.03	0.02
Block 5	RePLAce	156	254.6	1,477,283	3.24	31.83	0.04	0.03
	Manual	107	97.2	1,480,881	3.23	37.99	0.00	0.01
	Ours	68	141.0	1,472,302	3.28	36.59	0.01	0.03

Here, we compare our method with the state-of-the-art method (RePLAce<sup>14</sup>) and with manual placements using an industry-standard EDA tool. For all metrics in this table, lower is better. H, horizontal; V, vertical.



**Fig. 3 | Training from scratch versus fine-tuning for varying amounts of time.** For each block, we show zero-shot results, results after fine-tuning for 2 h and 12 h, and results for policies trained from scratch. As can be seen in the table, the pre-trained policy network consistently outperforms the policy network trained from scratch, demonstrating the effectiveness of learning from training data offline.

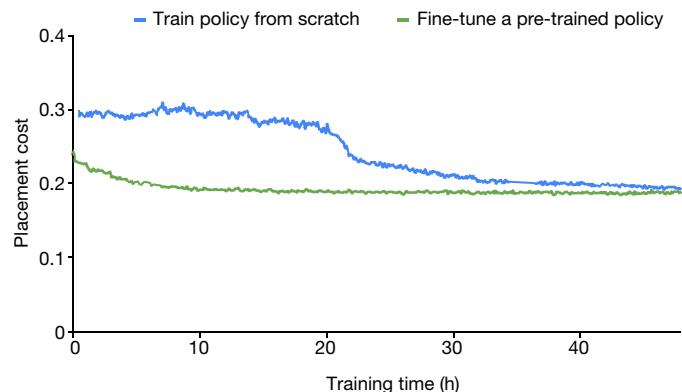
The supervised model is trained via regression to minimize the weighted sum of mean squared loss (negative reward). This supervised task allowed us to find the features and architecture necessary to generalize reward prediction across netlists. To incorporate Edge-GNN into our RL policy network, we removed the prediction layer and then used it as the encoder of the policy network, as shown in Fig. 2.

To place a new netlist at inference time, we load the pre-trained weights of the policy network and apply it to the new netlist. We refer to placements generated by a pre-trained policy with no fine-tuning as zero-shot placements. Such a placement can be generated in subsecond times, because it requires only a single forward pass through the pre-trained policy for each macro. We can further optimize placement quality by fine-tuning the policy network. Doing so gives us the flexibility to either use the pre-trained weights (which have learned a rich representation of the input state) or further fine-tune these weights to optimize for the properties of a particular chip netlist.

Figure 1 shows an overview of the proposed policy network (modelled by  $\pi_\theta$  in equation (1)) and value network architectures. The input is the netlist hypergraph (represented as an adjacency matrix and list of node features), the identity of the current node to be placed, the metadata of the netlist, and the process technology node (for example, 7 nm). The netlist is fed into our Edge-GNN architecture to generate embeddings of the partially placed netlist and of the current node. We use a feed-forward network to embed the metadata. These embedding vectors are then concatenated to form the state embedding, which is passed to another feed-forward neural network to generate a final representation of the state. This state is then fed into the policy network (composed of five deconvolutions, batch normalization<sup>28</sup> and rectified linear unit (ReLU) activation layers<sup>29</sup>) to generate a probability distribution over actions and into a value network (composed of a feed-forward network) to predict the value of the input state. The deconvolution layers have a kernel size of  $3 \times 3$ , a stride of 2, and 16, 8, 4, 2 and 1 filter channels.

### Empirical evaluation

In this section, we evaluate the ability of our method to generalize, explore the impact of using pre-trained policies and compare our



**Fig. 4 | Convergence plots on Ariane RISC-V CPU.** Placement cost of training a policy network from scratch versus fine-tuning a pre-trained policy network for a block of Ariane RISC-V CPU.

method to state-of-the-art baselines. We also inspect the visual appearance of generated placements and provide insights into the behaviour of our policy.

In terms of resource usage, for pre-training we used the same number of workers as blocks in the training dataset (for example, for the largest training set with 20 blocks, we pre-trained with 20 workers) and the pre-training runtime was 48 h. To generate the fine-tuning results in Table 1, our method ran on 16 workers for up to 6 h, but the runtime was often considerably lower owing to early stopping. For both pre-training and fine-tuning, a worker consists of an Nvidia Volta graphics processing unit (GPU) and 10 central processing units (CPUs), each with 2 GB of RAM. For the zero-shot mode (applying a pre-trained policy to a new netlist with no fine-tuning), we can generate a placement in less than a second on a single GPU.

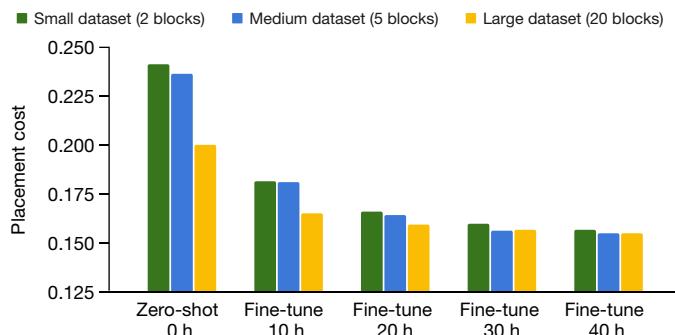
### Domain adaptation results

Figure 3 compares the quality of placements generated using pre-trained policies to those generated by training the policy from scratch. The training dataset is composed of blocks of TPU and of the open-source Ariane RISC-V CPU<sup>30</sup>. In each experiment, we pre-train the policy on all blocks except for the target block on which we evaluate. We show results for the zero-shot mode, as well as after fine-tuning the pre-trained policy on a particular design for 2 h and 12 h.

The policy trained from scratch takes much longer to converge, and even after 24 h the results (as evaluated by the reward function) are worse than what the fine-tuned policy achieves in 12 h. This demonstrates that exposure to many different designs during pre-training enables faster generation of higher-quality placements for new unseen blocks.

Figure 4 shows the convergence plots for training from scratch versus training from a pre-trained policy network for Ariane RISC-V CPU<sup>30</sup>. Not only does the pre-trained policy start with a lower placement cost, but it also converges more than 30 h faster than the policy trained from scratch.

**Learning from larger datasets.** In the following, we explore the impact of the training data on the learning ability of our policy. TPU chip blocks are quite diverse, and we carefully selected blocks across a representative range of functionalities (for example, on-chip and inter-chip network blocks, computation cores, memory controllers, data transport buffers and logic, and various interface controllers), saturations (ratio of total area of macros to that of the canvas, <30%, 30–60% and >60%) and macro counts (up to a few hundred). The small training set contains 2 blocks, the medium set contains 5 blocks and the large one contains 20 blocks. As we pre-train on more chip blocks, we are able to more quickly generate higher quality placements for new



**Fig. 5 | Effect of pre-training dataset size.** We pre-train the policy network on three different training datasets (the small dataset is a subset of the medium one, and the medium dataset is a subset of the large one). We then fine-tune this pre-trained policy network on the test block and report cost at various training durations. As the dataset size increases, both the time to convergence and the quality of generated placements increase.

unseen chip blocks. Figure 5 shows the impact of a larger training set on performance. As we increase the training set from 2 to 5 blocks, and finally to 20 blocks, the policy network generates better placements both at zero-shot and after being fine-tuned for the same number of hours. This suggests that as we expose the policy network to a greater variety of distinct chip designs, it becomes less prone to overfitting and better at generalizing to new unseen designs.

**Comparing with baseline methods.** In this section, we compare our method with the state-of-the-art RePLAe<sup>14</sup> and with the production design of the previous generation of TPU, which was generated by a team of human physical designers. The results are shown in Table 1.

To perform a fair comparison, we ensured that all methods had the same experimental setup, including the same inputs and the same EDA tool settings. We note that we ran all of the evaluations of RePLAe and our method ourselves, but we relied on the TPU physical design team to share metrics for their best-performing manual placements, and they may have evaluated with a slightly different EDA version. For more details, see Extended Data Table 1.

For our method, we use a policy pre-trained on the largest dataset (20 TPU blocks) and then fine-tune it on five target unseen blocks (denoted as blocks 1–5) for no more than 6 h. For confidentiality reasons, we cannot disclose the details of these blocks, but each contains up to a few hundred macros and millions of standard cells.

When evaluating the quality of a chip floorplan, there are several metrics that are important and that trade off against each other. There is no single metric that can be used to capture the overall quality of a placement, so we report all key metrics, including the total wirelength, timing, routing congestion (horizontal and vertical), area and power. Timing is reported via total negative slack (TNS) and worst negative slack (WNS). Negative slack is a measure of the extent to which the latency of the signal exceeds the expected latency. Timing and congestion are constraints, whereas wirelength, power and area are metrics to optimize.

To compare with RePLAe, which has a different objective function, we treat the output of a commercial EDA tool as ground truth. To perform this comparison, we fix the macro placements generated by our method and by RePLAe, and allow the commercial EDA tool to further optimize the standard cell placements with settings drawn from our production workflow. We used the version of RePLAe provided in ref.<sup>31</sup>, based on the version of the code published on 9 January 2020. Except for the density threshold (where RePLAe benefited from a lower threshold than its default), we used the default settings and did not use the timing-driven capability of RePLAe.

