DeepBurning-GL: an Automated Framework for Generating Graph Neural Network Accelerators

Shengwen Liang[1,2], Cheng Liu[1], Ying Wang[1,2], Huawei Li[1,2], Xiaowei Li[1,2]
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences[1],
University of Chinese Academy of Sciences[2]

ABSTRACT

Building FPGA-based graph learning accelerators is very time-consuming due to the low-level RTL programming and the complicated design flow of FPGA development. It also requires the architecture and hardware expertise from the Graph Neural Network (GNN) application developers to tailor efficient accelerator designs on FPGAs. This work proposes an automation framework, DeepBurning-GL, which is compatible with state-of-the-art graph learning frameworks such as Deep Graph Library so that the developers can easily generate application-specific GNN accelerators from the software-described models. First, DeepBurning-GL employs a GNN performance analyzer to locate the performance bottleneck of specific GNN applications and decide the major design architectures and parameters that meet the user-specified constraints. Second, DeepBurning-GL provides a series of pre-built design templates such as computing templates and memory templates, which can be parameterized and fused to generate the final accelerator design. It also includes an optimizer that conducts automatic optimization by adjusting the accelerator architectural parameters. In evaluation, we use DeepBurning-GL to generate customized accelerators on three different FPGA platforms for various GNN models and workloads. The experimental results show that the generated accelerators achieve 179.4X and 40.1X energy-efficiency boost over the CPU and GPU solutions on average and deliver a 6.28X speedup and 6.73X energy-efficiency improvement on average compared to the latest GNN accelerator HyGCN on Alveo U50.

CCS CONCEPTS

• **Hardware** → **Integrated circuits**; **Reconfigurable logic and FPGAs**;

KEYWORDS

graph neural networks, automated framework, FPGA

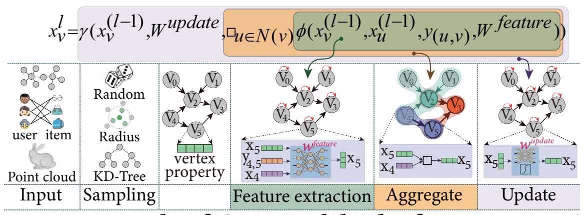
ACM Reference Format:
Shengwen Liang[1,2], Cheng Liu[1], Ying Wang[1,2], Huawei Li[1,2], Xiaowei Li[1,2]. 2020. DeepBurning-GL: an Automated Framework for Generating Graph Neural Network Accelerators. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415645

1 INTRODUCTION

Recently, graph neural networks (GNNs) that operate on unstructured data are becoming a rapidly progressing field with diverse applications such as social networks [16], knowledge graph [14], and point cloud [22]. The success of GNNs propelled the deployment of GNNs to the production system on the cloud and edge platform, such as Pinterest [28], Alibaba [27], and Baidu [13].

Similar to DNNs, a typical GNN-layer is depicted in Fig. 1 and it includes three primary stages: ***feature extraction, aggregate, and update***, followed by an optional ***sampling*** stage. The ***feature extraction*** and ***update*** stage acts like neural network inference and thus involves regular computational and memory access patterns. The ***aggregate*** stage walks on the graph topology to aggregate vertex/edge features, while the ***sampling*** stage constructs a new graph by user-defined sampling operations. These two stages operating on a large and sparse graph can incur dynamic computational data flow and numerous irregular memory access. The coexistence of regular neural network operations and irregular graph level operations in GNN does not favor conventional CPU and GPU solutions showing low efficiency when facing the divergent computation patterns of the four stages [25, 29].

Although specialized GNN accelerators of ASICs such as HyGCN [26] and EnGN [7], are tailored as alternatives to CPU and GPU, they are still not flexible enough to meet the requirements of various GNN-based applications from the cloud to edge. First and foremost, because graph learning is a fast-developing field and novel GNN architectures keep emerging, ASIC-based accelerators of fixed architecture sometimes cannot run state-of-the-art GNN models. For example, HyGCN does not support graph recurrent network [6] and graph attention network [21] while EnGN cannot run EdgeConv [22]. Second, according to the application scenarios, the GNN accelerator architectures may have different design goals such as high-throughput, low latency, and low energy. For instance, the GNN-based recommendation system in Taobao [27] typically needs to perform high-throughput recommendations while point cloud applications such as SLAM for autonomous vehicles require short-latency and low energy consumption, which results in entirely different GNN processing architectures. Third, different GNN systems favor different configurations since their performance bottleneck varies from case to case. Specifically, the execution time of the four GNN stages is impacted by several essential factors including the characteristics of the input graphs, the GNN network architecture, and the user-specified design constraints of logic resources and power budget. These factors cause the variation of compute and memory resource requirements in different GNN stages, which all have the possibility to become the bottleneck in a GNN application. Hence, a fixed GNN accelerator of ASIC chip such as HyGCN and EnGN cannot well satisfy the requirements of different applications.



$x_v^l = \gamma(x_v^{(l-1)}, W^{update}, \square_{u \in N(v)} \phi(x_v^{(l-1)}, x_u^{(l-1)}, y_{(u,v)}, W^{feature}))$
user item
Point cloud
Random
Radius
KD-Tree
vertex property
Input
Sampling
Feature extraction
Aggregate
Update

Fig. 1. An example of GNN model. The feature extraction, aggregate, and update stage are performed iteratively.

Fortunately, compared to the ASIC solutions, FPGAs offer much more flexibility as a reconfigurable and programmable architecture. However, because the conventional design flow of RTL programming is complicated and error-prone, it takes considerable design efforts to realize the customized GNN accelerator case by case for the target GNN workload or on the target FPGA devices. Though there is a lot of prior study on automating the flow of DNN accelerator development for FPGA [5, 24, 31], it is non-trivial to directly apply prior DNN-FPGA automation frameworks to the GNN-based applications. The reason is mainly attributed to three factors including 1) they are built on the traditional deep learning frameworks such as Caffe and Pytorch. However, these frameworks struggle in their inefficiency and even inability to describe the latest GNN architectures. Therefore, GNN development frameworks such as DGL and PyG must be supported in the GNN automation tools to realize diverse FPGA-based GNN solutions efficiently; 2) the DNN-to-FPGA automation frameworks generate the hardware of dataflows and memory hierarchy that are only optimized for DNNs, not the best choice for GNNs; and 3) Compared to DNNs which have deterministic execution graph independent of input, a GNN accelerator generation framework must be aware of input-dependent computing patterns, and it must be able to deal with resource allocation issue brought by the diversity of input graph types.

In this work, we propose DeepBurning-GL, an end-to-end automation framework for generating an optimized FPGA-based accelerator for GNN applications. DeepBurning-GL simplifies the procedure of transforming GNN-based applications described with the GNN frameworks such as DGL into FPGA implementations. The main contributions of this paper are as follows:

(1) **A GNN-to-FPGA automation framework.** Unlike prior works focused on DNNs, DeepBurning-GL concentrates on the hardware generation of the GNN accelerator from an application perspective. DeepBurning-GL analyzes the common bottleneck stage of the popular GNN-based applications and constructs three pre-defined hardware templates to generate high-quality RTL codes under the user constraints.

