BiG-Flow: Program-Conditioned Discrete Flow Matching for High-Level Synthesis Pragma Design

Anonymous authors
Paper under double-blind review

Abstract

High-level synthesis (HLS) allows software developers to build FPGA accelerators from C/C++ kernels, but the quality of the resulting hardware depends on the inserted pragmas that take hardware expertise to choose, and every candidate design needs minutes to hours of synthesis. Learned HLS tools predict the quality of results (QoR) of a design and then search the design space of each new kernel with that predictor, which costs many predictions and synthesis runs per kernel and generalizes poorly to kernels outside the training set. We present BiG-Flow, a bidirectional graph flow-matching model conditioned on the program. One network generates pragma designs for a requested QoR and predicts the QoR of a given design, directly and without per-kernel search or fine-tuning. BiG-Flow encodes the control and data flow graph of a kernel once as context, generates designs with a discrete flow over the pragma sites of the kernel, predicts QoR with a continuous flow, and keeps every generated pragma within its legal values. We evaluate on an expanded HLSyn benchmark and 22 test kernels that are never seen in the training stage, with Vitis HLS as the oracle. BiG-Flow predicts latency with at least 17% lower error than state-of-the-art graph predictors. Given the QoR of a held-out design as the request, it returns a valid design within a factor of two of the requested latency for 58% of requests, against 41% for unconditional sampling. Asked for 10× to 1000× speedups with 20 synthesis runs per kernel, it reaches a geometric-mean speedup of 8.6× over the unoptimized design and up to 445× on a single kernel, which is 1.7× the speedup of random legal designs, 1.6× that of the strongest LLM-based baseline, and 1.5× that of AutoDSE given the same synthesis time. On one kernel it finds a design 13.4× faster than the best design of a full AutoDSE search.

1 Introduction

High-level synthesis (HLS) compiles a C/C++ program, called a kernel, into an FPGA accelerator (Cong et al., 2011; 2022). A software developer can write the kernel, but the speed and size of the resulting hardware are decided by optimization pragmas inserted into the code, such as loop pipelining and loop unrolling. Choosing pragmas well takes hardware expertise, and the effect of a choice is known only after synthesis, which takes minutes to hours for one design. The number of possible pragma designs grows exponentially with the number of loops in a kernel, each corresponding to one or two pragma sites, and a kernel with ten pragma sites already has millions of possible designs. Automatic pragma design for new kernels is therefore a central step toward making FPGAs usable by software developers (Sohrabizadeh et al., 2022b; Bai et al., 2023).

Most current methods search the design space of each kernel separately. AutoDSE runs the HLS tool inside a bottleneck-guided search and spends hundreds of synthesis runs and about a day of synthesis time per kernel (Sohrabizadeh et al., 2022b). Learned methods such as GNN-DSE (Sohrabizadeh et al., 2022a), HARP (Sohrabizadeh et al., 2023), and HierarchicalMoE (Li et al., 2025) train a graph neural network to predict the quality of results (QoR) of a design and use the prediction to rank candidates during a search. A predictor tells how good a given design is but not which design to try, so every new kernel still needs its own search, and predictors lose accuracy on kernels that differ from the training set, which HierarchicalMoE addresses by incorporating the Mixture-of-Experts structure and fine-tuning on labeled designs of the target kernel. Generative models have been placed inside this per-kernel loop as well, where DID4HLS learns a distribution of designs from the synthesis

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

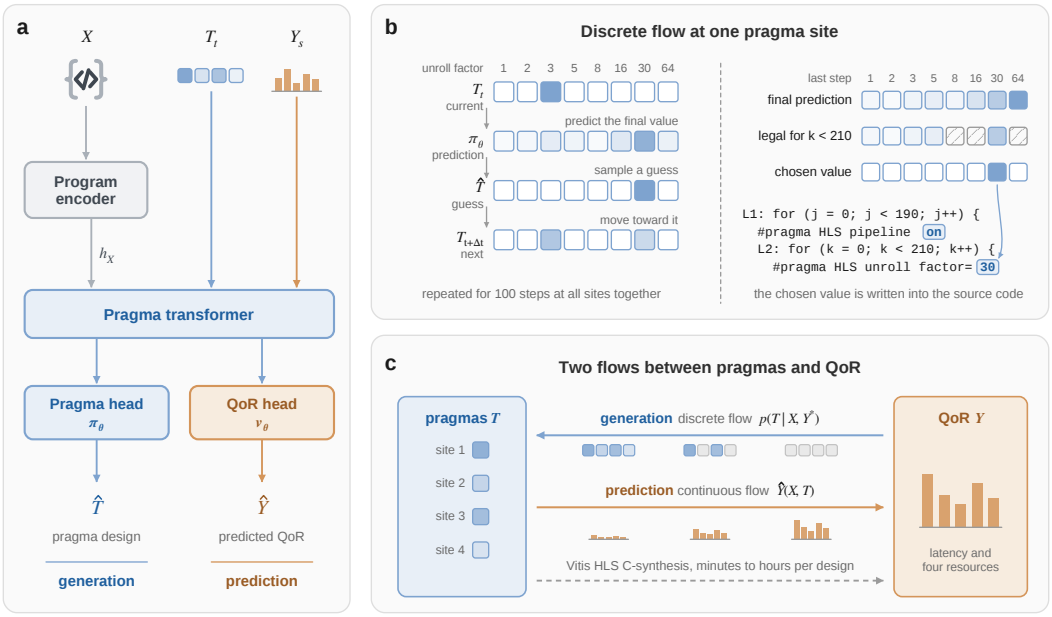

Figure 1: **Overview of BiG-Flow.** (a) A program encoder summarizes the CDFG of kernel X once. The pragma transformer, a graph transformer over the pragma sites, reads the current pragma values $\mathbf{T}_t$ and the paired QoR state $\mathbf{Y}_s$, its pragma head predicts the final value of every site, and its QoR head predicts the velocity of the QoR state. (b) The discrete flow at one pragma site. At each step the network predicts a distribution over the final unroll factor, a guess is sampled from it, and the site moves toward the guess, for 100 steps at all sites together. At the last step, values that are not legal options for the site are removed, here the factors that do not divide the trip count of 210, and the most likely legal value is written into the source code. (c) Pragma generation runs the discrete flow from noise to a design, conditioned on a requested QoR $Y^\star$. QoR prediction runs the continuous flow from zero to the QoR of a given design T. Vitis HLS synthesis measures the QoR of a design in minutes to hours.

results of the target kernel itself (Chang et al., 2024). Language-model methods such as ForgeHLS (Peng et al., 2025), LLM-DSE (Wang et al., 2025), and MailoHLS (Vouvali et al., 2026) propose pragmas for a new kernel from its source code, but they do not predict QoR, they accept at most a coarse objective such as low latency, and even with recent language models ForgeHLS and LLM-DSE find designs about half as fast as those of a full AutoDSE run (Section 5.4). No existing method learns, across kernels, one model that both predicts the QoR of a design and generates designs for a requested QoR on a kernel it has never seen.

We observe that QoR prediction and pragma design are the two conditionals of one object, the joint distribution of pragma designs (T) and QoR (Y) for a given program (X), denoted as $p(T, Y|X)$. QoR prediction asks for $p(Y|X, T)$, and pragma design for a requested QoR asks for $p(T|X, Y)$. One model that learns both conditionals answers both questions, and when it is conditioned on the program and trained across many kernels, it can generate designs for a kernel it has never seen, with no search and no data from that kernel.

We introduce BiG-Flow, short for Bidirectional Graph Flow matching, which learns both conditionals with flow matching (Figure 1). Its structure follows the structure of the problem. The control and data flow graph (CDFG) of a kernel is large but is the same for every design of that kernel, so BiG-Flow encodes it once and uses it as context. What varies between designs is the value at each pragma site. BiG-Flow generates these values with a discrete flow on a small graph of pragma sites, conditioned on the requested QoR, and predicts QoR with a continuous flow conditioned on the design. One graph transformer drives both flows, and a final decoding step keeps every pragma within its legal values.

We evaluate BiG-Flow on an expanded version of HLSyn (Bai et al., 2023) with Vitis HLS as the oracle. Every test kernel is held out from training, and every generated design is synthesized. BiG-Flow predicts latency more accurately than state-of-the-art graph neural networks, lowering the

kernel-averaged error of HARP by 17%, and its mean error over all five QoR coordinates is lower than that of HARP on 16 of the 22 test kernels. Given the measured QoR of a held-out design as a request, it returns a valid design within a factor of two of the requested latency for 58% of requests, against 41% for unconditional sampling from the same model, and it lowers the joint error over latency and four resources by 37%. Asked for speedups between 10× and 1000×, BiG-Flow reaches a geometric-mean speedup of 8.6× over the unoptimized design with only 20 synthesis runs per kernel, and up to 445× on a single kernel. This is 1.7× the speedup of random legal designs, 2.0× that of a design-space search guided by HARP, 1.6× that of the strongest language-model baseline, and 76% of the speedup that a full AutoDSE run reached with a median of 156 synthesis runs and 24.4 hours per kernel. Stopped after the synthesis hours that BiG-Flow used, AutoDSE reaches 5.8×. On one unseen kernel BiG-Flow finds a design 13.4× faster than the best design of the full AutoDSE run.

Our contributions are as follows.

- We identify QoR-conditioned pragma generation for unseen kernels as a problem that per-kernel search, QoR predictors, and language-model generators leave open. We formulate it together with QoR prediction as the two conditionals of the program-conditioned joint distribution $p(T, Y|X)$, so that one model can serve both tasks on a new kernel without search or data from that kernel.
- We propose BiG-Flow, which generates designs with a discrete flow over the pragma sites and predicts QoR with a continuous flow. Both flows share one network that reads the program as context. Compared with a separate generator and predictor, sharing costs neither task any accuracy, and the shared network proposes more distinct designs.
- On 22 unseen kernels with every generated design synthesized, BiG-Flow predicts latency more accurately than state-of-the-art graph predictors, follows requested QoR, and with 20 synthesis runs finds designs with a geometric-mean speedup of 8.6× over the default design, more than any other method at this budget, including AutoDSE and a language-model agent that both use synthesis feedback. A model trained across kernels can thus give a developer strong designs for a new kernel before any kernel-specific search.

2 Related Work

Machine learning for HLS design space exploration. Learned QoR predictors are the dominant abstraction in HLS design space exploration (DSE) (Schafer & Wang, 2020). Graph neural networks such as GNN-DSE (Sohrabizadeh et al., 2022a), HARP (Sohrabizadeh et al., 2023), and HierarchicalMoE (Li et al., 2025) predict performance from the program graph and rank candidates produced by an enumeration or search over the design space of one kernel. AutoDSE searches with the HLS tool in the loop and serves as the reference for learned DSE (Sohrabizadeh et al., 2022b; Li et al., 2025). Language-model methods such as LIFT (Prakriya et al., 2025), ForgeHLS (Peng et al., 2025), LLM-DSE (Wang et al., 2025), and MailoHLS (Vouvali et al., 2026) insert pragmas from the source text. BiG-Flow differs from all of these in that one model both predicts QoR and generates designs for an arbitrary requested QoR.

Generative models for design optimization. Generative models can replace per-instance search by learning a distribution of good solutions, as in DIFUSCO (Sun & Yang, 2023), GFlowNets (Zhang et al., 2023; Kim et al., 2024), and offline model-based optimization (Trabucco et al., 2022; 2021; Krishnamoorthy et al., 2023; Yuan et al., 2025). In hardware design, DID4HLS learns a design distribution from the synthesis results of the target kernel (Chang et al., 2024), DiffuSE applies diffusion to accelerator and EDA parameters (Ren et al., 2025), and FlowPlace applies flow matching to macro placement (Xie et al., 2026). These generators either learn from evaluations of the design space they optimize or rely on objectives and constraints that are cheap to compute. A new HLS kernel is a new design space whose objective requires synthesis, so neither assumption holds. BiG-Flow learns across kernels from an offline database, proposes designs for a new kernel without evaluating any of its designs, and predicts QoR with the same model. We compare with the generative approaches that propose designs for an unseen kernel before any of its designs is synthesized, namely language-model pragma generators, a conditional generator paired with a separate QoR predictor, which is the two-model design of offline model-based optimization, and the same BiG-Flow model sampled unconditionally. Methods that fit a model to the synthesis results of each target kernel solve a different problem. Appendix C extends this discussion, and Appendix M explores fine-tuning BiG-Flow with synthesis rewards in the style of Graph-GRPO (Zhu et al., 2026; Kim et al., 2025).

Discrete and bidirectional flows. Discrete diffusion (Vignac et al., 2023) and discrete flow matching (Gat et al., 2024; Campbell et al., 2024) generate categorical data such as graph labels, and DeFoG adapts discrete flow matching to graphs (Qin et al., 2025). MultiFlow couples discrete and continuous modalities in one network (Campbell et al., 2024), and Diagonal Flow Matching trains one network for both inverse design and prediction of continuous designs (de Campos et al., 2026). Section 3 describes how BiG-Flow builds on these formulations.

3 Preliminaries

HLS pragma design. We follow the notation of Li et al. (2025), in which a sample consists of a kernel X, a pragma design T, and a label Y. The kernel is represented by its CDFG G_X, and this graph is the same for every design of the kernel. The kernel exposes n_X pragma sites, each a fixed location in the source code that holds either a pipeline pragma or the unroll factor of a loop. Site $i \in \{1, \dots, n_X\}$ takes a value T_i from its legal options $\mathcal{A}_{X,i}$, which depend on the pragma type and on the trip count of the loop, and all sites draw their values from one shared vocabulary $\mathcal{A}$. A pragma design is the vector $T = (T_1, \dots, T_{n_X})$. Synthesizing kernel X with design T returns the label

$$Y = \left[\log_{10} \text{cycles},\ u_{\text{LUT}},\ u_{\text{FF}},\ u_{\text{DSP}},\ u_{\text{BRAM}}\right] \in \mathbb{R}^5,$$

where each u is the utilization of one resource, the fraction of the device capacity that the design uses, divided by 0.8. A design is valid when synthesis finishes and it uses at most 80% of every resource, including URAM, the constraint of HierarchicalMoE (Li et al., 2025) that leaves room for other logic on the device. The division by 0.8 turns this constraint into a normalized utilization of at most one. QoR prediction estimates Y for a given pair (X, T). Pragma generation samples designs T for a kernel X and a requested label $Y^\star$.

