HLPerf: Demystifying the Performance of HLS-based Graph Neural Networks with Dataflow Architectures

CHENFENG ZHAO, Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, St. Louis, USA
CLAYTON J. FABER, Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, St. Louis, USA
ROGER D. CHAMBERLAIN, Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, St. Louis, USA
XUAN ZHANG, Dept. of Electrical and Computer Engineering, Northeastern University, Boston, USA

The development of FPGA-based applications using HLS is fraught with performance pitfalls and large design space exploration times. These issues are exacerbated when the application is complicated and its performance is dependent on the input data set, as is often the case with graph neural network approaches to machine learning. Here, we introduce HLPerf, an open-source, simulation-based performance evaluation framework for dataflow architectures that both supports early exploration of the design space and shortens the performance evaluation cycle. We apply the methodology to GNNHLS, an HLS-based graph neural network benchmark containing 6 commonly used graph neural network models and 4 datasets with distinct topologies and scales. The results show that HLPerf achieves over 10 000× average simulation acceleration relative to RTL simulation and over 400× acceleration relative to state-of-the-art cycle-accurate tools at the cost of 7% mean error rate relative to actual FPGA implementation performance. This acceleration positions HLPerf as a viable component in the design cycle.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; • **Computer systems organization →** *Reconfigurable computing*; • **General and reference →** *Performance.*

Additional Key Words and Phrases: Performance evaluation, discrete-event simulation, graph neural networks

1 INTRODUCTION

Graphs are widely adopted to model the relational-structured data in social networks, bioinformatics, etc. Machine learning (ML) on graphs has experienced a surge of popularity in the past decade, since traditional ML models, which are designed to process Euclidean data with regular structure, are ineffective at performing prediction tasks on graphs. Due to their superior representation learning ability, Graph Neural Networks (GNNs) [19, 24, 37, 38, 42] have achieved impressive utility on graph learning tasks, such as node classification [39], graph classification [44], link prediction [17], recommendation [12], etc. Given the advent of IoT and edge computing platforms, the execution of GNNs on edge devices (especially those with constraints on energy usage) has similarly received recent attention [21, 47, 48].

With the diminishing of Moore's law [33, 36] and the end of Dennard scaling [4, 14], FPGAs have become an attractive approach to application-specific acceleration, due to their massive fined-grained parallelism to exploit

Authors' addresses: Chenfeng Zhao, Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, St. Louis, Missouri, USA, chenfeng.zhao@wustl.edu; Clayton J. Faber, Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, St. Louis, Missouri, USA, cfaber@wustl.edu; Roger D. Chamberlain, Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, St. Louis, Missouri, USA, roger@wustl.edu; Xuan Zhang, Dept. of Electrical and Computer Engineering, Northeastern University, Boston, USA, xuan.zhang@northeastern.edu.

ACM 1936-7406/2024/4-ART
https://doi.org/10.1145/3655627

the inherent parallelism of algorithms, better performance per watt than general-purpose computing platforms (of particular importance for edge computing), flexible reconfigurability for the adoption of various applications, and data path customization capability.

Due to the conflict between the non-trivial development efforts with conventional HDL design methodology and rapid emergence of new algorithms to be accelerated, High-Level Synthesis (HLS) tools, such as AMD Xilinx Vitis and Intel OpenCL HLS, have been developed to automatically translate algorithms expressed in commonly-used software languages such as C/C++ to hardware implementations, substantially shortening the development time. Due to their expression at a higher level of abstraction, modern HLS tools and techniques provide greater opportunities to ease design space exploration and optimization which can be impractical in HDL designs.

Although HLS bridges the gap between software and hardware development, optimizing HLS codes is substantially distinct from conventional software programming. In fact, due to the FPGAs' inherent attributes, such as lack of built-in cache mechanisms, low clock frequency (relative to traditional processor cores), and fine-grained configurability, the performance difference between a well-optimized version and naive version of the same kernel can be two or three orders of magnitude [6, 15, 35]. Therefore, to achieve the best performance, HLS developers need to explore a large optimization space for HLS designs with various optimization pragmas, coding paradigms, etc.

As the complexity of kernels increases, optimizing (or auto-optimizing) such kernels is difficult via conventional HLS workflows for several reasons:

(1) Since pure C emulation is only designed for functionality verification, current HLS developers have to use RTL simulation to understand performance by manually mapping the results of individual signals in the generated waveform back to the HLS code. However, since all the signal names are auto-generated, they are not easily comprehensible by users. Besides, RTL simulation usually takes a very long time, making the tuning effort arduous. Even worse, it is exacerbated by the fact that tuning with a small example data set is less meaningful for GNN kernels in terms of performance estimation because of the inherent irregularity of graph datasets and algorithms. In other words, distinct graph topologies can significantly impact the final performance achieved. Therefore, when it comes to large-scale graphs, RTL simulation is impractical to be used to optimize GNN kernels with these graphs.
(2) The notion of dataflow architectures which exploit task-level parallelism, where multiple functions are connected by FIFOs and executed concurrently instead of sequentially, further mystifies the optimization process because it induces a wider set of design space challenges including: task partitioning, FIFO depth tuning, and bottleneck identification, which are distinct from conventional computation platforms.

The critical missing piece in the optimization task is the availability of fast, high-quality understanding of the performance implications of the design choices that are made. Our focus in this work is to address this missing element, providing the the designer (whether it be a human or an automatic design space exploration tool [32]) with performance predictions both quickly and with sufficient accuracy that they can be used effectively.

Traditional approaches to performance assessment either involve static assessment (i.e., compile-time analysis) or cycle-accurate simulation. In this work, we propose a different method, effectively between the approaches of static estimation and cycle-accurate simulation, to investigate the impact of irregularity of data and algorithms on performance. Due to the existence of other HLS tools for functional verification (e.g., software emulation in Vitis), our method decouples functional verification from performance estimation, so that the runtime of the estimation process is independent of the computational details of the FPGA algorithms.

Here, we introduce HLPerf, a performance evaluation methodology that supports the performance variations inherent in data-dependent algorithms (it is simulation based), but relaxes the notion of cycle accuracy and replaces it with "approximate" cycle accuracy. The result is a simulation-based performance estimate that is two orders-of-magnitude faster than state-of-the-art simulations that perform cycle-accurate functional verification.

The paper makes the following contributions:

- We propose HLPerf, an open-source[1], approximately-cycle-accurate performance evaluation method, to estimate the dynamic performance of GNN HLS kernels with a dataflow architecture. It gives useful performance guidance with dramatically better simulation speed than both RTL simulation and more recently developed cycle-accurate simulators.
- We describe an approach to automatically transform the HLS C-based source code describing several GNN operations into simulation components.
- We propose a set of high-level quantitative expressions in HLPerf to model the performance impact of various optimization techniques. Decoupling performance estimation from functional verification, HLPerf is faster and can be used to guide dataflow pipeline designs even prior to the authoring of the constituent HLS kernels.
- We provide a comprehensive evaluation of HLPerf using 6 different GNN models on 4 graph datasets plus several additional general-purpose applications, assessing both accuracy of the performance predictions and performance of the simulator itself. Our evaluation results show that the error rate of HLPerf is 7% on average and it is $13,500\times$ faster than RTL simulation and over $400\times$ faster than a state-of-the-art cycle-accurate simulator.

The result is a capability that enables performance tuning of deeply pipelined dataflow architectures with a dramatically reduced design/evaluation cycle. It does not replace cycle-accurate simulation, as there is some accuracy sacrificed when relaxing the requirement for cycle accuracy. Rather, it is intended for use early in the design cycle, when variations in performance due to simple design modifications can impact performance by large factors.

The manuscript is organized as follows. Section 2 gives background information on HLS workflows and graph neural network applications as well as a description of related work. Section 3 describes the methodology, including the design workflow with HLPerf, the event-driven simulation, and the modeling of HLS kernels. Section 4 gives two options for an application developer's experience using HLPerf. Section 5 articulates the evaluation methods, and Section 6 gives quantitative evaluation results. Section 7 provides conclusions and directions for future work.

2 BACKGROUND AND RELATED WORK

2.1 High-Level Synthesis Workflow

We start with an introduction to the basics of an HLS workflow, which is adopted by most of the mainstream HLS tools, such as Xilinx Vitis and Intel OpenCL HLS. This is illustrated in Figure 1. In this paper, we use Xilinx Vitis [2] as the running example, but the underlying principles are the same for many other HLS tools. It contains the following steps.

(1) **HLS C Code**: HLS users are first required to build HLS kernels based on target algorithms using high-level languages (e.g., HLS C) annotated with pragmas that can have substantial impact on the final performance. Among all the various optimization techniques, `#pragma HLS dataflow` is frequently first used to build a dataflow architecture in which all the functions in the dataflow scope are connected by FIFOs to form a pipeline-style architecture and scheduled to be performed concurrently. In our GNN applications, each function contains one or more loops. Then they use other pragmas such as `Pipeline`, `Loop Unroll`, `Loop Merge`, `Burst Memory Access`, `Memory Port Widening`, etc., to optimize each function or loop in terms of throughput, iteration, and

[1]https://github.com/ChenfengZhao/HLPerf

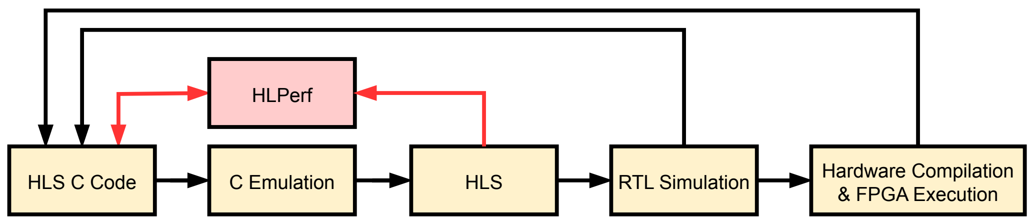

Fig. 1. Conventional HLS workflow (in black) and our contribution (in red).

memory accesses. In addition, distinct coding paradigms with the same functionality can also influence the final performance of the HLS kernel.

(2) **C Emulation**: The HLS kernel file, along with the host file, configuration files, and input dataset are compiled and executed under software emulation mode. Note that C emulation only focuses on functionality verification of the HLS kernel. Thus, it doesn't involve performance estimation.