(2) **Automatic generation of memory and caching policy.** By analyzing the data access patterns for different graph types on different applications, DeepBurning-GL constructs memory templates and provides a hybrid degree aware cache replacement policy, to fully explore the data reuse in GNN algorithms.

(3) **Automatic resource allocation and design parameter exploration.** To meet the given performance and resource constraints, DeepBurning-GL provides a performance analyzer and a resource allocation mechanism to automatically adjust hardware design parameters provided as knobs of the hardware templates to optimize the hardware performance before deployment, delivering efficient and instant hardware customization.

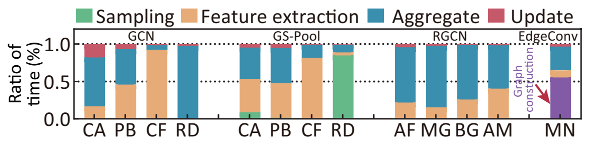


Fig. 2. Execution time of GNN models on Intel Xeon CPU.

(4) **Framework compatible APIs** DeepBurning-GL provides friendly APIs that are compatible with state-of-the-art graph learning frameworks (e.g., DGL [20] and PyG [2]) ease the deployment of GNN applications. It is proved in our experiments that the DeepBurning-GL generated FPGA solutions surpass the baseline of the CPU, GPU solutions, and also the specialized processor in energy-efficiency.

2 BACKGROUND&MOTIVATION

2.1 Primer on GNNs

A typical GNN model example is shown in Fig. 1. Its input graph varies in different applications and can be an isomorphic graph, heterogeneous graph, and point cloud. The graph can be described as $\mathcal{G} = (V, E)$ where V represents the set of vertices in the graph and E represents the set of edges in the graph. The feature of a vertex v in V and the feature of an edge $e_{u,v}$ in E can be represented with $\mathbf{x}_v \in \mathbb{R}^F$ and $\mathbf{y}_{(u,v)} \in \mathbb{R}^D$ respectively, where F and D stand for the vertex feature dimension and edge feature dimension respectively. The model may include multiple dependent layers as formulated in the top of Fig. 1 where l refers to the l-th layer of the GNN model. Each layer usually consists of four processing stages including sampling, feature extraction, aggregate, and update. Sampling is optional and is usually invoked to pre-build a graph from raw data like point cloud or extract a summarized graph from the original large graph. Feature extraction represented with ϕ in the formula is used to condense the vertex feature with processing functions like MLPs (Multi-Layer Perceptrons). Aggregate represented with $\square$ in the formula is utilized to aggregate features of a vertex's incoming neighbors in the graph and it can be conducted with processing functions like sum, mean, or max. Update represented with γ in the formula is used to perform the non-linear transformations such as activation function, GRU, and MLP. Note that $\mathcal{W}^{feat}$ and $\mathcal{W}^{update}$ are the learnable weights of the feature extraction and the update processing stages respectively. This GNN computing paradigm is representative and covers many different GNN-based applications such as DiffPool, graph attention network, and graph LSTM. Because of the distinct GNN models and various types of graphs, it is difficult to develop a dedicated ASIC accelerator with both higher performance and energy efficiency.

2.2 Related work

GNN accelerators. The increasing deployment of domain-specific accelerator design for neural network [1, 18] and graph [4, 8] motivates the development of customized GNN accelerators on both FPGAs and ASICs recently. The authors in [26] abstracted the execution flow of GCN into the aggregation stage and the combination stage and then developed an accelerator called HyGCN based on the processing stages. They leveraged both the SIMD and the systolic arrays to deal with the *coexistence problem* of GNNs. In order to support more GNN models, Liang, etc. [7] proposed a fine-grained GNN processing model and developed a corresponding GNN accelerator EnGN. They had a 2D computing array combined with

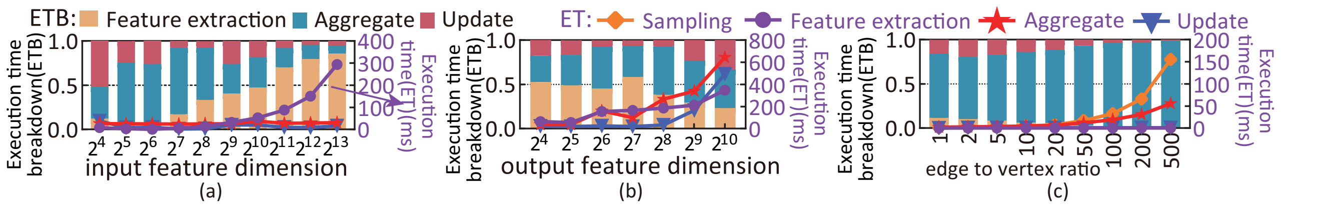


Fig. 3. The factors that affect the performance of GNNs. (a) Varied input feature dimension. (b) Varied output feature dimension. (c) Varied edge to vertex ratio.

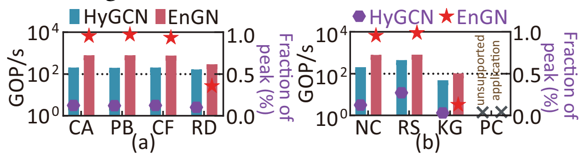


Fig. 4. Inefficiency of general purpose GCN processors. (a) The performance of GCN models on different scale datasets. (b) The performance of GNN models for different applications, where they fail to support point cloud processing.

ring-reduce interconnection to facilitate the GNN computing and investigated intensive on-chip memory hierarchy optimizations to address the irregular memory accesses in GNNs. GraphACT [29] is a GNN accelerator targeting at FPGAs. Unlike HyGCN and EnGN, it accelerates both the GCN training and inference on heterogeneous systems. Nevertheless, none of these works address the customization problem required to handle various GNN algorithms and graphs for distinct design goals in terms of performance, throughput, and energy efficiency.

Automatic accelerator generation. Accelerator design and customization automation has been very popular for various applications especially deep neural networks [9–11, 19, 30, 31]. For example, DnnWeaver [17] is proposed to automatically generate synthesizable neural network accelerators using hand-optimized RTL templates. DeepBurning [23] is also a novel design framework proposed to lower the design entry of the NN-based accelerator with pre-built RTL templates. [3, 12] attempted to leverage HLS-RTL templates to map DNNs onto FPGAs automatically. These prior DNN design frameworks demonstrate the great advantages of customizing accelerators for domain specific applications in terms of design productivity, performance, and energy efficiency. Nevertheless, the dramatic difference between DNN and GNN on both the data flows and memory access patterns hinders the use of previous frameworks for the GNN accelerator design directly. Thereby, it remains highly demanded to exploit automatic design tools for the various GNN applications on FPGAs.