Flow matching. Flow matching (Lipman et al., 2023) learns to transport samples of a simple source distribution p_0 into samples of the data distribution p_1 along a path of intermediate distributions p_t, $t \in [0, 1]$. For continuous data, a common path pairs a source sample z_0 with a data sample z_1 and moves along the straight line $z_t = (1 - t)\, z_0 + t\, z_1$. A network $v_\theta(z_t, t)$ is trained to predict the constant velocity $z_1 - z_0$, and new samples are drawn by integrating $\mathrm{d}z/\mathrm{d}t = v_\theta(z, t)$ from $z_0 \sim p_0$ at $t = 0$ to $t = 1$. For BiG-Flow, the data are all synthesized designs of the training kernels and the source is a categorical distribution over pragma values (Section 4.1).

Discrete flow matching. Pragma values are categories, and a straight line between two one-hot vectors passes through fractional vectors that never occur when sampling (Appendix D.2). Discrete flow matching (Gat et al., 2024; Campbell et al., 2024) defines the path on the categories themselves. For one categorical variable with a data value a_1, the value that the path reaches at $t = 1$, and a source distribution p_0 over the categories, such as the uniform distribution, the mixture path of Gat et al. (2024) with the scheduler $\kappa_t = t$ and the source value averaged out is the path used by Campbell et al. (2024) (Appendix D),

$$p_t(a \mid a_1) = t\, \delta(a, a_1) + (1 - t)\, p_0(a), \tag{1}$$

where $\delta(a, b)$ is one if $a = b$ and zero otherwise. At time t, the state equals the data value with probability t and is otherwise drawn from p_0. A network $\pi_\theta(a_1 \mid z_t, t)$ is trained with a cross-entropy loss to recover the data value from a sample z_t of this path, and during sampling its prediction defines how much probability moves from the current value toward each candidate value in a small time step. DeFoG (Qin et al., 2025) applies this construction to the node labels of graphs.

Bidirectional flow matching. A generator of designs conditioned on a label solves the inverse problem, and a predictor of the label solves the forward problem. Diagonal Flow Matching (de Campos et al., 2026) trains one network for both. It pairs every design state with a label state that holds the full label while the design is still noise and is empty once the design is complete, so the network learns to generate a design from a label and to predict the label of a complete design. This construction was developed for continuous designs, where the flow is an invertible map and prediction is its inverse. BiG-Flow applies the pairing to categorical pragma designs on a graph, conditions it on the program, and restricts each pragma to its legal values. A map onto finitely many categorical designs cannot be inverted, so BiG-Flow runs generation and prediction as two directions of one network.

4 BiG-Flow

BiG-Flow serves the two conditionals of the joint distribution of pragma designs T and labels Y given a kernel X with one network. A discrete flow over the pragma sites generates designs from

$p(T \mid X, Y)$, and a continuous flow over the label predicts the QoR of a given design. BiG-Flow is bidirectional in that one network maps a requested QoR to designs and a design to its QoR. This section describes the two flows, the network, its training objective, and its use for both tasks.

4.1 Two conditional flows

The training samples are all synthesized designs of the training kernels, not only the fastest or Pareto-optimal ones. For a sample (X, T, Y) we define one flow for the pragmas and one for the QoR.

Pragma flow. The pragma state $\mathbf{T}_t$ assigns one value to every site at time $t \in [0, 1]$, and each site follows the mixture path of Equation 1 toward its value T_i in the design,

$$p_t(T_{t,i} = a \mid T_i) = t\,\delta(a, T_i) + (1 - t)\,p_0(a), \qquad a \in \mathcal{A}, \tag{2}$$

so that the state reaches the design at $t = 1$, $T_{1,i} = T_i$. The source distribution p_0 is the frequency of each value of $\mathcal{A}$ in the training designs, smoothed so that no value has zero probability. The posterior over each final value sets the velocity of this path, so learning it with cross-entropy is the discrete flow-matching objective (Campbell et al., 2024). Conditioned on the kernel and a requested label, this flow generates designs.

QoR flow. The QoR state follows the straight path from zero to the label,

$$\mathbf{Y}_s = s\,Y, \qquad s \in [0, 1], \tag{3}$$

whose velocity is Y. Conditioned on the kernel and a design, this flow predicts the label of the design.

Pairing. Following Diagonal Flow Matching (de Campos et al., 2026), the two flows share one time. Every training sample draws one t and pairs $\mathbf{T}_t$ with $\mathbf{Y}_s$ at $s = 1 - t$, which forms one flow from a random design with the full label to the complete design with an empty QoR state. Generation runs it forward from a request, and prediction runs it backward from a design, that is, the QoR flow forward in s. Clocks running together would never pair these two starting states.

4.2 Network

A network f_θ reads the pragma state, the paired QoR state, the time, and the kernel and returns two outputs,

$$f_\theta(\mathbf{T}_t, \mathbf{Y}_s, t, X) = \left(\{\pi_{\theta,i}\}_{i=1}^{n_X},\ v_\theta\right), \qquad s = 1 - t,$$

where the pragma head $\pi_{\theta,i}$ is a distribution over the final value of site i and the QoR head $v_\theta \in \mathbb{R}^5$ is the velocity of the QoR state. The kernel enters through its CDFG, which is the same for every design. We encode G_X once per kernel with the graph encoder of HARP (Sohrabizadeh et al., 2023), six edge-aware TransformerConv layers (Shi et al., 2021) with max jumping knowledge, and pool the node embeddings into a program vector $\mathbf{h}_X$. Unlike in HARP, no pragma values are attached to the CDFG, and the encoder is trained together with the rest of the network. The pragma values enter through the pragma transformer, a graph transformer in the style of DiGress (Vignac et al., 2023) with six layers of multi-head self-attention over all pragma sites of the kernel. Its graph has one node for every pragma site, 4 to 22 in Expanded HLSyn, and its edges follow the loop nest and modulate the attention between sites. Each site token carries its current value $T_{t,i}$ and static features such as pragma type, loop depth, trip count, and number of legal options. Every layer receives the program vector $\mathbf{h}_X$, the time t, and the QoR state $\mathbf{Y}_s$ as global conditioning. The pragma head and the QoR head are small two-layer perceptrons on the output of the last layer, which together hold less than 1% of the 6.75M parameters, so both flows share the program encoder and all six transformer layers. The pairing of Section 4.1 shows the network designs at every stage of completion together with the matching share of the label, so the same layers learn how pragma values and QoR determine each other in both directions. In Appendix B, the shared network matches separately trained networks in both prediction and request following and proposes more distinct designs, so sharing costs neither task any accuracy. Because the pragma graph is small, one network evaluation is cheap even for kernels with large CDFGs. Appendix B also compares this pooled program vector with fusing CDFG node embeddings into individual pragma sites.

4.3 Training objective

A training step draws a time t, samples a pragma state from Equation 2 at every site, and forms the paired QoR state of Equation 3 at $s = 1 - t$. The pragma loss is the cross-entropy of the final values

averaged over all sites of a minibatch $\mathcal{B}$,

$$\mathcal{L}_T = -\mathbb{E}_t \left[\frac{\sum_{(X,T,Y)\in\mathcal{B}} \sum_{i=1}^{n_X} \log \pi_{\theta,i}(T_i \mid \mathbf{T}_t, \mathbf{Y}_s, t, X)}{\sum_{(X,T,Y)\in\mathcal{B}} n_X} \right], \tag{4}$$

and the QoR loss regresses the velocity of the QoR state,

$$\mathcal{L}_Y = \mathbb{E}_t \left[\tfrac{1}{5} \left\| v_\theta(\mathbf{T}_t, \mathbf{Y}_s, t, X) - Y \right\|_2^2 \right]. \tag{5}$$

For $s > 0$ the QoR state $s\,Y$ reveals the label, and the network could fit $\mathcal{L}_Y$ by reading its own input. We therefore replace $\mathbf{Y}_s$ by zero on a random fraction p of the training samples while keeping the target Y. On these samples the network has to infer QoR from the kernel and the pragmas alone, which is exactly what QoR prediction asks of it. The rate p is chosen on validation kernels, and Appendix E lists the value used in each experiment. The training loss is $\mathcal{L} = \mathcal{L}_T + \lambda\,\mathcal{L}_Y$, and we use $\lambda = 1$. Every training sample updates both heads. Training reads only the design database and never calls the HLS tool.

4.4 Prediction and generation

Figure 1c shows how the network serves the two tasks.

QoR prediction. Given a kernel X and a design T, we start from a zero QoR state at $s = 0$ and integrate the QoR head to $s = 1$ with 100 Euler steps. The design stays the pragma input throughout, and the network receives the paired time $t = 1 - s$. The QoR state grows from zero to the prediction $\hat{Y} = \mathbf{Y}_1$. The pragma head is not used.

Pragma generation. Given a kernel X and a requested label $Y^\star$, we start at $t = 0$ from a pragma state drawn from p_0 at every site and integrate the pragma flow to $t = 1$ in 100 steps. At each step, the pragma head predicts the final value of every site, and the state of each site moves toward a value sampled from this prediction, which is the Euler step of discrete flow matching in expectation (Appendix D). The request enters as the paired QoR state, which either follows $(1-t)\,Y^\star$ or is updated with the QoR head from $Y^\star$, as listed per experiment in Appendix E. Setting the QoR input to zero at every step gives an unconditional sampler of the same model, which samples designs from $p(T \mid X)$ without a request and which the network learns from the training samples whose QoR state is replaced by zero. We use it as the control for conditioning, since it differs from conditional generation only in whether the model receives the request.

Legal decoding. The vocabulary $\mathcal{A}$ is shared across kernels, while site i admits only the options $\mathcal{A}_{X,i}$. For example, we only use unroll factors that divide the loop bound. At the final step we remove every value outside $\mathcal{A}_{X,i}$ and take the most likely remaining value. This applies a known constraint to the sampled trajectory and costs no additional synthesis run. The decoded values are inserted into the pragma placeholders of the source code.

5 Experiments

We evaluate prediction, request following, and fast design on unseen kernels. Ablation studies in Appendix B support the two main design choices, a pooled program vector and one network shared by both flows.

5.1 Setup

Dataset. We use an expanded version of the HLSyn benchmark (Bai et al., 2023), which we call Expanded HLSyn. HLSyn contains MachSuite kernels (Reagen et al., 2014) and some Polybench kernels (Pouchet & Yuki, 2016) at some of their problem sizes. Expanded HLSyn extends these Polybench kernels to all three problem sizes, small, medium, and large. We use 75 of its kernels and 26,153 designs explored by AutoDSE and synthesized with Vitis HLS.

Unseen kernels. All experiments test on kernels that are never seen during training, the setting in which learned HLS models are most useful and most fragile (Li et al., 2025). Each split divides the 75 kernels at random into 56 training, 8 validation, and 11 test kernels, and no design of a validation or test kernel is used for training. Splits 1 and 2 have disjoint test kernels, which gives 22 distinct unseen kernels, and we report their average metrics. Every model, including every baseline, selects its checkpoint on the validation kernels of its split. The synthesis-budget study and the ablations use a third split, split 3. Appendix E.2 lists all test kernels, and Appendices G and H report per-split results.

Table 1: QoR prediction on unseen kernels, averaged over splits 1 and 2. Entries are mean absolute errors of $\log_{10}$ cycles and of normalized resource utilization, averaged over all test designs, except the per-kernel latency error, which first averages within each kernel so that every kernel counts equally. The last column averages the five per-design errors. Lower is better, and bold marks the best entry.

	Latency		Resources				Mean of
Model	Per design	Per kernel	LUT	FF	DSP	BRAM	the five
HARP	0.374	0.660	0.168	0.105	0.236	0.093	0.195
HierarchicalMoE	1.005	0.972	0.187	0.093	**0.168**	0.150	0.321
BiG-Flow	**0.338**	**0.549**	**0.116**	**0.074**	0.174	**0.053**	**0.151**

Synthesis. Every generated design that is not already in the database is synthesized with Vitis HLS 2025.1 C-synthesis targeting an AMD Alveo U200 at 250 MHz with a 30-minute timeout and checked for validity as defined in Section 3. A design found in the database counts as one synthesis run, so no method gains budget from designs that AutoDSE already synthesized.

Baselines. For QoR prediction we compare with HARP (Sohrabizadeh et al., 2023) and HierarchicalMoE (Li et al., 2025), retrained on the same splits with their public code and full training schedules. For pragma generation, the control is BiG-Flow sampled unconditionally. For finding fast designs we add AutoDSE (Sohrabizadeh et al., 2022b), search guided by HARP or HierarchicalMoE, LLM-DSE (Wang et al., 2025), ForgeHLS (Peng et al., 2025), and random legal designs (Appendix N).

5.2 QoR prediction on unseen kernels

Table 1 compares the predictors. BiG-Flow has the lowest latency error under both averages, 10% below HARP over designs and 17% below HARP over kernels. Averaged over designs, it also predicts every resource more accurately than HARP and three of the four more accurately than HierarchicalMoE. Averaged over all five QoR coordinates, its error is lower than that of HARP on 16 of the 22 test kernels and lower than that of HierarchicalMoE on 14. The kernel average weights every unseen kernel equally and measures how well a model transfers to a new kernel, while the design average is dominated by the few kernels with many designs in the dataset. HARP and HierarchicalMoE are accurate on many kernels but miss the latency by more than an order of magnitude on five of the 22 kernels each, whereas BiG-Flow does so on two (Figure 4 in Appendix G).

5.3 Generating designs for a requested QoR

Designers usually want the fastest design that fits their resource budget, the goal of AutoDSE and HierarchicalMoE (Sohrabizadeh et al., 2022b; Li et al., 2025). Section 5.4 expresses this goal as a request for a large speedup, and this section first tests how closely generated designs follow a request. Following a request also matters on its own when the application sets the latency, such as a kernel that must keep pace with its input stream, and the other logic on the device sets the resource budget. For every test kernel we use the measured QoR of held-out designs as requests, so every request has a solution, and let BiG-Flow generate one design per request. The same model also generates a design for each request unconditionally with the same random seed, which isolates the effect of the request.

A request succeeds when synthesis returns a valid design whose latency lies between half and twice the requested latency, that is, whose absolute $\log_{10}$-cycle error is at most 0.3. The criterion is two-sided, because a design much faster than requested does not follow the request and usually uses more resources than requested. Invalid designs count as failures. On requests where both samplers return valid designs we also report the error in each QoR coordinate.

The request steers generation. Conditional generation yields a valid design within a factor of two of the requested latency for 58% of requests, against 41% for unconditional sampling, a gain of 16.7 percentage points at a validity of 0.790 against 0.769. Conditioning lowers the error in every QoR coordinate and the mean error over the five coordinates by 37%, so the generated designs match the requested resources as well as the requested latency (Table 10 in Appendix H). Figure 2 shows that the gain holds at every tolerance and on 16 of the 22 kernels. Appendix Q details one request.