(3) **High-Level Synthesis**: In this step, HLS tools convert the HLS Kernel from the high-level C-like description to an RTL-level hardware description language (e.g., Verilog, VHDL). To achieve this goal, the HLS tool initially preprocesses the source code of the HLS kernel and conducts transformations based on user-defined pragmas. Subsequently, operations are scheduled in accordance with the corresponding dependency and optimization techniques, and then bound to hardware resources. After this process, coarse-grained control flows are typically implemented as finite-state machines, while fine-grained instruction collections are realized as variants of pipelines. Following these steps, the HLS tool provides estimated intermediate results of the scheduling such as the latency and throughput of pipelines. Finally, RTL code is generated. Note that due to the inherent dynamic characteristics of GNN applications and their significant dependence on input graph datasets, the HLS tool is incapable of providing conclusive outcomes through static performance estimation.

(4) **RTL Simulation**: To obtain estimated cycle-accurate performance results of HLS kernels, the generated RTL code, along with host files, configuration files, and graph datasets, are compiled and executed under hardware emulation mode for RTL simulation. Due to the inherent irregularity of GNNs and graph datasets, the simulation needs to include at least a representative subset of the input graphs to be processed. The results are stored in a waveform file containing cycle-accurate transitions of all the signals in the RTL code. To debug or improve the performance of the HLS kernels, users are required to trace these RTL signals (often with incomprehensible names) back to HLS code, modify the HLS code, and repeat the procedure until the performance goals are satisfied. However, the whole procedure takes a significant amount of time due to the incorporation of extensive architectural details and the desire to be cycle accurate. Consequently, RTL simulation is usually impractical to be used for estimating the performance of complicated GNN HLS kernels with large-scale graph datasets.

(5) **Hardware Compilation & FPGA Execution**: To get the physical layout, the generated RTL code is converted into a gate-level representation (i.e., netlist) for a specific architecture and then mapped to specific locations of the target device via the place & route process. A series of back-end strategies on physical implementations are performed to get a trade-off among design's performance, area, and power. The finalized circuit description is encapsulated into a bitstream file, which is then executed on an FPGA, enabling the measurement of the actual execution times. This step is quite time consuming (e.g., 4.5-12 hours for compilation of GNN HLS kernels), and even though direct execution is clearly the gold standard for performance understanding, it is the length of these build times that makes the inclusion of direct execution in the iterative design cycle unattractive.

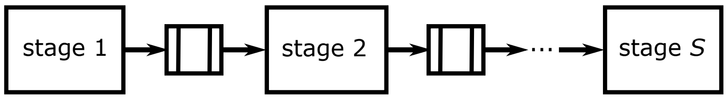

Fig. 2. GNN dataflow pipeline.

2.2 Graph Neural Networks

Here, we use GNNHLS [46], a HLS benchmark suite containing 6 well-tuned GNN kernels with multiple optimization techniques applied. These kernels with various structures can be classified into two categories: (1) isotropic GNNs in which every neighbor contributes equally to the update of the target vertex and (2) anisotropic GNNs in which edges and neighbors contribute unequally to the update due to the adoption of operations such as attention and gating mechanisms. Note that although these kernels have distinct HLS implementations, all of them adopt a dataflow architecture (illustrated in Figure 2) in which 7 to 27 functions are connected with FIFOs, making them suitable targets for HLPerf. Table 1 provides some information about each of the GNN models, further details (including the implementations) can be found in [46].

Table 1. GNN models.

GNN Model	Reference	Stages (S)	Category
Graph Convolutional Network (GCN)	[24]	7	isotropic
GraphSage (GS)	[20]	14	isotropic
Graph Isomorphism Network (GIN)	[42]	11	isotropic
Graph Attention Network (GAT)	[37]	27	anisotropic
Mixture Model Networks (MoNet)	[29]	11	anisotropic
Gated Graph ConvNet (GatedGCN)	[5]	19	anisotropic

The input data of the GNN kernels consists of 2 elements: the input graphs themselves (stored in CSR, compressed sparse row, format) and the corresponding features (stored in separate matrices).

2.3 Related Work

To address the performance evaluation challenge, several works have been proposed, which can be classified into 2 main classes: static estimation [8, 11, 13, 28] and cycle-accurate simulation [1, 10, 31]. Static estimation is performed at compile time so has difficulty with performance that is input dependent, and cycle-accurate simulation substantially accelerates the speed of RTL-level simulation while maintaining quality of the performance predictions. We will report research happening in each area in turn.

Legup [8] estimates the speedup of the accelerated function in the HLS kernel in the straightforward way of multiplying the number of iterations recorded by software profiling tools and the single-iteration execution time extracted from RTL simulation. This method does depend on RTL simulation. Additionally, it assumes the FPGA algorithms are performed sequentially, so it doesn't consider a number of HLS optimization techniques, such as pipelined execution. HLScope+ [11] proposes a method to perform pipelined loop analysis by inserting hooks to HLS C code and extracting HLS abstraction information. However, it fails to capture the irregularity of data sets (e.g., graph topologies) and respond to the dynamic properties of irregular algorithms and HLS kernels. Pyramid [28] uses machine learning techniques to estimate both FPGA area requirements and achievable clock rates. De Fine Licht et al. [13] propose a static expression for pipelined loop analysis with some optimization techniques. However, it is also not sufficient for irregular data and algorithms.

On the other hand, Flash [10] uses scheduling information to build a C cycle-accurate simulation model. Fastsim [1] translates generated RTL code to an equivalent C++ cycle-accurate model. LightningSim [31] proposes a LLVM-IR-trace-based method to reconstruct a cycle-accurate model. In spite of the ability of these cycle-accurate methods to analyze the dynamic behavior of many FPGA algorithms, there are still some drawbacks: (1) These works are designed to provide both functional correctness verification and performance estimation, increasing the workload of the evaluation process. (2) Since these methods are related to the construction and execution of cycle-accurate time models with many low-level details, the simulation speed is limited.

Design space search and custom architectures aimed specifically at GNNs have also received attention recently [18, 21, 26, 27, 43, 45, 47, 48]. HGNAS [47, 48] targets edge devices for execution of GNNs, explicitly considering reduction in memory requirements as well as execution speed. DeepBurning-GL [26] proposes an automated framework to convert specific component of GNN models based on DGL to RTL codes using pre-defined hardware templates. G-CoS [45] works to match GNN structure with the available execution platform(s), and Hao et al. [21] exploit reinforcement learning as part of the design space search. HyGCN [43] is a custom ASIC design aimed at graph neural network inference that is evaluated using the TSMC 12 nm CMOS process. EnGN [27] targets the need to scale up to large graphs by introducing a ring-edge-reduce dataflow to handle graphs with arbitrary dimensions. AWB-GCN [18] is a custom FPGA design that addresses the variability in graph topology by auto-tuning the accelerator during the execution of the GNN application itself. While the authors of all of the above studies evaluate performance on a variety of graphs, none of the design space exploration investigations incorporated simulation into the performance evaluation that specifically guides the search of the design space, and only AWB-GCN has explicit mechanisms for adapting to variations in properties of the input graphs. Our intention is to make simulation sufficiently fast that it can be seriously considered in an automated design space search context.

GNN models are but one example of applications that execute on graphs. Chen et al. [9] introduce ThunderGP, a framework for developing general graph applications for deployment on FPGAs. ThunderGP uses a dataflow architecture for its designs, so a GNN model developed using ThunderGP could likely utilize HLPerf as a companion performance evaluation tool.

3 METHODOLOGY

3.1 Overall Workflow

As depicted in black in Figure 1, the conventional design flow exhibits a deficiency in effective and practical methods for estimating the performance of GNN HLS kernels, which is crucial for rapid iterative tuning. To address this shortcoming, we devise a new workflow, HLPerf, incorporating the new element highlighted in red in Figure 1. The main idea is to circumvent the need for RTL simulation or FPGA execution each design iteration and build a high-level "approximately-cycle-accurate" abstraction of the HLS kernel that supports much higher simulation speed to accelerate the iterative design space exploration of HLS kernels. HLPerf is composed of 3 core elements: (1) a discrete-event simulation system built upon SimPy to emulate the inherent concurrency of the dataflow architecture and capture its dynamic execution behavior; (2) quantitative expressions of pragma-driven patterns to model the performance impact of various optimization techniques, decoupling performance estimation from functional verification; and (3) a front-end source-to-source compilation step to automatically transform the HLS C-based source code of diverse GNN applications into corresponding simulation components. These elements are used to build "approximately-cycle-accurate" models focusing only on the performance estimation of fundamental loops with distinct optimization techniques. Since HLPerf (1) decouples the performance estimation from computational intricacies of the algorithm, (2) involves fewer signals to be simulated, and (3) adopts coarser granularity of runtime simulation rather than cycle-accurate simulation, the performance estimation process and the resulting iterative tuning procedure are substantially accelerated.

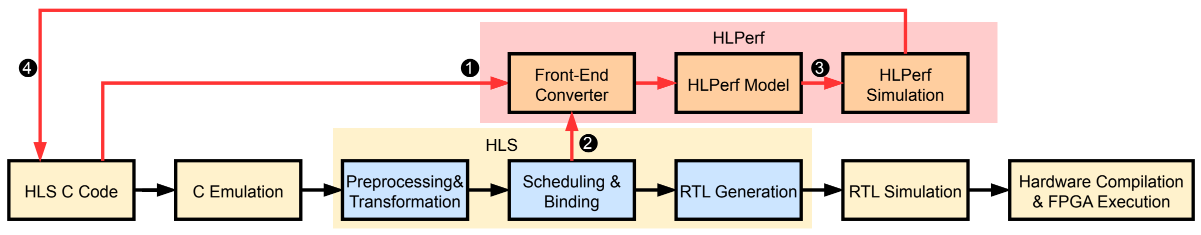

Fig. 3. The overall workflow of HLPerf.

The overall workflow of HLPerf is shown in Figure 3. It consists of 4 steps: model creation, parameter loading, HLPerf simulation, and kernel tuning.

❶ **Model Creation:** In addition to the conventional HLS workflow where HLS C code is checked with functionality via C emulation and converted to RTL code via HLS, in HLPerf we build a front-end converter to analyze the source code of HLS kernels to extract the key information, such as the code structure, the topology of dataflow scopes, high-level expression of loops and FIFOs, and HLS optimization techniques. Then the corresponding approximately-cycle-accurate HLPerf model is automatically generated by the converter.

❷ **Parameter Loading**: After the scheduling and binding procedure of HLS and before the start of RTL generation, the front-end converter also automatically extracts realistic performance parameters (e.g., latency and initiation interval) from the synthesis log file and uses them to complete the generated HLPerf model. Note that this step does not rely on the first step, so they can be executed in parallel.

