Balor: HLS Source Code Evaluator Based on Custom Graphs and Hierarchical GNNs

Emmet Murphy
ETH Zürich
Zürich, Switzerland

Lana Josipović
ETH Zürich
Zürich, Switzerland

ABSTRACT

While High-Level Synthesis (HLS) enables circuit generation directly from languages like C/C++ and OpenCL, optimal implementations require additional design specification through compiler directives. Automated optimization of these directives requires evaluation of candidate designs, but is bottlenecked by the high computational cost. Graph Neural Networks (GNNs) have recently emerged as the state-of-the-art for estimating the required Quality of Results (QoR) directly from high-level source code, but they come with associated challenges: the difficulty of information propagation, the complexity of software-oriented graph representations, and high levels of extraneous computation increase both error and computational cost. We present Balor, an HLS source code evaluator, which consists of a graph compiler and a GNN-based QoR estimator. The modular graph compiler is tailor-made for HLS QoR estimation, producing smaller graphs that contain only HLS-specific information. Balor analytically propagates important information across the graph, allowing us to pivot to small, local GNNs, while outperforming more expensive networks. Additionally, we make use of the natural hierarchy of an HLS kernel, and cluster the graph by basic block, to further propagate and process information—without the need for additional datasets. Combined, our contributions simultaneously reduce estimation error by 41%, and computational cost by 82%. Balor is open-sourced at github.com/emmet-murphy/balor

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; **Software tools for EDA**; • **Computing methodologies →** *Supervised learning by regression.*

KEYWORDS

HLS, GNNs, QoR estimation

ACM Reference Format:
Emmet Murphy and Lana Josipović. 2024. Balor: HLS Source Code Evaluator Based on Custom Graphs and Hierarchical GNNs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676788

1 INTRODUCTION

In the modern computing landscape, FPGAs are an attractive choice of accelerator for many workloads. They offer flexibility of implementation and low power consumption when compared to GPUs, and low turnaround times offer quicker adaptation versus ASIC implementations. High-level synthesis (HLS) aims to reduce the barriers to entry to FPGA accelerators by generating implementations directly from high-level languages such as C/C++ and OpenCL. However, the use of HLS is still impeded by several issues.

Hardware Expertise Requirements: Although HLS abstracts away many low-level details of hardware design, it still requires hardware expertise—unoptimized software kernels often perform several orders of magnitude worse on an FPGA than on a CPU. Issues such as buffer management, memory access patterns, loop pipelining, parallelism, and more must be dealt with in the kernel's source code by a hardware engineer, a process known as design space exploration (DSE). This problem is compounded by the fact that any difference in use case requires the kernel to be re-optimized: different operating frequencies, different chips, different HLS tools or tool versions require different source code implementations in order to achieve optimal performance. For years, research has aimed to relieve this burden by automating the process of DSE for HLS.

Difficulty of Optimization Evaluation: For HLS, even the most commonly applied optimizations must be both well-parameterized and judiciously applied. The possible optimizations therefore form a large search space of designs to be evaluated. For economic searching, DSE for HLS relies on sophisticated techniques from the field of multi-objective optimization problems (MOOPs) such as search space pruning, search space partitioning, analytical modeling of optimization dependencies, and heuristic-guided searching [21, 25].

High Cost of Candidate Design Evaluation: The cost of candidate design evaluation further impedes DSE. For a large kernel, a MOOP solver would need a budget exceeding thousands of evaluations, but the computational cost of HLS means a single evaluation can take hours. As the Quality of Results (QoR) estimates after HLS have high error, placement and routing must also be performed in order to find truly Pareto-optimal designs, pushing the computational cost even higher. As such direct executions make HLS DSE computationally infeasible, much work has gone into exploring HLS QoR estimation.

Why Graph Neural Networks? Graph neural networks (GNNs) are the current state-of-the-art in data-driven HLS QoR estimators [8, 24]. This is because, uniquely among data-driven estimators, they fully encode the interactions between source code structure and compiler directives. By representing an HLS kernel as a graph, a GNN can encode both its contents and topological structure into a single vector representation. With all the relevant information represented numerically, traditional estimation techniques (most commonly deep learning models) can be used to map to the QoR.

While traditional compiler graph representations benefit from decades of refinement, new interactions with GNN architectures require us to re-evaluate many details of their construction. Previous works [8, 15, 23, 24] have taken steps in this direction, but are still hindered by oversmoothing, poor information propagation, and extraneous computation. These problems come from the program representation as well as the GNN architecture itself, so any solution must engage with both.

Balor: An Open-Source, Cost-Effective, Directive-Focused HLS Source Code Evaluator. In this work, we present Balor, an HLS source code evaluator, which consists of a custom graph compiler and a GNN QoR estimator. Balor capitalizes on the following insights:

• Typical compiler graph representations contain complex software implementation details irrelevant to the HLS process and omit HLS-relevant hardware information. In Section 4, we introduce a graph representation tailor-made for the HLS process. Our graph representation is small enough to strongly reduce computational cost, and results in similar error.

• Compiler directives must be carefully incorporated into the HLS graph representation due to their impact on the HLS process, but existing methods are impaired by the fundamental weakness of GNNs. In Section 5, we push for an entirely new philosophy of graph annotation, strongly reducing estimation error by analytically propagating the directive information to all relevant nodes.

• Until now, GNN architectures have prioritized long-range interactions, sending information as far as possible across the graph. But due to the structure of GNNs, this leads to extremely high levels of redundant processing, and fails to deliver on its promise of increased accuracy. In Section 6, we discuss a series of GNN design choices that alleviate this issue: by making our GNN architectures local, deep, and hierarchical, we significantly reduce their computational cost while also reducing estimation error.

For evaluation, we perform QoR estimation on a set of 25 Machsuite [20] kernels, both with and without these insights, and show that they reduce the resource estimation error by 41%, timing estimation error by 41%, and computational cost by 82%. As these insights are general and transferable, both to recent state-of-the-art works [8, 24] and future ones, we envision that our open-source framework can serve as a foundation for accurate, cost-effective HLS DSE.

2 BACKGROUND

2.1 Graph Neural Network Theory

Figure 1 shows the process of GNN-based QoR estimation, which has three data structures: graph representation, graph embedding, and the QoR estimates. The input is a list of vectors (one for each node), and a list of edges. The vectors consist of the metrics of interest: for Balor's graph compiler, these are primarily node type, data type, and directive information. The output of a GNN is also a list of vectors, the node embeddings. The output node embeddings contain information on nodes within a certain radius of the original, and the topological structure of the local neighborhood. The key characteristic of GNNs are message-passing layers, which propagate information along the graph edges [37].