As shown in Table 1, our method outperforms RePLAe in generating placements that meet design criteria. Although RePLAe is

faster and runs in under an hour on a single Intel CPU of 3.7 GHz, the placements are generally of lower quality. Given the constraints imposed by the underlying process technology node, placements will not be able to meet timing constraints in the later stages of the design flow if WNS is considerably above 150 ps or if the horizontal or vertical congestion is over 1%, rendering many RePLAe placements (blocks 1, 2, 3) unusable. These results demonstrate that our approach is effective in generating high-quality placements that meet design criteria.

Table 1 also shows the results for the manual baseline, which is the actual production design of the previous TPU chip. This baseline was generated by the TPU's physical design team, and involved many iterations of placement optimization, guided by feedback from a commercial EDA tool over a period of several months. Both our method and human experts consistently generate viable placements that meet timing and congestion requirements. However, our method also outperforms or matches manual placements in area, power and wirelength. Furthermore, our end-to-end learning-based approach takes far less time to meet design criteria.

## Conclusion

In this work, we propose an RL-based approach to chip floorplanning that enables domain adaptation. The RL agent becomes better and faster at floorplanning optimization as it places a greater number of chip netlists. We show that our method can generate chip floorplans that are comparable or superior to human experts in under six hours, whereas humans take months to produce acceptable floorplans for modern accelerators. Our method has been used in production to design the next generation of Google TPU.

## Online content

Any methods, additional references, Nature Research reporting summaries, source data, extended data, supplementary information, acknowledgements, peer review information; details of author contributions and competing interests; and statements of data and code availability are available at <https://doi.org/10.1038/s41586-021-03544-w>.

- Markov, I. L., Hu, J. & Kim, M. Progress and challenges in VLSI placement research. *Proc. IEEE* **103**, 1985–2003 (2015).
- Tang, M. & Yao, X. A memetic algorithm for VLSI floorplanning. *IEEE Trans. Syst. Man Cybern. B* **37**, 62–69 (2007).
- Breuer, M. A. A class of min-cut placement algorithms. In *Proc. 14th Design Automation Conference (DAC 1977)* 284–290 (IEEE, 1977).
- Fiduccia, C. M. & Mattheyses, R. M. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference 175–181* (IEEE, 1982).
- Roy, J. A., Papa, D. A. & Markov, I. L. In *Modern Circuit Placement* (eds Nam, G.-J. & Cong, J. J.) 97–133 (Springer, 2007).
- Kirkpatrick, S., Gelatt, C. D. & Vecchi, M. P. Optimization by simulated annealing. *Science* **220**, 671–680 (1983).
- Sechen, C. M. & Sangiovanni-Vincentelli, A. L. TimberWolf3.2: a new standard cell placement and global routing package. In *23rd ACM/IEEE Design Automation Conference* 432–439 (IEEE, 1986).
- Sarrafzadeh, M., Wang, M. & Yang, X. In *Modern Placement Techniques* 57–89 (Springer, 2003).
- Luo, T. & Pan, D. Z. DPlace2.0: a stable and efficient analytical placement based on diffusion. In *2008 Asia and South Pacific Design Automation Conference* 346–351 (IEEE, 2008).
- Hu, B. & Marek-Sadowska, M. Multilevel fixed-point-addition-based VLSI placement. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **24**, 1188–1203 (2005).
- Viswanathan, N., Pan, M. & Chu, C. In *Modern Circuit Placement* (eds Nam, G.-J. & Cong, J. J.) 193–228 (Springer, 2007).
- Kim, M., Lee, D., Markov, I. L. & Sim, P. L. An effective placement algorithm. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **31**, 50–60 (2012).
- Lu, J. et al. ePlace: electrostatics-based placement using fast Fourier transform and Nesterov's Method. *ACM Trans. Des. Autom. Electron. Syst.* **20**, 17 (2015).
- Cheng, C.-K., Kahng, A. B., Kang, I. & Wang, L. RePLAe: advancing solution quality and routability validation in global placement. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **38**, 1717–1730 (2019).
- Silver, D. et al. Mastering the game of Go without human knowledge. *Nature* **550**, 354–359 (2017).
- Aslam, B., Arnjad, F. & Zou, C. C. Optimal roadside units placement in urban areas for vehicular networks. In *2012 IEEE Symposium on Computers and Communications (ISCC)* 000423–000429 (IEEE, 2012).

# Article

17. Medlock, J. & Galvani, A. P. Optimizing influenza vaccine distribution. *Science* **325**, 1705–1708 (2009).
18. Cherniak, C., Mokhtarzada, Z., Rodriguez-Esteban, R. & Changizi, K. Global optimization of cerebral cortex layout. *Proc. Natl Acad. Sci. USA* **101**, 1081–1086 (2004).
19. Langford, J. & Zhang, T. The Epoch-Greedy algorithm for multi-armed bandits with side information. In *Advances in Neural Information Processing Systems* Vol. 20, 817–824 (2008).
20. Usunier, N., Synnaeve, G., Lin, Z. & Chintala, S. Episodic exploration for deep deterministic policies: an application to starcraft micromanagement tasks. In *Proc. International Conference on Learning Representations* (2017).
21. Bello, I., Pham, H., Le, Q. V., Norouzi, M. & Bengio, S. Neural combinatorial optimization with reinforcement learning. Preprint at <https://arxiv.org/abs/1611.09940> (2016).
22. Mirhoseini, A. et al. Device placement optimization with reinforcement learning. In *Proc. International Conference on Machine Learning* 2430–2439 (PMLR, 2017).
23. Mirhoseini, A. et al. A hierarchical model for device placement. In *Proc. International Conference on Learning Representations* (2018).
24. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. Proximal policy optimization algorithms. Preprint at <https://arxiv.org/abs/1707.06347> (2017).
25. Obermeier, B., Ranke, H. & Johannes, F. M. Kraftwerk: a versatile placement approach. In *Proc. 2005 International Symposium on Physical Design* 242–244 (ACM, 2005).
26. Spindler, P., Schllichtmann, U. & Johannes, F. M. Kraftwerk2 – a fast force-directed quadratic placement approach using an accurate net model. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **27**, 1398–1411 (2008).
27. Viswanathan, N. et al. RQL: global placement via relaxed quadratic spreading and linearization. In *Proc. Design Automation Conference* 453–458 (ACM/IEEE, 2007).
28. Ioffe, S. & Szegedy, C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proc. 32nd International Conference on Machine Learning* 448–456 (JMLR, 2015).
29. Nair, V. & Hinton, G. E. Rectified linear units improve restricted Boltzmann machines. In *Proc. International Conference on Machine Learning* 807–814 (Omnipress, 2010).
30. Zaruba, F. & Benini, L. The cost of application-class processing: energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **27**, 2629–2640 (2019).
31. RePLAce software – the OpenROAD project <https://github.com/The-OpenROAD-Project/RePLAce> (2020).

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© The Author(s), under exclusive licence to Springer Nature Limited 2021

## Methods

In the following, we provide details of the proposed methodologies.

### Problem statement

In this work, we target the chip floorplanning problem, in which the objective is to map the nodes of a netlist (a hypergraph describing the chip) onto a chip canvas (a bounded two-dimensional space), so that final power, performance and area (PPA) is optimized. In this section, we provide an overview of how we formulate the problem as an RL problem, followed by a detailed description of the reward function, action and state representations, policy architecture, and policy updates.

### Overview of our approach

We take a deep RL approach to the chip floorplanning problem, in which an RL agent (policy network) sequentially places the macros. Once all macros are placed, we use a force-directed (FD) method<sup>11,25–27</sup> to place clusters of standard cells, as shown in Fig. 1.

In this section, we define the reward  $r$ , state  $s$ , actions  $a$ , policy network architecture  $\pi_\theta(a|s)$  parameterized by  $\theta$ , and finally the optimization method that we use to train those parameters. In our setting, at the initial state,  $s_0$ , we have an empty chip canvas and an unplaced netlist. At each step one macro is placed, and the final state,  $s_T$ , corresponds to a completely placed netlist. Thus,  $T$  is equal to the total number of macros in the netlist. At each time step  $t$ , the agent begins in state  $s_t$ , takes an action ( $a_t$ ), arrives at a new state ( $s_{t+1}$ ), and receives a reward ( $r_t$ ) from the environment (0 for  $t < T$  and negative proxy cost for  $t = T$ ).

We define  $s_t$  to be a concatenation of features representing the state at time  $t$ , including a graph embedding of the netlist (including both placed and unplaced nodes), a node embedding of the current macro to place, metadata about the netlist, and a mask representing the feasibility of placing the current node onto each cell of the grid.

The action space is all valid placements of the  $t$ th macro, which is a function of the density mask. Action  $a_t$  is the cell placement of the  $t$ th macro predicted by the RL policy network.  $s_{t+1}$  is the next state, which includes an updated representation containing information about the newly placed macro, an updated density mask and an embedding for the next node to be placed. In our formulation,  $r_t$  is 0 for every time step except for the final,  $r_T$ , where it is a weighted sum of approximate wirelength, congestion and density.

Through repeated episodes (sequences of states, actions and rewards), the policy network learns to take actions that will maximize cumulative reward. We use PPO<sup>24</sup> to update the parameters of the policy network, given the cumulative reward for each placement.

### Detailed methodology

Our goal is to minimize PPA, subject to constraints on routing congestion and density. Our true reward is the output of a commercial EDA tool, including wirelength, routing congestion, density, power, timing and area. However, RL policies require 10,000s of examples to learn effectively, so it is critical that the reward function be fast to evaluate, ideally running in a few milliseconds. In order to be effective, these approximate reward functions must also be positively correlated with the true reward. Therefore, a component of our cost is wirelength, because it is not only much cheaper to evaluate, but also correlates with power and performance (timing).