❸ **HLPerf Simulation**: After performing the simulation of the HLPerf model with graph datasets, a detailed result summary report is generated, offering a holistic view of the dataflow architecture's performance. This report provides the overall estimated performance of the GNN HLS kernel, and can help users to explore the impact of distinct graph topologies on the performance of the dataflow architecture and pinpoint performance bottlenecks.

❹ **Kernel Tuning**: With the help of the HLPerf result report, users can tune the HLS C code for one or more kernels with an alternative strategy, such as code paradigm, FIFO sizes, optimization technique choices, etc. Even better, if the code modification is not related to the parameters achieved from the scheduling and binding procedure, users can directly re-run the workflow of HLPerf without performing the HLS step, which can save additional time.

3.2 Discrete-Event Simulation

In cycle-accurate simulations, such as RTL simulation, activity within the design is modeled at each cycle, whether or not that activity is relevant to the performance estimation of the design (e.g., it might be relevant for functional correctness, but not performance). However, many cycles do not have activity that is relevant to the performance estimation, and can therefore be skipped in a purely performance-oriented simulation.

In contrast to the cycle-based progression described above, HLPerf's Discrete-Event Simulation (DES) adopts an event-driven paradigm. The key idea is to generate discrete sequences of coarse-grained events occurring at specific time intervals. These events represent a change of state in the system and are scheduled with a time-skipping strategy which advances the simulated time forward directly to the subsequent event's occurrence, Therefore, it enables the creation of simplified system models that concentrate on key processes of significant influence, while avoiding the intricacies of low-level details. By substantially diminishing the complexity and resource intensity of the simulation, DES is more efficient and time-effective than time-stepped simulation methodologies.

In this paper, we build the high-level performance models within HLPerf for a variety of GNN kernels using SimPy [34], a Python-based generic library for discrete-event simulation. In essence, SimPy operates as an asynchronous event dispatcher. It generates and schedules events at specified time intervals by storing all the events in a heap-based event list and ordering them by simulation time, priority, and increasing event ID. In SimPy, systems are modeled through the creation of process functions. These functions simulate entities whose behaviors evolve over time. Rather than exploiting multiple threads to replicate the inherent parallelism in processes, SimPy utilizes Python's generator functions in each process function. These generators are characterized by Python's `yield` keyword and act as suspend/resume points. This feature allows the temporary suspension of process functions and subsequent resumption at the point of last suspension. Specifically, when a generator issues a yield command within a process, the process is paused, and a new event is yielded and inserted into the event list with the position in a given order. Subsequently, SimPy's internal functions inspect the scheduled events, extracting and removing the one with the earliest simulation time. The system's simulation time, `SimTime`, is then updated to the time of that event, and the corresponding process is resumed immediately following the last executed `yield` statement. Through this mechanism, the parallelism of process functions can be simulated by alternate execution of effective co-routines.

In addition, process functions are not standalone entities. SimPy provides shared resource classes to model the interaction between processes. These resource classes serve as containers with a user-defined capacity, so that process functions can either write data into the resource instances using the `put()` method or retrieve data from it using the `get()` method. Since both of these methods are Python generator functions with `yield` statements, they can return an event that is triggered when the corresponding action is to be executed. Therefore, when the resource is empty or full, processes are required to wait for the state change of the resource. This system ensures a controlled and orderly interaction among processes for shared resources. All aspects of the simulation, including process functions, resources, simulation time, and event scheduling, are managed by SimPy's `Environment` class. Once the `Environment.run()` method is executed, the simulation is activated.

In order to simulate dataflow architectures in HLPerf we use process functions to represent individual stages. Given that SimPy offers three types of shared resources, we choose the `Store()` class to model FIFOs due to its capability to store Python objects. Figure 4 illustrates an example of employing SimPy for dataflow architectural simulation. For the sake of simplicity, this example considers only Stage1, Stage2, and a connecting FIFO1 with a capacity of two. These components are correspondingly modeled as `process1`, `process2`, and `store1`. In this case, assume `process1` has data to dispatch to `store1` at times t_0, t_1, t_2, and t_7, while `process2` retrieves data at t_3, t_4, t_5, and t_6, with $t_i < t_{i+1}$. Upon initiation of the simulation, a process from the "runnable" list is selected for execution. Assume `process1` is chosen and successfully sends data at t_0 and subsequently again at t_1, the FIFO reaches its full capacity, preventing the transmission at t_2 and resulting in the suspension of `process1`. Subsequently, `process2` is activated, retrieves data at t_3, and is then suspended. As FIFO1 is no longer full, `process1` resumes and succeeds in transmitting data at t_3, and is suspended again. Next, `process2` then resumes, successfully retrieves data at t_4 and t_5, and is suspended. At t_6, `process2`'s attempt to retrieve data fails due to FIFO1 being empty, prompting `process1` to resume and successfully send data at t_7, followed by `process2` retrieving data at the same time. Note that while ping-pong buffers are also used in some cases, they are not utilized in the benchmarks discussed in this paper. As such, they are not elaborated here. However, a ping-pong buffer can be implemented using two instances of SimPy's `Store` and simple switching logic. More details on constructing performance models in HLPerf based on SimPy are discussed in Section 3.4.

3.3 HLPerf Model Converter

The front-end converter takes the source code of the target HLS kernel and the intermediate results of the HLS scheduling and binding procedure as inputs, and automatically generates HLPerf models as outputs. Figure 5

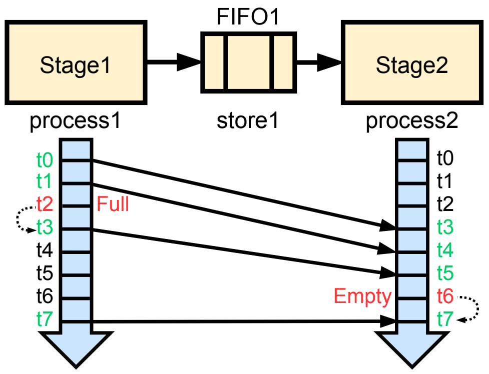

Fig. 4. A example of SimPy for a dataflow architecture, where process1 sends data to store1 at t_0-t_2 and t_7; process2 receives data from store1 at t_3-t_6; and $t_0 < t_1 < t_2 < t_3 < t_4 < t_5 < t_6 < t_7$.

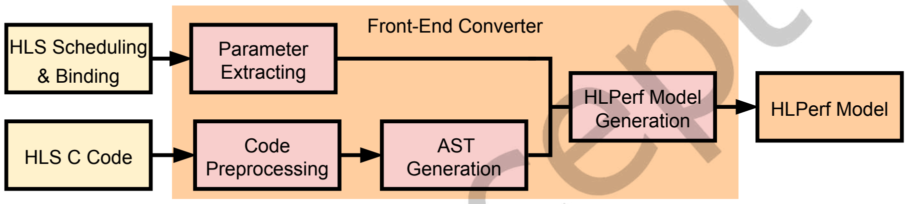

Fig. 5. The workflow of the Front-End Converter in HLPerf.

depicts the workflow of the front-end converter. Initially, the converter preprocesses the HLS C source code of the target GNN kernel by parsering the code using pycparser[3], a generic python library for C language parsing. Since the source code incorporates 2 specific C++ data types: hls::vector and hls::stream, which are not inherently recognizable by pycparser, these data types are substituted with a placeholder C data type (e.g., int) during the preprocessing procedure. We note that this substitution does not influence the subsequent HLPerf model simulation. The irrelevance of vector data to the simulation stems from our approach of decoupling the simulation from functional correctness, one of the key advantages of HLPerf. Additionally, the stream type's relevance is mitigated as the front-end converter can identify FIFOs through the stream pragma (i.e., #pragma HLS stream), ensuring a seamless simulation process. Meanwhile, the converter also parses the the synthesis log file (e.g., vitis_hls.log in Vitis) to extract intermediate results of HLS scheduling and binding. For example, in vitis_hls.log it will report the II and latency L in the log information Pipelining result : Target II = 1, Final II = 1, Depth = 75 where final II is 1 and latency is 75. Besides, the converter also extracts the pre-defined memory latency (e.g., set by parameter m_axi_latency in Vitis).

After parsing and preprocessing, the source code of the target HLS kernel is converted into an abstract syntex tree (AST) representation by pycparser. An AST is a tree representation of the syntactic structure of the source code, in which each node represents the information of a code part such as function, loop, statement, variable, etc, and each edge represents the relationships among different code parts. The root of the AST is the top-level kernel function. The front-end converter automatically generates the HLPerf model of the target GNN HLS kernel following AST traversal.

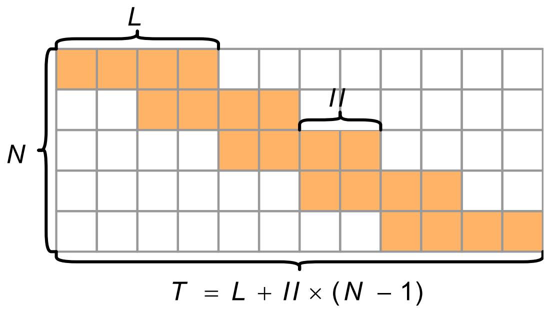

Fig. 6. The diagram of the latency (L), initiation interval (II), and interation number (N) of a pipelined loop.

Algorithm 1 shows the pseudocode for the HLPerf model generation. This method accepts the AST representation of the target kernel and intermediate parameters as inputs. The process begins with the instantiation of a new model for the target High-Level Synthesis (HLS) kernel. Subsequently, the converter identifies and logs all memory-related arguments from the top-level kernel design by analyzing the `#pragma HLS INTERFACE` directive. In our case, these memory ports adhere to the `m_axi` protocol and are essential for the subsequent integration of memory latency into the HLPerf model. Notably, among these arguments, `node_src`, the array representing source vertex indices, is uniquely tied to real input data as it reflects the irregular topology of the input graph. The converter then proceeds to locate all dataflow functions, starting from the root. These functions are identified by the `#pragma HLS dataflow` directive. Typically, each HLS kernel in our GNN applications corresponds to a single dataflow function. For each identified dataflow function, a new environment within the model is established. Given that dataflow functions in our applications consist of two node types, variables with a stream pragma denoting FIFOs and functions representing stages in the dataflow architecture, they are processed distinctly. A new FIFO instance for each stream variable is created within a unified model environment. Moreover, as each function generally encompasses several loops, a new process within the model is created for each function, with a detailed examination of its arguments to discern FIFOs and memory-related arguments. Subsequently, a recursive Depth-First Search (DFS) algorithm traverses all `for` loops within the function. For each loop, the boundary is first analyzed, with distinct processing based on the boundary's type (e.g., constant or variable). The pragmas within the loop are then checked, and the loop body is replaced with pragma-driven patterns. Finally, the intermediate parameters from HLS are incorporated into the loop. Upon the comprehensive processing of all dataflow functions, the HLPerf model is automatically generated.