Receptive Field: For GNNs, we often measure a particular path in "hops": the number of edges traversed. For a single message-passing layer, a node receives information about nodes one hop away. For each message-passing layer added, the node receives information about nodes further and further away. This gives the concept of a node's receptive field: the set of nodes it receives information from. A GNN with N message-passing layers has an N-hop receptive field.

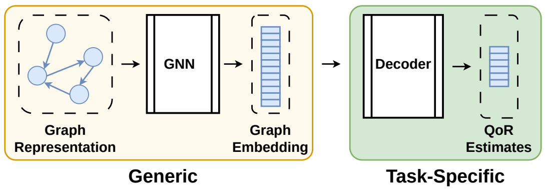


Figure 1: Balor employs an encoder-decoder structure for GNN QoR estimation, as shown in the figure. With a compact HLS-specific graph representation, preprocessor-directive-focused node annotation, and local-focused GNN architectures, it provides cheap and accurate QoR estimates.

Over-smoothing: Since every node of the graph has its own embedding, GNNs are inherently predisposed towards redundancy. This can be seen in a phenomenon known as over-smoothing. With every GNN layer, the receptive field of each node increases in size. It also becomes more similar to the receptive field of the neighboring nodes. This means their computations become more similar, and so their embeddings become more similar. At the limit, a GNN layer calculates the same output once for every node in the graph [1].

The optimal receptive field size decides the optimal number of message-passing layers to include in the network. A smaller receptive field results in a local-focused network, while a larger receptive field results in a global-focused network. Section 6.1 discusses the impact of this on QoR estimation.

Over-squashing: For information to pass from one node on the graph to a distant node, it must pass through nodes connecting them. For distantly connected nodes, information bottlenecks form on the path(s), with connecting nodes becoming responsible for passing more information than their dimensionality allows. This is known as over-squashing, and limits the ability of GNNs to propagate information across the graph [26].

Section 5 discusses the impact of over-squashing on adding compiler directives to the graph.

Graph Embeddings: Once an output embedding has been generated for each node in the graph, they can be aggregated by graph or sub-graph to produce a representation for any region of interest. These representations can then be used as inputs to traditional neural networks for any classification or regression task. The GNN therefore acts as an encoder, transforming graphs into vector representations in feature space. As an encoder, ideally it should be generic: usable as a front-end for any decoder rather than highly coupled to a particular use case. While having a powerful GNN is important, the often-overlooked task-specific decoder must also be complex enough to map from a graph encoding to the desired output. The encoder-decoder structure can be seen in Figure 1.

2.2 GNN QoR Estimators for HLS

Applications: GNNs offer an extremely cheap method of estimating QoR directly from HLS source code, but they currently only achieve rough accuracy. This means they cannot be relied upon to decide the final solution, but their speed of evaluation makes them an excellent tool to prune the search space. By using them to generate a list of candidate designs, which will be evaluated using an HLS tool, a high-quality design can be found in a fraction of the time [5, 23].

Fine-Tuning Cost: Additionally, for GNNs to be of use, the amount of data required to fine-tune them to new kernel types, synthesis settings, FPGAs, or HLS tools must be low enough to justify them: transfer learning has been shown to be effective for GNNs [24], but the cost of changing these conditions cannot be denied.

3 RELATED WORKS

3.1 Analytical HLS QoR Estimators

Analytical estimation models take kernel source code and extract metrics of interest. These metrics are then used as inputs to hand-crafted formulae for QoR estimation. Their analytical nature results in very low computational costs, offering speed-ups of 100×-1000×. Several works on analytical models for DSE were published from 2016-2018 [28, 34, 35]. However, analytical models have high maintenance and portability costs, requiring updates each time HLS tools change. Few analytical models have appeared in recent years: most work in this area focuses on data-driven estimators. Additionally, these works [28, 34, 35] focus heavily on their low performance estimation error, but either do not estimate resource usage or do not present their results on the accuracy of their resource usage estimation. Since there are many cases in which analytical models would struggle to predict an HLS tool's decision-making [25], we conclude that while analytical models perform excellently for performance estimation when well-designed and well-maintained, they fail to estimate resource usage adequately.

3.2 Data-Driven Methods for HLS

Data-driven estimation models are appealing due to the reduced man-hours required to both construct and maintain them, and the possibility of accurate resource estimation. While the cost of generating the data required for good performance is high, the process is automatic when moving to a new use case, and scalable in a way code maintenance is not. However, achieving acceptable estimation error is challenging, as few numerical representations of an HLS kernel properly encode the complex interactions between source code structure and compiler directives.

Non-GNN Data-Driven HLS QoR Estimators: One approach is to estimate the desired QoR from a different, related QoR [4, 16–18] (including pre-route QoR, CPU performance, and on a different FPGA). This is a much easier estimation problem than directly from source code, but the cost of obtaining the input QoR still prohibits DSE. Kwon and Carloni [12] estimate the QoR purely from the compiler directives, encoded in a vector, but this is limited by the fact the estimator receives no explicit information about the source code. Wang and Schafer [29] and Ferretti *et al.* [7] both build a library of micro-benchmarks to compare sections of the kernel to, but their strongly local methods do not capture more global interactions. MPSeeker [36] builds on an analytical model, Lin-Analyzer [35], extracting metrics of interest from the source code and analytical latency estimate. The constructed feature vector is processed using gradient boosting to estimate flip-flop (FF) and look-up table (LUT) results. Goswami *et al.* [9] instead extract metrics from a graph representation of the kernel. Hand-picked metrics struggle to represent the source code structure in a way that encodes all characteristics. While graph metrics do add topological information to the embedding, kernels with vastly different properties may still end up with similar feature vectors.

Non-DSE GNNs for HLS: Ustun *et al.* [27] show that GNNs have superior performance to profiling libraries for estimating the latency of arithmetic chains. PowerGear [15] and HGBO-DSE [11] use GNNs to estimate post-route QoR, but both use post-HLS QoR estimates from an HLS tool, and so are too expensive to be used for DSE. Wu *et al.* [31] do GNN QoR estimation purely from source code, but do not consider the impact of compiler directives.