2.3 Motivation

To gain insight into the computing characteristics of GNNs, we analyze four representative GNN models which cover the major areas of GNN applications including node classification (NC), recommendation system (RS), knowledge graph (KG), and point cloud (PC). The GNN models utilized in the four applications are GCN, GraphSAGE-Pool (GS-Pool), RGCN, and EdgeConv respectively. The datasets used for the four GNN models are selected from Table 3. Since some of the GNN models are specific to certain datasets, we can not have all the GNN models executed on all the datasets. GCN and GS-Pool is executed on datasets of CA, PB, CF, and RD. R-GCN is used in KG open datasets including AF, MG, BG, and AM while EdgeConv is only executed on ModelNet10 (MN) dataset.

We leverage a state-of-the-art GNN software framework, DGL, to deploy the models on a server equipped with Xeon 6151 processor and 696GB DRAM. The execution time breakdown of the four GNN models on corresponding datasets is shown in Fig. 2. According to the experiments, it can be observed that the execution time of the different processing stages varies across different applications. More specifically, it is mainly affected by both the input graphs and the GNN algorithms. For instance, the aggregation stage dominates the GCN execution time on CA and RD while the feature extraction stage takes up the majority of the GCN execution time on PB and CF. It demonstrates that the two different GNN algorithms lead to distinct execution time distribution even on the same input graphs. The execution time distribution varies even more dramatically when both the GNN algorithms and input graphs are different.

To further analyze the influence of the input graphs on the GNN execution, we take GCN as an example and compare the GCN execution when the input graph varies in input feature dimension, output feature dimension, and edge to vertex ratio respectively. The experiment is shown in Fig. 3. It can be observed that the execution time of the feature extraction increases rapidly with larger input feature dimension as presented in Fig. 3 (a). In contrast, the other processing stages are much less sensitive to the input feature dimension. Unlike the input feature dimension, the output feature dimension induces larger execution time on all the processing stages except sampling as shown in Fig. 3 (b). Similarly, the edge to vertex ratio also shows unique influence on the different processing stages of GCNs according to Fig. 3 (c). The experiment in Fig. 3 reveals that the execution of even the same GNN algorithm can be greatly affected by the input graph structure in terms of the feature dimension and the sparsity of the graph.

While the above analysis is conducted on CPUs, we further investigate the influence of GNN algorithms and input graphs on the GNN execution on state-of-the-art GNN accelerators including HyGCN and EnGN. As GNN accelerators mostly have different processing stages pipelined, it is difficult to obtain the execution time distribution and determine the influence of the different factors directly. In this case, we mainly measure the computing power which reveals the computing efficiency of the GNN algorithms because the peak computing power is determined for the GNN accelerators. The experiment results in Fig. 4 (a) shows the measured computing power of the GNN accelerators when GCN executes on different input graphs while Fig. 4 (b) shows the measured computing power of the GNN accelerators when different GNN algorithms execute on distinct input graphs. It can be observed that the measured computing power varies considerably. EnGN achieves less than 50% of the peak computing power on RD though it achieves near optimal performance on CA, PB, and CF. The computing efficiency exhibits even larger variations when both the GNN algorithms and input

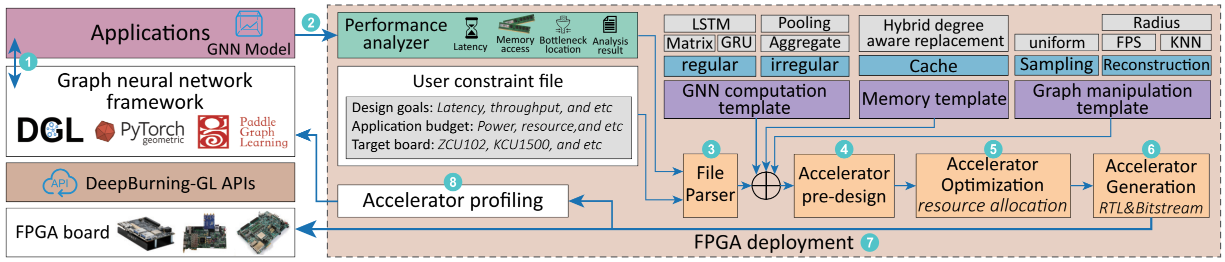


Fig. 5. DeepBurning-GL design framework.

Table 1: DeepBurning-GL (DB-GL) APIs.

	API functions	Description
Data	DB-GL_storage_transfer( )	Transfer data from storage to HOST/FPGA DRAM
	DB-GL_SetInput( )	Set the input data of the accelerator to a specific address
APIs	DB-GL_GetOutput( )	Get the result of the accelerator from a specific address
	DB-GL_Open( )	Open the generated accelerator
Control	DB-GL_Close( )	Close the generated accelerator
	DB-GL_Create_Task( )	Offload task to the generated accelerator
APIs	DB-GL_Run_Task( )	Run offload task
	DB-GL_Destory_Task( )	Close offload task
	DB-GL_Task_Monitor( )	Profiling status of accelerator

graphs are different. Clear variations can also be observed in the experiments. In a nutshell, the experiments demonstrate that it remains a great challenge to achieve optimized computing efficiency when deploying GNN applications with dramatic variations on a fixed GNN accelerator.

3 DEEPBURNING-GL DESIGN FRAMEWORK

In order to achieve high computing efficiency for various GNN applications, we take advantage of the reconfiguration capability of FPGAs to customize specific GNN accelerators for the different GNN applications. Particularly, we propose DeepBurning-GL, an automatic GNN acceleration framework targeting at specific GNN applications and design goals to avoid the time-consuming handcrafted customization process. The framework is shown in Fig. 5. It roughly contains eight processing stages as marked in Fig. 5 and will be illustrated as follows.

Step 1: Users start with the GNN model design and then train the GNN model with labeled application data using the GNN frameworks such as DGL.

Step 2: When the GNN model is determined, we have a performance analyzer to inspect the different processing stages of the GNN model and identify the performance bottleneck from different angles like memory access efficiency and execution time with profiling on top of a general DeepBurning-GL simulator. Eventually, this step will output a series of analysis results including GNN operators, GNN parameters, input graph statistic information, performance bottleneck, memory access preference, and so on.

Step 3: File parser extracts the key information from both the GNN model analysis results and the user constraints, with which it determines the required hardware templates that are required to execute the GNN model. The templates are essentially selected from the template library pre-built in DeepBurning-GL. While the template library includes three categories, i.e., GNN computation templates, memory templates, and graph manipulation templates. **GNN computation templates** can be further divided into two different types. One of them usually consists of an array of processing elements and is used for regular operations like matrix computing, GRU, and LSTM. The other type mainly targets at irregular operations like pooling and aggregation. **Memory templates** can either be distributed on-chip buffer, cache, or degree-aware hierarchy on-chip buffer such that both regular memory accesses and irregular memory accesses can be supported efficiently. **Graph manipulation templates** include a series of different graph sampling and reconstruction modules which may be required by different GNN algorithms.