To combine multiple objectives into a single reward function that can be optimized, we take the weighted sum of proxy wirelength, congestion and density, where the weights can be used to explore the trade-off between these metrics. While we treat congestion as a soft constraint (that is, lower congestion improves the reward function), we treat density as a hard constraint, masking out actions (grid cells to place nodes onto) the density of which exceeds the target density.

To keep the runtime per iteration small, we apply several approximations to the calculation of the reward function:

(1) We group millions of standard cells into a few thousand clusters using hMETIS<sup>32</sup>, a partitioning technique based on the minimum cut objective. Once all macros are placed, we use an FD method to place the standard cell clusters. Doing so enables us to generate an approximate but fast standard cell placement that facilitates policy network optimization.

(2) We discretize the grid to a few thousand grid cells and place the centre of macros and standard cell clusters onto the centre of the grid cells.

(3) When calculating wirelength, we make the simplifying assumption that all wires leaving a standard cell cluster originate at the centre of the cluster.

(4) To calculate the routing congestion cost, we consider only the average congestion of the top 10% most congested grid cells.

A chip netlist typically consists of hundreds of macros and millions of standard cells. Owing to their negligible area, standard cells can be approximated as points with zero area, allowing for analytic solvers to optimally place them with a small margin of error. Macros, on the other hand, have much larger area and cannot be optimally placed with these same analytic techniques. We chose to target macro placement, as it is a much more challenging problem, which previously required human experts to iterate for months to generate a high-quality placement.

**Synthesis of the input netlist.** We use a commercial tool to synthesize the netlist from RTL. Synthesis is physical-aware, in the sense that it has access to the floorplan size and the locations of the input/output pins, which were informed by inter- and intra-block-level information.

**Selection of grid rows and columns.** Given the dimensions of the chip canvas, there are many choices to discretize the two-dimensional canvas into grid cells. This decision affects the difficulty of optimization and the quality of the final placement. We limit the maximum number of rows and columns to 128. We treat choosing the optimal number of rows and columns as a bin-packing problem and rank different combinations of rows and columns by the amount of wasted space that they incur. We use an average of 30 rows and columns in our experiments.

**Selection of macro order.** To select the order in which the macros are placed, we sort macros by descending size and break ties using a topological sort. By placing larger macros first, we reduce the chance of there being no feasible placement for a later macro. The topological sort can help the policy network learn to place connected nodes close to one another. Another potential approach would be to learn to jointly optimize the ordering of macros and their placement, making the choice of which node to place next part of the action space. However, this enlarged action space would considerably increase the complexity of the problem, and we found that this heuristic worked in practice.

**Clustering of standard cells.** To quickly place standard cells to provide a signal to our RL policy, we first cluster millions of standard cells into a few thousand clusters. There has been a large body of work on clustering for chip netlists<sup>33–38</sup>. As has been suggested in the literature<sup>39</sup>, such clustering helps not only with reducing the problem size, but also helps to ‘prevent mistakes’ (for example, prevents timing paths from being split apart). We also provide the clustered netlist to each of the baseline methods with which we compare. To perform this clustering, we employed a standard open-source library, hMETIS<sup>39</sup>, which is based on multilevel hypergraph partitioning schemes with two important phases: (1) coarsening phase, and (2) uncoarsening and refinement phase.

**Generation of adjacency matrix.** To convert the netlist hypergraph into an adjacency matrix that can be consumed by the Edge-GNN encoder, we apply the following transformation. For each pair of nodes in the clustered netlist (either macros or clusters of standard cells), we generate an edge in the adjacency matrix with the following weight:

# Article

If the register distance between the two nodes is greater than 4, then no edge is created. Otherwise, we apply an exponentially decaying weight as the distance grows, starting with 1 if the distance is 0 and halved with each additional unit of distance.

**Placement of standard cells.** To place standard cell clusters, we use an approach similar to classic FD methods<sup>40</sup>. We represent the netlist as a system of springs that apply force to each node, according to the weight  $\times$  distance formula, causing tightly connected nodes to be attracted to one another. We also introduce a repulsive force between overlapping nodes to reduce placement density. After applying all forces, we move nodes in the direction of their force vector. To reduce oscillations, we set a maximum distance for each move.

**Postprocessing.** To prepare the placements for evaluation by a commercial EDA tool, we perform a simple legalization step to snap macros to the nearest power grid. We then fix the macro placements and use an EDA tool to place the standard cells and evaluate the placement.

## Reward

**Wirelength.** Following the literature<sup>40–43</sup>, we employ the half-perimeter wirelength (HPWL), the most commonly used approximation for wirelength. HPWL is defined as the half-perimeter of the bounding boxes for all nodes in the netlist. The HPWL for a given net (edge)  $i$  is:

$$\text{HPWL}(i) = (\max_{b \in i} \{x_b\} - \min_{b \in i} \{x_b\} + 1) + (\max_{b \in i} \{y_b\} - \min_{b \in i} \{y_b\} + 1). \quad (4)$$

Here  $x_b$  and  $y_b$  are the  $x$  and  $y$  coordinates of the end points of net  $i$ . The overall HPWL cost is then calculated by taking the normalized sum of all half-perimeter bounding boxes, as shown in equation (5). Here  $q(i)$  is a normalization factor that improves the accuracy of the estimate by increasing the wirelength cost as the number of nodes increases, where  $N_{\text{netlist}}$  is the number of nets. We calculate the total HPWL as follows:

$$\text{HPWL}(\text{netlist}) = \sum_{i=1}^{N_{\text{netlist}}} q(i) \text{HPWL}(i). \quad (5)$$

The wirelength also has the advantage of correlating with other important metrics, such as power and timing. Although our method does not optimize directly for these other metrics, it generates placements that meet design criteria with respect to power and timing (as shown in Table 1).

**Routing congestion.** We also followed convention in calculating proxy congestion<sup>44</sup>, using a simple deterministic routing based on the locations of the driver and loads on the net. The routed net occupies a certain portion of available routing resources (determined by the underlying semiconductor fabrication technology) for each grid cell through which it passes. We keep track of vertical and horizontal allocations in each grid cell separately. To smooth the congestion estimate, we run  $5 \times 1$  convolutional filters in both the vertical and horizontal direction. After all nets are routed, we take the average of the top 10% congestion values, drawing inspiration from the ABA10 metric in MAPLE<sup>44</sup>. The congestion cost in equation (2) is the top 10% average congestion calculated by this process.

**Density.** We treat density as a hard constraint, disallowing the policy network from placing macros in locations that would cause density to exceed the target ( $\text{max}_{\text{density}}$ ) or that would result in infeasible macro overlap. This approach has two benefits: (1) it reduces the number of invalid placements generated by the policy network, and (2) it reduces the search space of the optimization problem, making it more computationally tractable.

A feasible placement of a standard cell cluster must meet the following criterion: the density of placed items in each grid cell must not

exceed a given target density threshold ( $\text{max}_{\text{density}}$ ). We set this threshold to be 0.6 in our experiments to avoid over-utilization, which would render placements unusable. To meet this constraint, during each RL step, we calculate the current density mask, a binary  $m \times n$  matrix representing grid cells onto which we can place the centre of the current node without violating the density threshold. Before selecting an action, we first take the dot product of the mask and the policy network output and then sample from the resulting probability distribution over feasible locations. This approach prevents the policy network from generating placements with overlapping macros or dense standard cell areas. We also enable blockage-aware placements (such as clock straps) by setting the density function of the blocked areas to 1.

## Action representation

For policy optimization purposes, we convert the canvas into an  $m \times n$  grid. Thus, for any given state, the action space (or the output of the policy network) is the probability distribution of placements of the current macro over the  $m \times n$  grid. The action is then sampled from this probability distribution.

## State representation

Our state contains information about the adjacency matrix corresponding to the clustered netlist, its node features (width, height, type), edge features (number of connections), current node (macro) to be placed, and metadata of the netlist and the underlying technology (for example, routing allocations, total number of wires, macros and standard cell clusters). Next, we discuss how we process these features to learn effective representations for the chip floorplanning problem.

## Enabling transfer learning

To discover domain-adaptive architectures, we propose grounding the policy architecture search in the supervised task of predicting the value of reward functions. We take this approach because exploration would be far costlier in an RL setting, and the underlying complexity of training a domain-adaptive policy network would be prohibitively high, as it involves an immense state space encompassing all possible placements of all possible chips. Furthermore, different netlists and grid sizes can have very different properties, including differing numbers of nodes, macro sizes, netlist topologies and canvas widths and heights.

The intuition behind this approach is that a policy network architecture capable of transferring placement optimization across chips should also be able to encode the state associated with a new unseen chip into a meaningful signal at inference time. We therefore propose training a neural network architecture capable of predicting reward on new netlists, with the ultimate goal of using this architecture as the encoder layer of our policy network.

To train this supervised model, we needed a large dataset of chip floorplans and their corresponding reward labels. We therefore created a dataset of 10,000 chip floorplans where the input is the state associated with a given floorplan and the label is the reward for that floorplan (wirelength and congestion). We built this dataset by generating 2,000 floorplans for each of five TPU blocks. To collect diverse floorplans, we trained a vanilla policy network with various congestion weights (ranging from 0 to 1) and random seeds, and collected snapshots of floorplans throughout the course of policy training. An untrained policy network starts off with random weights and the generated floorplans are of low quality, but as the policy network trains, the quality of generated floorplans improves, allowing us to gather a diverse dataset with floorplans of varying quality.