GNNs for HLS DSE: IronMan-Pro [30] is a GNN-based QoR estimator which takes only the source code as input. However, it only considers resource directives and so cannot explore the full design space. The GNN approach from Ferretti *et al.* [5], GNN-DSE [23], and HARP [24] were the first works to explore the relevant use case: they do not rely on HLS tools, and consider a wide range of pragmas. Ferretti *et al.* 's approach [5] takes an augmented control flow graph as input, with a specific focus on properly encoding read and write dependencies. Without instruction-level encoding, transferability is limited, as the resource cost of each basic block must be learned from scratch for each new kernel. Therefore the instruction-level representation used in GNN-DSE [23] and HARP [24] has won out as the state-of-the-art approach. However, these works miss opportunities to make the graph representation simpler and more concise, causing high levels of redundant calculation. Their accuracy is also strongly impaired by oversquashing. The approach from Gao *et al.* [8] contributes post-route results, as well as additional graph augmentations which reduce estimation error. But those augmentations are highly dependent on an analytical back-end estimation tool and strongly increase the redundant processing.

As our approach is most similar to GNN-DSE [23], HARP [24], and the approach from Gao *et al.* [8]; we compare to them in detail in Sections 4.2 and 5.

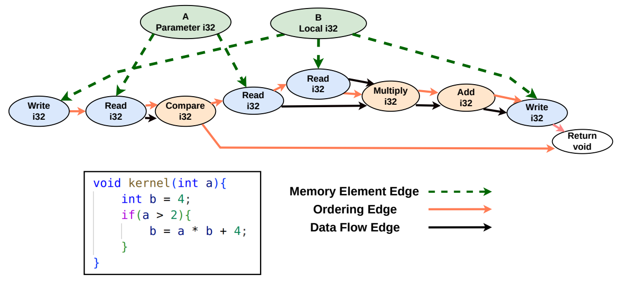
4 GRAPH REPRESENTATIONS: FROM HIGH-LEVEL CODE TO AN HLS-TAILORED GRAPH

While software-oriented graph representations can be extended to include HLS-specific details, many details of their construction conflict with how HLS is performed. Information important to HLS is also often absent in these representations, due to not affecting software compilation. Section 4.1 details how we construct our graph representation, while Section 4.2 compares it to recent works.

4.1 Graph Construction

Balor's graph compiler takes as input an abstract syntax tree (AST). However, it could equally work with lower IRs such as LLVM [13] or MLIR [14]. The output graph representation is heavily inspired by ProGraML [3] and is built as follows:

Action Nodes: A line of high-level code can be broken down into several atomic actions: arithmetic operations, reads and writes, comparisons, function calls, etc. We add a node to the graph for each atomic action, with each type of action having a corresponding node



```
void kernel(int a){
    int b = 4;
    if(a > 2){
        b = a * b + 4;
    }
}
```

Figure 2: HLS-tailored graph representation of a simple kernel. The structure mitigates the inherent weaknesses of GNNs and is economical for processing.

type. These nodes also contain data type and directive information, which we discuss in detail in Section 5.

Ordering Edges: Although actions are not executed sequentially in hardware, we still want to order them sequentially on the graph, as it is a simple way to connect independent actions that are temporally close. We therefore connect every action node to the action node that follows it in program order, using an ordering edge. In the case of variable program order, such as if statements or loops, we add additional ordering edges so that each action node is connected to all possible program order neighbours.

Data Flow Edges: We then add data flow edges to the graph: Atomic actions have data dependencies, and these edges connect data-producing nodes directly to where that data is consumed.

Memory Element Nodes and Memory Element Edges: Each variable declaration is then added to the graph as a memory element node, such as variable 'b' in Figure 2. Different declaration types (local or parameter, scalar or array) are added as different node types. The nodes are then directly connected to all the read and write nodes of that variable using memory element edges.

Subfunctions and Call Edges: Once finished with the main kernel, Balor adds each subfunction to the graph representation, initially leaving them disconnected from the main kernel. The sub-functions are then connected to the correct function call nodes using call edges: one edge to the first action node in the sub-function, and one from the last action node.

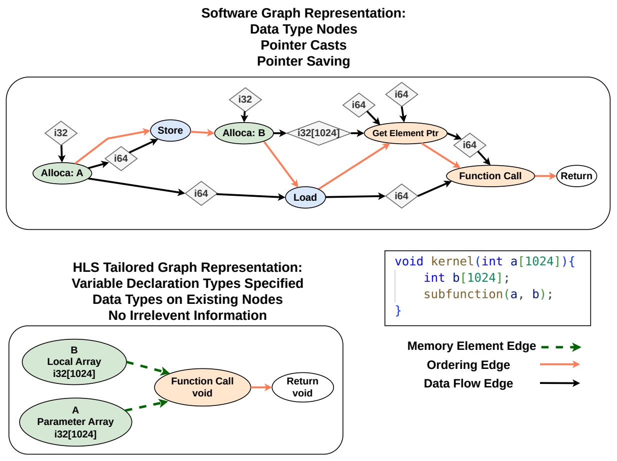
A simple example kernel can be seen in Figure 2, showing the different node and edge types. Our graph representation is as concise as possible, has all relevant information, and directly connects each node to all other relevant nodes. By prioritizing small graphs and direct connections, we optimize GNN performance for both accuracy and cost, as we show in Section 8.

4.2 Comparison to Recent Works

When compared with recent approaches [8, 23, 24] our representation differs in the following ways:

Trivial Nodes: Instructions such as extending and truncating bit width, or casting pointer types, are trivial or nonexistent from a hardware perspective, and are additionally encoded through operation data type. These are all omitted from our representation.

Pointer Saving: Pointers passed to a function as arguments are stored in a pointer to a pointer. This requires both a memory allocation and a store, neither of which happens in a hardware setting, and so neither of these are included in our representation.



```
void kernel(int a[1024]){
    int b[1024];
    subfunction(a, b);
}
```

Figure 3: Comparison of previous graph representations to our HLS-tailored representation. Previous approaches add information inapplicable to an HLS context, and add expensive data type nodes. The reduced graph size results in cheaper, faster inference with improved accuracy.

Pointer Loading: A pointer to a pointer requires two loads in software, but one in hardware. Our representation has only a single load on the graph no matter the address type.

Arithmetic Operation Bit Width: The graph representations used in previous works [8, 23, 24] did not support all arithmetic operations at all bit widths. An example of this is the bitwise or, which would have its inputs extended to 32 bits, and then the output truncated back to 8 bits. Balor's graph compiler instead supports all operations at all bit widths.