Step 4: When the selected hardware templates are determined, DeepBurning-GL framework fuses these templates with corresponding data flow into a unified GNN accelerator. For instance, we can map the feature extraction operation to the regular computing array with output stationary data flow. Similarly, we may equip the accelerator with different memory access modules based on the performance analysis results. For example, we can apply conventional on-chip memory structure to buffer the edges or output vertex IDs. In contrast, we may deploy a degree aware on-chip memory hierarchy for the vertex property buffering such that high degree vertex properties that will be frequently referenced can be reused. In addition, note that all the hardware modules utilized in the pre-designed accelerator adopt the default configurations to ensure the basic functionality at this stage.

Step 5: To fulfill both the performance requirements and the power or resource constraints, we need to further tune the design parameters including the generated GNN accelerator architectural parameters and the dataflow options based on the input graph structure and the GNN algorithm. Basically, we build a series of regression models for the performance, power, and resource consumption based on experiments mentioned in Section 2.3. On top of the models, we can conduct the design space exploration (DSE) with DeepBurning-GL which has multiple DSE methods integrated.

Step 6: When GNN accelerator architectural parameters as well as the dataflow options are determined after the Step 5, DeepBurning-GL will have the generated accelerator wrapped up as an IP with standard interfaces like AXI such that the accelerator can be integrated and utilized in an FPGA board conveniently.

Step 7: This step will have the wrapped GNN accelerator implemented on target FPGA platforms. On top of the hardware implementation, DeepBurning-GL also generates platform-specific APIs utilized to bridge the gap between the high-level Python-based GNN processing framework and the low-level GNN accelerator. The general definition of the DeepBuring-GL APIs are defined in Table 1. It can be divided into two groups. One group of the APIs mainly target at data plane operations such as data transfer, input setup, and output collection. The other group of APIs target at control plane operations and they can be used to invoke, manipulate, and monitor the GNN accelerator.

Step 8 This step is essentially utilized to evaluate the GNN models deployed on the customized GNN accelerator generated by

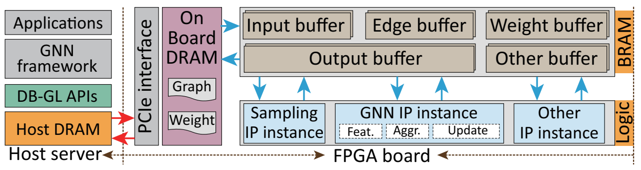


Fig. 6. Accelerator architecture.

DeepBurning-GL. With the evaluation, DeepBurning-GL can determine if the design goals are met. If the design goal is achieved, the automatic GNN customization is done. Otherwise, DeepBurning-GL will repeat the customization processing steps until the design goal is achieved.

With the proposed DeepBurning-GL, an end-to-end customized GNN acceleration solution can be achieved on a target FPGA board. It enables the designers to take advantage of the FPGAs for GNN acceleration without touching the underlying low-level details.

4 GNN ACCELERATOR ARCHITECTURE

In order to generate a GNN accelerator based on the optimized design parameters, we need both a GNN accelerator template and the accelerator component templates to instantiate the design. The GNN accelerator template defines the general architecture and the connection of the components while the component template covers the different processing operations or architectures. These templates will be detailed in the rest of this section.

4.1 Overall GNN accelerator template

The overall GNN accelerator template is shown in Fig. 6. It assumes the input graphs, GNN weights stored in the on-board DRAM. In order to fetch the data from DRAM efficiently for the GNN processing, it has a series of on-chip memory blocks implemented to buffer the IO data as well as the intermediate data. These memory blocks include input buffer, edge buffer, output buffer, weight buffer, and so on. They must be customized appropriately to meet the requirements of the GNN models. When the data are ready in the buffers, GNN models that can be split into four different processing stages including sampling, feature extraction, aggregation, and update will be executed according to the order of the processing stages. They operate data stored in the on-chip buffers, so they are connected with the on-chip buffers directly. As sampling operates on graphs and differs greatly across the different GNN algorithms, it is classified as sampling IP instance in Fig. 6. The rest of the three processing stages can usually be mapped to the same computing infrastructures and are classified as GNN computing IPs as shown in Fig. 6. In addition, there are also some minor operations like the various activation functions. They can be implemented with customized circuits like vector processing unit and are denoted as other IP instances in Fig. 6. All the components can be configured and replaced to fit the various GNN model acceleration.

4.2 GNN computation template

As shown in Fig. 5, DeepBurning-GL contains two categories of GNN computation templates. One of the templates is used for regular computing which is mostly the feature extraction and update in GNNs while the other template is designed for irregular computing which is mostly graph-based aggregation. While the major regular computing in feature extraction and update is essentially matrix-matrix multiplication, LSTM, and GRU, we adopt a systolic array or a dot-production array as the template for these regular processing

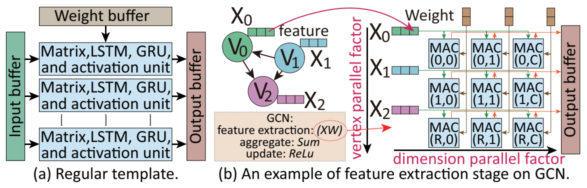


Fig. 7. Regular computing template and data mapping.

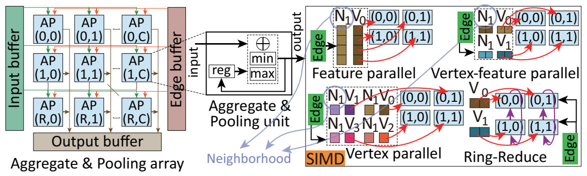


Fig. 8. Irregular computing template and data mapping.

as shown in Fig. 7 (a). The mapping of the feature extraction to a typical systolic array is illustrated in Fig. 7 (b). With both weights and input features streamed into the array, the computing array can be efficiently utilized.

The irregular computing template is presented in the left of Fig. 8 and it contains an array of homogeneous processing units. Since the processing units mainly target at graph-based aggregation, they have various aggregation operations such as Maximization, Minimization, accumulation integrated. Unlike the regular computing array which usually adopts a fixed mesh topology, the irregular computing array configurable to meet the various requirements of the graph aggregation. Currently, the irregular computing array can be configured to be either an array of SIMD units or a ring-reduce topology as displayed in the right of Fig. 8. For the SIMD array, each SIMD unit exploits the parallelism in the vertex property dimension and is utilized to perform the aggregation for a destination vertex sequentially. Different SIMD units conduct the aggregation independently. For the ring-reduce architecture, each row of the computing array is responsible for the aggregation of a destination vertex. Different rows can share the source vertex property across the ring without reloading data from the on-chip buffers, which improves the data reuse and thus the computing efficiency. The major disadvantage of the ring-reduce architecture is the larger hardware resource consumption caused by the increased interconnections.

In addition, note that the irregular computing templates are designed for the aggregation of sparse graphs. Although they can also be utilized for dense graph aggregation, they typically incur considerable overhead when parsing the dense graphs with edges. In fact, the aggregation over a dense graph can be considered as matrix-matrix multiplication, which is more efficient to deploy on regular computing templates. With this observation, DeepBurning-GL will evaluate the sparsity of the input graphs and choose the optimized architectural design option for the resulting GNN accelerator.