3.4 Pragma-Driven Pattern Modeling

In GNN implementations, the HLS kernels are invariably loops, which can have their performance analyzed analytically without any need for full cycle accuracy. In HLPerf, we build pragma-driven patterns based on analytical expressions [13] to model the performance impact of various optimizations on loops in each stage of the dataflow architecture. A pragma driven pattern consists of 3 parameters: **Latency** (L) represents the number of cycles for an input element to propagate from the entry to exit of a pipeline. **Initiation interval** (II) denotes the number of cycles required for successive input elements to be ingested. **Iteration number** (N) of the loop represents the number of data elements to be processed by the pipeline. Figure 6 shows the relationship of these 3 parameters in a pipeline loop where L, II, and N are 4, 2, and 5 respectively. The overall execution time is $T = L + II \times (N - 1)$. In essence, the performance benefit of the prevalent optimization techniques will be attributed to their impact on these parameters. Thus, applying distinct optimization techniques to the same loop

Algorithm 1: HLPerf Model Generation Algorithm

```
   input  : AST of kernel's source code
   input  : Intermediate parameters of HLS
   output : Generated HLPerf model
   // Create a new model instance
1  model ← createModel();
2  $S_m$ ← Find all the top-level memory arguments via #pragma HLS INTERFACE;
3  $S_d$ ← Find all the dataflow functions starting from root;
4  foreach $df \in S_d$ do
       // create a new environment
5      env ← model.createEnv(df);
6      foreach $node \in df.childList$ do
7          if node.pragma is stream then
               // create a new fifo instance
8              model.addFIFO(env, node);
9          else if node.type is function then
               // create a new process instance
10             func ← model.addProcess(env, node, $S_m$);
11             while Recursively DFS traverse all the for loops ∈ func do
12                 Analyze the boundary of the loop;
13                 Replace loop body with pragma-driven patterns;
14                 Load intermediate parameters of HLS;
   // Save the generated HLPerf model
15 model.save();
```

will result in different parameterizations of this quantitative expression. To quantify the parameterization, we use SimPy's `Environment.timeout()` method which is a `yield` statement to schedule the execution of the HLPerf model by suspending a process for a given time. Note that for loops without any pragmas, we use the number of unoptimized operations in the loop body to schedule the process. Here, we use code snippets of the GCN kernel as examples to illustrate how these optimization techniques affect the kernel performance and the mapping between HLS kernel and HLPerf models.

Pipeline is one of the essential optimization techniques in the HLS community for effective hardware acceleration. It enables instruction-level concurrent execution to to reduce *II* and improve the overall throughput. Vitis provides a pragma, `#pragma HLS pipeline`, to convert a regular sequential loop to a hardware pipeline whose execution is illustrated in Figure 6. To figure out how to map HLS C with pipelined loops to the HLPerf model, we use the aggregation module in the GCN kernel as an example, Listing 1 and Listing 2 show the corresponding code snippet in HLS C and HLPerf, respectively. There are 3 loops in Listing 1. The first, `loop1`, is a non-perfect loop with a variable boundary representing the in-degree of each node. After reading the neighbors' feature vectors from the input FIFO (i.e., `fifoIn`) which is implemented as a `stream` class in Vitis (line 2), these features are aggregated in `loop1.1` to which the pipeline pragma is applied. The equivalent HLPerf model is shown in Listing 2 (lines 1-3). The input FIFO is realized by the `store` class of SimPy and the read operation is performed via the `get()` method. We note that although FIFO operations are performed, the accessed data is ignored because

```
loop1: for(uint64_t e=tmp_begin; e<tmp_end;
    e++){
  vec vec1 = fifoIn.read();
  loop1.1: for(int i=0; i<2; i++){
    #pragma HLS pipeline II=1
    //aggregate feature of each neighbor
  }
}
loop2: for(int i=0; i<2; i++){
  #pragma HLS pipeline II=1
  fifoOut << vec_agg[i] // write the
      aggregated result.
}
```

Listing 1. Aggregation module in HLS C.

```
1  for e in range(deg):
2    _ = yield fifoIn.get()
3    yield env.timeout(L[0]+II[0])
4  for i in range(2):
5    if i == 0:
6      yield env.timeout(L[1])
7    else:
8      yield env.timeout(II[1])
9    fifoOut.put(1)
```

Listing 2. Aggregation module in HLPerf.

we are only focusing on performance estimation. Now that `loop1.1` doesn't contain any FIFO operations, we can use the expression illustrated in Figure 6 to calculate its overall execution time serving as the delay, which is simulated via the `timeout()` method of the SimPy environment.

The remaining loop in Listing 1, `loop2`, is also a pipelined loop designed to write the result feature vector to `fifoOut`. However, since it contains FIFO accesses, we cannot use a single formula to build the HLPerf model. Because the first input data needs to pass through the pipline before being written to the output FIFO, we establish a loop with iteration-level delay analysis, shown in Listing 2 (lines 4-9), where the time spent on generating the first and the rest of output data is L and II, respectively.

Loop Unroll leverages instruction-level parallelism by executing multiple copies of loop iterations in parallel to increase throughput at the cost of resource utilization. It is enabled with `#pragma HLS unroll`. In essence, this pragma reduces the number of data elements to be processed sequentially. Thus, it improves the kernel performance by reducing N. The `factor` option represents the number of generated hardware replications. Here we use the core loop of the grouped VMM module as an example to illustrate its HLPerf Model. Listing 3 shows the HLS C code where `loop1.1` is unrolled by a factor of 2. The corresponding HLPerf model is shown in Listing 4 where the impact of the unroll pragma is to reduce the iteration number by 2.

```
loop1: for(int k=0; k<FT/D; k++){
  loop1.1: for(int kd = 0; kd < D; kd++){
    #pragma HLS pipeline II=1
    #pragma HLS unroll factor=2
    // computation details of grouped VMM
  }
}
```

Listing 3. Grouped vector-matrix multiplication in HLS C.

```
yield env.timeout(L + II * (FT/D * D/2 - 1))
```

Listing 4. Grouped vector-matrix multiplication in HLPerf.

Loop Merge optimizes the finite state machine (FSM) of nested loops to remove the impact of inner loop latency on the overall throughput. This optimization technique is usually automatically inferred by Vitis. In essence, its performance benefit is enabling the latency of inner loops to be counted at every iteration of the outer loops. Let's still take the grouped VMM module as an example. In Listing 3, `loop1` and `loop1.1` are merged automatically by Vitis because this nested loop is a perfect loop. Therefore, in the HLPerf model shown in Listing 4, L is counted only once. Note that for the loops containing FIFO operations, the HLPerf model can be built by adding the iteration index of the outer loop into the if condition of the latency.

```
1  loop1: for(uint64_t e=tmp_begin; e<tmp_end;
        e++){
2    uint64_t tmp_src = fifoIn.read();
3    loop1.1: for(int i=0; i<2; i++){
4      #pragma HLS pipeline II=1
5      fifoOut << ft_in_mat[tmp_src*2+i];
6    }
7  }
```

Listing 5. Memory read module in HLS C.

Memory Burst Access enables large chunks of data accesses in contiguous address to be executed in burst mode to improve the overall memory bandwidth. During memory burst accesses, the memory latency (i.e., the accumulated latency of the DDR controller, AXI interconnect, M-AXI adapter, and kernel design) is paid only for the first data element and the successive data elements are accessed at every clock cycle. This optimization technique and associated parameters (e.g., burst length) are automatically inferred by Vitis. From a high-level perspective, the performance benefit of burst access lies in moving the request latency out of the memory access loop, which is similar to the principle of Loop Merge.

Taking a memory read module as an example, which enables memory read accesses in burst mode, the HLS C kernel and HLPerf model are shown in Listing 5 and Listing 6, respectively. In Listing 5, there is a nested loop. Since it is an edge-wise operation, the indices of the source node need to be read from the input FIFO `fifoIn` in the order of edges (lines 1-2). Then in `loop1.1` each feature vector is read from memory according to

```
1  for e in range(deg):
2    _ = yield fifoIn.get()
3    yield env.timeout(L_mem) # memory latency
4    for i in range(2):
5      if i == 0:
6        yield env.timeout(L)
7      else:
8        yield env.timeout(II)
9      yield fifoOut.put(1)
```

Listing 6. Memory read module in HLPerf.

the source node index (i.e., `ft_in_mat`) and written to the output FIFO `fifoOut`. Because of the emergence of `fifoIn` read operation (line 2), induced by the inherent irregularity of graph topology, `loop1` is a non-perfect loop. Therefore, the burst mode is only inferred in `loop 1.1` and the burst length is the feature size d. Listing 6 shows the equivalent HLPerf model. Since the burst access is constrained in the inner loop (lines 4-9), the impact of memory latency L_mem is applied prior to the start of the inner loop. Therefore, the timeout statement of L_mem is placed at line 3, between the outer loop and the inner loop.

```
1  loop1: for(uint64_t e=tmp_begin; e<tmp_end;
       e++){
2    #pragma HLS pipeline II=1
3    uint64_t tmp_src = fifoIn.read();
4    fifoOut << ft_in_mat[tmp_src];
5  }
```

Listing 7. Pipelined memory read requests in HLS C.

```
1  yield env.timeout(L_mem) # memory latency
2  for e in range(deg):
3    _ = yield fifoIn.get()
4    if e == 0:
5      yield env.timeout(L)
6    else:
7      yield env.timeout(II)
8    yield fifoOut.put(1)
```

Listing 8. Pipelined memory read requests in HLPerf.

Memory Port Widening increases the memory port width of the kernel to improve the throughput of memory access logic. Users can enable it by defining the memory port arguments of the top-level function using the `vector` data type, so that the kernel can fetch or store the whole vector instead of a single data element (e.g., `float`) at a time. Therefore, this optimization technique improves the performance by reducing the number of iterations N of the memory access loop, which is similar to the principle of Loop Unroll from a high-level perspective. Thus, after applying this optimization technique to loop1.1 in Listing 5 by setting the vector length to $d/2$, the iteration number is reduced from d to 2.

Furthermore, beyond its impact on N of the memory loop, widening memory ports provides more opportunities to improve memory access throughput for GNN implementations by enabling pipelined memory requests. For example, Listing 7 and Listing 8 show the HLS C design and the HLPerf model of the Memory Read Module after setting the vector length to d. As can be seen, the nested loop is changed to a regular pipelined loop, meaning that the memory read requests are pipelined. Therefore, in the HLPerf model in Listing 8, we can put the `timeout` statement of L_mem (line 1) to the outside of the loop1 (line 2) so that L_mem will not affect the performance of loop1 at each iteration. Note that given the width of physical pins on FPGA is limited (e.g., 512 bits), it will result in the increase of II if the memory port width is higher than the physical boundary. Therefore, there might be a trade-off to widening the memory ports in some cases.