To train a supervised model capable of accurately predicting wirelength and congestion labels and generalizing to unseen data, we developed a graph neural network architecture (Edge-GNN) to embed information about the netlist. The role of Edge-GNN is to distill information about the type and connectivity of a node into a low-dimensional vector representation that can be used in downstream tasks. Some examples

of such downstream tasks are node classification<sup>45</sup>, device placement<sup>46</sup>, link prediction<sup>47</sup> and design rule check (DRC) violations prediction<sup>48</sup>.

We create a vector representation of each node by first concatenating its features, including node type, width, height, and x and y coordinates. We also pass node adjacency information as input to our algorithm. We then repeatedly perform the following updates: (1) each edge updates its representation by applying a fully connected network to an aggregated representation of intermediate node embeddings, and (2) each node updates its representation by taking the mean of adjacent edge embeddings. The node and edge updates are shown in equation (3).

Node embeddings are denoted by  $v_i$  for  $1 \leq i \leq N$ , where  $N$  is the total number of macros and standard cell clusters. The vector representations of the edge connecting nodes  $v_i$  and  $v_j$  is represented as  $e_{ij}$ . Both edge ( $e_{ij}$ ) and node ( $v_i$ ) embeddings are 32-dimensional.  $v_i$  is initialized by passing the node features (type, width, height, x, y) through a feed-forward network.  $fc_e$  is a  $65 \times 32$  feed-forward network and  $w_j^e$  is a  $1 \times 1$  weight corresponding to the number of nets between adjacent nodes.  $\text{Neighbours}(v_i)$  denotes the neighbours of  $v_i$ . The outputs of the algorithm are the node and edge embeddings.

Our supervised model consists of the following. (1) The graph neural network (Edge-GNN) described earlier, which embeds information about node type and the netlist adjacency matrix. (2) A fully connected feed-forward network that embeds netlist metadata, including information about the underlying semiconductor technology (horizontal and vertical routing capacity), the total number of nets (edges), macros, and standard cell clusters, canvas size and number of rows and columns in the grid. (3) A fully connected feed-forward network (the prediction layer) whose input is a concatenation of the netlist adjacency matrix and metadata embeddings, and whose output is the reward prediction. The netlist embedding is created by applying a reduced mean function on the edge embeddings. The supervised model is trained via regression to minimize the weighted sum of the mean squared loss of wirelength and congestion.

This supervised task allowed us to find the features and architecture necessary to generalize reward prediction across netlists. To incorporate this architecture into our policy network, we simply removed the prediction layer and then used the remaining network as the encoder of the policy network, as shown in Fig. 2.

To handle different grid sizes corresponding to different choices of rows and columns, we set the grid size to  $128 \times 128$ , and mask the unused L-shaped section for grid sizes smaller than 128 rows and columns. To place a new test netlist at inference time, we load the pre-trained weights of the policy network and apply it to the new netlist. We refer to placements generated by a pre-trained policy network with no fine-tuning as zero-shot placements. Such a placement can be generated in less than a second, because it requires only a single inference step of the pre-trained policy network for each macro. We can further optimize placement quality by fine-tuning the policy network, meaning that we have the option to either use the pre-trained weights (which have learned a rich representation of the input state) directly at inference or further fine-tune these weights to optimize for the properties of a particular chip netlist.

## Policy network architecture

Figure 1 depicts an overview of the policy network (modelled by  $\pi_\theta$  in equation (1) and the value network architecture that we developed for chip floorplanning. The input to these networks is the adjacency matrix and node features corresponding to the netlist hypergraph, the identity of the current node to be placed, and the metadata of the netlist and the semiconductor technology. The netlist is passed through our graph neural network architecture (Edge-GNN) as described earlier. Edge-GNN generates embeddings of (1) the partially placed hypergraph and (2) the current node. We use a simple feed-forward network to embed (3) the metadata. These three embedding vectors are then concatenated to form the state embedding, which is passed to a feed-forward neural network. The output of the feed-forward network is then fed into

the policy network (composed of five deconvolutions, batch normalization, and ReLU activation layers) to generate a probability distribution over actions and passed to a value network (composed of a feed-forward network) to predict the value of the input state.

**Policy network update: training parameters  $\theta$ .** In equation (1), the objective is to train a policy network  $\pi_\theta$  that maximizes the expected value ( $E$ ) of the reward ( $R_{pg}$ ) over the policy network's placement distribution. To optimize the parameters of the policy network, we use PPO<sup>24</sup> with a clipped objective as shown below:

$$L^{\text{CLIP}}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)],$$

where  $\hat{E}_t$  represents the expected value at timestep  $t$ ,  $r_t$  is the ratio of the new policy and the old policy, and  $\hat{A}_t$  is the estimated advantage at timestep  $t$ .

## Experimental setup

To perform a fair comparison, we ensured that our method and all baseline methods had access to the same inputs and the same evaluation settings. Extended Data Fig. 1 shows the flow that we used to conduct the evaluations.

Once each method finishes placing the netlist, the macro locations are frozen and snapped to the power grid. Next, the EDA tool performs standard cell placement. The settings for the EDA tool are drawn directly from our production flow and thus we cannot share all details. The final metrics in Table 1 are reported after PlaceOpt, meaning that global routing has been performed by the EDA tool.

Clustering standard cells allowed our method to more effectively optimize the placement of macros. We therefore gave RePIAe access to clustered standard cells and found that its performance also improved, so we reported results of RePIAe on the netlist with clustered standard cells. Although RePIAe has a default density threshold of 1.0, we found that our setting of 0.6 resulted in better performance, so that is what we used to report RePIAe performance. In all other cases, we used the default settings and cost functions for RePIAe. For reproducibility, we provide all architectural details and hyperparameter settings for our RL algorithm in Extended Data Table 1, as well as for the FD method used to place standard cells in Extended Data Table 2.

The deconvolutions layers have a  $3 \times 3$  kernel size with stride 2 and 16, 8, 4, 2 and 1 filter channels. To cluster the standard cells for each chip block, we used hMETIS<sup>32</sup>, which partitions millions of standard cells into thousands of clusters. The hyperparameters for hMETIS are listed in Extended Data Table 3. For all other hMETIS hyperparameters, we simply use the default settings (see the hMETIS manual<sup>49</sup> for the values of these defaults and for more detailed information about each hyperparameter). We note that we use a licensed version of hMETIS but, to our knowledge, the same features are available in the open-source version.

To avoid overfitting, we employ an early stopping mechanism that halts RL training once the policy converges. More precisely, training stops when it has been two hours since the evaluation return improved by at least 0.5% over the best return so far.

## Open-source benchmark: Ariane RISC-V

For the Ariane benchmark, we used the following open-source design<sup>30</sup> (<https://github.com/pulp-platform/ariane>) and mapped all logical memories to physical memories of size  $256 \times 16$ , resulting in 133 macros. In Extended Data Fig. 4, we compare a placement generated by our method trained from scratch and one that was generated in zero-shot mode by a pre-trained policy.

## Use in a production setting

Our method was used in the product tapeout of a recent Google TPU. We fully automated the placement process through PlaceOpt, at which

# Article

point the design was sent to a third party for post-placement optimization, including detailed routing, clock tree synthesis and post-clock optimization. This is a standard practice for many hardware teams, and physical designers spend months iterating with commercial EDA tools to produce designs that meet the strict requirements to move to this next stage.

In the production flow, we use the same RL method described in Table 1 and the same EDA workflow to place standard cells. Although the RL placements were already comparable to manual designs, we performed an additional fine-tuning step with simulated annealing (SA) to further boost performance, which helped to improve macro orientation, as we do not currently perform macro mirroring in RL. Adding this fine-tuning step improved wirelength by an average of 1.07% (s.d. = 0.04%), slightly reduced timing (average 1.18 ns reduction in TNS; s.d. = 2.4 ns) and negligibly affected congestion (less than 0.02% variation in vertical or horizontal congestion in all cases). The resulting end-to-end runtime was 8 h on average. Since that production launch, we have replaced SA in our production workflow with a greedy postprocessing step that tunes the macro orientation in a few minutes, considerably reducing our end-to-end runtime without degrading quality.

## Impact of cost trade-offs

In Extended Data Table 4, we perform an ablation study to examine the impact of congestion weight on the quality of post-PlaceOpt results (final quality of result from the commercial EDA tool). As expected, increasing congestion weight improves both horizontal and vertical congestion up to a point, but results in wirelength degradation, due to the inherent trade-off between these two metrics. A congestion weight of 0.1 represents a ‘sweet spot’ in this case, as routing congestion is already low but wirelength has not yet overly degraded, which together contribute to lower TNS and WNS as well.

## Robustness to noise

To demonstrate the sensitivity of our method to noise, we performed eight runs of fine-tuning on the Ariane RISC-V block, each time with a different random seed, and we report the results (in proxy wirelength, congestion and density) in Extended Data Table 5. Our evaluations demonstrate that the choice of random seed had negligible impact on all the metrics, including proxy wirelength, congestion and density, with a standard deviation of 0.0022 in the overall cost across all runs.

## Generalization versus training data

As we train on more chip blocks, we are able to speed up the fine-tuning process on new blocks and generate higher-quality results faster. As discussed earlier, as we increase the training set from 2 blocks (small dataset) to 5 blocks (medium dataset) and finally to 20 blocks (large dataset), the policy network generates better placements both at zero-shot and after being fine-tuned for the same number of hours. Extended Data Fig. 3 shows the placement cost on one test block (a TPU block not included in training) as the policy network is being pre-trained. We can see that for the small training dataset, the policy network quickly overfits to the training data and performance on the test data degrades, whereas it takes longer for the policy network to overfit on the largest dataset, and the policy network pre-trained on this larger dataset yields better results on the test data. This plot suggests that as we expose the policy network to a greater variety of different training blocks, it will take longer for the policy network to pre-train, but the policy network will become less prone to overfitting and better at finding optimized placements for new unseen blocks.