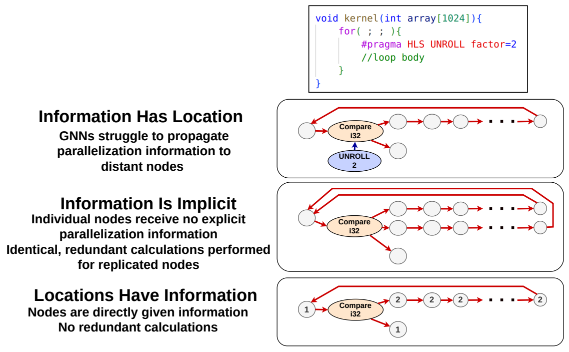
Struct Indexing: In software, a struct is the same as an array, but using names instead of indices. In hardware, a struct becomes an ultra-wide vector, so no indexing occurs. Our representation does not add indexing nodes for struct accesses.

External Functions: In previous works [8, 23, 24], all external functions were marked as the same operation. We use separate labels for common external functions present in HLS kernels, including cosine, sine, and square root.

Variable Types: In a hardware context, an array passed as a parameter is very different from a locally declared array. Yet, from a software perspective, both are simply pointers. Our graph representation labels use different node types for different types of variable declarations.

Improved Data Reuse: The graph representations from previous works [8, 23, 24] often recalculate the pointer to an array element. Balor's graph compiler detects multiple loads and stores to a single memory address in a single basic block, and will only calculate the address once.

Data Type: Previous works [8, 23, 24] all add data type information to the graph by adding a data type node for every use of a variable or a constant. Since this more than doubles the computational cost of a GNN, and also complexifies the topological structure of the graph unnecessarily, Balor instead annotates the data information directly onto the existing nodes. Operations without an output data type have a custom data type label.



```
void kernel(int array[1024]){
    for( ; ; ){
        #pragma HLS UNROLL factor=2
        //loop body
    }
}
```

Information Has Location
GNNs struggle to propagate parallelization information to distant nodes
Compare i32
UNROLL 2
Information Is Implicit
Individual nodes receive no explicit parallelization information
Identical, redundant calculations performed for replicated nodes
Compare i32
Locations Have Information
Nodes are directly given information
No redundant calculations
Compare i32
1
2
1

Figure 4: Visualization of Compiler Directive Philosophies for a Parallelization Directive. "Locations Have Information" is the only philosophy in which all nodes receive the directive information and has no redundant calculations—improving accuracy and reducing computational cost.

Figure 3 shows two graph representations for a simple kernel. Above is a software graph representation of the types used in recent works [8, 23, 24], based on ProGraML [3]. In the lower left is our HLS-tailored graph representation, which removes irrelevant software implementation details, and includes the important information of whether each array was locally declared. The sub-function and call edges have been omitted from both for readability. As we show in Section 8.3, any reduction in graph size reduces the computational cost by the same factor—our graph representation gives cheaper, faster inference with no reduction in accuracy.

5 COMPILER DIRECTIVES

An important question for GNN-based QoR estimation is how to augment the graph representation with compiler directive information. This has been approached from two different angles, which differ fundamentally on a philosophical level. We introduce a third approach, with a third philosophy, which we show achieves high performance at a fraction of the cost. How these affect graph construction for a parallelization directive is shown in Figure 4.

5.1 Philosophies

1) Information Has Location: GNN-DSE [23] and HARP [24] are two works that showcase the first philosophy, "Information Has Location". This approach augments the graph with a node for each compiler directive, and connects that node to the graph using a single edge. This edge connects to the node most relevant to the compiler directive, e.g. the loop comparison node for a loop directive. As compiler directives can impact many nodes on the graph, including distant ones, this technique is strongly impaired by over-squashing. Several of HARP's main contributions are attempts to mitigate this effect, including making the graph denser, as well as directive-specific multi-layer perceptrons (MLPs).

However, increasing graph density also worsens over-smoothing, as the receptive fields expand at a faster rate. While the directive-specific networks are computationally cheap, the approach means most of the network receives no directive information at all. Since these mitigations have associated drawbacks, approaches that avoid the issue entirely can achieve improved performance.

2) Information Is Implicit: The approach from Gao *et al.* [8] instead uses the directives to construct the graph, and to affect how the graph is processed. The preprocessor information is therefore implicit, rather than added to the graph directly.

For parallel directives, an instruction parallelized by a factor of N is added to the graph N times. This direct replication means that after the nodes are aggregated to form a graph embedding, it contains parallelization information. But GNNs are deterministic: a node with the same input embedding and the same neighbors has the exact same output embedding. Since parallelization factors often reach more than a hundred, this approach means only a tiny fraction of the calculations are unique. It is also limiting: parallelization factors interact unpredictably with the resource cost increase. The implicit parallelization information does propagate to nodes near the edges of the loop, allowing them to respond non-linearly, but for large loops many of the nodes receive no parallelization information at all: this reduces the approach to linearly scaling these nodes by their parallelization factor.

For pipelining directives, Gao *et al.* [8] use a multi-model approach, training separate sub-GNNs for pipelined loops versus non-pipelined loops, with additional annotation from an analytical scheduling estimator. Training separate GNNs is a clean solution to the problem of information propagation, but is unfortunately limited to directives with few parameters, such as pipeline and function inline directives. While hybrid (combining analytical and data-driven) approaches give improved performance, we consider only the data-driven side in this work; any benefit to data-driven approaches also benefits hybrid ones.

3) Locations Have Information: We propose a third philosophy, of "Locations Have Information". Compiler directives add information to the graph, each relevant to a subset of nodes. We analytically propagate this information to those relevant nodes by directly annotating them. With the information already propagated, the GNN is relieved of this responsibility. With each node being on the graph only once, there are no redundant calculations. For resource usage, the main piece of information to annotate on each node is its total parallelization factor, as well as the constituent parallelization factors: multi-dimensional array accesses parallelize differently depending on the specific loop. We annotate a total of 4 parallelization factors on each node, but datasets with deeper nesting may require more. Pipelining directive information is almost equally important from a timing point of view, as this type of directive can impact many nodes. It also applies implicit parallelization directives to the sub-loops, which must be annotated with the resulting parallelization factors. We annotate array directives such as partition style and partition factor, which specify how an array should be divided among separate memory elements, directly on the array nodes, and we annotate function inlining directive information on the call nodes.

While "Information Is Implicit" and "Locations Have Information" both overcome the limitations of "Information Has Location", "Locations Have Information" does it a fraction of the computational cost. It is also the only approach in which compiler directive information, the most important information for estimation accuracy, is visible to every node.