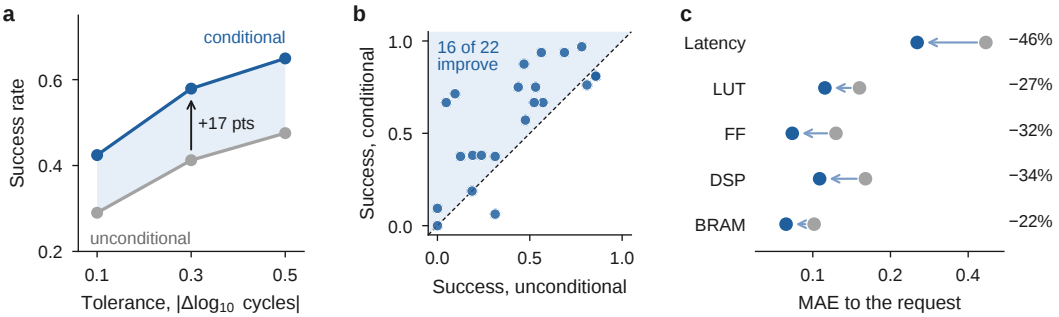

Figure 2: **Pragma generation for a requested QoR on the 22 unseen test kernels.** (a) Fraction of requests that yield a valid design whose absolute $\log_{10}$-cycle error is at most a given tolerance, for conditional and unconditional sampling. (b) Success on each kernel at tolerance 0.3. Points above the diagonal favor conditioning. (c) Mean absolute error between the requested and the generated design's QoR, in $\log_{10}$ cycles for latency and normalized utilization for resources, on requests where both samplers return valid designs, for unconditional (gray) and conditional (blue) sampling.

Table 2: Finding fast designs for the 11 unseen kernels of split 3. Feedback marks methods that choose each design after seeing the synthesis results of earlier designs. Runs and hours are the synthesis runs and the total Vitis HLS time per kernel, as medians over kernels. Speedup is the geometric mean over kernels of the default latency divided by the best valid latency. AutoDSE, same hours stops the AutoDSE run on each kernel after the synthesis hours that the method of the row used there, and the last column divides the speedup by that of the full AutoDSE run. The unconditional sampler is the same model with its QoR input set to zero. LLM-DSE and ForgeHLS use the strongest of the language models we ran, and Table 22 lists all variants. Bold marks the highest speedup at 20 runs.

Method	Feedback	Runs	Hours	Speedup	AutoDSE, same hours	vs. full AutoDSE
AutoDSE, first 20 runs	✓	20	1.6	6.2×		0.55
LLM-DSE, GPT-5 mini	✓	20	1.6	5.2×	5.0×	0.46
HARP-guided search		20	5.8	4.4×	6.0×	0.39
HierarchicalMoE-guided search		20	10.0	1.3×	10.0×	0.12
ForgeHLS, Llama 3.2 3B		≤20	2.6	5.5×	6.4×	0.49
Random legal designs		20	7.5	5.1×	6.9×	0.45
BiG-Flow, unconditional		20	2.4	7.6×	5.8×	0.67
BiG-Flow		20	5.1	**8.6**×	5.8×	**0.76**
AutoDSE, full run	✓	156	24.4	11.3×		1.00

5.4 Finding fast designs under a synthesis budget

We ask BiG-Flow for designs that are 10, 30, 100, 300, and 1000 times faster than the all-default design of each test kernel, a latency that any DSE flow knows after one synthesis, and for a normalized utilization of 0.8 on every resource. BiG-Flow draws 32 designs per request, removes duplicates, and submits 20 of the remaining designs, chosen uniformly at random, to synthesis. Every method gets 20 synthesis runs per kernel, and the default design is free. We report the best valid latency per kernel as a speedup over the default design and as a fraction of the speedup of AutoDSE, the reference used by prior learned DSE work (Li et al., 2025), and average over the 11 test kernels of split 3 with a geometric mean. A kernel with no valid design keeps the default. We also report the synthesis time per kernel, the sum of the Vitis HLS run times of the designs a method submits (Appendix K).

Table 2 compares all methods at a budget of 20 synthesis runs per kernel. BiG-Flow reaches 8.6×, the highest speedup at this budget. AutoDSE and LLM-DSE choose each design after seeing the synthesis results of earlier ones. Within 20 runs AutoDSE reaches 6.2×, and LLM-DSE reaches 5.2× with GPT-5 mini, the strongest of the language models we ran. BiG-Flow chooses all 20 designs before any synthesis and still exceeds both. Among the methods without feedback, it reaches 1.7× the speedup of random legal designs, 1.6× that of ForgeHLS with Llama 3.2 3B, and 2.0× that of the better predictor-guided search, which synthesizes the 20 designs with the lowest predicted latency among 75,000 legal designs, following the evaluation of HierarchicalMoE (Appendix N).

a

```
L0: for (i = 0; i < 1000; i++) {
  #pragma HLS pipeline PIPE_L0
  L2: for (k = 0; k < 1200; k++) {
    #pragma HLS pipeline PIPE_L2
    #pragma HLS unroll factor= PARA_L2
    L3: for (j = 0; j < 1100; j++) {
      #pragma HLS unroll factor= PARA_L3
      C[i][j] += alpha*A[i][k]*B[k][j];
}}}
```

b

site	default	AutoDSE	BiG-Flow
PIPE_L0	none	off	**none**
PIPE_L2	none	off	**none**
PARA_L2 (k)	1	400	**20**
PARA_L3 (j)	1	2	**50**
synthesis runs	1	116	20
synthesis hours	<0.01	28	5.1

c

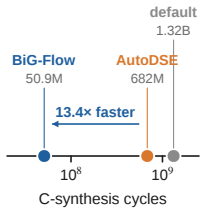

Figure 3: **A design for an unseen kernel that outperforms the AutoDSE search.** (a) Main loop nest of `gemm-p_1200` with its four decisive pragma sites. (b) Values of the all-default design, the best design of the full AutoDSE run, and the best of 20 BiG-Flow designs, with the synthesis runs and hours each one required. (c) Synthesized cycles of the three designs.

Measured in synthesis time, the AutoDSE run stopped after the hours that BiG-Flow used on each kernel, a median of 5.1 hours, reaches 5.8×, and the two BiG-Flow samplers are the only methods that clearly exceed AutoDSE at their own synthesis time. The full AutoDSE run synthesized a median of 156 designs in 24.4 hours per kernel and reached 11.3×. BiG-Flow reaches 76% of this speedup with 8× fewer runs and 5× fewer hours. Prior learned methods surpass AutoDSE only after fine-tuning on designs of the target kernel (Li et al., 2025), which BiG-Flow never sees. Even so, BiG-Flow finds a faster design than the full AutoDSE run on two of the 11 kernels, by 13.4× on `gemm-p_1200` (Section 5.5). The same model sampled unconditionally reaches 7.6×.

Broken down by request, every request level yields valid designs, at rates of 29% to 44%, and at every level the fastest valid design across the 11 kernels is 153× to 445× faster than the default design. Pooling the five levels, 5 of the 11 kernels receive a design at least 10× faster than the default (Table 17 in Appendix K).

As the budget grows, BiG-Flow stays ahead of AutoDSE at every budget up to 20 synthesis runs, and AutoDSE needs 10 hours per kernel to reach the speedup that BiG-Flow reaches with its 20 designs (Figure 7 in Appendix K). The 20 proposals do not depend on each other, so they can also be synthesized in parallel within one timeout. Spending the same 20 runs on five proposals refined by local search finds slower designs than 20 direct proposals (Appendix L), so diverse direct proposals are the better use of a small budget.

5.5 Case study: a design 13× faster than the AutoDSE search

Figure 3 opens `gemm-p_1200`, an unseen kernel of split 3 that multiplies a 1000×1200 matrix by a 1200×1100 matrix. The AutoDSE run synthesized 116 designs of this kernel in 28 hours. It kept the inner loop j at an unroll factor of 2 or less in 102 of them, and the 14 designs that unroll j further keep the reduction loop k at 1 and run as slowly as the default. Its best design turns pipelining off, unrolls k by 400, and takes 682 million cycles, 1.9× faster than the default. BiG-Flow has never seen this kernel. Across the five requests, it proposes designs that unroll both loops together, and 143 of its 160 proposals unroll j by at least 10. The best of its 20 synthesized designs leaves pipelining to the HLS tool to decide and unrolls k by 20 and j by 50. Vitis HLS reports 50.9 million cycles, 26× faster than the default and 13.4× faster than the best AutoDSE design, using at most 71% of any resource budget. It is the fastest design that any method finds for this kernel, 1.6× faster than the next best, from LLM-DSE with GPT-5 mini. AutoDSE improves one pragma at a time, and on this kernel unrolling k alone gains 1.9× while unrolling j alone gains nothing, so the search never combines the two. BiG-Flow generates all pragma values jointly and proposes the combination directly.

6 Conclusion

QoR prediction and pragma design in HLS are two conditionals of one distribution over programs, pragma designs, and QoR. BiG-Flow learns both conditionals with one network, a program-conditioned discrete flow for designs and a continuous flow for QoR. On kernels it has never seen, it predicts latency more accurately than state-of-the-art graph predictors, generates designs that follow a requested QoR, and finds designs 8.6× faster than the default with 20 synthesis runs and no search, including one 13.4× faster than the best design of a full AutoDSE search. Generative models of this kind can give developers good pragma designs for a new kernel from the start and leave kernel-specific search to refine them. Appendix A discusses the limitations of this work.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods: AI coding assistants helped write parts of the software implementation, including experiment orchestration scripts and figure-generation code. We have not used generative AI tools to generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, clean and reformat the dataset, support qualitative and thematic data analysis, or interpret results. Additionally, we used generative AI tools to identify and summarize relevant literature during the related-work survey and to edit the paper to improve readability after the authors had written the full draft. We have reviewed all AI-assisted work. AI-generated code was tested for correctness and verified against saved per-design records by the authors. Literature identified by AI was verified through manual inspection of the original sources. All experiments were designed and executed by the authors, and all reported numbers were checked against the saved synthesis outcomes. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work presents a machine-learning method for automating pragma selection in high-level synthesis for FPGAs. It does not involve human subjects, personally identifiable data, or surveillance applications. The dataset is built from publicly available benchmark suites (MachSuite and Polybench) and synthesis outcomes from a commercially available tool (Vitis HLS). The method is intended to reduce the engineering effort required for hardware design, and we are not aware of direct negative societal consequences. We have read and adhered to the ICLR Code of Ethics.

REPRODUCIBILITY STATEMENT

The appendix specifies the data split (Appendix E.2), normalization and static site features (Appendix F), model architecture and optimization hyperparameters (Appendix F, Table 5), oracle accounting and evaluation protocol (Appendix E), statistical estimators (Appendix E), metric definitions (Appendix P), and the commands that regenerate each result (Appendix R.3). The supplementary code includes checkpoint metadata, generated design manifests, synthesis outcomes, and the scripts that build every table and figure from per-design records (Appendix R.4).

REFERENCES

Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, and Jason Cong. Towards a comprehensive benchmark for high-level synthesis targeted to FPGAs. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, volume 36, 2023.

Andrew Campbell, Jason Yim, Regina Barzilay, Tom Rainforth, and Tommi Jaakkola. Generative flows on discrete state-spaces: Enabling multimodal flows with applications to protein co-design. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 5453–5512, 2024.

Ping Chang, Tosiron Adegbija, Yuchao Liao, Claudio Talarico, Ao Li, and Janet Roveda. Deep inverse design for high-level synthesis. *arXiv preprint arXiv:2407.08797*, 2024.

Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. FPGA HLS today: Successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4):1–42, 2022.

Miguel de Campos, Werner Krebs, and Hanno Gottschalk. Generative inverse design with abstention via diagonal flow matching. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026.

Itai Gat, Tal Remez, Neta Shaul, Felix Kreuk, Ricky T. Q. Chen, Gabriel Synnaeve, Yossi Adi, and Yaron Lipman. Discrete flow matching. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Minsu Kim, Taeyoung Yun, Emmanuel Bengio, Dinghuai Zhang, Yoshua Bengio, Sungsoo Ahn, and Jinkyoo Park. Local search GFlowNets. In *International Conference on Learning Representations*, 2024.

Sunwoo Kim, Minkyu Kim, and Dongmin Park. Test-time alignment of diffusion models without reward over-optimization. In *International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=vi3DjUhFVm`.

Siddarth Krishnamoorthy, Satvik Mehul Mashkaria, and Aditya Grover. Diffusion models for black-box optimization. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 17842–17857, 2023.

Weikai Li, Ding Wang, Zijian Ding, Atefeh Sohrabizadeh, Zongyue Qin, Jason Cong, and Yizhou Sun. Hierarchical mixture of experts: Generalizable learning for high-level synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 18476–18484, 2025. doi: 10.1609/aaai.v39i17.34033.

Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. In *International Conference on Learning Representations*, 2023.

Zedong Peng, Zeju Li, Mingzhe Gao, Qiang Xu, Chen Zhang, and Jieru Zhao. ForgeHLS: A large-scale, open-source dataset for high-level synthesis. *arXiv preprint arXiv:2507.03255*, 2025.

Louis-Noël Pouchet and Tomofumi Yuki. PolyBench/C: The polyhedral benchmark suite. `http://sourceforge.net/projects/polybench/`, 2016.

Neha Prakriya, Zijian Ding, Yizhou Sun, and Jason Cong. LIFT: LLM-based pragma insertion for HLS via GNN supervised fine-tuning. *arXiv preprint arXiv:2504.21187*, 2025.

Yiming Qin, Manuel Madeira, Dorina Thanou, and Pascal Frossard. DeFoG: Discrete flow matching for graph generation. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 50269–50326, 2025.

Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization*, pp. 110–119, 2014.

Yi Ren, Chenhao Xue, Jiaxing Zhang, Chen Zhang, Qiang Xu, Yibo Lin, Lining Zhang, and Guangyu Sun. DiffuSE: Cross-layer design space exploration of DNN accelerator via diffusion-driven optimization. In *International Symposium of Electronics Design Automation*, pp. 756–761, 2025. doi: 10.1109/ISEDA65950.2025.11100315.

Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39 (10):2628–2639, 2020.

Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, and Yu Sun. Masked label prediction: Unified message passing model for semi-supervised classification. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pp. 1548–1554, 2021.

Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Automated accelerator optimization aided by graph neural networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 55–60, 2022a.

Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems*, 27(4):1–27, 2022b. doi: 10.1145/3494534.

Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Robust GNN-based representation learning for HLS. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2023. doi: 10.1109/ICCAD57390.2023.10323853.