4.3 Memory template

Memory template essentially refers to the on-chip memory blocks utilized to buffer the data for GNN computing. While there are both regular memory accesses and irregular memory accesses, thus we have corresponding memory templates provided. For the regular memory accesses, we utilize normal on-chip buffers directly and they can be configured for arbitrary banks to meet the various on-chip data access requirements. For the irregular memory accesses especially the vertex property retrieval based on the vertex neighbor indices, we presented a hybrid degree-aware on-chip cache structure to improve the data reuse. The basic idea is to assign the high-degree vertex entries with higher priorities such that they will not be replaced by low-degree vertex entries frequently. Since the high-degree vertex properties are more likely to be reused, this approach improves the on-chip data reuse. Nevertheless, the degree-aware requires additional degree comparison, it needs to be adjusted to balance the overhead and the potential benefits. This design option is also taken into consideration in DeepBurning-GL and will be optimized during the parameter tuning step.

4.4 Graph manipulation template

Graph manipulation template is used to either sample from a large graph or construct graph from raw data like point cloud. The processing varies greatly across the different GNN applications. For instance, GCN may have graph sampling to select a subset of edges from the edge list of the original graph. Point cloud application may have different algorithms including farthest point sampling (FPS), K-nearest neighbors (KNN), and radius to construct a graph from raw point data in different scenarios. In addition, it is also difficult to have a unified accelerator to support all the different processing. Thereby, we have a series of different templates to fulfill the requirements of the different GNNs. These accelerators can also be configured to compromise between the performance and the resource consumption, and they can be tuned by DeepBurning-DL as well.

5 GNN ACCELERATOR CUSTOMIZATION

GNN accelerator customization that optimizes the GNN accelerator architectural parameters for the design goal under multiple users' constraints is the key processing step of DeepBurning-GL. Essentially, this step is to explore the design space and search for the optimal design parameters. As discussed in the previous section, the template-based GNN accelerator includes a number of design parameters such as the computing array sizes, on-chip buffer sizes and structures, dataflow options, and other sub component parameters, so the design space is rather large. On the other hand, GNNs mostly operate on large graphs and it is expensive to evaluate the design goal metrics like performance with either simulation or hardware implementation. To address this problem, we propose a simulated annealing algorithm in combination with a model-based design space pruning algorithm for the customization. The model-based pruning is utilized to reduce the design space rapidly while the simulated annealing is adopted for the complex DSE.

Algorithm 1 illustrates the proposed DSE algorithm and it includes two stages, i.e., the design space pruning (line1-line16) and the parameters fine tuning (line18). While the utilized fine tuning approach is a well-known simulated annealing algorithm, we will not dwell on it. In this work, we will focus on the design space pruning method. The basic idea is to evaluate the major design parameters such as the regular computing array sizes and the on-chip buffer sizes with analytical models, with which we can prune the inappropriate design options rapidly. We take the feature extraction processing performance model as an example. Suppose we have the feature extraction processing mapped to a systolic array with $R \times C$ processing elements (PEs) and the array works at $freq$ Hz. Assume the input graph has N vertices, the input feature dimension, and the output feature dimension are F and H respectively. The processing time of the feature extraction which is essentially a matrix-matrix multiplication can be roughly estimated with Eq. 1. Similarly, we can also build performance models for other regular GNN processing and hardware resource consumption models for the regular hardware modules. For irregular processing, it is difficult to build accurate models, but we can still obtain the lower bound with a roofline model.

$$processing\_time = \left\lceil \frac{N}{R} \right\rceil \times \left\lceil \frac{H}{C} \right\rceil \times F \times \frac{1}{freq} \tag{1}$$

The detailed parameter pruning is presented in Algorithm 1 from Line 1 to Line 16. It starts with the performance evaluation of the different GNN processing stages based on the computing array sizes according to Eq. 1. On top of the model, we can identify the performance bottleneck and alleviate the bottleneck by adding more computing resources to the bottleneck processing stage. Meanwhile, we observe that the on-chip buffers determine the input computing tiling, which eventually affects the data reuse and potential peak performance of the processing. Thus, we need to allocate more on-chip buffers to the bottleneck processing stages as well. In addition, larger on-chip buffer sizes and computing array sizes further require more memory bandwidth which is usually fixed and limits the increase of the computing array sizes and on-chip buffer sizes. Thereby, it is also considered during parameter customization.

When these parameters, i.e., $\boldsymbol{P_m}$ that are closely relevant to the performance are determined, we will invoke the simulated annealing algorithm to determine the rest of the design parameters, i.e., $\boldsymbol{F_m}$ like cache sizes that are difficult to characterize with analytical models. As many of the design parameters are already determined, the simulated annealing that searches on a much smaller design space can be much faster.

Algorithm 1 GNN Accelerator Parameter Customization

```
Input: Pruning parameters $\boldsymbol{P_m}$, Fine-tuned parameters $\boldsymbol{F_m}$, Constraint parameters:
    Total compute resource $CR$, Total BRAM resource $BR$, Total bandwidth $BW$.
Output: Parameter $\boldsymbol{P = \{P_m, F_m\}}$ for optimized performance.
 1: Initialize $\boldsymbol{P_m}$ and $\boldsymbol{F_m}$ based on constraint parameters.
 2: while $\sum_{s=0}^{s} CR \le CR$ and $\sum_{s=0}^{s} BW_s \le BW$ do
 3:     Select stage $p$ with maximum execution latency in pipeline data path.
 4:     if $\sum_{s=0}^{s} CR_s + \Delta CR_p \le CR$ then
 5:         if $\sum_{s=0}^{s} BW_s + \Delta BW_p \ge BW$ then
 6:             Select stage $q$ with maximum bandwidth consuming in data path.
 7:             if $p \ne q$ And $\sum_{s=0}^{s} BR_s + \Delta BR_q \le BR$ then
 8:                 $BR_q \leftarrow BR_q + \Delta BR_q$      ▹ adjust $\boldsymbol{P_m}$ and $\boldsymbol{F_m}$ related to BRAM
 9:             else break
10:             end if
11:         else
12:             $CR_p \leftarrow CR_p + \Delta CR_p$      ▹ adjust $\boldsymbol{P_m}$ and $\boldsymbol{F_m}$ related to DSP
13:         end if
14:     else break
15:     end if
16: end while
17: Fixed parameters $\boldsymbol{P_m}$
18: $P \leftarrow$ Simulated_Annealing_Algorithm($\boldsymbol{P_m}, \boldsymbol{F_m}$)      ▹ Fine tune $\boldsymbol{F_m}$
```

6 EVALUATION

6.1 Experimental setup

Target Platforms & Baselines In order to evaluate the capability and scalability of the proposed DeepBurning-GL framework, we utilize DeepBurning-GL to customize GNN accelerators for various GNN models targeting at three classical FPGA platforms including ZC706, KCU1500, and Alveo U50, respectively. The generated GNN accelerators adopt 16-bit fixed point and are implemented with

Table 2: Baseline architecture.