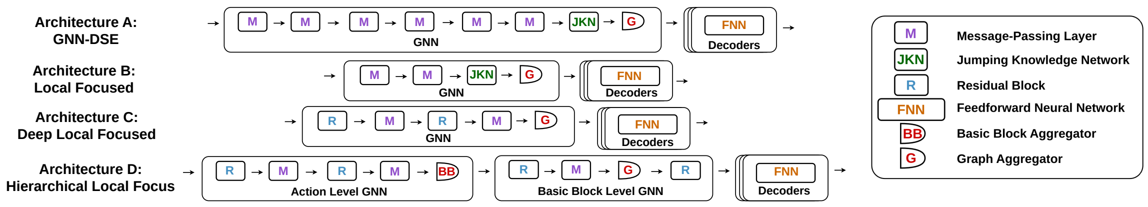


Figure 5: Layerwise GNN Architecture Comparisons. Our local architecture reduces computational cost, the deeper architecture replaces the removed layers with residual blocks for more effective processing, and our hierarchical architecture adds an aggregation by basic block for improved information propagation.

6 GNN ARCHITECTURES: FROM GRAPH TO VECTOR

6.1 Local Architectures

Previously, GNN architectures have been designed to maximize information propagation. They use a relatively high number of message-passing layers: these pull as much information as possible from across the graph, and do very little processing of the currently available information before obtaining more. However, the GNN literature shows that fewer message-passing layers perform best on many graph datasets [1, 33], with local-focused networks outperforming more global ones on both accuracy and cost.

Local networks were not feasible with the compiler directive augmentation philosophy "Information Has Location", as they do not propagate the compiler directive information far enough. "Locations Have Information" analytically propagates the directive information, taking this responsibility away from the GNN, allowing the use of cheaper architectures while reducing estimation error. However, directly pivoting to local networks also means pivoting to smaller networks, and the reduction in computational power can cause increased error, as can be seen in Section 8. While our experiments show that for QoR estimation, graph topology matters only within 1 or 2 nodes, it is undeniable that message-passing layers cannot be simply removed- they must be replaced.

6.2 Deeper Architectures

After switching to a local focus, the next step is therefore to attempt to deepen the network. Deeper networks can give improved performance at increased cost, but the benefits are not guaranteed: network depth is limited both by the richness of the input vector and by the size of the training dataset.

As the network has become shallow due to the removal of message-passing layers, the question becomes what to add instead of them: we want increased depth without increasing the receptive field. Residual blocks [10] preserve gradients well during training, and so stand out as a natural alternative. While our contribution is that adding residual blocks in place of the message-passing layers is beneficial to both accuracy and cost, we do not explore the possible engineering optimizations beyond this obvious first choice.

6.3 Hierarchical Architectures

While we show that very local GNNs have the best performance, it is undeniable that HLS kernels still have some longer-range interactions that cannot be localized. Hierarchical approaches have been shown to improve performance by coarsening the graph, allowing easier propagation of information [8, 11]. But previous approaches discard much of the information they generate, and none have taken advantage of the natural hierarchy of HLS kernels: basic blocks. The state-of-the-art for hierarchical architectures is the approach by Gao *et al.* [8]. First, they perform full QoR estimation on the innermost loops, using both a GNN encoder and a decoder (and additionally requiring the creation of additional innermost-loop-only datasets). The "hierarchical" aspect of their approach is to then replace all the innermost loops in the graph with a single node, containing only that QoR estimation. But as Figure 1 shows, the GNN encoder produced much richer graph embeddings, containing all the information of the QoR estimate and more, and the QoR estimate is a poor second choice. More importantly, using the subgraph embeddings does not require expensive additional datasets, increasing the viability of fine-tuning and transfer learning.

Balor therefore directly uses the subgraph embeddings. Additionally, instead of only clustering nodes in innermost loops, it clusters by basic block, a strategy that applies to every node in the graph.

Our hierarchical architecture can be seen as Architecture D in Figure 5. Our approach adds an interim aggregation by basic block, and treats the resulting embeddings as nodes in the control flow graph. Combined with the control flow edges, message-passing layers can now propagate information along the much coarser control flow graph. The original aggregation now also happens on the control flow graph, but still results in a single graph embedding.

With a simple change to the approach, we can process longer-range interactions without the high computational cost and associated limitations of many message-passing layers. With rich vectors, the network can be deepened further. Basic-blocks-aggregation gives us a natural coarse graph to use, providing benefit to the entire graph. Compared to the approach from Gao *et al.* [8], which separately trains sub-models on specially generated auxiliary datasets, our approach is trained end-to-end on a single dataset.

7 EXPERIMENTAL SETUP

7.1 Graph Compiler

Balor's graph compiler is built on ROSE [19], a compiler infrastructure designed for program analysis and source-to-source transformations. Unlike lowering IRs, which discard high-level information as compilation progresses, ROSE allows the graph compiler access to every high-level detail present in the source code.

7.2 Database

For evaluation, we use the open-source database DB4HLS [6]. From this, we take post-HLS results for 25 kernels from the MachSuite benchmark [20], comprising 36,296 data points. All data is generated using Vivado HLS 2018.2. This means our models only estimate post-HLS results, instead of post-implementation, introducing an additional source of error. However, other works [8] have used post-implementation results successfully, and none of our insights are post-HLS specific: while post-implementation results are required for a final implementation, post-HLS datasets are cheaper to generate and therefore larger, and so are useful for building insight. Additionally, DB4HLS [6] contains explicit compiler directives only for parallelization, array partitioning, and function inlining. While Balor does support explicit pipelining directives, for this evaluation pipelining decisions are done automatically by the HLS tool, and are constant for each kernel.

7.3 GNN Implementation

Our models were trained on NVIDIA GeForce GTX 1080 GPUs. We used a 70:15:15 split for training, validation and testing, the ADAM optimizer with an initial learning rate of 0.0005, and trained for 1000 epochs. Models were selected based on the validation set, and final results are from the test set. Based on previous works [8, 23, 24], the message-passing layers chosen were TransformerConv [22] with a feature size of 64. Graph aggregation was attention-based summation. All residual blocks were two layers deep.