Zhiqing Sun and Yiming Yang. DIFUSCO: Graph-based diffusion solvers for combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Brandon Trabucco, Aviral Kumar, Xinyang Geng, and Sergey Levine. Conservative objective models for effective offline model-based optimization. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 10358–10368, 2021.

Brandon Trabucco, Xinyang Geng, Aviral Kumar, and Sergey Levine. Design-bench: Benchmarks for data-driven offline model-based optimization. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 21658–21676, 2022.

Clément Vignac, Igor Krawczuk, Antoine Siraudin, Bohan Wang, Volkan Cevher, and Pascal Frossard. DiGress: Discrete denoising diffusion for graph generation. In *International Conference on Learning Representations*, 2023.

Elena Vouvali, Dimosthenis Masouros, Aggelos Ferikoglou, Dimitrios Soudris, and Sotirios Xydis. MailoHLS: Multi-adapter structure-aware learning for pareto-driven HLS pragma optimization. *arXiv preprint arXiv:2606.07246*, 2026.

Hanyu Wang, Xinrui Wu, Zijian Ding, Su Zheng, Chengyue Wang, Neha Prakriya, Tony Nowatzki, Yizhou Sun, and Jason Cong. LLM-DSE: Searching accelerator parameters with LLM agents. *arXiv preprint arXiv:2505.12188*, 2025.

Peng Xie, Ke Xue, Yunqi Shi, Ruo-Tong Chen, Chengrui Gao, Siyuan Xu, Chenjian Ding, Mingxuan Yuan, and Chao Qian. FlowPlace: Flow matching for chip placement. In *Proceedings of the 63rd ACM/IEEE Design Automation Conference*, 2026. URL https://arxiv.org/abs/2604.23658.

Ye Yuan, Can Chen, Christopher Pal, and Xue Liu. ParetoFlow: Guided flows in multi-objective optimization. In *International Conference on Learning Representations*, 2025.

Dinghuai Zhang, Hanjun Dai, Nikolay Malkin, Aaron C. Courville, Yoshua Bengio, and Ling Pan. Let the flows tell: Solving graph combinatorial optimization problems with GFlowNets. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Baoheng Zhu, Deyu Bo, Delvin Ce Zhang, and Xiao Wang. Graph-GRPO: Training graph flow models with reinforcement learning. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026.

A Discussion

The CDFG of a kernel is fixed context, the pragma values are the decision, and synthesis maps the pair to QoR. BiG-Flow follows this structure. It encodes the program once, runs a discrete flow only over the pragma sites, and serves prediction and generation with one network. Because it is trained across kernels, it proposes designs for a new kernel before any synthesis of that kernel, which is where search-based methods are weakest. Generation and search are complementary. Proposals from BiG-Flow can seed a search that refines them with synthesis feedback, and under a small budget we found diverse direct proposals to be the most valuable first step (Section 5.4).

Limitations. All outcomes are Vitis HLS C-synthesis estimates for one device and clock, and we do not run place-and-route. BiG-Flow assigns pipeline and unroll values at known pragma sites and does not restructure the code or insert other pragma types. QoR prediction integrates 100 network evaluations, which is slower than a single-pass predictor although far faster than synthesis. Like new kernels in practice, most test kernels share an algorithm family with some training kernel at a different problem size or implementation, and three of the 22 come from families that are absent from training (Appendix E.2).

B Ablation studies

Program conditioning. Table 9 in Appendix G compares ways of giving the pragma transformer access to the CDFG on split 3. Local fusion maps each pragma site to its CDFG nodes and adds their embeddings to the site token. It raises the kernel-averaged error from 0.536 without program information to 0.838, and to 0.925 when combined with the pooled program vector. Pooling the CDFG into one program vector gives the best single model, with a kernel error of 0.446, and all three of its training seeds are more accurate than HierarchicalMoE. Attenuating the local path by a factor of 0.1 on top of the pooled vector gives 0.453. Program information should therefore enter as a summary of the whole program, and aligning it to individual pragma sites consistently hurts transfer.

One network or two. A common alternative to a shared model is a conditional generator paired with a separately trained QoR predictor, which doubles the parameter count. We train this two-network variant with the same program vector, features, and selection rule. On unseen kernels its latency error is close to that of BiG-Flow, 0.577 against 0.549 averaged over kernels (Figure 5). In generation with legal decoding, the two networks reach a validity of 0.789 against 0.746 for BiG-Flow, while BiG-Flow has the lower latency error on requests valid under both, 0.397 against 0.427, and produces more distinct designs, 0.906 against 0.785 unique. One shared network therefore serves both tasks as well as two separate networks and proposes more distinct designs. Appendix I reports all comparisons.

C Extended related work

Machine learning for HLS design space exploration. Learned QoR predictors are the dominant abstraction in HLS design space exploration (DSE) (Schafer & Wang, 2020). GNN-DSE predicts performance from the program graph with pragmas attached (Sohrabizadeh et al., 2022a), HARP adds a hierarchical graph and a separate pragma encoder (Sohrabizadeh et al., 2023), and HierarchicalMoE mixes experts across graph granularities to generalize to new kernels (Li et al., 2025). All of them rank candidates produced by an enumeration or search over the design space of one kernel. AutoDSE searches with the HLS tool in the loop and serves as the reference for learned DSE (Sohrabizadeh et al., 2022b; Li et al., 2025). Language-model methods insert pragmas from the source text, supervised by a graph model in LIFT (Prakriya et al., 2025), trained on a large synthesis corpus in ForgeHLS (Peng et al., 2025), driven by agents in LLM-DSE (Wang et al., 2025), or combined with graph embeddings and objective-specific adapters in MailoHLS (Vouvali et al., 2026). BiG-Flow differs from all of these in that one model both predicts QoR and generates designs for an arbitrary requested QoR.

Generative models for design optimization. Generative models can replace per-instance search by learning a distribution of good solutions. DIFUSCO generates combinatorial solutions with diffusion (Sun & Yang, 2023), and GFlowNets sample solutions in proportion to a reward (Zhang et al., 2023; Kim et al., 2024). Offline model-based optimization learns from a fixed table of designs and scores

(Trabucco et al., 2022; 2021), with inverse models such as DDOM (Krishnamoorthy et al., 2023) and multi-objective flows such as ParetoFlow (Yuan et al., 2025). In hardware design, DID4HLS learns a conditional design distribution from the synthesis results of the target kernel inside an iterative loop (Chang et al., 2024), DiffuSE applies diffusion to DNN accelerator and EDA parameters (Ren et al., 2025), and FlowPlace generates macro placements with flow matching and projects the predicted placement onto overlap-free positions at every sampling step (Xie et al., 2026). These generators either learn from evaluations of the design space they optimize, as offline model-based optimization, DID4HLS, and DiffuSE do, or rely on objectives and constraints that are cheap to compute, as in combinatorial optimization and placement. A new HLS kernel is a new design space whose objective requires synthesis, so neither assumption holds. BiG-Flow learns across kernels from an offline database, proposes designs for a new kernel without evaluating any of its designs, and predicts QoR with the same model. We compare with the generative approaches that propose designs for an unseen kernel before any of its designs is synthesized, namely language-model pragma generators, a conditional generator paired with a separate QoR predictor, which is the two-model design of offline model-based optimization, and the same BiG-Flow model sampled unconditionally. Methods that fit a model to the synthesis results of each target kernel solve a different problem. Appendix M explores fine-tuning BiG-Flow with synthesis rewards in the style of Graph-GRPO (Zhu et al., 2026; Kim et al., 2025).

Discrete and bidirectional flows. Discrete diffusion (Vignac et al., 2023) and discrete flow matching (Gat et al., 2024; Campbell et al., 2024) generate categorical data such as graph labels, and DeFoG adapts discrete flow matching to graphs (Qin et al., 2025). MultiFlow couples discrete and continuous modalities in one network (Campbell et al., 2024). Diagonal Flow Matching pairs designs with labels so that a single network performs both inverse design and prediction for continuous designs (de Campos et al., 2026). BiG-Flow applies this pairing to categorical pragma designs on a graph, conditions it on the program, and restricts each pragma to its legal values. Section 3 gives the formulations that BiG-Flow builds on.

D Discrete formulation

D.1 Mixture path

Gat et al. (2024) define the mixture path

$$p_t(a \mid a_0, a_1) = \kappa_t\, \delta(a, a_1) + (1 - \kappa_t)\, \delta(a, a_0)$$

for a data value a_1, a source value a_0 drawn from p_0, and a scheduler κ_t. With $\kappa_t = t$ and the source value averaged out, it becomes Equation 1, the path used by Campbell et al. (2024).

D.2 Categorical path versus simplex interpolation

Flow matching on categorical data can interpolate one-hot vectors continuously, $\mathbf{T}_t = (1-t)\,\mathbf{T}_0 + t\,\mathbf{T}_1$, or sample a categorical state from the mixture path of Equation 2. The two choices differ in what the network can read off the state. Under continuous interpolation every coordinate of $\mathbf{T}_t$ lies in $\{0,\ t,\ 1-t,\ 1\}$. Given t, the magnitude of a coordinate therefore tells whether its category is active at the source, at the data, at both, or at neither, and recovering the final value reduces to a table lookup. The network learns this lookup and never learns to denoise. At sampling time the state is one-hot, which the lookup reads as "source equals data," and the sampler returns its source sample unchanged. We observed this fixed point on synthetic graph benchmarks and on the pragma sites.

A state sampled from the mixture path removes the shortcut because only zeros and ones are visible. The network must infer the final value from the kernel, the current value, and t, which is the same information it has during sampling. All BiG-Flow results use the mixture path.

D.3 Sampling step and the discrete flow-matching rate

The sampler moves the one-hot state of a site toward a sampled final value $\hat{\mathbf{T}}_i \sim \pi_{\theta,i}$ with the velocity $(\hat{\mathbf{T}}_i - \mathbf{T}_{t,i})/(1-t)$, takes an Euler step, projects onto the simplex, and draws a new one-hot state. For current value a, destination $b \neq a$, source distribution p_0, and prediction π_θ, discrete flow matching

(Gat et al., 2024) defines the rate $R_{\theta,t}(a, b)$ obtained by averaging the conditional rates over π_θ, and the Euler transition

$$P_\theta(T_{t+\Delta t,i} = b \mid T_{t,i} = a) = \Delta t\, R_{\theta,t}(a, b), \qquad P_\theta(a \mid a) = 1 - \sum_{b \neq a} \Delta t\, R_{\theta,t}(a, b),$$

with transitions rescaled when the off-diagonal mass exceeds one. The two schemes agree in expectation for the mixture path and produced indistinguishable sample statistics in a matched comparison on a synthetic graph benchmark. Offline sampling uses the first form. The policy-optimization study in Appendix M needs an exact trajectory likelihood and therefore evaluates the rate, with legal options enforced before normalization. The loop-nest edges are fixed, so transitions act on pragma values only.

D.4 QoR path and input dropout

The paired QoR state $\mathbf{Y}_s = s\,Y$ with $s = 1 - t$ reveals the label for every $s > 0$. On a fraction p of training samples we replace this input by zero and keep the target velocity Y, which makes the input seen during QoR prediction part of the training distribution. The rate is chosen by validation error on held-out kernels. The unconditional sampler sets the QoR input to zero at every network query.

E Data, splits, and evaluation protocol

E.1 Expanded HLSyn

HLSyn (Bai et al., 2023) contains MachSuite kernels and Polybench kernels at some of their problem sizes. Expanded HLSyn adds the remaining problem sizes so that every Polybench kernel of HLSyn appears at all three sizes, and it was synthesized with Vitis HLS 2025.1. The dataset contains 77 kernels and 26,191 designs. We drop 38 designs and exclude the `aes_32` and `nw_129` kernels, whose records are inconsistent, leaving 75 kernels and 26,153 designs. Each design records latency in cycles and LUT, FF, DSP, BRAM, and URAM utilization. Every design comes from one AutoDSE run per kernel, and the resource labels are divided by 0.8 times the device capacity, so the usual rule that every resource fits within 80% of the device becomes a normalized cutoff of one.

The CDFG of a kernel is the same for every design of that kernel. The generated object is the design on the known pragma sites. Site features encode pragma type, loop depth, trip count, and number of legal options. Loop-nest edges and the CDFG are fixed.

Table 3: Expanded HLSyn setting.

Quantity	Value
Kernels / designs	75 / 26,153
Train / validation / test kernels per split	56 / 8 / 11
Splits	1 and 2 with disjoint test kernels, and 3
Pragma sites per kernel	4–22
CDFG nodes per kernel	345–1473, median 656
QoR coordinates	$\log_{10}$ cycles and LUT, FF, DSP, BRAM
Synthesis	Vitis HLS 2025.1, Alveo U200, 250 MHz
Timeout	30 minutes
Validity	five resources $\leq$ 80% of capacity

E.2 Test kernels

All splits use disjoint validation and test kernels. Split 1 tests on

`bicg_124`, `doitgen-red_60`, `gemm-blocked_64`, `gemver_400`, `gesummv_1300`, `jacobi-1d_118`, `jacobi-2d_248`, `spmv-ellpack_494`, `stencil-3d_32`, `symm_1200`, and `trmm-opt_240`.

Split 2 tests on

`2mm_220`, `doitgen-red_30`, `gemm-ncubed_64`, `gemm-p_80`, `gemver_2000`, `gesummv_250`, `gesummv_90`, `md_256`, `spmv-crs_1666`, `syrk_1200`, and `trmm-opt_1200`.

Split 3, used for the synthesis-budget study and the ablations, tests on

`atax_124`, `doitgen_60`, `gemm-blocked_64`, `gemm-p_1200`, `gesummv_1300`, `jacobi-2d_1298`, `seidel-2d_398`, `spmv-ellpack_494`, `symm-opt_1200`, `syrk_1200`, and `trmm_80`.

The exact (kernel, design) membership of every split is exported before any baseline is trained, and HARP and HierarchicalMoE read these files directly. Splits 1 and 2 serve QoR prediction and pragma generation because they give 22 distinct test kernels. Each split requires training every model and synthesizing every generated design, so the synthesis-budget study uses one split.

E.3 Algorithm-family overlap

We normalize the C++ sources by removing comments, numeric constants, and pragma values, then compare token shingles and canonical algorithm-family names. Nine of the eleven split-1 test kernels and ten of the eleven split-2 test kernels have a training kernel from the same family at another size or in another implementation, and several normalized sources are identical after constants are removed. The three test kernels without a same-family training kernel are `spmv-ellpack_494`, `stencil-3d_32`, and `md_256`. The splits therefore test transfer to unseen kernels, problem sizes, and pragma spaces, mostly within algorithm families that appear in training.

Table 4: Algorithm-family overlap of the test kernels of splits 1 and 2.