## Insights and visualizations

Here we share some observations about our method’s behaviour that may provide insight into the metrics in Table 1. One observation is

that the RL policy tends to place macros on the same datapath close to each other, which results in better timing performance. The Edge-GNN encoder embeds the features of each node by iteratively averaging and applying nonlinear transformations to the node’s  $k$ -hop neighbouring nodes and edges, where  $k$  is the number of iterations applied. Therefore, one hypothesis is that the representation of nodes in a given datapath are similar to one another, causing our policy network to generate similar predictions about where they should be placed on the canvas. This naturally results in nodes in the same datapath being placed near to one another, improving timing performance.

Another observation is that our policy learns to reserve sufficient area for the subsequent placement of standard cells, as this effectively optimizes its reward function. Even at zero-shot (meaning that we run inference on our policy network in less than one second), our method already exhibits this behaviour, as shown in Extended Data Fig. 4.

Extended Data Fig. 5 juxtaposes a placement generated by a human physical designer (on the left) with that of our method (on the right) for a recent TPU block. The white area shows the macro placements and the green area shows the standard cell placements. Our method creates donut-shaped placements of macros surrounding standard cells, which results in a reduction in the total wirelength. These placement images are blurred to preserve confidentiality.

## Comparing with simulated annealing

In the main text, we compare our method with two baselines: the academic state-of-the-art RePIAe and human expert placements. In this section, we provide an additional comparison with SA. To generate results for our method, we use the same procedure as in Table 1, pre-training a policy on the largest dataset (20 TPU blocks) and then fine-tuning it on the same five unseen test blocks.

SA is known to be a powerful, but slow, optimization method. However, similar to RL, SA is capable of optimizing arbitrary non-differentiable cost functions. To show the relative sample efficiency of RL, we ran experiments in which we replaced it with an SA optimizer. Our SA algorithm works as follows: in each SA iteration (step), we perform  $2N$  macro actions (where  $N$  is the number of macros). A macro action takes one of three forms: swap, shift and mirror. Swap selects two macros at random and swaps their locations, if feasible. Shift selects a macro at random and shifts that macro to a neighbouring location (left, right, up or down). Mirror flips a macro at random across the  $x$  axis, across the  $y$  axis, or across both the  $x$  and  $y$  axes. We apply a uniform probability over the three move types, meaning that at each time step there is a  $1/3$  chance of swapping, a  $1/3$  chance of shifting and a  $1/3$  chance of flipping. After  $N$  macro actions, we use an FD method to place clusters of standard cells while keeping macro locations fixed, just as we do in our RL method. For each macro action or FD action, the new state is accepted if it leads to a lower cost. Otherwise, the new state is accepted with a probability of  $\exp[\text{prev}_{\text{cost}} - \text{new}_{\text{cost}}]/t$ , where  $t = t_{\max} \exp\{-\log[(t_{\max}/t_{\min})] (\text{step}/\text{steps})\}$ . Here  $\text{prev}_{\text{cost}}$  refers to the cost at the previous iteration;  $\text{new}_{\text{cost}}$  refers to the cost at the current iteration;  $t$  is the temperature, which controls the willingness of the algorithm to accept local degradations in performance, allowing for exploration; and  $t_{\max}$  and  $t_{\min}$  correspond to the maximum and minimum allowable temperature, respectively.

To make comparisons fair, we ran 80 SA experiments sweeping different hyperparameters, including maximum temperature ( $10^{-5}, 3 \times 10^{-5}, 5 \times 10^{-5}, 7 \times 10^{-5}, 10^{-4}, 2 \times 10^{-4}, 5 \times 10^{-4}, 10^{-3}$ ), maximum SA episode length ( $5 \times 10^4, 10^5$ ) and seed (five different random seeds), and report the best results in terms of proxy wirelength and congestion costs in Extended Data Table 6. Each of the 80 SA workers runs an experiment corresponding to one particular choice of the five random seeds, two episode lengths and eight maximum temperatures.

The SA baseline uses more compute ( $80 \text{ SA workers} \times 2 \text{ CPUs per SA worker} \times 18 \text{ h of runtime} = 2,880 \text{ CPU-hours}$ ) than our method ( $16 \text{ RL workers} \times (1 \text{ GPU} + 10 \text{ CPUs per RL worker}) \times 6 \text{ h} = 1,920 \text{ CPU-hours}$ ). Here, we treat the cost of one GPU as roughly 10 times that of a CPU.

If we had stopped SA after 12 h (and if we did not use early stopping in RL), then the two methods would have used equivalent compute, but the SA results after 12 h were not even close to competitive. Even with additional time (18 h of SA versus 6 h of RL), SA struggles to produce high-quality placements compared to our approach, generating placements with 14.4% higher wirelength and 24.1% higher congestion on average.

### Implications for a broader class of problems

We believe that the proposed method has broader implications for other stages of chip design and other placement optimization tasks. For example, our zero-shot mode allows design space explorations through rapid evaluation of computer architectures grounded in the physical reality. Automating and optimizing architectural exploration and its interface with physical design can not only further accelerate the chip design process, but also lead to additional improvements in critical hardware metrics, such as power and timing.

Furthermore, this method is applicable to a broad class of placement optimization problems outside of chip design, such as city planning (for example, traffic light placement), compiler optimization (for example, datacentre resource allocation) and environmental engineering (for example, dam placement).

### Related work

Chip floorplanning is a longstanding challenge, requiring multi-objective optimization over circuits of ever-growing complexity. Since the 1960s, many approaches have been proposed, so far falling into three broad categories: (1) partitioning-based methods, (2) stochastic/hill-climbing methods and (3) analytic solvers.

Starting in the 1960s, industry and academic laboratories used partitioning-based<sup>34,50</sup> and resistive-network<sup>51,52</sup> approaches to chip floorplanning, as well as resistive-network based methods<sup>51,52</sup>. These methods are characterized by a divide-and-conquer approach; the netlist and the chip canvas are recursively partitioned until sufficiently small sub-problems emerge, at which point the sub-netlists are placed onto the sub-regions using optimal solvers. Such approaches are quite fast to execute and their hierarchical nature allows them to scale to arbitrarily large netlists. However, by optimizing each sub-problem in isolation, partitioning-based methods sacrifice quality of the global solution, especially routing congestion. Furthermore, a poor early partition may result in an unsalvageable end placement.

In the 1980s, analytic approaches emerged but were quickly overtaken by stochastic/hill-climbing algorithms, particularly SA<sup>6–8</sup>. SA is named for its analogy to metallurgy, in which metals are first heated and then gradually cooled to induce, or anneal, energy-optimal crystalline surfaces. SA applies random perturbations to a given placement (for example, shifts, swaps or mirroring of macros) and then measures their effect on the objective function (for example, HPWL). If the perturbation is an improvement, it is applied; if not, it is still applied with some probability, referred to as temperature. Temperature is initialized to a particular value and is then gradually annealed to a lower value. Although SA generates high-quality solutions, it is very slow and difficult to parallelize, thereby failing to scale to the increasingly large and complex circuits of the 1990s and beyond.

The 1990s–2000s were characterized by multi-level partitioning methods<sup>5,53</sup>, as well as the resurgence of analytic techniques<sup>54</sup>, such as FD methods<sup>9–11,25–27</sup> and nonlinear optimizers<sup>55–58</sup>. The renewed success of quadratic methods was due in part to algorithmic advances but also to the large size of modern circuits (10–100 million nodes), which justified approximating the placement problem as that of placing nodes with zero area. However, despite the computational efficiency of quadratic methods, they are generally less reliable and produce lower-quality solutions than their nonlinear counterparts.

Nonlinear optimization approximates cost using smooth mathematical functions, such as the log-sum-exp<sup>59</sup> and weighted-average<sup>65</sup>

models for wirelength, as well as Gaussian<sup>60</sup> and Helmholtz models for density. These functions are then combined into a single objective function using a Lagrange penalty or relaxation. Owing to the higher complexity of these models, it is necessary to take a hierarchical approach, placing clusters rather than individual nodes, an approximation that degrades the quality of the placement.

The last decade has seen the rise of modern analytic techniques, including more advanced quadratic methods<sup>12,44,61–63</sup>, and more recently electrostatics-based methods such as ePlace<sup>13</sup> and RePlAce<sup>14</sup>. Modeling netlist placement as an electrostatic system, ePlace<sup>13</sup> proposed a new formulation of the density penalty in which each node (macro or standard cell) of the netlist is analogous to a positively charged particle whose area corresponds to its electric charge. In this setting, nodes repel each other with a force proportional to their charge (area), and the density function and gradient correspond to the system's potential energy. Variations of this electrostatics-based approach have been proposed to address standard cell placement<sup>13</sup> and mixed-size placement<sup>64,65</sup>. RePlAce<sup>14</sup> is a recent state-of-the-art mixed-size placement technique that further optimizes ePlace's density function by introducing a local density function that tailors the penalty factor for each individual bin size. DREAMPlace<sup>66</sup> further speeds up RePlAce by taking a deep-learning-based approach to optimize the placement and leveraging GPU acceleration. However, the focus of DREAMPlace is standard cell placement optimization, rather than macro placement, and reports comparable quality to RePlAce. Therefore, we compare the performance of our method against RePlAce.