7.4 Architectures

Figure 5 shows the 4 architectures used in our evaluation: GNN-DSE, Local Focused, Deep Local Focused and Hierarchical Local Focused. Recent works [8, 23, 24] all use 6 message-passing layers, in addition to other techniques which all increase the receptive field size. As an example, we take the network from GNN-DSE [23] (Architecture A) and reduce it to only 2 message-passing layers (Architecture B), creating a local-focused variant, which uses a jumping knowledge network (JKN) [32] to combat the effects of over-smoothing. With fewer message-passing layers, this is no longer required, but we maintain it for direct comparison. We then deepen the network using residual blocks (Architecture C), which do not expand the receptive field. Finally, we also use our hierarchical approach (Architecture D), for additional processing on a basic block level.

7.5 Comparison

As the graph constructors of previous works were tightly coupled to their closed-source databases, we create a "baseline" implementation to compare against, which is as similar as possible to GNN-DSE [23]. We chose GNN-DSE for this baseline as both HARP [24] and the approach from Gao *et al.* [8] build directly on GNN-DSE's design decisions. While we can compare "Locations Has Information" directly to "Information Has Location", we cannot compare to "Information is Implicit"—these comparisons would require additional datasets, analytical back-ends, and complex database structures. Our contributions relate to the foundations of GNN estimation, and prioritize simplicity to enable application directly into more complex optimization techniques. Our baseline differs from GNN-DSE in three ways:

Compiler Directives: GNN-DSE uses Merlin Compiler [2] as a front-end, and so only has loop compiler directives. As our database uses Vitis HLS, our baseline also has compiler directives that affect arrays. As the natural parallel of connecting loop directive nodes directly to the loop comparison node, we connect these array directive nodes directly to the array nodes.

Improved Data Reuse: GNN-DSE uses ProGraML as the basis for its graph construction, which does not detect all forms of pointer reuse. Rather than replicating the exact cases where it does and does not detect reuse, our baseline detects all types of pointer reuse.

More Constant Nodes: The rules for adding data type nodes in GNN-DSE [23] come also for ProGraML. A node is added for every variable use, but only once per unique constant. Our baseline instead adds a node for each constant use, regardless of uniqueness.

8 RESULTS AND DISCUSSION

8.1 Estimation Error

Resource percentage estimation error statistics can be seen Table 1, with corresponding timing statistics in Table 2. Both tables show statistics by individual metric, with the final column reporting the average relative to the baseline.

Balor "Information Has Location" A: Relative resource error drops to 0.87 while timing drops to 0.91, showing that Balor's compact graph representation enables more accurate inference. Balor "Information Has Location" B: Our smallest architecture is local-focused, but the size reduction means it does not have enough computational power. It has a relative error of 1.15 for resources and 1.20 for timing. Balor "Information Has Location" C: Our deep, local architecture has very similar results to Arch. A for resources, but a better relative error of 0.83 for timing. Balor "Information Has Location" D: Our hierarchical architecture gives strong error reductions. It has relative errors of 0.68 for resources and 0.65 for timing. Balor "Locations Have Information" A: "Locations Have Information" worsens Arch. A by removing the reason for its large receptive field, giving relative errors of 1.01 for resources and 0.99 for timing. Balor "Locations Have Information" B: The lack of computational power again impedes accuracy, with 0.89 for resource error and 1.06 for timing. Balor "Locations Have Information" C: Our directive annotation philosophy and deep local-focused architecture combine for increased accuracy at lower cost. It has a relative resource error of 0.70 and timing error of 0.67. Balor "Locations Have Information" D: Our best architecture's hierarchical nature provides relative errors of 0.59 for resources and 0.59 for timing. It can be seen that our directive philosophy, "Locations Have Information" gives more desirable results than "Information Has Location", especially when combined with the architectural insights presented in Section 6.

8.2 Graph Size

Table 3 contains statistics on node counts for the different graph representations. Our graph representation has an average relative node count of 0.34. This reduces overall computational cost by a similar factor, which we discuss in Section 8.3. The majority of the reduction comes from the removal of data type nodes, but removing

Table 1: Resource Percentage Estimation Error Statistics (Value Relative to Baseline in Brackets): Our hierarchical architecture with "Information Has Location" gives strong reductions in estimation error.

Graph Repr.	Arch.	Resource Percent Estimation Error Statistics												
		LUTs			FFs			DSPs			BRAMs			Relative Error (All)
		Median	Mean	98th Percentile	Median	Mean	98th Percentile	Median	Mean	98th Percentile	Median	Mean	98th Percentile	Mean
Baseline	A	9.26 (1.00)	17.20 (1.00)	87.99 (1.00)	13.27 (1.00)	26.76 (1.00)	118.85 (1.00)	11.23 (1.00)	22.20 (1.00)	100.20 (1.00)	0.24 (1.00)	0.71 (1.00)	5.27 (1.00)	1.00
Balor with "Information Has Location"	A	7.68 (0.83)	14.19 (0.82)	76.22 (0.87)	12.45 (0.94)	24.46 (0.91)	102.85 (0.87)	9.19 (0.82)	17.84 (0.80)	85.25 (0.85)	0.20 (0.84)	0.59 (0.84)	4.61 (0.88)	0.87
	B	11.10 (1.20)	19.86 (1.15)	100.79 (1.15)	16.09 (1.21)	31.30 (1.17)	159.22 (1.34)	12.52 (1.11)	25.89 (1.17)	127.00 (1.27)	0.20 (0.81)	0.65 (0.92)	4.90 (0.93)	1.15
	C	8.26 (0.89)	14.13 (0.82)	71.19 (0.81)	11.89 (0.90)	24.61 (0.92)	128.85 (1.08)	8.77 (0.78)	18.16 (0.82)	87.76 (0.88)	0.23 (0.94)	0.64 (0.90)	4.71 (0.89)	0.88
	D	5.14 (0.56)	10.35 (0.60)	55.88 (0.64)	7.83 (0.59)	17.96 (0.67)	100.00 (0.84)	7.69 (0.68)	15.91 (0.72)	71.30 (0.71)	0.18 (0.74)	0.54 (0.77)	4.40 (0.84)	0.68
Balor with "Locations Have Information"	A	8.71 (0.94)	16.03 (0.93)	86.94 (0.99)	13.07 (0.98)	24.67 (0.92)	117.00 (0.98)	11.64 (1.04)	22.66 (1.02)	121.24 (1.21)	0.25 (1.01)	0.77 (1.08)	6.50 (1.23)	1.01
	B	7.84 (0.85)	14.55 (0.85)	78.80 (0.90)	12.06 (0.91)	24.49 (0.92)	108.28 (0.91)	10.64 (0.95)	19.50 (0.88)	89.29 (0.89)	0.16 (0.67)	0.58 (0.82)	4.25 (0.81)	0.89
	C	5.07 (0.55)	9.83 (0.57)	53.06 (0.60)	7.82 (0.59)	16.94 (0.63)	98.45 (0.83)	7.72 (0.69)	16.50 (0.74)	90.62 (0.90)	0.21 (0.86)	0.58 (0.82)	4.01 (0.76)	0.70
	D	**4.24 (0.46)**	**8.45 (0.49)**	**45.65 (0.52)**	**6.55 (0.49)**	**14.95 (0.56)**	**97.43 (0.82)**	**6.44 (0.57)**	**13.86 (0.62)**	**64.76 (0.65)**	**0.15 (0.62)**	**0.49 (0.69)**	**3.93 (0.75)**	**0.59**

Table 2: Timing Percentage Estimation Error Statistics (Value Relative to Baseline in Brackets). Timing estimation error shows the same reduction as resources (Table 1).