Split	Test kernels	Same family in training	Family absent from training
1	11	9	spmv-ellpack, stencil-3d
2	11	10	md

E.4 Evaluation tasks

QoR prediction. Given an unseen kernel and a complete design, predict QoR. Every model uses the same split and its own validation-selected checkpoint. We report design and kernel averages.

Pragma generation for a requested QoR. Given an unseen kernel and a requested QoR, generate a design without observing any synthesis result. The conditional and unconditional samplers share the model, the seed, and the number of samples.

Finding fast designs. Given an unseen kernel, find the fastest valid design with a fixed number of synthesis runs. Budget selection happens before database lookup, and duplicate designs do not buy extra runs. For every method we report the synthesis runs and hours it needs, including those of the AutoDSE run for methods that choose among its designs.

E.5 Model configuration by result

Table 5 lists the model, weights, QoR path, and sampling budget behind each result.

Table 5: Model, weights, and inference configuration behind each result.

Result	Model	Weights	QoR state during generation	Decoding and budget
Prediction and generation, splits 1 and 2	pooled program vector, $p = 0.05$	raw, epochs 174 / 49	known path $(1 - t)Y^\star$	legal decoding
Fast designs, split 3	pooled program vector, $p = 0.5$	EMA, epoch 99	integrated with the QoR head	160 proposals, 20 synthesized
Two-network variant	matched program vector and p	validation selected	fixed condition	legal decoding
Policy study	$p = 0.05$ model as initializer	raw	categorical policy path	transition masks

E.6 Statistics

For prediction, let $e_{m,k}$ be the mean absolute error of model m on kernel k. The kernel average is $K^{-1}\sum_k e_{m,k}$. For generation, validity and success are fractions of all requests, and target error is compared on requests for which both samplers return a valid design. For fast designs, per-kernel speedups are averaged in log space and exponentiated, which gives a geometric mean. Bootstrap intervals over requests and over kernels for the generation results are reported in Tables 11 and 12.

Design and kernel averages are both reported because the archived designs are unevenly distributed. The largest split-1 kernel, `bicg_124`, contributes 48.9% of the test designs, and `md_256` contributes 59.0% in split 2. The design average measures accuracy on the archived mixture, whereas the kernel average estimates accuracy on a newly drawn kernel. Split 2 shows the difference. HARP has the lower design average, 0.316 against 0.353, because it is accurate on `md_256`, while BiG-Flow is more accurate on six of the eleven kernels and has the lower kernel average, 0.620 against 0.756.

E.7 Checkpoint selection

BiG-Flow, the two-network variant, HARP, and HierarchicalMoE checkpoints are selected independently by validation $\log_{10}$-cycle error. Raw and EMA weights are both eligible where both are trained. The BiG-Flow checkpoints are epoch 174 raw for split 1 and epoch 49 raw for split 2. No test result participates in selection.

F Architecture and optimization

Table 6: BiG-Flow architecture.

Component	Setting
Pragma transformer	6 layers, 8 heads
Node / edge / global hidden width	256 / 64 / 128
Node / edge / global feed-forward width	256 / 128 / 256
Static site features	10
Program encoder	6 TransformerConv layers, 4 heads
Encoder width / dropout	128 / 0.10
Encoder aggregation	max jumping knowledge and graph pooling
Pragma vocabulary $\lvert\mathcal{A}\rvert$	35 shared values
QoR coordinates	5
Parameters	6.75M

Table 7: Training settings.

Hyperparameter	Value
Epochs	300
Optimizer / learning rate	AdamW / $2 \cdot 10^{-4}$
Weight decay	10^{-12}
Batch size	64
Two-network micro-batch / accumulation	32 / 2
EMA decay	0.999
Pragma state during training	sampled from the mixture path
Sampling steps	100
QoR input dropout p	0.05
Loop-nest edges	fixed
Evaluation width	padded to 22 sites
Checkpoint rule	minimum validation $\log_{10}$-cycle error, raw or EMA

Each training sample contributes the pragma cross-entropy and the QoR velocity loss. CDFG tensors are deduplicated by kernel within a batch, and the program vector is broadcast to every design of that kernel.

F.1 Static site features

The ten features are indicators for pipeline versus unroll site and for the loop families L, R, and W, loop depth, log trip count, number of options, log total nested iterations, and an innermost-loop indicator. They distinguish pragma sites that look alike in the small loop graph but play different roles in the program. All compared models, including the predictor of the two-network variant, receive the same features.

F.2 Pooled program vector versus local fusion

The program encoder produces CDFG node embeddings and a whole-program vector. Local fusion maps each pragma site back to CDFG nodes and adds the aligned embeddings to the site token. This mapping is noisy across compiler graphs, and Appendix G shows that it degrades transfer. BiG-Flow therefore uses the pooled program vector alone. The degradation is not explained by conflicting gradients. Across eight batches, the cosine between the pragma and QoR gradients averages +0.065 on both the encoder and the shared transformer, so the two objectives are nearly orthogonal.

G Complete prediction results

G.1 Per-kernel latency error

Figure 4 shows the latency error of each predictor on each of the 22 unseen test kernels of splits 1 and 2.

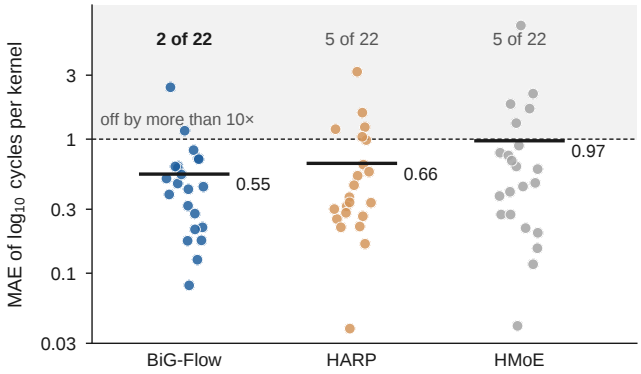

Figure 4: **Latency prediction on the 22 unseen test kernels.** Each point is the mean absolute error of $\log_{10}$ cycles over the test designs of one kernel, and black bars mark the mean over kernels. The shaded band holds predictions that are off by more than a factor of 10, and the numbers count the kernels in it.

G.2 Per-split results

Table 8 reports each split separately. The main text averages the two columns of each metric. HierarchicalMoE was designed to adapt to a new kernel by fine-tuning on its designs (Li et al., 2025), and without that data it degrades on split 2. The two-network variant uses the same program vector and input dropout as BiG-Flow, with checkpoints at epoch 224 raw for split 1 and epoch 24 EMA for split 2, both chosen by validation error.

Per kernel, the mean error of BiG-Flow over all five coordinates is lower than that of HARP on 16 of the 22 kernels and lower than that of HierarchicalMoE on 14. Figure 5 adds the two-network variant of Appendix B.

G.3 Program conditioning

Table 9 and Figure 6 report the program-conditioning ablation on split 3. The pooled-vector model is trained with three seeds. The best seed reaches design and kernel errors of 0.345 and 0.446, the other

Table 8: Latency prediction error per split, as the mean absolute error of $\log_{10}$ cycles. Design averages over the test designs and Kernel over the 11 test kernels of a split.

	Split 1		Split 2	
Model	Design	Kernel	Design	Kernel
HARP	0.433	0.563	**0.316**	0.756
HierarchicalMoE	0.432	0.568	1.578	1.376
BiG-Flow	**0.322**	**0.477**	0.353	**0.620**

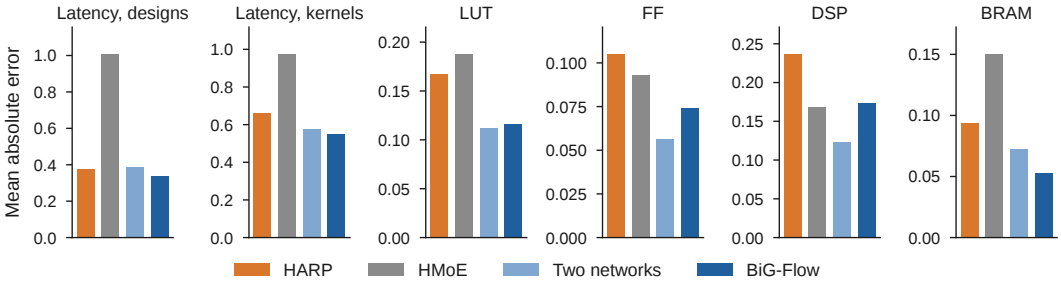

Figure 5: Prediction error of BiG-Flow, its two-network variant, HARP, and HierarchicalMoE on unseen kernels, averaged over splits 1 and 2. Entries are mean absolute errors of $\log_{10}$ cycles and of normalized resource utilization. Latency error is averaged over test designs and over test kernels, and resource errors over test designs. Lower is better.

two reach 0.419 and 0.577 and 0.658 and 0.817, and a fixed ensemble of the three reaches 0.414 and 0.516. Every seed has a better kernel average than HierarchicalMoE.

Table 9: Program conditioning and prediction baselines on split 3. Errors are mean absolute errors of $\log_{10}$ cycles. The pooled vector lists the three-seed mean with the seed range in brackets.

Model	Validation error	Test error, designs	Test error, kernels
Pragma sites only	0.346	0.381	0.536
Pooled vector, 3 seeds	–	0.474 [0.345, 0.658]	0.613 [0.446, 0.817]
Pooled vector, ensemble	–	0.414	0.516
Pooled vector + 0.1 local fusion	0.450	0.419	0.453
Local fusion only	0.443	0.516	0.838
Pooled vector + local fusion	0.452	0.709	0.925
Two networks, full CDFG	0.378	0.389	0.556
HARP	–	0.400	0.649
HierarchicalMoE	–	0.759	0.878

G.4 Splits within kernels

When 90% of the designs of every kernel are visible in training, HARP is 2.4 to 2.7 times more accurate than BiG-Flow under both averages and wins 71 of the 75 kernels. This setting rewards memorizing each kernel, and six of the eleven best generated designs in it are already present in training. We report it only as a diagnostic, since the unseen-kernel setting is the one in which a model must transfer.

H Complete generation results

Table 10 reports the request-following study of Section 5.3 averaged over splits 1 and 2.

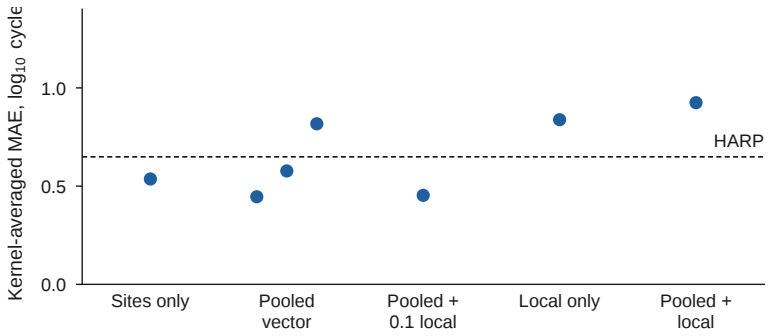

Figure 6: Kernel-averaged mean absolute error of $\log_{10}$ cycles for each program-conditioning variant on split 3. The pooled vector shows its three seeds. The dashed line is HARP.

Table 10: Pragma generation for a requested QoR on unseen kernels, averaged over splits 1 and 2. The unconditional sampler is the same model with its QoR input set to zero. Success requires a valid design within a factor of two of the requested latency in either direction. Errors are mean absolute errors between the requested and the synthesized QoR, in $\log_{10}$ cycles for latency and in normalized utilization for each resource, on requests where both samplers return valid designs.

	BiG-Flow, unconditional	BiG-Flow
Validity ↑	0.769	**0.790**
Success ↑	0.412	**0.579**
Latency error ↓	0.468	**0.254**
LUT error ↓	0.152	**0.111**
FF error ↓	0.123	**0.083**
DSP error ↓	0.160	**0.106**
BRAM error ↓	0.101	**0.079**
Mean of the five ↓	0.201	**0.127**

H.1 Per-split results

Split 1 evaluates 32 requests for each of eleven kernels. The conditional and unconditional samplers produce 285 and 276 valid designs. On the 239 requests valid under both, the latency error is 0.287 against 0.423, and conditioning has the lower error on nine of the eleven kernels. Split 2 uses 21 requests for each of eleven kernels, the largest number available for every kernel. Validity is 0.771 against 0.753, and the latency error on requests valid under both is 0.221 against 0.514. One kernel has no request valid under both, and conditioning has the lower error on seven of the other ten.

H.2 Success against tolerance and per-coordinate error

Table 11 reports success at three tolerances and Table 12 the error on every QoR coordinate, with bootstrap intervals over requests and over kernels.

Table 11: Success rate over all requests against the log-cycle tolerance. A request succeeds when the design is valid and its absolute log-cycle error is at most the tolerance.

Split	Tolerance	Conditional	Unconditional	Δ	Request 95% CI	Kernel 95% CI
1	0.1	0.446	0.273	0.173	[0.122, 0.227]	[0.080, 0.264]
1	0.3	0.574	0.401	0.173	[0.122, 0.224]	[0.060, 0.273]
1	0.5	0.628	0.457	0.170	[0.122, 0.222]	[0.085, 0.250]
2	0.1	0.403	0.307	0.095	[0.030, 0.160]	[0.004, 0.203]
2	0.3	0.584	0.424	0.160	[0.087, 0.234]	[0.039, 0.307]
2	0.5	0.671	0.494	0.177	[0.108, 0.247]	[0.039, 0.346]

Table 12: Per-coordinate mean absolute error between the requested and the synthesized QoR on requests valid under both arms, in $\log_{10}$ cycles for latency and in normalized utilization for resources.

Split	Coordinate	Conditional	Unconditional	Δ	Request 95% CI	Kernel 95% CI
1	log latency	0.287	0.423	-0.136	[-0.193, -0.083]	[-0.253, -0.041]
	LUT	0.074	0.112	-0.038	[-0.051, -0.024]	[-0.056, 0.003]
	FF	0.045	0.080	-0.035	[-0.047, -0.024]	[-0.061, -0.007]
	DSP	0.056	0.097	-0.040	[-0.062, -0.020]	[-0.061, -0.003]
	BRAM	0.058	0.072	-0.014	[-0.030, 0.002]	[-0.059, 0.015]
	mean of five	0.104	0.157	-0.053	[-0.069, -0.037]	[-0.077, -0.024]
2	log latency	0.221	0.514	-0.293	[-0.392, -0.200]	[-0.591, -0.047]
	LUT	0.148	0.191	-0.043	[-0.066, -0.021]	[-0.095, -0.000]
	FF	0.122	0.166	-0.044	[-0.063, -0.027]	[-0.084, -0.006]
	DSP	0.156	0.223	-0.067	[-0.104, -0.032]	[-0.165, -0.000]
	BRAM	0.099	0.130	-0.031	[-0.058, -0.005]	[-0.110, 0.040]
	mean of five	0.149	0.245	-0.096	[-0.128, -0.065]	[-0.198, -0.020]

H.3 Legal decoding

The shared vocabulary lets the pragma head name an unroll rank that a site does not offer. Table 14 reports the fraction of designs whose most likely values all lie inside their legal options before legal decoding. Legal decoding applies the known constraint to the sampled trajectory and needs no extra synthesis run. In the comparison with the two-network variant on split 3, 82 and 48 designs fail only because of an unsupported rank. Table 13 reports the comparison after decoding every design within its legal options.