There are numerous opportunities for machine learning to advance physical design<sup>67–69</sup>. Recent work<sup>70</sup> proposes training a model to predict the number of DRC violations for a given macro placement. DRCs are rules that ensure that the placed and routed netlist adheres to tape-out requirements. To generate macro placements with fewer DRCs, ref. <sup>70</sup> uses the predictions from this trained model as the evaluation function in simulated annealing. Although this work represents an interesting direction, it reports results on netlists with no more than six macros, far fewer than any modern block, and does not consider the effect of place and route optimizations, which can dramatically alter the number of DRCs. Furthermore, although adhering to the DRC criteria is a necessary condition, the primary objective of macro placement is to optimize for wirelength, timing (for example, WNS and TNS), power and area, and this work does not even consider these metrics.

To address this classic problem, we propose a new category of approach: an end-to-end learning-based method. This new approach is most closely related to analytic solvers, particularly nonlinear ones, in that all of these methods optimize an objective function via gradient updates. However, our approach differs from prior approaches in its ability to learn from past experience to generate higher-quality placements on new chips. Unlike existing methods that optimize the placement for each new chip from scratch, our work leverages knowledge gained from placing prior chips to become better over time. In addition, our method enables direct optimization of the target metrics, such as wirelength, density and congestion, without having to define convex approximations of those functions, as is done in other approaches<sup>13,14</sup>. Not only does our formulation make it easy to incorporate new cost functions as they become available, but it also allows us to weight their relative importance according to the needs of a given chip block (for example, timing-critical or power-constrained).

Domain adaptation is the problem of training policies that can learn across multiple experiences and transfer the acquired knowledge to perform better on new unseen examples. In the context of chip floorplanning, domain adaptation involves training a policy across a set of chip netlists and then applying that trained policy to a new unseen netlist. The use of deep learning for combinatorial optimization is an area of growing interest, including approaches to the Travelling Salesman problem<sup>21,71</sup>, neural architecture search<sup>72</sup> and model parallelism<sup>22,23</sup>. More recently, there has been work on domain adaptation for compiler

# Article

optimization<sup>46,73–75</sup> and the Maximum Cut problem<sup>76</sup>. Our approach not only leverages past experience to reduce training time, but also produces higher-quality results when exposed to more instances of the problems. To our knowledge, our method is the first deep RL approach used in production to solve a combinatorial optimization problem, namely, in the design of the latest generation of Google TPU.

## Data availability

The data supporting the findings of this study are available within the paper and the Extended Data.

## Code availability

The code used to generate these data is available from the corresponding authors upon reasonable request.

32. Karypis, G. & Kumar, V. Metis: a hypergraph partitioning package <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview> (1998).
33. Alpert, C. J., Hagen, L. W. & Kahng, A. B. A hybrid multilevel/genetic approach for circuit partitioning. In Proc. APCCAS'96 – Asia Pacific Conference on 1012 Circuits and Systems 298–301 (IEEE, 1996).
34. Caldwell, A. E., Kahng, A. B. & Markov, I. L. Improved algorithms for hypergraph 1014 bipartitioning. In Proc. 2000 Design Automation Conference 661–666 (IEEE, 2000).
35. Chen, H. et al. An algebraic multigrid solver for analytical placement with layout 1017 based clustering. In Proc. 40th annual Design Automation Conference 794–799 (ACM, 2003); 10.1145/77583.776034.
36. Alpert, C., Kahng, A., Nam, G.-J., Reda, S. & Villarrubia, P. A semi-persistent clustering technique for vlsi circuit placement. In Proc. 2005 International Symposium on Physical Design, 200–207 (ACM, 2005).
37. Fogaca, M., Kahng, A. B., Reis, R. & Wang, L. Finding placement-relevant clusters with fast modularity-based clustering. In Proc. 24th Asia and South Pacific Design Automation Conference 569–576 (ACM, 2019); <https://doi.org/10.1145/3287624.3287676>.
38. Fogaca, M. et al. On the superiority of modularity-based clustering for determining placement-relevant clusters. *Integration* **74**, 32–44 (2020).
39. Kahng, A. B. Futures for partitioning in physical design (tutorial). In Proc. 1998 International Symposium on Physical Design 190–193 (ACM, 1998); <https://doi.org/10.1145/274535.274563>.
40. Shahookar, K. & Mazumder, P. VLSI cell placement techniques. *ACM Comput. Surv.* **23**, 143–220 (1991).
41. Caldwell, A. E., Kahng, A. B., Mantik, S., Markov, I. L. & Zelikovsky, A. On wirelength estimations for row-based placement. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **18**, 1265–1278 (1999).
42. Kahng, A. B. & Xu, X. Accurate pseudo-constructive wirelength and congestion estimation. In Proc. 2003 International Workshop on System-Level Interconnect Prediction 61–68 (ACM, 2003); <https://doi.org/10.1145/639929.639942>.
43. Kahng, A. B. & Reda, S. A tale of two nets: studies of wirelength progression in physical design. In Proc. 2006 International Workshop on System-Level Interconnect Prediction 17–24 (ACM, 2006); <https://doi.org/10.1145/1117278.1117282>.
44. Kim, M.-C., Viswanathan, N., Alpert, C. J., Markov, I. L. & Ramji, S. MAPLE: Multilevel Adaptive Placement for Mixed-Size Designs. In Proc. 2012 ACM International Symposium on International Symposium on Physical Design 193–200 (ACM, 2012).
45. Nazi, A., Hang, W., Goldie, A., Ravi, S. & Mirhoseini, A. GAP: generalizable approximate graph partitioning framework. In International Conference on Learning Representations Workshop (2019).
46. Zhou, Y. et al. GDP: generalized device placement for dataflow graphs. Preprint at <https://arxiv.org/abs/1910.01578> (2019).
47. Zhang, M. & Chen, Y. Link prediction based on graph neural networks. In Proc. International Conference on Neural Information Processing 5171–5181 (Curran Associates Inc., 2018).
48. Xie, Z. et al. RouteNet: routability prediction for mixed-size designs using convolutional neural network. In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 1–8 (IEEE, 2018).
49. hMETIS – hypergraph and circuit partitioning manual <http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download>.
50. Dunlop, A. E. & Kernighan, B. W. A procedure for placement of standard-cell VLSI circuits. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **4**, 92–98 (1985).
51. Cheng, C. K. & Kuh, E. S. Module placement based on resistive network optimization. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **3**, 218–225 (1984).
52. Tsay, R.-S., Kuh, E. & Hsu, C.-P. Proud: a fast sea-of-gates placement algorithm. In Proc. Design Automation Conference 1988, 318–323 (IEEE, 1988).
53. Agnihotri, A., Ono, S. & Madden, P. Recursive bisection placement: Feng Shui 5.0 implementation details. In Proc. International Symposium on Physical Design 230–232 (ACM, 2005).
54. Alpert, C. et al. Analytical engines are unnecessary in top-down partitioning based placement. *VLSI Des.* **10**, 99–116 (2002).
55. Kahng, A. B., Reda, S. & Wang, Q. Architecture and details of a high quality, large-scale analytical placer. In *IEEE/ACM International Conference on Computer-Aided Design 2005* 891–898 (IEEE, 2005).
56. Kahng, A. B. & Wang, Q. An analytic placer for mixed-size placement and timing-driven placement. In *IEEE/ACM International Conference on Computer-Aided Design 2004* 565–572 (IEEE, 2004).
57. Kahng, A. B. & Wang, Q. Implementation and extensibility of an analytic placer. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **24**, 734–747 (2005).
58. Chen, T.-C., Jiang, Z.-W., Hsu, T.-C., Chen, H.-C. & Chang, Y.-W. A High-quality mixed-size analytical placer considering preplaced blocks and density constraints. In Proc. 2006 IEEE/ACM International Conference on Computer-Aided Design 187–192 (ACM, 2006).
59. Naylor, W., Donnelly, R. & Sha, L. Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer. US Patent US6301693B1 (2001).
60. Chen, T., Jiang, Z., Hsu, T., Chen, H. & Chang, Y. NTUplace3: an analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **27**, 1228–1240 (2008).
61. Kim, M.-C. & Markov, I. L. ComPLx: a competitive primal-dual Lagrange optimization for global placement. In *Design Automation Conference 2012* 747–755 (ACM, 2012).
62. Brenner, U., Struzyna, M. & Vygen, J. BonnPlace: placement of leading-edge chips by advanced combinatorial algorithms. *Trans. Comp.-Aided Des. Integ.Cir. Sys.* **27**, 1607–1620 (2008).
63. Lin, T., Chu, C., Shinnerl, J. R., Bustany, I. & Nedelchev, I. POLAR: placement based on novel rough legalization and refinement. In Proc. International Conference on Computer-Aided Design 357–362 (IEEE, 2013).
64. Lu, J. et al. ePlace-MS: electrostatics-based placement for mixed-size circuits. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **34**, 685–698 (2015).
65. Lu, J., Zhuang, H., Kang, I., Chen, P. & Cheng, C.-K. Eplace-3d: electrostatics based placement for 3d-ics. In *International Symposium on Physical Design 11–18* (ACM, 2016).
66. Lin, Y. et al. DREAMPlace: deep learning toolkit-enabled GPU acceleration for modern VLSI placement. In *Design Automation Conference 1–6* (ACM/IEEE, 2019).
67. Kahng, A. B. Machine learning applications in physical design: recent results and directions. In Proc. 2018 International Symposium on Physical Design 68–73 (ACM, 2018); <https://doi.org/10.1145/3177540.3177554>.
68. Kahng, A. B. Reducing time and effort in ic implementation: a roadmap of challenges and solutions. In Proc. 55th Annual Design Automation Conference (ACM, 2018); <https://doi.org/10.1145/3195970.3199854>.
69. Ajayi, T. et al. Toward an open-source digital flow: first learnings from the openroad project. In Proc. 56th Annual Design Automation Conference 2019 (ACM, 2019); <https://doi.org/10.1145/3316781.3326334>.
70. Huang, Y. et al. Routability-driven macro placement with embedded CNN-based prediction model. In *Design, Automation & Test in Europe Conference & Exhibition* 180–185 (IEEE, 2019).
71. Khalil, E., Dai, H., Zhang, Y., Dilksina, B. & Song, L. Learning combinatorial optimization algorithms over graphs. *Adv. Neural Inf. Process Syst.* **30**, 6348–6358 (2017).
72. Zoph, B. & Le, Q. V. Neural architecture search with reinforcement learning. In Proc. International Conference on Learning Representations (2017).
73. Addanki, R., Venkatakrishnan, S. B., Gupta, S., Mao, H. & Alizadeh, M. Learning generalizable device placement algorithms for distributed machine learning. *Adv. Neural Inf. Process Syst.* **32**, 3981–3991 (2019).
74. Paliwal, A. et al. Reinforced genetic algorithm learning for optimizing computation graphs. In Proc. International Conference on Learning Representations (2020).
75. Zhou, Y. et al. Transferable graph optimizers for ML compilers. Preprint at <https://arxiv.org/abs/2010.12438> (2021).
76. Barrett, T. D., Clements, W. R., Foerster, J. N. & Lvovsky, A. I. Exploratory combinatorial optimization with reinforcement learning. Preprint at <https://arxiv.org/abs/1909.04063> (2020).
77. Kipf, T. N. & Welling, M. Semi-supervised classification with graph convolutional networks. Preprint at <https://arxiv.org/abs/1609.02907> (2016).