Graph Repr.	Arch.	Timing Percent Estimation Error Statistics						
		Latency			Clock Period			Relative Error (All)
		Median	Mean	98th Percentile	Median	Mean	98th Percentile	Mean
Baseline	A	3.12 (1.00)	8.24 (1.00)	57.08 (1.00)	0.78 (1.00)	1.42 (1.00)	6.82 (1.00)	1.00
Balor with "Information Has Location"	A	2.80 (0.90)	8.01 (0.97)	59.76 (1.05)	0.66 (0.84)	1.18 (0.83)	5.80 (0.85)	0.91
	B	3.93 (1.26)	10.66 (1.29)	76.33 (1.34)	0.89 (1.14)	1.54 (1.08)	7.49 (1.10)	1.20
	C	2.56 (0.82)	7.36 (0.89)	47.01 (0.82)	0.62 (0.80)	1.14 (0.80)	5.65 (0.83)	0.83
	D	1.94 (0.62)	6.37 (0.77)	37.37 (0.65)	0.45 (0.58)	0.88 (0.61)	4.56 (0.67)	0.65
Balor with "Locations Have Information"	A	3.13 (1.00)	9.35 (1.14)	54.50 (0.95)	0.75 (0.96)	1.34 (0.94)	6.50 (0.95)	0.99
	B	3.60 (1.15)	9.82 (1.19)	64.59 (1.13)	0.75 (0.96)	1.36 (0.95)	6.71 (0.98)	1.06
	C	1.98 (0.63)	5.99 (0.73)	35.76 (0.63)	0.48 (0.62)	0.94 (0.66)	5.03 (0.74)	0.67
	D	**1.83 (0.59)**	**5.79 (0.70)**	**32.70 (0.57)**	**0.35 (0.45)**	**0.80 (0.56)**	**4.54 (0.67)**	**0.59**

Table 3: Kernel Node Counts for Different Graph Representations (Value Relative to Baseline in Brackets): Balor results in much smaller graphs with no loss of information.

Kernel	Node Count For Different Graph Representations		
	Smallest Node Count	Mean Node Count	Largest Node Count
Baseline	70 (1.00)	440 (1.00)	2,828 (1.00)
Baseline w/o Datatype Nodes	32 (0.46)	189.7 (0.43)	1,202 (0.43)
Balor	25 (0.36)	149.9 (0.34)	926 (0.33)

irrelevant software-specific nodes does also reduce the graph size, without any loss of HLS information. This points to the effectiveness of Balor's graph construction strategy.

8.3 Computational Cost

Finally, Table 4 has statistics on cost metrics. The number of weights shows the size of the network, but since computational cost is also dependent on graph size, we also show the fewest, mean, and most multiplications required for inference. This table corresponds to the last 4 rows in Tables 1 and 2, using "Locations Have Information". Our smallest architecture requires a relative number of multiplications of only 0.11, giving a relative inference time of 0.65. With the expansions of this architecture for increased hierarchical processing, the relative multiplications are increased to 0.18, with a corresponding relative inference time of 0.73. We therefore Pareto-dominate GNN-DSE, reducing error and computational cost.

Table 4: Computational Cost Comparison for "Locations Have Information" (Value Relative to Baseline in Brackets): Balor's smaller graphs and local architectures result in substantial cost savings.

Graph Repr. / Arch.	Computational Cost For Different Graph Representations and Architectures					
	Weights	Fewest Multiplications	Mean Multiplications	Most Multiplications	Inference Time (ms)	Epoch Training Time (s)
Baseline A	121,863 (1.00)	442,138,376 (1.00)	2,779,141,256 (1.00)	17,862,284,168 (1.00)	1.43 (1.00)	56.5 (1.00)
Balor A	132,103 (1.00)	157,908,296 (0.36)	946,930,998 (0.34)	5,848,826,120 (0.33)	1.08 (0.75)	41.4 (0.73)
Balor B	54,279 (0.45)	52,436,296 (0.12)	314,436,508 (0.11)	1,942,143,240 (0.11)	0.92 (0.64)	32.0 (0.57)
Balor C	79,747 (0.65)	78,045,846 (0.18)	468,011,857 (0.17)	2,890,720,972 (0.16)	0.96 (0.67)	33.8 (0.60)
Balor D	109,188 (0.90)	84,361,878 (0.19)	508,813,42 (0.18)	3,203,364,556 (0.18)	1.05 (0.73)	40.13 (0.71)

Cost Comparison to Gao et al. [8]: The computational cost of their approach is too challenging to quantify concisely: each kernel has both a different size and a different distribution of parallelization factors, leading to strongly varying costs for each inference. However, even if replication was applied only for their sub-models, the computational cost of some data points can reach up to 100× the baseline cost. Additionally, the cost of generating their auxiliary datasets cannot be overlooked: dataset generation cost is the largest barrier to GNN implementation.

9 CONCLUSION

This work presents Balor, an HLS source code evaluator, which increases estimation accuracy while strongly reducing the computational cost. The key insights that enable this are: (1) The shift away from traditional graph representations, containing software-specific details irrelevant to HLS, and towards HLS-tailored graph representations. (2) Our philosophy "Locations Have Information" enables efficient processing of compiler directive information, without the over-squashing or redundant processing issues of other approaches. (3) Local, deep, hierarchical GNNs strongly mitigate against the inherent weakness of GNNs, while being cheaper than previous architectures. Our final reduction in resource estimation error, timing estimation error and computational cost is 41%, 41% and 82% respectively, and come from foundational insights which transfer to all GNN approaches. With the proven success of these approaches on large databases of regular kernels, further work is now needed on annotation for transfer learning: both to new kernels, and to new operating conditions.