4 DEVELOPER EXPERIENCE

Here, we describe a number of potential developer experience use cases that are enabled by HLPerf. In the first use case, as indicated above, HLPerf can be helpful in the tuning process that is ubiquitous in FPGA design efforts [7, 13, 30]. In designs for which the performance is data dependent (often the case in GNN computations, which are frequently sensitive to graph topology), the time required to adjust a potentially performance impacting parameter (e.g., alter a pragma) and understand its performance implications can be quite long in conventional workflows. HLPerf's approach of simulating performance exclusively, rather than including functional correctness, supports a shorter turnaround time, which provides overall benefit to the development process. When the designer is verifying algorithm correctness, a longer evaluation method is totally appropriate. When all they want to know is, "Is this approach faster or slower, and by how much?", re-evaluating functional correctness simply slows down the process.

In the second use case, HLPerf is usable prior to authoring HLS kernels. Figure 2 illustrates a generic computational pipeline, in which the contents of each pipeline stage are (as yet) undetermined. In fact, even the number of stages, S, has not yet been specified.

As the developer makes initial decisions about the number of pipeline stages (which will become HLS kernels) and the particular functions that will be performed at each pipeline stage, an HLPerf model can be developed that utilizes estimates of the performance parameters of each stage. These estimates might come from the developer's experience (i.e., they have written similar kernels in the past) or measurements of existing kernels (e.g., when library kernels are being invoked). This model can then be simulated to assess the performance impact of the data dependencies present in the input graph data set. In this case, it is incumbent on the developer to manually author the HLPerf models, rather than have them automatically derived from the HLS kernel code, which doesn't yet exist.

As an example of this second use case, consider a circumstance where a streaming computation is implemented across two execution platforms and data flows from the upstream portion to the downstream portion via a network link. This could be between two stages of a GNN model, or as part of any general-purpose streaming computation. To minimize the performance impact of the network, the author of the application chooses to compress the data moving across the network, and to maintain security the data is encrypted as well. The resulting communications link pipeline is shown in Figure 7.

Estimating the performance of individual stages of this pipeline by referencing the available AMD Xilinx libraries, one can readily build a HLPerf model of this communications link. We will show the performance implications of varying compression ratios on this pipeline below in Section 6.5.

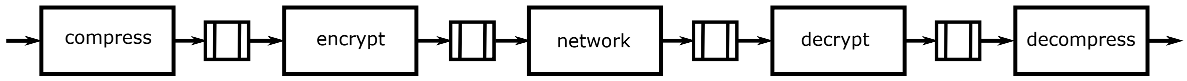

Fig. 7. Streaming data flow pipeline over a network.

As the development effort shifts to implementation of the HLS kernels themselves, the HLPerf model can be used to guide the kernel developer as to what should be the performance focus in each individual kernel's design. For example, if the HLPerf model indicates a particular stage in the pipeline is likely to be a performance bottleneck, the focus of the development can be on throughput. Alternatively, if the HLPerf model indicates some stage is unlikely to impact the overall throughput, the focus of the development can be on area savings. These types of area-speed tradeoffs occur frequently in FPGA designs, and HLPerf can be helpful in providing guidance as to which direction the tradeoff should be focused.

In this latter use case, the performance results from HLPerf are, of course, entirely dependent upon the quality of the individual kernel performance estimates provided by the developer. One possible path for the developer to pursue next is to prioritize the detailed design of individual HLS kernels that have significant performance uncertainty. Once authored, those kernels that previously had uncertain performance parameters can now exercise the HLS C Code to HLPerf Model link in Figure 3 to provide better knowledge of their individual performance to the HLPerf high-level model.

5 EVALUATION METHODOLOGY

GNN Models: Figure 8 shows the 6 GNN models that are used to evaluate HLPerf. They include GCN [24], one of the earliest GNN models; GraphSage (GS) [20]; and Graph Isomorphism Network (GIN) [42] as isotropic models. Also included are Graph Attention Network (GAT) [37]; Mixture Model Networks (MoNet) [29]; and the Gated Graph ConvNet (GatedGCN) [5] as anisotropic models. We configure the feature dimensions for various GNN kernels as follows: GCN and GraphSage have the same input and output dimensions at 128. GIN is assigned input and output dimensions of 64. The input, head, and output dimensions of GAT and MoNet are (128, 8, 16) and (64, 2, 64), respectively. GatedGCN is set with a feature dimension of 32.

Datasets: Table 2 shows the graph datasets used in our evaluation. All these graphs are collected from Open Graph Benchmark [22], a widely-used graph library for GNNs, and have a wide range of fields and scales. These graphs represent two classes of graphs with distinct topologies used in the GNN community: MH and MT consist of multiple small dense graphs, while AX and PT each consist of one single sparse graph. The maximum and average degree shown in Table 2 indicates their varying distributions ranging from regular-like to powerlaw-like. As mentioned above, we use GNNHLS [46], a benchmark suite of 6 GNN models, to evaluate HLPerf. HLPerf models are constructed with PyPy3 and SimPy [34] running on an Intel i7-8850H CPU at 2.6 GHz.

Table 8 lists the set of five general-purpose applications used to assess HLPerf on applications beyond GNN models. The data sets come from the original authors.

Table 2. Graph datasets [46].

Dataset	Node #	Edge #	Maximum Degree	Average Degree
OGBG-MOLTOX21 (MT)	145 459	302 190	6	2.1
OGBG-MOLHIV (MH)	1 049 163	2 259 376	10	2.2
OGBN-ARXIV (AX)	169 343	1 166 243	13 155	6.9
OGBN-PROTEINS (PT)	132 534	79 122 504	7 750	597.0

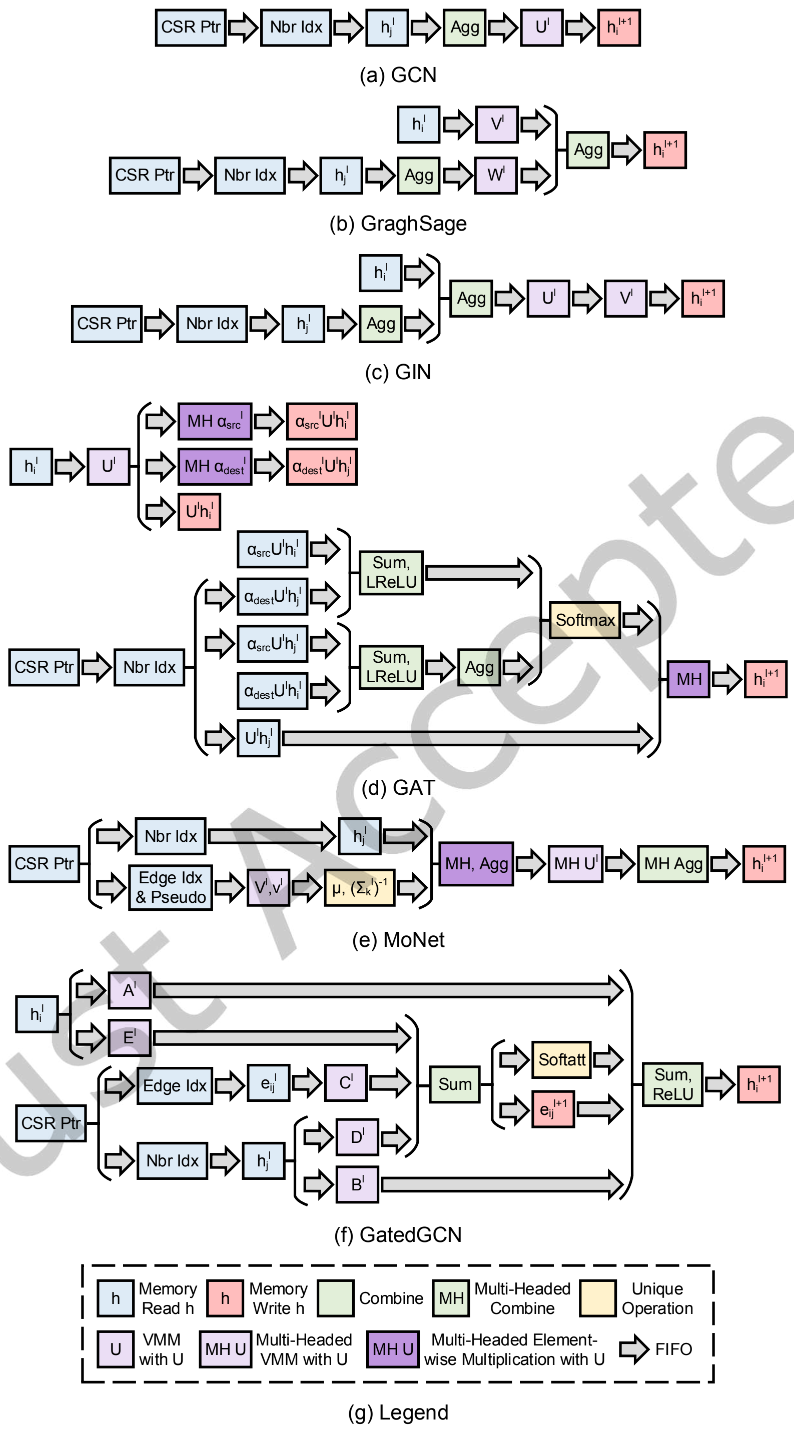

Fig. 8. Dataflow diagrams of GNN HLS kernels in GNNHLS [46].

Evaluation methods: To perform evaluation, we use a Xilinx Alveo U280 FPGA card, provided by the Open Cloud Testbed [25], to execute the HLS kernels. This FPGA card provides 8 GB of HBM2 with 32 memory banks at 460 GB/s total bandwidth, 32 GB of DDR memory at 38 GB/s, and 3 super logic regions (SLRs) with 1205K look-up tables (LUTs), 2478K registers, 1816 BRAMs, and 9020 DSPs. We adopt 32-bit floating point as the data format, and Vitis 2023.1 for hardware emulation, synthesis, and hardware linkage.

6 EVALUATION

As part of the evaluation, we will start by investigating the performance of the simulation execution within HLPerf. This will be followed by an assessment of the accuracy of the performance predictions, an illustration of the use of HLPerf for buffer size analysis, a description of HLPerf's utility in identifying and addressing performance bottlenecks within the application, and a look into the use of HLPerf on other applications.