	CPU	GPU	HyGCN	ZC706	KCU1500	Alveo U50
Compute unit	3.0GHz @ 65 cores	1.25GHz @ 5120 cores	4608 DSPs	900 DSPs	5520 DSPs	5952 DSPs
'On-chip memory	42.75MB	34MB	24.125MB	19.2Mb	75.9Mb	227.3Mb
Off-chip memory	255.9GB/s	~900GB/s	316GB/s	12.8GB/s	76.8GB/s	316GB/s

Table 3: GNN models and datasets.

Applications	Model	Graph	#Vertices	#Edges	#Feature/ #Relation	Label
Node Classification	GCN	Cora (CA)	2708	10556	1433	**7**
		Pubmed (PB)	19717	88651	500	**3**
Recommender system	GS-Pool	Cora-Full (CF)	19793	126842	8710	67
		Reddit (RD)	232965	114.6M	602	41
Knowledge graph	R-GCN	AIFB (AF)	8285	29043	91	4
		MUTAG (MG)	23644	192098	47	2
		BGS (BG)	333845	2166243	207	2
		AM (AM)	1666764	13643406	267	11
Point cloud	EdgeConv	ModelNet10 (MN)	1024	20480	6	10
		ModelNet40 (M4)	2048	51200	6	40

Vivado 2019.2. The different implementations work at 100 MHz, 200 MHz, and 200MHz, respectively. The accelerators seated with an Intel Xeon Silver 4216 processor can be utilized by the graph learning framework DGL with the APIs provided by DeepBurning-GL. Then we compared the generated accelerators with three typical baseline systems including a CPU platform equipped with Intel Xeon (Skylake) 6151 processor and 696GB DRAM, a GPU platform equipped with NVIDIA Tesla V100 and 32GB HBM2, and a state-of-the-art GCN accelerator, HyGCN [26]. HyGCN is implemented on Alveo U50 and it can run at 200MHz, but it can not be implemented on the other two FPGA platforms due to the large requirements of DSPs and memory bandwidth. More detailed configurations of the different platforms can be found in Table 2.

GNN models & Datasets. Table 3 shows the GNN models and datasets used for the benchmarking of GNN acceleration systems. Particularly, the representative GNN models cover various applications including node classification, recommender system, knowledge graph, and point cloud analysis. As the GNN models and datasets are closely related, different GNN models only run a sub set of the datasets as detailed in Table 3. The abbreviation of the datasets can also be found in Table 3. In addition, GCN, GS-Pool, and R-GCN have two layers and the hidden output dimension is 16. Meanwhile, GCN adopts a mechanism similar to GS-Pool to perform inductive inference. EdgeConv has only the first layer executed as the following layers have the same structures. The hidden output dimension of EdgeConv is 64. We uniformly sample 1,024 and 2,048 points from MN and M4 datasets and set the number of nearest neighbors to be 20 and 25 respectively.

6.2 Experimental results

Accelerator configuration on GCN model DeepBurning-GL is able to customize the GNN accelerator for a specific GNN algorithm and input graph under specific resource constraints. In this experiment, we leverage DeepBurning-GL to customize the design parameters for higher performance. We just put the major parameters of the accelerators customized for GCN in Table 4. It can be observed that the optimized design parameters under different scenarios vary considerably. For instance, the customized regular computing array for PB on ZC706 is one fourth of that on Alveo U50. Another significant difference between the accelerator configurations is the input buffer sizes. It can be seen that Alveo U50 with the largest on-chip memory typically allocates more on-chip memory blocks to the input buffers. On the other hand, the input buffer sizes for the different datasets are also distinct. Basically,

Table 4: Accelerator configuration on GCN model.

		Computation unit ($R \times \underline{C}$)		On-chip memory (KB)					
		Feature extraction	Aggregate	Input	Output	Weight	Edge	Intermediate	Dataflow
ZC706	CA	$32 \times \underline{\mathbf{7}}$	8×7	256	40	64	64	128	$FE \rightarrow A$
	PB	$63 \times \underline{\mathbf{3}}$	18×3	512	128	32	512	704	$FE \rightarrow A$
	CF	4×67	1×67	640	256	384	320	704	$A \rightarrow FE$
	RD	1×41	3×41	768	512	32	768	256	$A \rightarrow FE$
KCU1500	CA	64×7	16×7	3072	40	64	64	128	$FE \rightarrow A$
	PB	128×3	32×3	4096	128	32	512	704	$FE \rightarrow A$
	CF	8×67	2×67	5120	2048	384	320	704	$A \rightarrow FE$
	RD	1×41	32×41	5120	2048	32	1024	1024	$A \rightarrow FE$
Alveo U50	CA	128×7	32×7	8192	40	64	64	128	$FE \rightarrow A$
	PB	256×3	64×3	12288	128	32	512	704	$FE \rightarrow A$
	CF	12×67	3×67	16384	2688	384	320	704	$A \rightarrow FE$
	RD	3×41	84×41	16384	2048	32	2048	4096	$A \rightarrow FE$

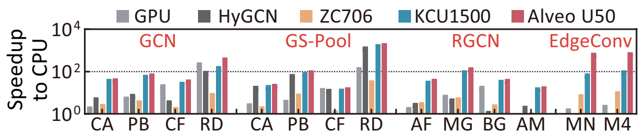


Fig. 9. Speedup compared to CPU.

large datasets like CF and RD require larger input buffers. Nevertheless, we notice that the weight buffer sizes remain the same for different configurations in spite of the different FPGA boards and datasets. This is because the number of weights in GNN models is usually small and the accesses to the weights are critical to the GNN accelerator performance. Thereby, GNN weights are usually fully buffered unless it can not be fitted to the on-chip memory. The configurations in the table also confirm that a fixed GNN accelerator can not fulfill the diverse GNN computing requirements and intensive customization is required and can be efficient.

Performance comparison. DeepBurning-GL generates specific accelerators according to the characteristics of an input graph and the architecture of a GNN model. Then we compared the DeepBurning-GL generated accelerators to the three state-of-the-art solutions on CPU, GPU, and general GNN accelerator (HyGCN). As shown in Fig. 9, all the performance is normalized to that on CPU. It can be observed that the average performance speedup of the customized accelerators on ZC706, KCU1500, and Alveo U50 over the CPU is 7.43X, 199.98X, and 346.98X, respectively. The accelerators on ZC706 have relatively lower performance mainly because of the much lower off-chip memory bandwidth. As a result, the DSPs of ZC706 are under-utilized due to the bandwidth bound. In contrast, the customized accelerators on Alveo U50 with the highest memory bandwidth achieve the highest performance speedup. In addition, we notice that the performance of the customized accelerators on KCU1500 and Alveo U50 are 14.6X and 16.5X higher on average than that of GPU on small datasets such as CA and PB. For larger datasets such as CF and RD, the performance speedup drops to 3.7X and 4.5X. This is mainly caused by the fact that the on-chip buffer sizes limit the data reuse considerably especially for the large graphs. Basically, large graphs need to be tiled to fit the on-chip buffers in the accelerators and the tiled graph data may be repeatedly loaded during the GNN computing. Alveo U50 with the most on-chip memory allows larger on-chip buffer configurations, which improves the data reuse and is beneficial to the performance of the accelerators accordingly. Fig. 9 also shows that the customized accelerators on KCU1500 and Alveo U50 outperform HyGCN on most datasets. This is because that the customized accelerators fit better to the corresponding GNN models and achieve much higher computing resource utilization. As shown in Fig. 11 (a), the computing array utilization of HyGCN is only 26.9% for the GCN model. In