Table 13: Generation with legal decoding on split 3 for BiG-Flow and its two-network variant. Latency error is the mean absolute $\log_{10}$-cycle error between the requested and the synthesized latency on requests for which both return a valid design.

Model	Validity	Latency error, both valid	Unique designs
BiG-Flow	0.746	0.397	0.906
Two networks	0.789	0.427	0.785

Table 14: Fraction of designs whose values all lie inside the legal options before legal decoding.

Evaluation	Conditional	Unconditional	Random legal
Split 1 generation	0.790	0.696	–
Split 2 generation	0.788	0.701	–
Fast designs, split 3	0.662	0.840	1.000

I One network versus two

The two-network variant trains a QoR-conditioned generator and a separate predictor with the same input dropout, static features, fixed edges, padded width, and selection rule as BiG-Flow. Without program information it has 10.74M parameters against 5.37M for BiG-Flow.

On the unseen kernels of split 3 without program information, the latency error of BiG-Flow is 0.381 against 0.487 for the two networks, or 0.536 against 0.598 averaged over kernels, and BiG-Flow is more accurate on seven of the eleven kernels. On 256 generation requests in this setting, the two networks produce more valid designs before legal decoding, 0.594 against 0.434, while BiG-Flow has the lower error on the 88 requests valid under both, 0.477 against 0.739. With the pooled program vector and the full CDFG, validity before legal decoding is 0.445 for BiG-Flow and 0.613 for the two networks, with errors of 0.407 and 0.680, and legal decoding brings the two to 0.746 and 0.789 validity and 0.397 and 0.427 error. On splits 1 and 2 the two networks reach kernel-averaged errors of 0.497 and 0.657 against 0.477 and 0.620 for BiG-Flow.

In the synthesis-budget protocol of Section 5.4 without program information, the best designs of BiG-Flow are 1.36 times faster than those of the two-network variant on the nine kernels where both find a valid design, and BiG-Flow is faster on six of them. Relative to AutoDSE the two reach 0.67 and 0.49 on the kernels each solves.

QoR prediction with BiG-Flow integrates 100 network evaluations and takes 15,789 ms for 256 designs padded to 22 sites, against 162 ms for the single-pass predictor. This cost is negligible next to synthesis but matters when screening very large numbers of designs.

J Background on HLS pragma design

HLS compiles a behavioral C/C++ kernel into a cycle-accurate hardware microarchitecture. Pragmas do not change the function that the kernel computes. They change its implementation by selecting parallelism, pipelining, and memory behavior. The design object in this work is the set of values at the pragma locations of a fixed source. The following excerpt from the held-out `gesummv_1300` kernel shows the generated fields.

```
L2: for (i = 0; i < 1300; i++) {
  #pragma HLS pipeline auto{__PIPE__L2}
  #pragma HLS unroll factor=auto{__PARA__L2}
  L3: for (j = 0; j < 1300; j++) {
    #pragma HLS unroll factor=auto{__PARA__L3}
    tmp[i] += A[i][j] * x[j];
  }
}
L4: for (i = 0; i < 1300; i++) {
  #pragma HLS pipeline auto{__PIPE__L4}
  L5: for (j = 0; j < 1300; j++) {
    #pragma HLS unroll factor=auto{__PARA__L5}
    y[i] += B[i][j] * x[j];
  }
}
```

A pipeline site chooses among off, on, and omitted. A parallel site chooses a rank in that site's ascending list of legal unroll factors, and the list depends on the loop trip count. The output vocabulary is shared across kernels, which is why raw endpoints can name an unsupported rank and why the terminal decode restricts each site to its own options.

Pragmas alter the amount and schedule of replicated hardware. Increasing an unroll factor can reduce loop latency while consuming more DSPs, LUTs, and memory ports, and an outer pipeline can conflict with an inner pipeline or with a data dependence. The landscape is therefore non-smooth and kernel-specific. In the `gesummv_1300` example of Figure 9, changing seven values lowers latency by 5.41× while raising normalized BRAM from 0.636 to 0.850. A forward model has to capture this coupling, and an inverse model has to represent several feasible assignments for one target.

K Finding fast designs under a synthesis budget

K.1 Requests and outcomes

Let $L_{\text{default},X}$ be the cycle count of the all-default design of kernel X. A speedup request $s \in \{10, 30, 100, 300, 1000\}$ sets the requested latency to $\log_{10} L_{\text{default},X} - \log_{10} s$, and every requested resource utilization is 0.8. No achievable range of the test kernel enters the request. Every synthesized design receives exactly one outcome, valid, over budget, timeout, or tool failure. Preempted jobs are retried and never counted as failures.

The study uses 660 synthesis runs over three samplers and eleven kernels. Seventy-three are database or cache hits, and 587 are new Vitis HLS projects, of which 275 reach the 30-minute timeout. Candidate selection is fixed before lookup, so a design found in the database still uses its run.

K.2 Speedup against budget

Figure 7 follows the best design found as the budget grows, and Table 15 lists the speedups at 5, 10, 15, and 20 runs. The curve of BiG-Flow flattens after about fifteen runs, at 8.4× for 15 runs and 8.6× for 20, so 20 runs capture nearly all of the value of the proposals.

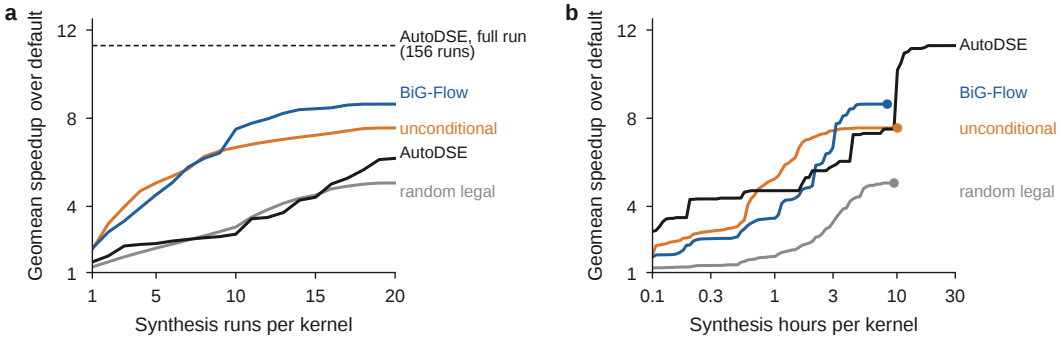

Figure 7: **Best design found against the synthesis budget.** Curves show the geometric-mean speedup over the default design when every test kernel receives the same budget. (a) Budget in synthesis runs. A point at k uses k of the 20 synthesized designs of a sampler, as the median over 4,000 random orders, and the first k runs of the AutoDSE run. The dashed line is the full AutoDSE run. (b) Budget in synthesis hours on a logarithmic axis, with runs executed one after another. Dots mark where a sampler has synthesized all 20 designs.

Table 15: Geometric-mean speedup over the default design when k of the 20 synthesized designs are used, as the median over 4,000 random orders of the designs of each method.

Synthesis runs k	5	10	15	20
BiG-Flow	4.527	7.504	8.430	8.641
BiG-Flow, unconditional	5.066	6.674	7.227	7.559
Random legal designs	2.109	3.065	4.507	5.061
AutoDSE, first k runs	2.316	2.740	4.414	6.178
AutoDSE, full run		11.295		

K.3 Synthesis time

The AutoDSE run of every kernel recorded when each design finished synthesis. Some runs were interrupted and resumed, and the recorded exploration time excludes the pauses. The runs synthesized one design at a time with a timeout of 15, 30, or 40 minutes depending on the kernel, so the time between two consecutive designs gives the synthesis time of the second. A design that a method finds in the database is charged this time, and a design that we synthesize is charged the run time reported by Vitis HLS, or 30 minutes if it reaches our timeout. Every design of Expanded HLSyn appears in

these runs. The lookup database additionally contains designs from a second AutoDSE run for 3 of the 11 test kernels, which does not change the best design of any of them. Table 16 lists the hours and the matched comparison per kernel.

Table 16: Synthesis hours per kernel and AutoDSE stopped after the hours of BiG-Flow. Speedups are over the default design, and bold marks kernels where BiG-Flow is faster than AutoDSE with the same hours.

Kernel	AutoDSE, full run runs	AutoDSE, full run hours	BiG-Flow hours	AutoDSE, same hours runs	AutoDSE, same hours speedup	BiG-Flow speedup	AutoDSE, full speedup
atax_124	1486	24.0	1.2	15	7.3	**28.1**	29.1
doitgen_60	387	24.0	2.4	29	1.2	1.0	1.8
gemm-blocked_64	1148	30.0	5.9	86	176.9	**257.3**	288.8
gemm-p_1200	116	28.0	5.1	51	1.9	**26.0**	1.9
gesummv_1300	156	28.0	2.1	31	11.2	**15.4**	12.3
jacobi-2d_1298	60	24.2	7.8	22	14.3	1.9	14.3
seidel-2d_398	277	24.1	8.5	29	1.1	1.0	1.1
spmv-ellpack_494	3713	36.0	1.6	23	1.0	**2.1**	2.1
symm-opt_1200	110	28.0	4.5	44	1.2	1.1	1.2
syrk_1200	95	17.7	6.7	41	43.6	3.7	45.5
trmm_80	128	24.4	7.2	27	10.0	**445.1**	638.9

K.4 Response to the requested speedup

Table 17: Designs generated for each requested speedup on the 11 unseen kernels of split 3. Best speedups are over the all-default design, as a geometric mean over kernels and for the single best kernel.

Requested speedup	Synthesized designs	Valid rate	Kernels with ≥ 10×	Best speedup	
				geo. mean	best kernel
10×	64	0.42	4	6.9×	338×
30×	62	0.44	4	6.0×	445×
100×	63	0.29	2	2.7×	173×
300×	69	0.36	4	5.7×	445×
1000×	46	0.35	3	3.4×	153×

Table 17 breaks the result of Section 5.4 down by request. Most requests exceed the best speedup that AutoDSE found for the kernel, and requests this aggressive are unfamiliar to the model, so they act less as a precise target than as a push toward aggressive designs. The best speedup therefore does not grow with the requested speedup, yet every request level yields valid designs at a rate of 29% to 44%, and at every level the fastest valid design across the 11 kernels is 153× to 445× faster than the default design. Each request level receives about four of the 20 runs per kernel, and with this share its best designs reach a geometric mean of 2.7× to 6.9× over the 11 kernels, with 2 to 4 kernels receiving a design at least 10× faster than the default. Pooling the five levels gives 8.6×, and 5 of the 11 kernels receive a design at least 10× faster.

Table 32 in Appendix S lists, for each requested speedup, the number of synthesized designs, the valid rate, and the median and best achieved speedup over the default. The median achieved speedup stays between 1.9 and 3.7 at every request, while the best designs are 153 to 445 times faster than the default. Aggressive designs are invalid more often than unconditional samples but include much faster designs, and the budget of 20 runs keeps the best of them.

K.5 Outcomes

A timeout means that the tool did not finish within 30 minutes. Over budget means that synthesis returned a latency but at least one resource exceeds 80% of the device. Requests for large speedups lead to aggressive designs that time out or overflow more often, which lowers validity and raises the speedup of the best designs.

Table 18: Synthesis outcomes of the 20 runs per kernel.

Method	Valid	Over budget	Timeout	Failed	Valid rate
BiG-Flow	81	42	85	12	0.368
BiG-Flow, unconditional	128	29	59	4	0.582
Random legal designs	61	5	131	23	0.277

K.6 Database overlap and budget accounting

Model proposals hit the database of AutoDSE designs more often than random designs. Treating a hit as free would give learned methods a larger budget, so we select unique designs before lookup and count every returned measurement as one synthesis run, whether it comes from the database or from a new synthesis. Duplicate proposals are kept for analysis but never create additional projects.

L Flow-initialized local search

The hybrid spends a total budget of 20 synthesis runs per kernel. The first k runs are BiG-Flow proposals. The remaining runs evaluate legal one-site neighbours of the b best valid designs so far, in rounds, and update the seeds after every round. This study uses the $p = 0.05$ model of split 3 with a single request of 10× speedup. The allocation k and beam width b were selected on the eight validation kernels of split 3, where $k = 5$ and $b = 2$ reached a best valid speedup over the default of 6.30× against 6.14× for $k = 3$. The test run evaluates the hybrid, 20 direct BiG-Flow proposals, and the five seed proposals alone on the eleven test kernels of split 3. All 165 designs were synthesized, and no cached result was visible before its run was counted.

Table 19: Fixed-budget test of the hybrid. Speedup is the geometric mean over the eleven kernels of the best valid latency relative to the default design, retaining the default where an arm has no valid design. Ratios are kernel bootstraps over that estimand.

Arm	Evaluations	Speedup over default	Kernels with a valid design
One-shot BiG-Flow	20 proposals	6.894	11/11
Hybrid	5 proposals + 15 neighbours	4.695	9/11
Seeds only	5 proposals	2.411	6/11

Hybrid over one-shot = 0.681×, with 6 wins and 4 losses. Hybrid over seeds = 1.948×. One-shot over seeds = 2.860×.

Under the same budget of 20 runs, twenty direct proposals beat five proposals refined by local search. The refinement does improve on its own seeds by 1.948× and wins six kernels against one-shot, including a 5.6× gain on `gesummv_1300`, but on `trmm_80` the fast design that one-shot sampled among its twenty proposals was absent from the five seeds and unreachable by one-site moves, which costs the hybrid a factor of 386 on that kernel and dominates the geometric mean. Table 20 lists every kernel. The result favours spending a small budget on diverse direct proposals before local refinement, which is the division of labour that Section 5.4 adopts.

Table 20: Best valid cycles per kernel under a 20-evaluation budget for twenty direct proposals (one-shot), the five seed proposals alone, and the hybrid that refines those seeds with fifteen legal one-site neighbours. The last column is one-shot cycles over hybrid cycles.