**Acknowledgements** This project was a collaboration between Google Brain and the Google Chip Implementation and Infrastructure (CI2) Team. We thank M. Bellemare, C. Young, E. Chi, C. Stratakos, S. Roy, A. Yazdanbakhsh, N. Myung-Chul Kim, S. Agarwal, B. Li, S. Bae, A. Babu, M. Abadi, A. Salek, S. Bengio and D. Patterson for their help and support.

**Author contributions** A.G. and A.M. are co-first authors and the order of the names was determined by coin flip. M.Y., J.W.J., E.S., S.W. and Y.-J.L. were major contributors to this work. The following authors contributed to the overall evaluation and provided insights on physical design: E.J., O.P., A.N., J.P., A.T., K.S., W.H. and E.T. The following authors managed and advised on the project: Q.V.L., J.L., R.H., R.C. and J.D.

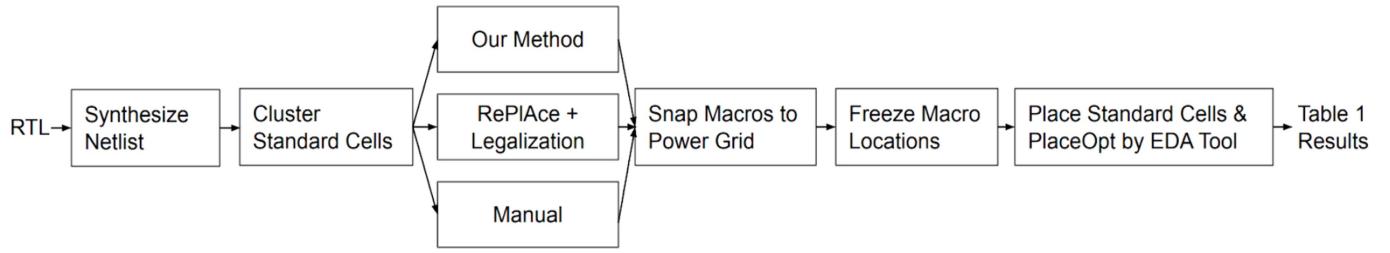
**Competing interests** The following US patents are related to this work: ‘Generating integrated circuit floorplans using neural networks’ (granted as US10699043) and ‘Domain adaptive reinforcement learning approach to macro placement’ (filed).

## Additional information

**Correspondence and requests for materials** should be addressed to A.M. or A.G.

**Peer review information** *Nature* thanks Jakob Foerster and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

**Reprints and permissions information** is available at <http://www.nature.com/reprints>.



**Extended Data Fig. 1 | Evaluation workflow for producing the results in**

**Table 1.** We allow each method access to the same clustered netlist hypergraph. We use the same hyperparameters (to the extent possible) in all the methods. Once the placement is completed by each method (this includes

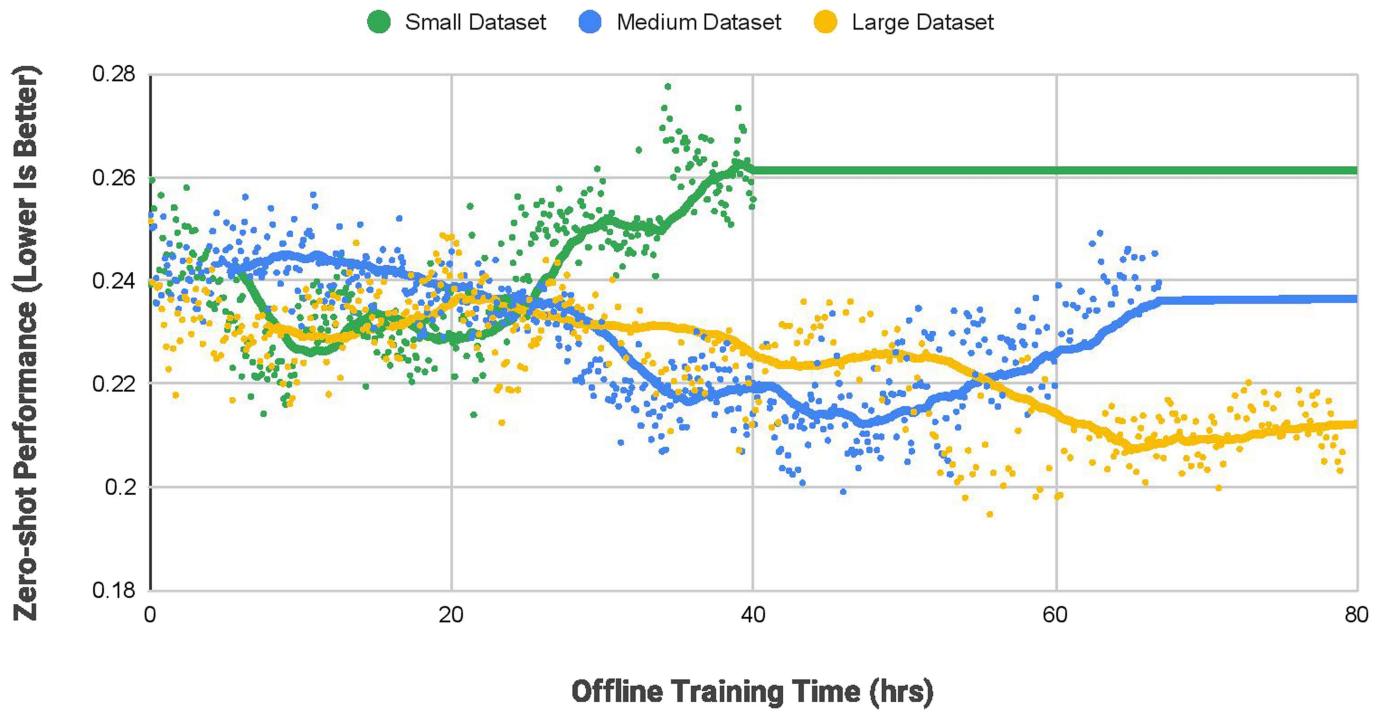
the legalization step for RePIAce), we snap the macros to the power grids, freeze the macro locations and use a commercial EDA tool to place the standard cells and report the final results.

## Article



**Extended Data Fig. 2 | Zero-shot performance of Edge-GNN versus GCN (graph convolutional neural network)<sup>77</sup>.** The agent with an Edge-GNN architecture is more robust to over-fitting and yields higher-quality results, as