6.1 Simulator Performance

We first examine the performance of HLPerf by comparing its simulation elapsed time with RTL simulation and several previous cycle-accurate simulators. We use Vitis hardware emulation mode to conduct the RTL simulation. Due to the low speed of RTL simulation, it will take a very long time on even the smallest-scale graph dataset MT used in this paper. Thus, in order to constrain the RTL simulation elapsed time to within 1 hour, we perform all the GNN kernels on just the first subgraph of MT with 16 nodes and 34 edges. Table 3 shows the simulation elapsed time and the speedup of HLPerf over RTL simulation. We observe that the speedup of HLPerf ranges from 1 200× to 35 700× across all the GNN applications, and the average speedup is 13 500×.

Table 3. Simulation elapsed time for HLPerf and RTL simulation, and the speedup of HLPerf relative to RTL simulation.

	HLPerf (s)	RTL Simulation (s)	Speedup
GCN	0.08	1 779	21 600×
GS	0.14	1 988	14 500×
GIN	0.10	3 538	35 700×
GAT	0.21	2 771	12 900×
MoNet	0.10	3 368	34 100×
GatedGCN	0.28	352	1 200×

Compared with the reported speed of the state-of-the-art cycle-accurate simulator, LightningSim [31], our work averages over 400× faster (primarily by giving up the requirement to be cycle accurate). In comparison to the reported performance of Flash [10], HLPerf exhibits an average speedup of just over 8×. Relative to FastSim [1], our methodology achieves an average speedup of over 300×. Note that although these comparisons might not be perfectly fair, for example Flash's C-based implementation is inherently more efficient than HLPerf's Python implementation and they are not all simulating the same designs, our work still achieves significant performance improvement. Note, the above comparisons are performed by examining the performance gain relative to RTL simulation as reported by the simulators' authors.

The performance benefit of HLPerf comes from 3 aspects: (1) it decouples the performance estimation from computational details of the algorithm, which is important for computation-heavy algorihtms like GNNs, (2) it simplifies the signal list to be simulated with a higher-level abstraction, and (3) HLPerf as a discrete-event simulator doesn't simulate every clock cycle. These results indicate that the "approximately-cycle-accurate" approach of HLPerf can substantially diminish the time required to estimate performance for HLS GNN kernels.

Since increasing the graph scale leads to long simulation times, this superior speedup makes HLPerf quite suitable for design exploration of GNNs with large-scale graphs. Table 4 shows the elapsed time of HLPerf across all 6 GNN kernels and 4 graph datasets. In contrast, Table 5 presents the elapsed time of conventional HLS workflow procedures including HLS synthesis steps (preprocessing & transformation, scheduling & binding, and RTL generation) and hardware compilation. The duration of these procedures are independent of the input graphs. Comparing Table 4 and the first two rows of HLS synthesis from Table 5, we observe that for small-scale graphs the elapsed time of HLS synthesis is comparable to HLPerf simulation, while for large-scale graphs HLPerf simulation becomes increasingly dominant. A comparison between Table 4 and hardware compilation time in Table 5, reveals that the smaller the graph scale, the higher performance benefit of HLPerf to be achieved against hardware compilation. Even for the largest graph adopted in the benchmark, PT, the performance benefit of HLPerf over hardware compilation ranges from 4.4× to 122×. In addition, we note that HLPerf is based on the single-threaded Python library, and therefore its performance could potentially be further improved by utilizing multiple threads and a more efficient implementation.

Table 4. HLPerf simulation elapsed time of all the GNN kernels on 4 graph datasets.

	MT (s)	MH (s)	AX (s)	PT (s)
GCN	13	83	18	359
GS	31	230	42	364
GIN	24	158	36	298
GAT	68	542	158	6 097
MoNet	19	128	31	699
GatedGCN	71	426	285	4 808

Table 5. Elapsed time of conventional HLS workflow procedures, including HLS synthesis steps and hardware compilation.

Conventional HLS Workflow Procedures		GCN (s)	GraphSage (s)	GIN (s)	GAT (s)	Monet (s)	GatedGCN (s)
HLS Synthesis	Preprocessing & Transformation	47	85	26	111	29	19
	Scheduling & Binding	58	95	51	146	55	59
	RTL Generation	77	168	83	251	94	139
Hardware Compilation		23 602	44 325	16 397	27 098	18 826	25 922

6.2 Application Performance Prediction Accuracy

We next quantitatively examine the accuracy of HLPerf by comparing the simulated execution time with the experimental execution time measured on the FPGA platform. For these experiments, and all those that follow, we are using the entire graph for performance evaluation, both in the HLPerf simulation and in the FPGA platform execution. We use measured execution time instead of RTL simulation for two reasons: first, it is the gold standard for understanding performance; and second, RTL simulation is prohibitively time-consuming and thus impractical for complex GNN HLS kernels with real-world graph datasets. We use error rate, defined as the percentage of the simulation result deviating from experimental measurement, to represent the accuracy of HLPerf. Figure 9 shows the normalized execution time predicted by HLPerf for all the 6 GNN kernels on 4 graph datasets relative to FPGA measurements, and Table 6 shows the absolute numbers for execution times and corresponding error rates. To enhance the clarity of comparison, in Table 6 we use the same clock frequency as on-board measurements. From the table, we observe that the error rate of HLPerf ranges from 3.3% to 14.7%, and

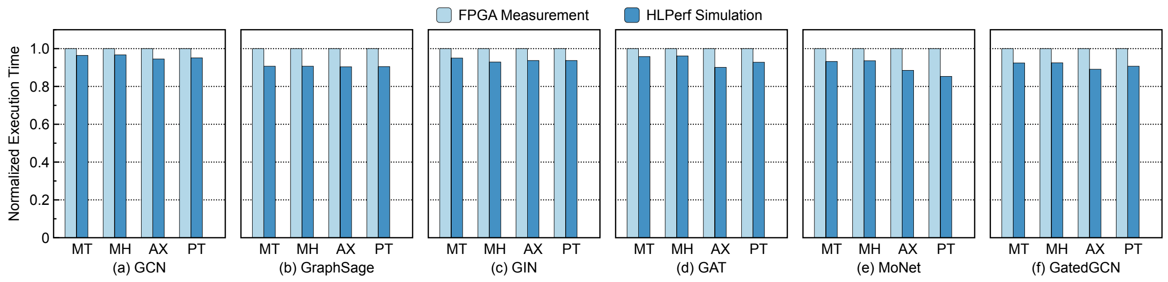

Fig. 9. Normalized HLPerf predicted execution time relative to measurements on physical FPGA. The closer to 1 the better.

is 7% on average. This level of imprecision is quite acceptable when assessing design alternatives that regularly exceed factors of 2 and more.

The observed inaccuracy in HLPerf's performance predictions can be be attributed to several factors. First, as HLPerf operates at a higher level of abstraction, it does not consider some low-level architecture details. For example, it cannot account for the impact of implementation strategies on onboard execution performance since it operates prior to hardware compilation. HLPerf relies on the estimated parameters (e.g., L and II) derived from the HLS report. These parameters are subject to further optimizations during the back-end compilation phase, which can affect the accuracy of HLPerf simulation. Furthermore, HLS synthesis does not provide all the detailed information needed for accurate HLPerf predictions. In scenarios with unoptimized loops where latency is not explicitly reported by HLS, operation count is used as a substitute for latency. Similarly, a default memory latency is applied in the absence of precise measurements, which are not available until actual execution on the FPGA board. Based on detailed examination of a small subset of the experimental runs, the reliance on latency and iteration interval estimates are the primary error contributor for compute-intensive kernels and the memory latency estimate is the primary error contributor for memory-intensive kernels.

Although the simulated and experimental results are not exactly the same, the timing results of HLPerf track the FPGA execution time closely across graphs with various topologies and GNN kernels with distinct structures, indicating that HLPerf is able to recognize the inherent irregularity of the graph datasets and the algorithm.

Table 6. Execution time of FPGA measurements (HLS), predicted execution time from HLPerf, and corresponding error rate relative to FPGA measurements.

	MT			MH			AX			PT		
	HLS (s)	HLPerf (s)	Error Rate	HLS (s)	HLPerf (s)	Error Rate	HLS (s)	HLPerf (s)	Error Rate	HLS (s)	HLPerf (s)	Error Rate
GCN	0.10	0.10	3.6%	0.74	0.71	3.3%	0.39	0.36	5.5%	25.72	24.44	4.9%
GraphSage	0.16	0.14	9.3%	1.17	1.06	9.3%	0.62	0.56	9.6%	40.24	36.42	9.5%
GIN	0.08	0.08	5.0%	0.63	0.59	7.1%	0.33	0.31	6.3%	22.04	20.65	6.3%
GAT	0.23	0.22	4.2%	1.65	1.59	3.9%	0.74	0.66	9.9%	31.10	28.86	7.2%
MoNet	0.08	0.07	6.8%	0.55	0.51	6.4%	0.09	0.08	11.5%	1.37	1.17	14.7%
GatedGCN	0.10	0.09	7.6%	0.72	0.67	7.5%	0.39	0.34	10.9%	25.40	23.04	9.3%

Turning our attention to the impact of graph topology on the accuracy of HLPerf, we find that HLPerf achieves lower error on regular-like graphs than powerlaw-like graphs. According to Table 6, the average error rate of GNN applications on MT, MH, AX, and PT are 5.8%, 5.9%, 8.6%, 8.1%, respectively. We believe this to be because

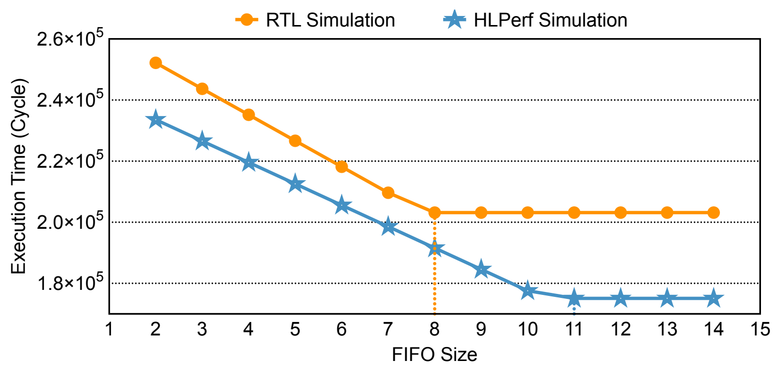

Fig. 10. FIFO size sensitivity.

irregular graphs (i.e., PT and AX) with higher edge-to-node ratio and variation of degree distribution brings increased uncertainty to the performance predictions. In contrast, regular graphs contain less irregularity. Thus, the performance predictions have lower uncertainty.