Kernel	Default	AutoDSE	One-shot	Seeds	Hybrid	Hybrid gain
atax_124	118.7k	4,085	4,370	4,401	4,232	1.03×
doitgen_60	572.3k	322.2k	322.2k	572.3k	362.2k	0.89×
gemm-blocked_64	1.631M	5,646	10.5k	12.1k	7,167	1.47×
gemm-p_1200	1.324B	682.090M	77.003M	1.324B	53.758M	1.43×
gesummv_1300	21.973M	1.791M	11.065M	11.910M	1.965M	5.63×
jacobi-2d_1298	1.688B	118.294M	884.114M	1.688B	1.688B	0.52×
seidel-2d_398	730.372M	689.695M	730.372M	730.372M	730.372M	1.00×
spmv-ellpack_494	6,110	2,897	2,898	2,941	2,922	0.99×
symm-opt_1200	8.468B	7.292B	7.413B	7.413B	7.413B	1.00×
syrk_1200	1.444B	31.739M	391.877M	1.444B	391.877M	1.00×
trmm_80	4.402M	6,890	11.4k	4.402M	4.402M	0.00259×

M Exploratory compiler-in-the-loop policy optimization

M.1 Objective

The endpoint model defines a categorical policy over trajectories through the analytic transition of Appendix D.2, so its trajectory log likelihood $\log p_\theta(\tau)$ is exact. Following Graph-GRPO (Zhu et al., 2026), each update draws G trajectories per training kernel from a shared source assignment and uses a leave-one-out baseline,

$$A_i = R_i - \frac{1}{G-1}\sum_{j\neq i} R_j, \qquad \mathcal{L}_{\mathrm{RLOO}} = -\frac{1}{G}\sum_i A_i \log p_\theta(\tau_i),$$

with a KL penalty to the pretrained policy and the offline endpoint-flow loss as an anchor. The reward R_i comes from Vitis HLS. A valid design receives a latency-quality reward after all five resource estimates are checked, an over-budget design receives a penalty that grows with its overflow, and a timeout or tool failure receives a fixed penalty. Every request of an update is synthesized before the update is applied, and the rollout policy is frozen while its synthesis batch is pending.

Table 21: Policy-optimization settings.

Setting	Value
Initialization	pretrained $p = 0.05$ model after ten surrogate-reward updates
Compiler-oracle updates	14
Kernels / trajectories per update	4 / $G = 8$
Rollout steps	10
Trainable scope	pragma-site transformer
Learning rate / weight decay	$2 \cdot 10^{-5}$ / 0
Sampling temperature	1.5
KL coefficient	0.02
Endpoint-flow anchor	0.15
Gradient clip	5
Failure reward	−1.65
Oracle	Vitis HLS 2025.1 C-synthesis

Training charges 448 requests for 442 unique assignments, of which 376 are fresh syntheses, 64 database hits, and two cache hits. Checkpoints must improve a validation DSE metric while staying inside a forward-error gate, and an increase in reward together with collapsing entropy or uniqueness is treated as policy collapse.

M.2 Results

We evaluate the tuned policy against the pretrained flow on the eleven test kernels of the first split with 20 charged evaluations per kernel and report the ratio of geometric-mean best valid latency. Two test seeds give 1.19× with CI $[0.97, 1.71]$ and 1.25× with CI $[1.02, 1.73]$. Four fixed-budget validation seeds give ratios of 1.13, 1.00, 1.07, and 1.05, and an equal-seed hierarchical bootstrap gives 1.06× with CI $[0.96, 1.23]$. In the second test seed the tuned policy produces 72 valid designs against 55 for the pretrained flow and reaches 0.754 of the AutoDSE latency against 0.605. The evidence is suggestive and does not establish an effect, so policy optimization stays out of the main claims. Figure 8 shows the budget curve of that seed and Figure 9 one held-out kernel.

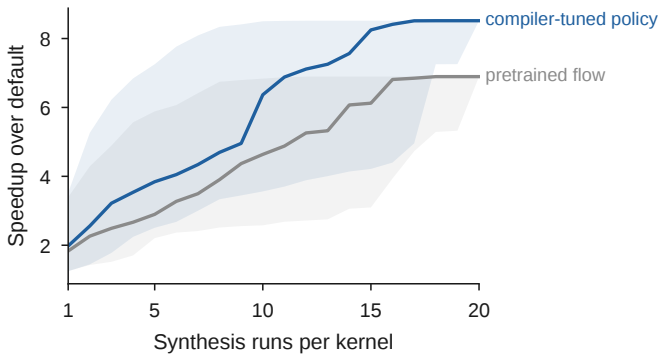

Figure 8: Best valid speedup over the default design against charged evaluations for the pretrained flow and the compiler-tuned policy on test seed 17031. Lines are medians and bands 2.5–97.5% quantiles over 4,000 random candidate orderings.

a

```
L2: for (i = 0; i < 1300; i++) {
  #pragma HLS pipeline PIPE_L2
  L3: for (j = 0; j < 1300; j++) {
    #pragma HLS unroll factor= PARA_L3
    tmp[i] += A[i][j] * x[j];
}}
L4: for (i = 0; i < 1300; i++) {
  #pragma HLS pipeline PIPE_L4
  L5: for (j = 0; j < 1300; j++) {
    #pragma HLS unroll factor= PARA_L5
    y[i] += B[i][j] * x[j];
}}
L6: for (i = 0; i < 1300; i++) {
```

b

site	pretrained	tuned
PARA_R1	2	1
PARA_R3	4	1
PIPE_L2	off	none
PARA_L3	130	65
PIPE_L4	off	none
PARA_L5	1	52
PARA_L6	10	1

c

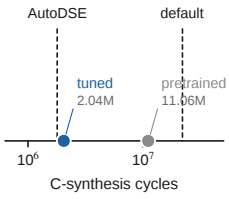

Figure 9: Held-out `gesummv_1300`. (a) Source with the decisive pragma sites. (b) Best valid assignment of the pretrained flow and of the tuned policy. Bold marks sites that differ. (c) C-synthesis cycles against the default design and the archived AutoDSE result.

N Baseline implementations

QoR predictors. HARP and HierarchicalMoE are retrained with their public code for their full 1,000-epoch schedules on the exact split files, select checkpoints by validation $\log_{10}$-cycle error, and are evaluated on the test kernels once. HARP's cosine schedule improves late, so the full schedule is kept. Kernels whose archived labels use a half-scale normalization are normalized per kernel.

Ranking the designs of the AutoDSE run. For the fast-design study, HARP and HierarchicalMoE predict the QoR of all 3,794 designs that the AutoDSE runs of the 11 test kernels of split 3 synthesized successfully, keep the designs predicted to fit the device, and take the 20 with the lowest predicted latency per kernel. These designs were already synthesized, so the baseline measures how well a predictor ranks the designs of a finished AutoDSE run, and its cost is the cost of that run.

Predictor-guided search. HARP and HierarchicalMoE also search the design space on their own, following the online evaluation of HierarchicalMoE (Li et al., 2025). For each test kernel we predict the QoR of 75,000 legal designs, keep the designs predicted to fit the device, and synthesize the 20 with the lowest predicted latency. The candidates are all legal designs when the design space is smaller and a uniform sample otherwise, which covers every pragma site of a large design space, whereas a depth-first order would change only the last sites. Designs are encoded exactly as the training data of the two predictors, which we verify on the stored designs of every test kernel.

Language-model methods. LLM-DSE (Wang et al., 2025) searches with language-model agents that select promising designs, propose new pragma values, and arbitrate between the proposals. We keep its agents and prompts, replace its compiler with Vitis HLS 2025.1, and state our 30-minute timeout in the prompts. Every iteration synthesizes its batch of proposals in parallel, as the original loop does, and returns the results to the agents before the next iteration. A proposal found in the database is read from it and counts as a run. We run LLM-DSE with GPT-4o, the model of the original work, and with the newer GPT-4.1, GPT-5 mini, and GPT-5. ForgeHLS (Peng et al., 2025) releases a recipe for fine-tuning a language model with LoRA on pragma-annotated code. We apply the recipe to the pragma sites of Expanded HLSyn with the released Llama 3.2 1B model, with Llama 3.2 3B, and with Qwen2.5 7B, the model size of the original work, and sample 20 designs per kernel with the prompt for low latency.

Table 22: All methods of the fast-design study on the 11 test kernels of split 3, with runs and hours per kernel as in Table 2. The last column stops AutoDSE on every kernel after the synthesis hours that the method of the row used there. Rows marked † are ours.

Method	Runs	Hours	Kernels	Speedup	vs. AutoDSE	AutoDSE, same hours
Search with synthesis feedback						
AutoDSE, full run	156	24.4	11/11	11.3×	1.00	
AutoDSE, first 20 runs	20	1.6	11/11	6.2×	0.55	
LLM-DSE, GPT-4o	20	0.2	11/11	3.5×	0.31	3.5×
LLM-DSE, GPT-4.1	20	0.4	11/11	3.2×	0.28	3.9×
LLM-DSE, GPT-5 mini	20	1.6	11/11	5.2×	0.46	5.0×
LLM-DSE, GPT-5	20	1.0	11/11	2.9×	0.26	4.6×
Designs of the full AutoDSE run ranked by HARP or HierarchicalMoE						
HARP	20 of 156	24.4	11/11	11.1×	0.98	
HierarchicalMoE	20 of 156	24.4	11/11	10.3×	0.91	
Designs chosen before any synthesis						
HARP-guided search	20	5.8	6/11	4.4×	0.39	6.0×
HierarchicalMoE-guided search	20	10.0	2/11	1.3×	0.12	10.0×
ForgeHLS, Llama 3.2 1B	≤20	0.9	11/11	3.4×	0.30	5.4×
ForgeHLS, Llama 3.2 3B	≤20	2.6	11/11	5.5×	0.49	6.4×
ForgeHLS, Qwen2.5 7B	≤20	4.3	11/11	4.9×	0.43	5.8×
Random legal designs	20	7.5	9/11	5.1×	0.45	6.9×
BiG-Flow, unconditional†	20	2.4	10/11	7.6×	0.67	5.8×
BiG-Flow†	20	5.1	9/11	8.6×	0.76	5.8×

O Algorithmic specification

O.1 Offline training

For each minibatch:

1. Load a design T, its label Y, the static site features, the fixed loop-nest edges, and the CDFG of the kernel.
2. Encode each distinct CDFG once and broadcast its program vector to all designs of that kernel in the batch.
3. Draw $t \sim \mathcal{U}(\epsilon, 1-\epsilon)$, sample every pragma site from the mixture path of Equation 2, and form the paired QoR state $s\,Y$ at $s = 1 - t$.
4. With probability p, replace the QoR state input by zero without changing the velocity target Y.
5. Predict the final pragma values and the QoR velocity, minimize Equations 4 and 5, and update the EMA weights.

Training never invokes an ODE solver or Vitis HLS. The compiler is used only for generated designs that are absent from the database.

O.2 Pragma generation

Given kernel X and requested label $Y^\star$, the sampler performs the following steps.

1. Standardize $Y^\star$ with statistics fit on training kernels and sample the initial pragma state from p_0.
2. For 100 steps, query the network with the program vector and the current QoR state, convert the predicted final values into a categorical transition, and draw a new one-hot state. Loop-nest edges and site features remain fixed.
3. At the final step, restrict each site to its legal pipeline or unroll options and decode one design.
4. Insert the design into the pragma placeholders of the source and run Vitis HLS if the design is absent from the database and the synthesis cache.

The unconditional sampler uses the same code, kernel, source distribution, sample count, and seed, and sets the QoR input to zero at every query.

O.3 Compiler-oracle policy update

For one policy update:

1. Freeze the rollout policy and draw G trajectories for each selected training kernel from a shared source assignment.
2. Select the requests to synthesize independently of cache membership, canonicalize the assignments, and merge duplicate trajectories into one request.
3. Resolve database and cache hits and submit the remaining projects as an index-stable job array. Retry preempted jobs without changing policy parameters or indices.
4. Block until every request has a terminal outcome, form same-kernel leave-one-out advantages, and apply one policy update with the KL and endpoint-flow anchors.

P Metric definitions

QoR prediction. For each design, latency error is $|\log_{10} \hat{L} - \log_{10} L|$. The design average runs over all test designs. The kernel average first averages within each kernel and then over kernels. Resource errors are computed in normalized units, and the five-coordinate mean averages latency and the four resources.

Pragma generation for a requested QoR. Validity divides the number of valid designs by the number of requests. Success at tolerance τ divides the number of requests whose design is valid and whose absolute $\log_{10}$-cycle error is at most τ by the number of requests. The main text uses $\tau = 0.3$, a factor of two in either direction. Target error is evaluated on requests for which both samplers return a valid design, which separates accuracy from validity while validity is reported alongside. Request-level intervals resample requests, and kernel-level intervals resample one difference per kernel.

Finding fast designs. For kernel k, the retained latency is the smallest valid cycle count among the free default design and the synthesized proposals. The speedup over the default is $r_k = L_{\text{default},k}/L_k$, and speedups are aggregated as $\exp(K^{-1} \sum_k \log r_k)$ together with the number of kernels with a valid design. A kernel without a valid proposal keeps a speedup of one. The speedup relative to AutoDSE divides the aggregated speedup of a method by that of the best design of the full AutoDSE run, which equals the geometric mean of $L_{\text{AutoDSE},k}/L_k$. Synthesis hours add the Vitis HLS run times of the designs of a method, one after another.

Policy optimization. Training reward, valid fraction, average measured reward, and best valid latency are reported separately. Multi-seed validation weights seeds equally.

Q GENERATED-DESIGN CASE STUDIES

Table 23: Two generation requests with conditional and unconditional sampling. Error gain is the reduction of the absolute $\log_{10}$-cycle error.