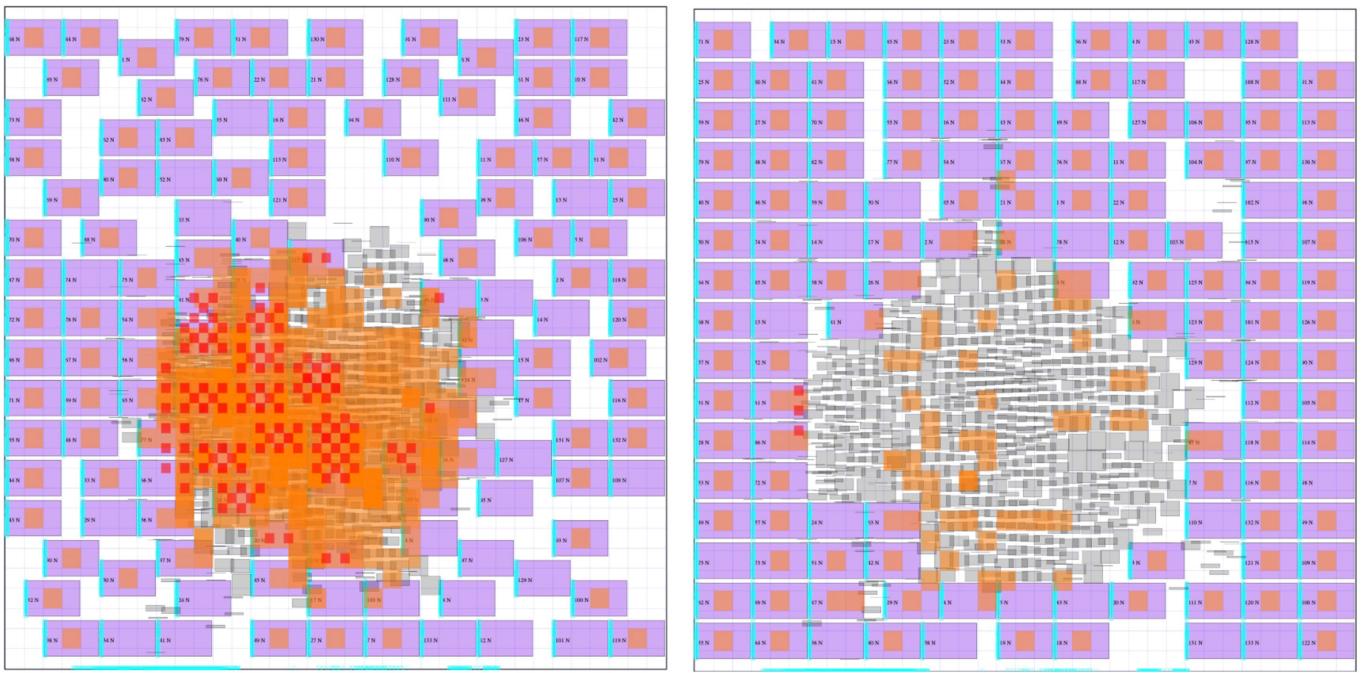
measured by average zero-shot performance on the test blocks shown in Extended Data Fig. 1.



**Extended Data Fig. 3 | Generalization performance as a function of pre-training dataset size.** We pre-train the policy network on three different training datasets (the small dataset with 2 blocks is a subset of the medium one with 5 blocks, and the medium dataset is a subset of the large one with 20 blocks). For each policy, at various snapshots during pre-training we report

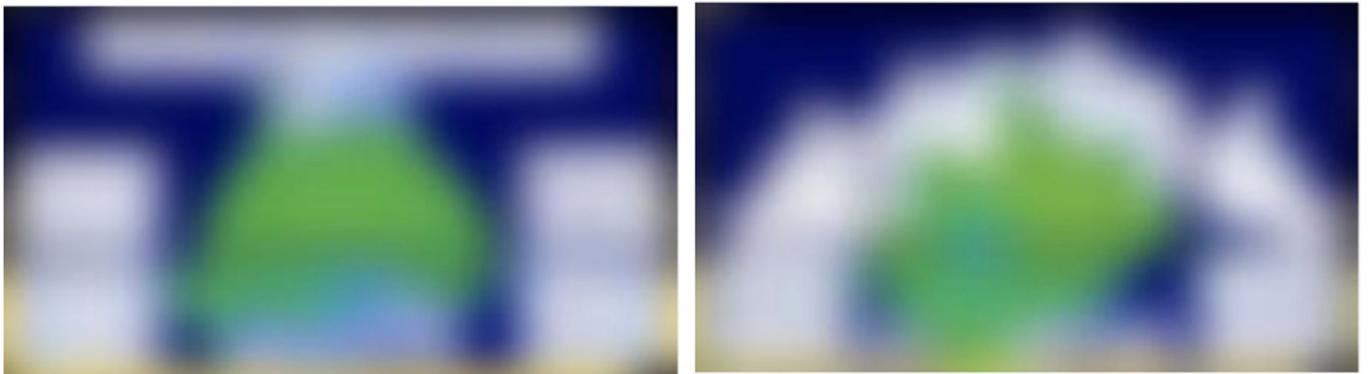
its inference performance on an unseen test block. As the dataset size increases, both the quality of generated placements on the test block and the generalization performance of the policy improve. The policy trained on the largest dataset is most robust to over-fitting.

# Article



**Extended Data Fig. 4 | Visualization of Ariane placements.** Left, zero-shot placements from the pre-trained policy; right, placements from the fine-tuned policy. The zero-shot placements are generated at inference time on a previously unseen chip. The pre-trained policy network (with no fine-tuning)

reserves a convex hull in the centre of the canvas in which standard cells can be placed, a behaviour that reduces wirelength and that emerges only after many hours of fine-tuning in the policy trained from scratch.



**Extended Data Fig. 5 | Visualization of a real TPU chip.** Human expert placements are shown on the left and results from our approach are shown on the right. The white area represents macros and the green area represents standard cells. The figures are intentionally blurred because the designs are

proprietary. The wirelength for the human expert design is 57.07 m, whereas ours is 55.42 m. Furthermore, our method achieves these results in 6 h, whereas the manual baseline took several weeks.

## Article

**Extended Data Table 1 | Hyperparameters used for fine-tuning the RL agent**

Hyperparameter	Value
Learning rate	1.00E-04
Optimizer	Adam
Num Epochs	4
Num GPUs	16
Num CPUs per GPU worker	10
Batch size per GPU	64
Effective batch size (Num GPUS × Batch size per GPU)	1024
Clip Grad	1
Num actors per GPU	32
Weight initializer	Xavier
Episodes per rollout	2
PPO loss:	
Clipping parameter ( $\epsilon$ )	0.2
Value Coeff	0.5
Entropy Coeff	0.01
Discount ( $\gamma$ )	1

The pre-training hyperparameters are the same, except for the number of GPUs and the effective batch size. For pre-training, we use one GPU per block in the training dataset (our largest dataset has 20 blocks).

**Extended Data Table 2 | Hyperparameters used for the FD algorithm that places standard cell clusters**

Hyperparameter	Value	Description
Number of schedules	3	Number of schedules to run the force-directed algorithm
Steps	[100, 100, 100]	Number of steps of the force-directed algorithm during each schedule
Move Distance Factors	[1.0, 1.0, 1.0]	Maximum distance relative to canvas size that a node can move in a single step of the force-directed algorithm
Attract Factors	[100, 1e-3, 1e-5]	The spring constants between two connected nodes in the force-directed algorithm
Repel Factors	[0, 1e6, 1e7]	The repellent factor for spreading the nodes to avoid congestion in the force-directed algorithm
I/O Factors	[1.0, 1.0, 1.0]	The I/O factor for multiplying the forces from the I/O ports to the nodes in the force-directed algorithm

## Article

**Extended Data Table 3 | Hyperparameters used to generate standard cell clusters with hMETIS<sup>32</sup>**

Name	Val	Definition
UBfactor	5	The extent to which unbalanced partitions are permitted.
Nruns	10	The number of bisections performed by hMETIS, the best of which is returned as the final solution.
CType	5	Edge Coarsening (EC) algorithm (heavy-edge maximal matching). In this mode, pairs of vertices are grouped together if they are connected by multiple hyperedges.
RType	3	Early-Exit Fiduccia-Mattheyses refinement scheme (FM-EE) algorithm. In this mode, the FM iteration is aborted if the quality of the solution does not improve after a relatively small number of vertex moves.
Vcycle	3	Performs Vcycle refinement on each intermediate solution, meaning that each one of the Nruns bisections is also refined using Vcycles.

---

**Extended Data Table 4 | Effect of different cost trade-offs on the post-PlaceOpt performance of Block 1 in Table 1**

Congestion Weight	WNS (ps)	TNS (ns)	Wirelength (m)	Congestion Horizontal (%)	Congestion Vertical (%)
0.01	163	22.85	48190736	0.30	0.03
0.1	154	11.80	50841227	0.06	0.03
1.0	118	34.73	53153141	0.07	0.02

As expected, increasing congestion weight improves both horizontal and vertical congestion up to a point, but results in wirelength degradation, owing to the inherent trade-off between these two metrics.

## Article

Extended Data Table 5 | Sensitivity of results to the choice of random seed, as measured on a Ariane RISC-V block

Seed	Proxy Wirelength	Proxy Congestion	Proxy Density
111	0.1187	0.9856	0.5780
222	0.1237	1.0251	0.5691
333	0.1207	0.9456	0.5714
444	0.1189	0.9559	0.5681
555	0.1174	0.9168	0.5561
666	0.1187	0.9676	0.5815
777	0.1200	0.9693	0.5772
888	0.1199	1.0087	0.5819
<b>mean</b>	0.1198	0.9718	0.5729
<b>std</b>	0.0019	0.0346	0.0086

We observed little sensitivity to random seed in all of the three cost functions, although variations in wirelength and density are lower than congestion. The overall cost for these eight runs had a standard deviation of 0.0022.

---

**Extended Data Table 6 | Performance of our method compared to SA**

	Replacing Deep RL with SA in our framework		Ours	
	Proxy Wirelength	Proxy Congestion	Proxy Wirelength	Proxy Congestion
Block 1	0.048	1.21	0.047	0.87
Block 2	0.045	1.11	0.041	0.93
Block 3	0.044	1.14	0.034	0.96
Block 4	0.030	0.87	0.024	0.78
Block 5	0.045	1.29	0.038	0.88

Proxy wirelength and congestion for each block. We note that because these proxy metrics are relative, comparisons are only valid for different placements of the same block. Even with additional time (18 h of SA versus 6 h of RL), SA generates placements with 14.4% higher wirelength and 24.1% higher congestion on average.