Note that there is always a trade-off between simulation speed and accuracy. Therefore, although HLPerf is less accurate than full cycle-accurate simulators (e.g., LightningSim reports a 0.1% error rate on average compared with RTL simulation on an example dataset, while Flash and FastSim report 0% error rate relative to RTL simulation on their respective benchmarks), HLPerf matches the "approximately-cycle-accurate" goal in which the simulation speed can be dramatically improved at the cost of a small accuracy loss. We contend that when design choices can have as dramatic a performance impact as three orders of magnitude (e.g., see [6, 15, 30, 35]), this is a tradeoff well worth considering.

6.3 FIFO Size Sensitivity

One of the clear benefits of a simulation-based performance modeling approach is that one can effectively observe more than just aggregate performance. Here, we will illustrate the use of HLPerf to assess whether or not inter-stage buffers (i.e., FIFOs) have been allocated enough storage so as to not become a performance bottleneck themselves. In order to investigate the effectiveness of HLPerf for tuning FIFO size, we build a micro-benchmark based on a common property across all the GNN applications. We do this, instead of directly using GNN applications, because the applications' dataflow architecture contains sufficiently many stages and FIFOs as to make it challenging to illustrate the technique.

This micro-benchmark is a dataflow architecture with 2 functions and 1 FIFO. The first function represents an edge-wise aggregation operation and the latter one is a node-wise update operation. These two functions are connected by a FIFO. We use a synthetic graph to make the FIFO alternatively full and empty, reproducing the FIFO size issue. Figure 10 shows the cycle counts of RTL simulation and HLPerf running on the micro-benchmark with the FIFO size ranging from 2 entries to 15 entries. We observe that HLPerf's performance predictions are sensitive to the FIFO size in a manner that is quite close to the more detailed simulation available at the RTL level. In addition, optimized FIFO sizes achieved from RTL simulation and HLPerf are 8 and 11, respectively. Although these optimized FIFO sizes are not the same because of the accuracy loss of HLPerf, it requires less iterations for users or heuristic tuning algorithms to find the optimized FIFO size if starting from HLPerf. Note that HLPerf, as a high-level software simulator designed for performance prediction, does not encompass functional verification. Consequently, it may not identify certain potential deadlock issues related to functional correctness.

6.4 Identifying Performance Bottlenecks

HLPerf can not only estimate overall application performance but can also identify the bottleneck kernel of a dataflow pipeline. By comparing the simulation results of each function without stalls and the simulation results of the whole dataflow architecture, the bottleneck can be identified. Here we take GCN as an example. After performing HLPerf on GCN and 4 graph datasets, we find that the bottleneck of the GCN kernel on regular-like graphs, MT and MH, is the node-wise update operation implemented as a vector-matrix multiplication. On the other hand, on powerlaw-like graphs such as AX and PT, the bottleneck function is the edge-wise memory read module. After replacing this latter kernel with the pipelined memory read request enabled, as illustrated in Listing 7, the GCN kernel is accelerated by 2.6× and 5.3× on AX and PT, respectively. And the cycle counts of GCN doesn't change on MT and MH (the execution time changes because of distinct clock frequency inferred by the HLS tool). Table 7 shows the execution time, error rate, and simulation time of HLPerf for the modified GCN kernel. From the table, we observe that the error rate ranges from 1.9% to 6.9% and the average error rate is 2.8%, which matches the accuracy of HLPerf on other GNN applications. Hence we conclude that HLPerf can be used to tune the GCN HLS kernel for better performance.

Table 7. Execution time, error rate, and simulation time of HLPerf for the GCN Kernel with pipelined memory requests.

	Execution Time (s)		HLPerf Performance	
Graphs	HLS	HLPerf	Error Rate	Simulation Time (s)
MT	0.09	0.09	2.3%	10
MH	0.68	0.67	1.9%	68
AX	0.11	0.11	6.9%	17
PT	3.82	3.74	2.1%	179

6.5 General-Purpose Application Evaluation

While the main motivation for the development of HLPerf is understanding the performance of GNNs, the techniques used are not limited exclusively to GNNs. In general, the set of applications to which HLPerf is applicable is constrained to dataflow architectures for which the developer has (or can learn) information about the performance of the constituent dataflow stages. In effect, if the performance estimates (e.g., latency and initiation interval) that come out of the high-level synthesis compilation process are accurate, HLPerf can be used to understand end-to-end performance.

In order to evaluate the ability of HLPerf to be used on other HLS kernels, beyond GNN models, we adopt 5 general-purpose benchmarks with dataflow architectures used in previous work [31]. Distinct from GNN applications, which are complicated irregular workloads with large-scale graph input datasets, these benchmarks have small enough input datasets that an RTL simulation with the full dataset is reasonable. Therefore, we use the results of RTL simulation as the baseline for comparison purposes.

Table 8 (left) presents both the predicted application execution time by HLPerf and RTL simulation, along with the accuracy of HLPerf relative to RTL simulation. From the table, we observe that the error rate ranges from 0.01% to 9.5% with an average of 2.0%, which is consistent with the results in Section 6.2.

Table 8 (right) details the simulation elapsed time of RTL simulation and HLPerf, and speedup of both HLPerf and LightingSim [31] relative to RTL simulation. From the table, we find that HLPerf speedup over RTL simulation ranges from 188× to 14 062× and it is 112× on average faster than LightingSim. Among the benchmarks, Vector Accumulator and Cascade Adder exhibit lower speedup through HLPerf relative to the other general-purpose benchmarks. This discrepancy is attributed to these kernels having a higher number of scheduling events,

Table 8. Predicted execution time and simulation elapsed time of RTL simulation and HLPerf, and corresponding error rate relative to RTL simulation for 5 general-purpose applications. The performance speedup of HLPerf and LightingSim are relative to RTL simulation.

	Predicted Results (cycles)			Simulation Elapsed Time (s)			
	RTL Sim	HLPerf	Error Rate	RTL Sim	HLPerf	HLPerf Speedup	LightingSim Speedup [31]
Vector Accumulator [40]	4 642	4 239	8.7%	37.18	0.1981	188×	15.2×
Cascade Adder [41]	9 212	8 333	9.5%	41.04	0.2292	179×	7.3×
Parallel Merge Sort [23]	73	68	6.8%	38.63	0.0027	14 062×	30.7×
Block Matrix Multiplication [23]	226	216	4.4%	21.28	0.0032	6 590×	14.3×
Multi-Stage FFT [23]	7 208	7 209	0.01%	87.83	0.0139	6 338×	29.7×

triggered by Python generator functions, leading to extended duration of interleaving suspension and resumption of process functions as discussed in Section 3.2. In addition, the speedup of HLPerf over RTL simulation for these two small benchmarks is lower than that observed for the GNN HLS kernels. This is because the performance of RTL simulation is closely related to the computational details of the HLS kernels, while HLPerf's efficiency depends on the quantity of processes and events, rather than the computational intricacies within the dataflow architecture. Hence, HLS kernels of varying computational complexity might yield similar performance under HLPerf models.

Returning to the network communications link example of Section 4, prior to deployment of the individual kernels, we have estimates of their performance because they are available as library elements. Similarly, we can estimate the performance of a network link from the literature (e.g., see [16]). What we may not know, however, early in the design, is the impact of the compression stage on the overall performance. How does the compression ratio achieved in the `compress` kernel influence the performance of the overall pipeline?

Since HLPerf does not perform functional simulation, and is therefore not actually executing the compression algorithm as part of the simulation process, this is clearly an example of what is being traded off to achieve faster simulation speeds. What we can do, however, is to sweep the compression ratio over a range of credible values and explore the performance implications for each value.

Figure 11 shows the results of simulating the pipeline of Figure 7, varying the compression ratio between 1 : 1 and 8 : 1. What we see is that the effective data rate (referenced at the input) increases as the compression ratio increases, up to a point of no additional benefit.

Here, while HLPerf cannot tell us the actual compression ratio, it does give the designer information about the performance implications of different compression ratios.

7 CONCLUSIONS AND FUTURE WORK

This paper has introduced HLPerf, an open-source, simulation-based performance evaluation methodology for dataflow architectures that is over 13 500× faster than RTL simulation and 400× faster than state-of-the-art cycle-accurate tools, on average, at the cost of 7% average error rate relative to FPGA measurements. This speed increase is attributable to three decisions inherent in HLPerf: (1) it does not try to be cycle-accurate, instead being satisfied with being "approximately" cycle-accurate, (2) it simplifies the signal list to be simulated with high-level abstractions, and (3) it does not try to verify functional correctness, focusing exclusively on performance prediction.

Because it is simulation-based, HLPerf can reflect the performance variations of computations that are dependent on input data sets. As such, it is well suited to evaluating the performance of graph neural network

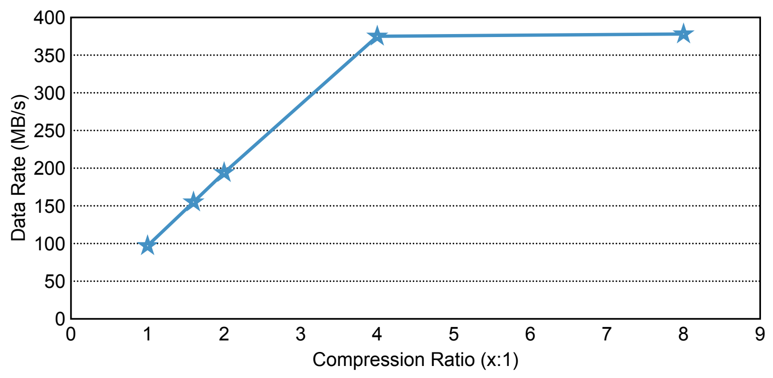

Fig. 11. Streaming data pipeline with compression.

computations, which can be heavily influenced by graph topology. In the empirical evaluation, we showed that distinct HLS kernels in the dataflow architecture can be performance bottlenecks for regular-like graphs versus powerlaw-like graphs.

While the fast performance predictions supported by HLPerf are useful for optimizing the design of individual HLS kernels (e.g., choosing specific pragmas, etc.), HLPerf can also be an effective design tool prior to authoring any of the HLS code itself. By using performance estimates of individual kernels, HLPerf can analyze the performance of the dataflow pipeline as a whole, focusing the attention of the developer on potential bottleneck kernels earlier in the design cycle.