Split	Kernel	Requested cycles	Conditional	Unconditional	Error gain
1	`trmm-opt_240`	57.70M	58.63M	385.8k	2.168
2	`2mm_220`	135.7k	155.5k	66.77M	2.633

For `trmm-opt_240`, conditioning raises one inner unroll factor from 1 to 6 and removes a pipeline that the unconditional sampler chose, moving the output from 385,752 cycles to 58,632,553 cycles against a 57,704,347-cycle request. The faster design does not answer this request, because generation is scored against the request. For `2mm_220`, Figure 10 shows the source locations. The unseen kernel computes two matrix products, and the request is the QoR of a measured design with 135,661 cycles. With conditional sampling, BiG-Flow pipelines the first product and unrolls the two inner reductions by 30 and 190, and Vitis HLS reports 155,461 cycles, 14.6% above the request. With unconditional sampling, the same model and seed omit the pipeline and unroll by 2 and 1, which gives a valid design with 66.8 million cycles. The conditional arm raises both inner unroll factors and enables the pipeline of the first product, whereas the unconditional sampler chooses minimal parallelism. Both designs are valid.

a

```
L1: for (j = 0; j < 190; j++) {
  #pragma HLS pipeline PIPE_L1
  L2: for (k = 0; k < 210; k++) {
    #pragma HLS unroll factor= PARA_L2
    tmp[i][j] += alpha*A[i][k]*B[k][j];
}}
L4: for (j = 0; j < 220; j++) {
  #pragma HLS pipeline PIPE_L4
  D[i][j] *= beta;
  L5: for (k = 0; k < 190; k++) {
    #pragma HLS unroll factor= PARA_L5
    D[i][j] += tmp[i][k]*C[k][j];
}}
```

b

site	conditional	unconditional
PIPE_L1	**on**	**none**
PARA_L2	**30**	**2**
PIPE_L4	none	none
PARA_L5	**190**	**1**

c

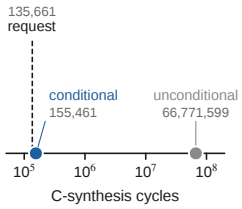

Figure 10: **A generated design for an unseen kernel.** (a) Source of `2mm_220` with the four decisive pragma sites. (b) Values decoded with conditional and unconditional sampling. Bold marks sites that differ. (c) Synthesized cycles of both designs against the request.

Table 24: Best valid cycles in three policy candidate sets of test seed 17031.

Kernel	Pretrained	Tuned	AutoDSE
`gesummv_1300`	11.06M	**2.04M**	1.79M
`gemm-p_1200`	77.00M	**50.16M**	682.09M
`gemm-blocked_64`	10,520	**8,487**	5,646

The `gemm-p_1200` row shows that the result of the AutoDSE run is not an upper bound. That run covered 116 points, and both candidate sets find compiler-evaluated designs outside its trajectory. AutoDSE remains better on `gesummv_1300` and `gemm-blocked_64`.

R Compute and reproducibility

R.1 Resources

Offline BiG-Flow runs use a single GPU. The shared field trains with batch 64. The two-network pipeline uses micro-batch 32 with two-step gradient accumulation to match the effective batch on 12 GiB devices. HARP and HierarchicalMoE run their complete 1,000-epoch schedules. Synthesis projects run as job arrays on CPU workers. Completed results are stored in an append-only cache keyed by kernel and canonical pragma point with file locking for concurrent access, and retried jobs keep their index and never overwrite a terminal outcome.

R.2 Tool and validity checks

The project generator was validated by resynthesizing archived database points and reproducing their cycle counts exactly. A design is valid when synthesis finishes with a positive cycle count and every normalized utilization is at most one, that is, every resource fits within 80% of the device. Dataset labels are normalized as utilization divided by 0.8 times device capacity, and all five device resources are checked at a normalized value of one. Eighteen archived designs pass the four modeled resources but exceed URAM, which is why validity is checked on all five resources after synthesis.

Checkpoints record the model configuration, data and source fingerprints, raw and EMA weights, optimizer state, epoch, step, and policy initialization. Candidate manifests store the requested target, seed, policy hash, canonical assignment, cache source, job index, and terminal outcome.

R.3 Commands

```
python scripts/hls_global_fold.py --fold 1 \
  --ckpt-dir checkpoints/hls_global_only_propdrop005_lko_fold1
python scripts/hls_global_fold.py --fold 2 \
  --ckpt-dir checkpoints/hls_global_only_propdrop005_lko_fold2

python scripts/hls_pipeline_fusion_fold.py --fold 1 \
  --ckpt-dir checkpoints/hls_pipeline_global_propdrop005_lko_fold1
python scripts/hls_pipeline_fusion_fold.py --fold 2 \
  --ckpt-dir checkpoints/hls_pipeline_global_propdrop005_lko_fold2

python scripts/hls_eval.py --ckpt <fold1-e174.pt> --weights raw \
  --split test --targets-per-kernel 32 --n-per-target 1 \
  --property-schedule training --decode legal --seed 0 \
  --out <fold1-cond.json> --emit-projects <fold1-cond-projects>
# Repeat with --uncond using the identical seed and target count.
# Split 2 uses <fold2-e49.pt> and --targets-per-kernel 21.
python scripts/hls_oracle.py --emit-projects <projects/designs.json> \
  --out <bundle>
bash scripts/hls_oracle_vast.sh <run-dir>
python scripts/hls_inverse_compare.py \
  --a <cond.json> --bundle-a <cond-bundle> \
  --b <uncond.json> --bundle-b <uncond-bundle> \
  --name-a conditional --name-b unconditional --out <compare.json>
python scripts/hls_inverse_success.py \
  --a <cond.json> --bundle-a <cond-bundle> \
  --b <uncond.json> --bundle-b <uncond-bundle> \
  --name-a conditional --name-b unconditional \
  --thresholds 0.1 0.3 0.5 --out <all-target-success.json>

python scripts/hls_dse.py \
  --ckpt <global-e99.pt> --weights ema \
  --split test --speedups 10 30 100 300 1000 \
  --n-per-target 32 --synth-budget 20 --seed 2026 \
```

```
  --arms cond uncond random \
  --out <dse.json> --emit-projects <projects>
python scripts/hls_dse_score.py --dse-json <dse.json> \
  --bundle <bundle> --budget-cap 20 --seed 0 --out <scored.json>

python -m pytest -q tests/test_graph_grpo.py tests/test_vitis_reward.py
python scripts/hls_dse_compare.py --a <policy-scored.json> \
  --b <base-scored.json> --name-a policy --name-b base
```

R.4 Artifact index

Table 25: Machine-readable sources of the main claims.

Claim	Artifact pattern
Inverse control on both splits	`runs/hls_nohup/global_only_propdrop005_lko_fold{1,2}*cond_vs_uncond.json`, `fold{1,2}_inverse_all_target_success.json`, and `inverse_folds_five_qor_target_error.json`
Forward HARP and HierarchicalMoE	`runs/harp/compare_fold{1,2}_bigflow_vs_{harp,hmoe}.json` and `compare_folds12_*.json`
Fast designs	`runs/hls_guidance/global_true_control_g1{,_scored}.json`, `global_true_control_all_kernel_pairwise.json`, and `scripts/hls_dse_score.py`
Program-context ablation	`runs/hls_full_cdfg/compare_*.json`
Pipeline controls	`compare_global_unified_vs_full_pipeline*.json` and `compare_global_vs_pipeline_inverse_legal.json`
Policy study	`final_warm_full_test_seed7031/`, `final_rwf6_full_test_seed17031/`, and `warm_final_four_seed_fixed_budget_summary.json`
Paper values	`iclr2027/make_tables.py`, `make_appendix_tables.py`, `make_figures.py`, and `make_schematics.py`

S Full per-kernel tables

The following tables are generated from the same per-design records as the main-text summaries and expose the kernel heterogeneity behind every mean.

Table 26: Per-kernel latency error of every program-conditioning variant on split 3, as the mean absolute error of $\log_{10}$ cycles.

Kernel	Sites	P+L	Local	P+.1L	Pooled	HARP	HMoE
atax_124	0.256	0.793	0.368	0.343	0.261	0.175	0.499
doitgen_60	0.205	0.187	0.214	0.320	0.190	0.047	0.202
gemm-blocked_64	0.395	0.432	0.440	0.493	0.434	0.579	1.188
gemm-p_1200	0.532	1.308	1.401	0.312	0.294	0.371	0.276
gesummv_1300	0.764	0.901	0.679	0.595	0.655	1.391	2.003
jacobi-2d_1298	0.211	1.965	1.514	0.221	0.514	1.451	1.410
seidel-2d_398	0.622	1.947	1.373	0.616	0.313	0.436	0.758
spmv-ellpack_494	0.426	0.265	0.320	0.340	0.261	0.627	0.802
symm-opt_1200	0.446	0.274	1.522	0.456	0.151	0.160	0.200
syrk_1200	1.529	1.644	0.787	0.886	1.248	1.462	1.434
trmm_80	0.515	0.456	0.602	0.401	0.583	0.440	0.890

Table 27: Split 1 per-kernel latency prediction error, as the mean absolute error of $\log_{10}$ cycles.

Kernel	n	BiG-Flow	HARP	HMoE	Lowest
bicg_124	2934	0.220	0.266	0.217	HMoE
doitgen-red_60	383	0.212	0.369	0.274	BiG-Flow
gemm-blocked_64	968	0.387	0.985	0.792	BiG-Flow
gemver_400	39	0.126	0.223	0.117	HMoE
gesummv_1300	129	0.720	1.187	0.444	HMoE
jacobi-1d_118	593	0.628	0.336	0.755	HARP
jacobi-2d_248	81	1.156	1.577	1.690	BiG-Flow
spmv-ellpack_494	223	0.423	0.645	0.405	HMoE
stencil-3d_32	246	0.467	0.254	1.317	HARP
symm_1200	367	0.081	0.039	0.040	HARP
trmm-opt_240	33	0.828	0.316	0.200	HMoE

Table 28: Split 2 per-kernel latency prediction error, as the mean absolute error of $\log_{10}$ cycles.

Kernel	n	BiG-Flow	HARP	HMoE	Lowest
2mm_220	310	0.708	0.571	0.472	HMoE
doitgen-red_30	150	0.609	0.534	0.274	HMoE
gemm-ncubed_64	265	0.545	1.228	1.828	BiG-Flow
gemm-p_80	67	0.318	0.338	0.626	BiG-Flow
gemver_2000	156	0.442	0.220	0.376	HARP
gesummv_250	309	0.176	0.453	0.691	BiG-Flow
gesummv_90	509	0.509	0.301	0.598	HARP
md_256	3184	0.278	0.166	2.186	HARP
spmv-crs_1666	21	2.437	3.184	7.040	BiG-Flow
syrk_1200	78	0.625	1.041	0.896	BiG-Flow
trmm-opt_1200	349	0.174	0.282	0.154	HMoE

Table 29: Split 1 per-kernel latency error of generated designs, as the mean absolute $\log_{10}$-cycle error between the requested and the synthesized latency on requests valid under both arms.

Kernel	n	Conditional	Unconditional	Δ	Lower error
bicg_124	32	0.135	0.534	-0.398	conditional
doitgen-red_60	29	0.090	0.282	-0.192	conditional
gemm-blocked_64	29	0.749	0.594	0.155	unconditional
gemver_400	11	0.156	0.167	-0.011	conditional
gesummv_1300	17	0.568	0.531	0.037	unconditional
jacobi-1d_118	31	0.066	0.263	-0.197	conditional
jacobi-2d_248	4	0.654	1.041	-0.387	conditional
spmv-ellpack_494	32	0.220	0.375	-0.154	conditional
stencil-3d_32	17	0.190	0.205	-0.016	conditional
symm_1200	24	0.026	0.075	-0.049	conditional
trmm-opt_240	13	0.997	1.395	-0.398	conditional

Table 30: Split 2 per-kernel latency error of generated designs, as the mean absolute $\log_{10}$-cycle error between the requested and the synthesized latency on requests valid under both arms.

Kernel	n	Conditional	Unconditional	Δ	Lower error
2mm_220	16	0.115	1.587	-1.472	conditional
doitgen-red_30	17	0.097	0.098	-0.001	conditional
gemm-p_80	19	0.652	1.029	-0.377	conditional
gemver_2000	8	0.112	0.135	-0.023	conditional
gesummv_250	12	0.202	0.158	0.044	unconditional
gesummv_90	20	0.218	0.184	0.034	unconditional
md_256	21	0.258	0.850	-0.592	conditional
spmv-crs_1666	21	0.122	0.175	-0.052	conditional
syrk_1200	9	0.385	0.761	-0.376	conditional
trmm-opt_1200	16	0.002	0.002	0.000	unconditional

Table 31: Best valid cycles by kernel in the fast-design study. AutoDSE runs is the number of designs synthesized by the full AutoDSE run.

Kernel	Default	AutoDSE	BiG-Flow	Unconditional	Random	AutoDSE runs
atax_124	118.7k	4,085	4,232	8,346	4,335	1486
doitgen_60	572.3k	322.2k	–	472.2k	512.3k	387
gemm-blocked_64	1.631M	5,646	6,336	10.0k	81.3k	1148
gemm-p_1200	1.324B	682.090M	50.873M	92.481M	90.665M	116
gesummv_1300	21.973M	1.791M	1.425M	10.642M	1.962M	156
jacobi-2d_1298	1.688B	118.294M	884.114M	884.115M	–	60
seidel-2d_398	730.372M	689.695M	–	–	730.252M	277
spmv-ellpack_494	6,110	2,897	2,922	4,381	4,383	3713
symm-opt_1200	8.468B	7.292B	8.014B	7.343B	7.520B	110
syrk_1200	1.444B	31.739M	391.787M	33.120M	–	95
trmm_80	4.402M	6,890	9,890	10.6k	12.4k	128

Table 32: Designs generated for each requested speedup on split 3.

Request	Measured	Valid	Valid rate	Median achieved	Best achieved
10×	64	27	0.422	1.92×	338.1×
30×	62	27	0.435	3.67×	445.1×
100×	63	18	0.286	2.09×	173.3×
300×	69	25	0.362	2.08×	445.1×
1000×	46	16	0.348	2.01×	152.9×

Table 33: Per-kernel best valid cycles for the pretrained flow and the compiler-tuned policy on test seed 17031.

Kernel	AutoDSE	Pretrained	Tuned	Tuned/pretrained
atax_124	4,085	4,370	4,347	1.01×
doitgen_60	322.2k	322.2k	326.3k	0.99×
gemm-blocked_64	5,646	10.5k	8,487	1.24×
gemm-p_1200	682.090M	77.003M	50.155M	1.54×
gesummv_1300	1.791M	11.065M	2.044M	5.41×
jacobi-2d_1298	118.294M	884.114M	884.114M	1.00×
seidel-2d_398	689.695M	808.619M	730.173M	1.11×
spmv-ellpack_494	2,897	2,898	2,898	1.00×
symm-opt_1200	7.292B	7.413B	7.413B	1.00×
syrk_1200	31.739M	391.877M	391.877M	1.00×
trmm_80	6,890	11.4k	11.4k	1.00×

Table 34: Synthesis outcomes of the fast-design study and the policy study.

Arm	Measured	Valid	Over budget	Timeout	Failed	Valid rate
Fast designs, BiG-Flow	220	81	42	85	12	0.368
Fast designs, unconditional	220	128	29	59	4	0.582
Fast designs, random legal	220	61	5	131	23	0.277
Policy, pretrained	194	55	41	97	1	0.284
Policy, tuned	194	72	39	82	1	0.371