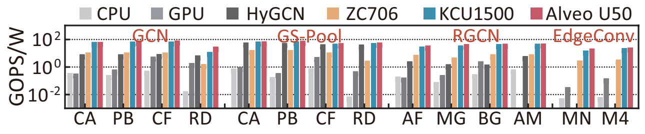


Fig. 10. Energy efficiency.

contrast, the computing array utilization of the DeepBurning-GL generated accelerators can be up to 88.4%, 94.8%, and 95.9% on ZC706, KCU1500, and Alveo U50, respectively.

In addition, HyGCN can not support EdgeConv that requires a KNN-based graph reconstruction operation, so the performance data for HyGCN on MN and M4 is left blank. According to the work in [15], KNN with a large number of random memory accesses is time-consuming and it occupies 55% and 94.5% of total execution time on CPUs and GPUs, respectively. To accelerate EdgeConv with the given resource constraints, DeepBurning-GL has a specialized KNN accelerator integrated along with the normal GNN operations and then performs the design parameter customization in a unified manner. With the customization, the generated accelerators achieve significant performance speedup on the point cloud applications.

Energy-efficiency comparison. We use *billion operations per second per Watt* (GOPS/W) as the metric to evaluate the energy-efficiency of the implementations. The power of CPU and GPU is measured using a power meter. The power of each FPGA platform is obtained from the power controller resided on the board. The energy efficiency of the different implementations is shown in Fig. 10. The average energy-efficiency of the DeepBurning-GL generated accelerators on ZC706, KCU1500, and Alveo U50 are 8.6 GOPS/W, 47.1 GOPS/W, and 53.5 GOPS/W respectively. They are 28.7X, 157.8X, and 179.4X higher than the implementations on CPUs, and are 6.4X, 35.2X, and 40.1X higher than the GPU implementations. According to the experiments, HyGCN implemented on Alveo U50 exhibits much lower energy efficiency compared to the customized accelerators implemented on the same FPGA board. This is mainly caused by the lower computing resource utilization as shown in Fig. 11 (a).

FPGA implementations. Table 5 shows the FPGA resource utilization of the DeepBurning-GL generated accelerators for each GNN model and each specific graph dataset. It can be found that the limited on-chip memory resource of ZC706 can only satisfy the requirements of small datasets like CA and PB. When the graph data gets larger, the on-chip memory becomes the bottleneck and hinders the use of larger computing array. In addition, the relatively low memory bandwidth (12.8GB/s) can also be part of the reason that fails to make use of the computing resources in ZC706. Unlike ZC706, the customized accelerators on KCU1500 and Alveo U50 have much larger on-chip memory and off-memory bandwidth according to Table 2. With the alleviated resource constraints, more computing resources can be utilized as listed in Table 5 and higher performance can be expected eventually.

Cache optimization. As discussed in subsection 4.3, the high degree vertices are more frequently accessed in GNNs. While the vertex properties used in the complex graphs can be large, equally buffering all the vertex properties can be a waste to the limited on-chip memory. Thus, we utilize a degree-aware cache structure to prioritize the buffering of the high-degree vertex properties, which can reduce the overall off-memory accesses. It is essentially built on top of a baseline 4-way set-associative cache structure with degree-aware comparison added to the original LRU cache replacement. The cache sizes of the customized GNN accelerators on ZC706, KCU1500, and Alveo U50 are set to be 64KB, 96KB, and 128KB, respectively. The normalized performance speedup over the accelerator with baseline cache architecture (Acc_base) on GCN model is presented in Fig. 11 (b). It can be observed that the degree-aware cache that reduces the amount of memory accesses improves the accelerator performance significantly. The performance speedup is relatively higher given larger cache sizes which enlarge the memory access savings.

Table 5: Resource utilization of the three FPGA platforms.

		ZC706				KCU1500				Alveo U50				
		LUT	FF	BRAM	DSP	LUT	FF	BRAM	DSP	LUT	FF	BRAM	URAM	DSP
Total		218K	437K	545	900	663K	1326K	2160	5520	872K	1743K	1344	640	5952
GCN	CA	52%	20%	23%	90%	38%	35%	35%	85%	77%	29%	29%	29%	99%
	PB	47%	19%	77%	86%	42%	42%	56%	84%	93%	36%	47%	46%	98%
	CF	54%	23%	94%	96%	45%	44%	88%	88%	90%	34%	71%	70%	98%
	RD	16%	7%	95%	31%	16%	16%	95%	**35%**	87%	34%	85%	84%	**99%**
GS-Pool	CA	53%	22%	82%	99%	36%	36%	76%	79%	88%	31%	43%	42%	91%
	PB	50%	19%	86%	93%	37%	37%	82%	83%	76%	30%	49%	48%	83%
	CF	53%	20%	94%	96%	38%	39%	94%	85%	90%	35%	83%	83%	100%
	RD	22%	9%	99%	38%	41%	21%	97%	**43%**	90%	34%	87%	87%	**95%**
RGCN	AF	52%	20%	37%	95%	34%	33%	9%	76%	81%	33%	5%	5%	89%
	MG	49%	19%	81%	86%	38%	37%	29%	84%	82%	30%	12%	12%	88%
	BG	52%	21%	91%	94%	42%	43%	71%	93%	91%	34%	19%	19%	95%
	AM	53%	22%	99%	99%	40%	41%	92%	89%	79%	30%	85%	85%	88%
Edge Conv	MN	90%	17%	39%	97%	51%	20%	9%	63%	67%	34%	5%	5%	88%
	M4	95%	19%	70%	98%	59%	27%	21%	81%	77%	37%	12%	11%	99%

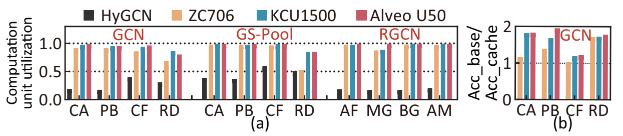


Fig. 11. (a) Computation unit utilization. (b) Normalized performance speedup of accelerators with degree-aware cache to that with the baseline cache on GCN model.

7 CONCLUSION

DeepBurning-GL is proposed to simplify the design flow of GNN accelerator for GNN-based applications. It is compatible with state-of-the-art graph learning framework such as DGL, PyG, and facilitates the developers to generate application-targeted GNN accelerator. DeepBurning-GL employ a GNN performance analyzer to locate the performance bottleneck and provides the pre-built computation and memory templates to guide the generation of hardware architecture. Meanwhile, it further optimizes the accelerator performance by adjusting the computation and memory resources allocation according to the property of graph, GNN architecture, and the resource constraints imposed by the target FPGA platform or the user specification. In evaluation, the DeepBurning-GL generated accelerators deliver superior performance and energy efficiency when compared to CPU, GPU, and state-of-the-art GCN accelerator.