REFERENCES

[1] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2020. Measuring and Relieving the Over-Smoothing Problem for Graph Neural Networks from the Topological View. In *Proceedings of The 34th AAAI Conference on Artificial Intelligence*, Vol. 34. New York, NY, 3438–3445.

[2] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. 2016. Source-to-Source Optimization for HLS. *FPGAs for Software Programmers* (2016), 137–163.

[3] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning*, Vol. 139. PMLR, Virtual, 2244–2253.

[4] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline FY Young, and Zhiru Zhang. 2018. Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines*. Boulder, CO, 129–132.

[5] Lorenzo Ferretti, Andrea Cini, Georgios Zacharopoulos, Cesare Alippi, and Laura Pozzi. 2022. Graph Neural Networks for High-Level Synthesis Design Space Exploration. *ACM Transactions on Design Automation of Electronic Systems* 28, 2 (2022), 1–20.

[6] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura Pozzi. 2021. DB4HLS: A Database of High-Level Synthesis Design Space Explorations. *IEEE Embedded Systems Letters* 13, 4 (2021), 194–197.

[7] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca P Carloni, and Laura Pozzi. 2020. Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3736–3747.

[8] Mingzhe Gao, Jieru Zhao, Zhe Lin, and Minyi Guo. 2023. Hierarchical Source-to-Post-Route QoR Prediction in High-Level Synthesis with GNNs. In *2024 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, Valencia, 1–6.

[9] Pingakshya Goswami, Benjamin Carrion Schaefer, and Dinesh Bhatia. 2023. Machine Learning Based Fast and Accurate High Level Synthesis Design Space Exploration: From Graph to Synthesis. *Integration* 88 (2023), 116–124.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. Las Vegas, NV, 770–778.

[11] Huizhen Kuang, Xianfeng Cao, Jingyuan Li, and Lingli Wang. 2023. HGBO-DSE: Hierarchical GNN and Bayesian Optimization based HLS Design Space Exploration. In *2023 International Conference on Field Programmable Technology*. IEEE, Yokohama, 106–114.

[12] Jihye Kwon and Luca P Carloni. 2020. Transfer learning for Design-Space Exploration with High-Level Synthesis. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. Virtual, 163–168.

[13] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *2004 International Symposium on Code Generation and Optimization*. IEEE, Palo Alto, CA, 75–86.

[14] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization*. Virtual, 2–14.

[15] Zhe Lin, Zike Yuan, Jieru Zhao, Wei Zhang, Hui Wang, and Yonghong Tian. 2022. Powergear: Early-Stage Power Estimation in FPGA HLS via Heterogeneous Edge-Centric GNNs. In *2022 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, Virtual, 1341–1346.

[16] Zhe Lin, Jieru Zhao, Sharad Sinha, and Wei Zhang. 2020. HL-Pow: A Learning-Based Power Modeling Framework for High-Level Synthesis. In *2020 25th Asia and South Pacific Design Automation Conference*. IEEE, Beijing, 574–580.

[17] Hosein Mohammadi Makrani, Hossein Sayadi, Tinoosh Mohsenin, Setareh Rafatirad, Avesta Sasan, and Houman Homayoun. 2019. XPPE: Cross-Platform Performance Estimation of Hardware Accelerators using Machine Learning. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, Tokyo, 727–732.

[18] Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. 2018. HLSPredict: Cross Platform Performance Prediction for FPGA High-Level Synthesis. In *2018 IEEE/ACM International Conference on Computer-Aided Design*. San Diego, 1–8.

[19] Dan Quinlan. 2000. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters* 10 (2000), 215–226.

[20] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *2014 IEEE International Symposium on Workload Characterization*. Raleigh, NC, 110–119.

[21] Benjamin Carrion Schafer and Zi Wang. 2019. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2628–2639.

[22] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. 2020. Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification. *arXiv preprint arXiv:2009.03509* (2020).

[23] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated Accelerator Optimization Aided by Graph Neural Networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. San Francisco, 55–60.

[24] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2023. Robust GNN-based Representation Learning for HLS. In *2023 IEEE/ACM International Conference on Computer Aided Design*. San Francsico, 1–9.

[25] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems* 27, 4 (2022), 1–27.

[26] Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M Bronstein. 2021. Understanding Over-squashing and Bottlenecks on Graphs via Curvature. In *Proceedings of the 9th International Conference on Learning Representations*. Virtual.

[27] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. 2020. Accurate Operation Delay Prediction for FPGA HLS using Graph Neural Networks. In *2020 IEEE/ACM International Conference on Computer-Aided Design*. San Diego, 1–9.

[28] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, Austin, TX, 1–6.

[29] Zi Wang and Benjamin Carrion Schafer. 2022. Learning From the Past: Efficient High-Level Synthesis Design Space Exploration for FPGAs. *ACM Transactions on Design Automation of Electronic Systems* 27, 4 (2022), 1–23.

[30] Nan Wu, Yuan Xie, and Cong Hao. 2022. IRONMAN-PRO: Multiobjective Design Space Exploration in HLS via Reinforcement Learning and Graph Neural Network-Based Modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 3 (2022), 900–913.

[31] Nan Wu, Hang Yang, Yuan Xie, Pan Li, and Cong Hao. 2022. High-Level Synthesis Performance Prediction using GNNs: Benchmarking, Modeling, and Advancing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. San Francisco, 49–54.

[32] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation Learning on Graphs with Jumping Knowledge Networks. In *Proceedings of the 35th International Conference on Machine Learning*. PMLR, Stockholm, 5453–5462.

[33] Jiaxuan You, Zhitao Ying, and Jure Leskovec. 2020. Design Space for Graph Neural Networks. *Advances in Neural Information Processing Systems* 33 (2020), 17009–17021.

[34] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A Comprehensive Model-Based Analysis Framework for High-Level Synthesis of Real Applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design*. Irvine,CA, 430–437.

[35] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. 2016. Lin-Analyzer: A High-Level Performance Analysis Tool for FPGA-Based Accelerators. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, Austin, TX, 1–6.

[36] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. 2017. Design Space Exploration of FPGA-Based Accelerators with Multi-Level Parallelism. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, Lausanne, 1141–1146.

[37] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph Neural Networks: A Review of Methods and Applications. *AI Open* 1 (2020), 57–81.