Future work will focus on assessing how well the techniques exploited by HLPerf can generalize to a wider set of other problems, outside the scope of GNNs, that have performance that is dependent on the input data set. This will require expansion of the model conversion (translating HLS C code to SimPy simulation models) and the pragma-driven pattern modeling (which is limited to the pragmas typically used in loop optimizations that are prevalent in GNN models). One approach is to rely initially on manually authored HLPerf models, as we did in Section 6.5, which would enable us to assess the techniques without yet implementing the source-to-source compiler required to author HLPerf models automatically.

With current design space exploration yielding performance variability over multiple orders of magnitude, a fast approach to performance estimation is an important tool. HLPerf seeks to do precisely that task.

ACKNOWLEDGEMENTS

This research is supported by NSF under grants CNS-1739643 and CNS-1763503 and a gift from BECS Technology, Inc. The authors are grateful for the use of the Open Cloud Testbed [25] as an experimentation platform.

REFERENCES

[1] Mohammed Abderehman, Jayprakash Patidar, Jay Oza, Yom Nigam, TM Abdul Khader, and Chandan Karfa. 2021. FastSim: A fast simulation framework for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 5 (2021), 1371–1385. https://doi.org/10.1109/TCAD.2021.3090339
[2] ARM Xilinx. 2023. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393). https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration. Accessed Aug. 2023.
[3] Eli Bendersky. 2023. `pcyparser`. https://github.com/eliben/pycparser. Accessed Oct. 2023.
[4] Mark Bohr. 2007. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter* 12, 1 (2007), 11–13. https://doi.org/10.1109/N-SSC.2007.4785534
[5] Xavier Bresson and Thomas Laurent. 2017. Residual Gated Graph ConvNets. arXiv preprint. , 11 pages. arXiv:1711.07553

[6] Nick Brown. 2020. Exploring the acceleration of Nekbone on reconfigurable architectures. In *Proc. of IEEE/ACM Int'l Workshop on Heterogeneous High-performance Reconfigurable Computing*. IEEE, 19–28. https://doi.org/10.1109/H2RC51942.2020.00008

[7] Anthony M. Cabrera and Roger D. Chamberlain. 2020. Design and Performance Evaluation of Optimizations for OpenCL FPGA Kernels. In *Proc. of High-Performance Extreme Computing Conference*. IEEE, 7 pages. https://doi.org/10.1109/HPEC43674.2020.9286221

[8] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. 2013. From software to accelerators with LegUp high-level synthesis. In *Proc. of Int'l Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE, 9 pages. https://doi.org/10.1109/CASES.2013.6662524

[9] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 69–80. https://doi.org/10.1145/3431920.3439290

[10] Young-Kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. 2020. Flash: Fast, parallel, and accurate simulator for HLS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4828–4841. https://doi.org/10.1109/TCAD.2020.2970597

[11] Young-kyu Choi, Peng Zhang, Peng Li, and Jason Cong. 2017. HLScope+: Fast and accurate performance estimation for FPGA HLS. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 691–698. https://doi.org/10.1109/ICCAD.2017.8203844

[12] Hanjun Dai, Zornitsa Kozareva, Bo Dai, Alex Smola, and Le Song. 2018. Learning steady-states of iterative algorithms over graphs. In *Proc. of International Conference on Machine Learning*. PMLR, 1106–1114.

[13] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2020. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1014–1029. https://doi.org/10.1109/TPDS.2020.3039409

[14] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proc. of 38th International Symposium on Computer Architecture*. ACM, 365–376. https://doi.org/10.1145/2000064.2000108

[15] Clayton J. Faber, Steven D. Harris, Zhili Xiao, Roger D. Chamberlain, and Anthony M. Cabrera. 2022. Challenges Designing for FPGAs Using High-Level Synthesis. In *Proc. of High-Performance Extreme Computing Conference*. IEEE, 7 pages. https://doi.org/10.1109/HPEC55821.2022.9926398

[16] Clayton J. Faber, Tom Plano, Samatha Kodali, Zhili Xiao, Abhishek Dwaraki, Jeremy D. Buhler, Roger D. Chamberlain, and Anthony M. Cabrera. 2021. Platform Agnostic Streaming Data Application Performance Models. In *Proc. of IEEE/ACM Workshop on Redefining Scalability for Diversely Heterogeneous Architectures*. IEEE, 10 pages. https://doi.org/10.1109/RSDHA54838.2021.00008

[17] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. *Advances in Neural Information Processing Systems* 30 (2017), 10 pages.

[18] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proc. of 53rd IEEE/ACM Int'l Symposium on Microarchitecture*. IEEE, 922–936. https://doi.org/10.1109/MICRO50266.2020.00079

[19] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proc. of IEEE International Joint Conference on Neural Networks*, Vol. 2. IEEE, 729–734. https://doi.org/10.1109/IJCNN.2005.1555942

[20] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Adv. Neural Inf. Process. Syst.* 30 (2017), 11 pages.

[21] Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wen-mei Hwu, and Deming Chen. 2019. FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge. In *Proc. of 56th Design Automation Conference*. ACM, 6 pages. https://doi.org/10.1145/3316781.3317829

[22] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in Neural Information Processing Systems* 33 (2020), 22118–22133.

[23] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel programming for FPGAs. arXiv preprint. , 235 pages. arXiv:1805.03648

[24] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *Proc. of International Conference on Learning Representations*. 14 pages. https://openreview.net/forum?id=SJU4ayYgl

[25] Miriam Leeser, Suranga Handagala, and Michael Zink. 2021. FPGAs in the Cloud. *Computing in Science & Engineering* 23, 6 (2021), 72–76. https://doi.org/10.1109/MCSE.2021.3127288

[26] Shengwen Liang, Cheng Liu, Ying Wang, Huawei Li, and Xiaowei Li. 2020. DeepBurning-GL: an automated framework for generating graph neural network accelerators. In *Proc. of 39th International Conference on Computer-Aided Design*. ACM, 72:1–72:9. https://doi.org/10.1145/3400302.3415645

[27] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. 2020. EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Trans. Comput.* 70, 9 (2020), 1511–1525. https://doi.org/10.1109/TC.2020.3014632

[28] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, and Setareh Rafatirad. 2019. Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design. In *Proc. of 29th International Conference on Field Programmable Logic and Applications*. IEEE, 397–403. https://doi.org/10.1109/FPL.2019.00069

[29] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. 2017. Geometric deep learning on graphs and manifolds using mixture model CNNs. In *Proc. of Conf. on Computer Vision and Pattern Recognition*. IEEE, 5115–5124. https://doi.org/10.1109/CVPR.2017.576

[30] Ahmed Sanaullah, Rushi Patel, and Martin Herbordt. 2018. An empirically guided optimization framework for FPGA OpenCL. In *Proc. of International Conference on Field-Programmable Technology*. IEEE, 46–53. https://doi.org/10.1109/FPT.2018.00018

[31] Rishov Sarkar and Cong Hao. 2023. LightningSim: Fast and Accurate Trace-Based Simulation for High-Level Synthesis. In *Proc. of 31st IEEE International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 11 pages. https://doi.org/10.1109/FCCM57271.2023.00010

[32] 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. https://doi.org/10.1109/TCAD.2019.2943570

[33] John Shalf. 2020. The future of computing beyond Moore's Law. *Philosophical Transactions of the Royal Society A* 378, 2166 (2020), 20190061. https://doi.org/10.1098/rsta.2019.0061

[34] Team SimPy. 2023. SimPy: Discrete event simulation for Python. https://simpy.readthedocs.io. Accessed Aug. 2023.

[35] 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), 32:1–32:27. https://doi.org/10.1145/3494534

[36] Thomas N Theis and H-S Philip Wong. 2017. The end of Moore's Law: A new beginning for information technology. *Computing in Science & Engineering* 19, 2 (2017), 41–50. https://doi.org/10.1109/MCSE.2017.29

[37] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. In *Proc. of International Conference on Learning Representations*. 12 pages. https://openreview.net/forum?id=rJXMpikCZ

[38] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2020), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[39] Shunxin Xiao, Shiping Wang, Yuanfei Dai, and Wenzhong Guo. 2022. Graph neural networks in node classification: survey and evaluation. *Machine Vision and Applications* 33 (2022), 1–19. https://doi.org/10.1007/s00138-021-01251-0

[40] Xilinx. 2023. Basic examples for Vitis HLS. https://github.com/Xilinx/Vitis-HLS-Introductory-Examples. Accessed Feb. 2024.

[41] Xilinx. 2023. Vitis Accel Examples. https://github.com/Xilinx/Vitis_Accel_Examples. Accessed Feb. 2024.

[42] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *Proc. of International Conference on Learning Representations*. 17 pages. https://openreview.net/forum?id=ryGs6iA5Km

[43] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGNC: A GCN accelerator with hybrid architecture. In *Proc. of International Symposium on High Performance Computer Architecture*. IEEE, 15–29. https://doi.org/10.1109/HPCA47549.2020.00012

[44] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An end-to-end deep learning architecture for graph classification. In *Proc. of AAAI Conf. on Artificial Intelligence*, Vol. 32. 4438–4445. https://doi.org/10.1609/aaai.v32i1.11782

[45] Yongan Zhang, Haoran You, Yonggan Fu, Tong Geng, Ang Li, and Yingyan Lin. 2021. G-CoS: GNN-accelerator co-search towards both better accuracy and efficiency. In *Proc. of IEEE/ACM International Conference On Computer Aided Design*. IEEE, 9 pages. https://doi.org/10.1109/ICCAD51958.2021.9643549

[46] Chenfeng Zhao, Zehao Dong, Yixin Chen, Xuan Zhang, and Roger D. Chamberlain. 2023. GNNHLS: Evaluating Graph Neural Network Inference via High-Level Synthesis. In *Proc. of 41st IEEE International Conference on Computer Design*. IEEE, 574–577. https://doi.org/10.1109/ICCD58817.2023.00092

[47] Ao Zhou, Jianlei Yang, Yeqi Gao, Tong Qiao, Yingjie Qi, Xiaoyi Wang, Yunli Chen, Pengcheng Dai, Weisheng Zhao, and Chunming Hu. 2021. Brief industry paper: Optimizing memory efficiency of graph neural networks on edge computing platforms. In *Proc. of Real-Time and Embedded Technology and Applications Symp.* IEEE, 445–448. https://doi.org/10.1109/RTAS52030.2021.00048

[48] Ao Zhou, Jianlei Yang, Yingjie Qi, Yumeng Shi, Tong Qiao, Weisheng Zhao, and Chunming Hu. 2023. Hardware-Aware Graph Neural Network Automated Design for Edge Computing Platforms. arXiv preprint. , 6 pages. arXiv:2309.10875