ACKNOWLEDGMENTS

This work was supported in part by the National Key Research and Development Program of China under Grant 2018AAA0102700, the National Natural Science Foundation of China under Grant 61874124, Grant 61876173, Grant 61532017, Grant 61772300, Grant 61902375, Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDC05030201, and YESS hip program No.YESS2016qnrc001. Shengwen Liang and Cheng Liu made equal contributions. Corresponding authors are Ying Wang and Huawei Li.

REFERENCES

[1] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. SIGPLAN Not. 49, 4 (Feb. 2014), 269–284. https://doi.org/10.1145/2644865.2541967

[2] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. arXiv:1903.02428 [cs.LG]

[3] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). 152–159. https://doi.org/10.1109/FCCM.2017.25

[4] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In The 49th Annual IEEE/ACM International Symposium on Microarchitecture (Taipei, Taiwan) (MICRO-49). IEEE Press, Article 56, 13 pages.

[5] 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 Proceedings of the 56th Annual Design Automation Conference 2019 (Las Vegas, NV, USA) (DAC '19). Association for Computing Machinery, New York, NY, USA, Article 206, 6 pages. https://doi.org/10.1145/3316781.3317829

[6] Xiao Huang, Qingquan Song, Yuening Li, and Xia Hu. 2019. Graph Recurrent Networks With Attributed Random Walks. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining (Anchorage, AK, USA) (KDD '19). Association for Computing Machinery, New York, NY, USA, 732–740. https://doi.org/10.1145/3292500.3330941

[7] S. Liang, Y. Wang, C. Liu, L. He, H. LI, D. Xu, and X. Li. 2020. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. IEEE Trans. Comput. (2020), 1–1. https://doi.org/10.1109/TC.2020.3014632

[8] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. 2019. Cognitive SSD: A Deep Learning Engine for in-Storage Data Retrieval. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (Renton, WA, USA) (USENIX ATC '19). USENIX Association, USA, 395–410.

[9] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. 2015. Automatic Nested Loop Acceleration on FPGAs Using Soft CGRA Overlay. CoRR abs/1509.00042 (2015). arXiv:1509.00042 http://arxiv.org/abs/1509.00042

[10] C. Liu, H. Ng, and H. K. So. 2015. QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In 2015 International Conference on Field Programmable Technology (FPT). 56–63.

[11] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In 2017 27th International Conference on Field Programmable Logic and Applications (FPL). 1–8.

[12] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In 2017 27th International Conference on Field Programmable Logic and Applications (FPL). 1–8. https://doi.org/10.23919/FPL.2017.8056824

[13] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming Graph Neural Networks. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (Virtual Event, China) (SIGIR '20). Association for Computing Machinery, New York, NY, USA, 719–728. https://doi.org/10.1145/3397271.3401092

[14] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. 2019. Estimating Node Importance in Knowledge Graphs Using Graph Neural Networks. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining (Anchorage, AK, USA) (KDD '19). Association for Computing Machinery, New York, NY, USA, 596–606. https://doi.org/10.1145/3292500.3330855

[15] R. Pinkham, S. Zeng, and Z. Zhang. 2020. QuickNN: Memory and Performance Optimization of k-d Tree Based Nearest Neighbor Search for 3D Point Clouds. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). 180–192.

[16] Jiezhong Qiu, Jian Tang, Hao Ma, Yuxiao Dong, Kuansan Wang, and Jie Tang. 2018. DeepInf: Social Influence Prediction with Deep Learning. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining (London, United Kingdom) (KDD '18). Association for Computing Machinery, New York, NY, USA, 2110–2119. https://doi.org/10.1145/3219819.3220077

[17] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From High-Level Deep Neural Models to FPGAs. In The 49th Annual IEEE/ACM International Symposium on Microarchitecture (Taipei, Taiwan) (MICRO-49). IEEE Press, Article 17, 12 pages.

[18] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. 2016. C-Brain: A Deep Learning Accelerator That Tames the Diversity of CNNs through Adaptive Data-Level Parallelization. In Proceedings of the 53rd Annual Design Automation Conference (Austin, Texas) (DAC '16). Association for Computing Machinery, New York, NY, USA, Article 123, 6 pages. https://doi.org/10.1145/2897937.2897995

[19] S. I. Venieris and C. Bouganis. 2016. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). 40–47.

[20] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. arXiv:1909.01315 [cs.LG]

[21] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. 2019. Heterogeneous Graph Attention Network. In The World Wide Web Conference (San Francisco, CA, USA) (WWW '19). Association for Computing Machinery, New York, NY, USA, 2022–2032. https://doi.org/10.1145/3308558.3313562

[22] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. 2019. Dynamic Graph CNN for Learning on Point Clouds. ACM Trans. Graph. 38, 5, Article 146 (Oct. 2019), 12 pages. https://doi.org/10.1145/3326362

[23] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: Automatic Generation of FPGA-Based Learning Accelerators for the Neural Network Family. In Proceedings of the 53rd Annual Design Automation Conference (Austin, Texas) (DAC '16). Association for Computing Machinery, New York, NY, USA, Article 110, 6 pages. https://doi.org/10.1145/2897937.2898003

[24] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. 2020. AutoDNNchip: An Automated DNN Chip Predictor and Builder for Both FPGAs and ASICs. In The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Seaside, CA, USA) (FPGA '20). Association for Computing Machinery, New York, NY, USA, 40–50. https://doi.org/10.1145/3373087.3375306

[25] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. 2020. Characterizing and Understanding GCNs on GPU. IEEE Computer Architecture Letters (2020), 1–1. https://doi.org/10.1109/LCA.2020.2970395

[26] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. ArXiv abs/2001.02514 (2020).

[27] Hongxia Yang. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining (Anchorage, AK, USA) (KDD '19). Association for Computing Machinery, New York, NY, USA, 3165–3166. https://doi.org/10.1145/3292500.3340404

[28] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining (London, United Kingdom) (KDD '18). Association for Computing Machinery, New York, NY, USA, 974–983. https://doi.org/10.1145/3219819.3219890

[29] Hanqing Zeng and Viktor Prasanna. 2019. GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms. ArXiv abs/2001.02498 (2019).

[30] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In Proceedings of the 35th International Conference on Computer-Aided Design (Austin, Texas) (ICCAD '16). Association for Computing Machinery, New York, NY, USA, Article 12, 8 pages. https://doi.org/10.1145/2966986.2967011

[31] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In Proceedings of the International Conference on Computer-Aided Design (San Diego, California) (ICCAD '18). Association for Computing Machinery, New York, NY, USA, Article 56, 8 pages. https://doi.org/10.1145/3240765.3240801