GCN Inference Acceleration using High-Level Synthesis

Yi Chien Lin, Bingyi Zhang, Viktor Prasanna
University of Southern California, Los Angeles, California
Email: {yichienl, bingyizh, prasanna}@usc.edu

***Abstract*—GCN (Graph Convolutional Network) has become a promising solution for many applications, such as recommendation systems, social data mining, etc. Many of these applications requires low latency GCN inference.**

In this paper, we provide a case study of a GCN inference acceleration on FPGA. We explore high-level synthesis programming model to achieve low-latency inference. First, we propose a partition-centric mapping strategy to map the execution tasks of GCN onto FPGA to exploit data reuse, which reduces external memory access overhead. Second, we provide HLS-based kernel design with improved memory performance and achieve massive data parallelism. Third, we perform design space exploration to facilitate feasible pre-placement which avoids potential Place-and-Route (PnR) failures. We evaluate our design on a state-of-the-art FPGA platform using three commonly used datasets: Reddit, Yelp and Amazon-2M. We compare our design with two state-of-the-art libraries PyTorch-Geometric (PyG) and Deep Graph Library (DGL) running on high-end CPU and GPU by evaluating their latency and energy efficiency to perform full-batch GCN inference on a two-layer Vanilla-GCN model. Compared with PyG CPU version, our design reduces the latency by $59.95\times$ and is $96.22\times$ more energy efficient on the average. Compared with DGL, our design achieves $2.9\times - 6.4\times$ speedup and is $5.87\times$ more energy efficient compared with the CPU version. Compared with the DGL GPU version, although the latency of our design is $1.67\times - 2.5\times$ that of DGL GPU, our design is $1.8\times$ more energy efficient.

I. Introduction

Graph Convolutional Network (GCN) has become popular solutions for many cloud-based applications, such as e-commerce [1] and recommendation systems [2]. Most GCN applications like recommendation systems are deployed on cloud. To achieve real-time performance, GCN acceleration has been studied on application-specific integrated circuit (ASIC) [3] and GPU platform [4]. FPGAs in the cloud become a promising solution in terms of performance, energy efficiency and flexibility. There are several challenges of deploying GCN on cloud-based FPGAs: (1) Heterogeneity of GCN workload: There are two major computation kernels in GCN [5]: *aggregation* and *transformation*. The aggregation kernel is used for graph traversal, and involves large number of irregular memory accesses. On the other hand, the transformation kernel involves regular neural network computation, such as multi-layer perceptron (MLP). Thus, GCN acceleration needs to efficiently utilize external memory bandwidth as well as achieve massive computation parallelism. (2) Time to market: While GCNs are widely used, their models evolve rapidly [6]–[8]. RTL-based accelerators [3], [9] are hard to adapt to new GCN models and require significant development effort. HLS-based kernel design can be easily adapted to evolving GCN models, but requires careful optimizations to achieve high performance. (3) Architectural constraints: FPGAs contain massive on-chip resources. They are suitable for GCN acceleration, which requires massive memory bandwidth and computation parallelism. However, state-of-the-art FPGAs usually consist of multi-die with limited inter-die wire connections. The on-chip resources, such as memory ports, block RAMs and DSPs are unevenly distributed into different dies [10]. Thus, placing a large design of GCN on state-of-the-art FPGAs frequently causes PnR failures and timing violations.

To address the above challenges, we provide a case study of GCN inference acceleration on a state-of-the-art FPGA. We explore the programming model of HLS, dramatically reducing the design efforts as well as achieving significant speedup. Our main contributions are:

- We provide a partition-centric mapping strategy for mapping GCN inference onto FPGA. By deploying partition-centric mapping, partitioned graph can be stored on-chip and data reuse can be exploited. Based on our proposed task mapping, we design synthesizable HLS-based kernel functions for GCN with several pragma-driven optimizations to enhance the overall kernel performance.
- We perform design space exploration to facilitate feasible design pre-placement. This avoids potential PnR failures and also improves the timing performance.
- We evaluate our work on three widely used large-scale datasets. Experimental results show that compared with PyG running on a state-of-the-art CPU, our design achieves $59.95\times$ speedup and $96.22\times$ energy efficiency on average. Compared with DGL-CPU, our design achieves $2.9\times - 6.4\times$ speedup and $5.87\times$ energy efficiency on average. Compared with DGL-GPU, latency of our design is $1.67\times - 2.5\times$, and is $1.8\times$ more energy efficient.

II. Preliminaries

Graph Convolutional Network (GCN) is a powerful machine learning model operating on unstructured graphs. Vanilla-GCN [5] is the first proposed GCN model, the Vanilla-GCN layer operation is:

$$\boldsymbol{X}^{l+1} = \sigma\left(\boldsymbol{A}\boldsymbol{X}^{l}\boldsymbol{W}^{l}\right), \tag{1}$$

where $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ is the adjacency matrix of the input graph, $\boldsymbol{X}^{l} \in \mathbb{R}^{N\times f_l}$ is the 2-D input feature matrix, $\boldsymbol{X}^{l+1} \in \mathbb{R} \in \mathbb{R}^{N\times f_{l+1}}$ is the 2-D output feature matrix, $\boldsymbol{W}^{l} \in \mathbb{R}^{f_l\times f_{l+1}}$ is the layer weight matrix and $\sigma(\cdot)$ is the element-wise activation function. N denotes the number of vertices in the input

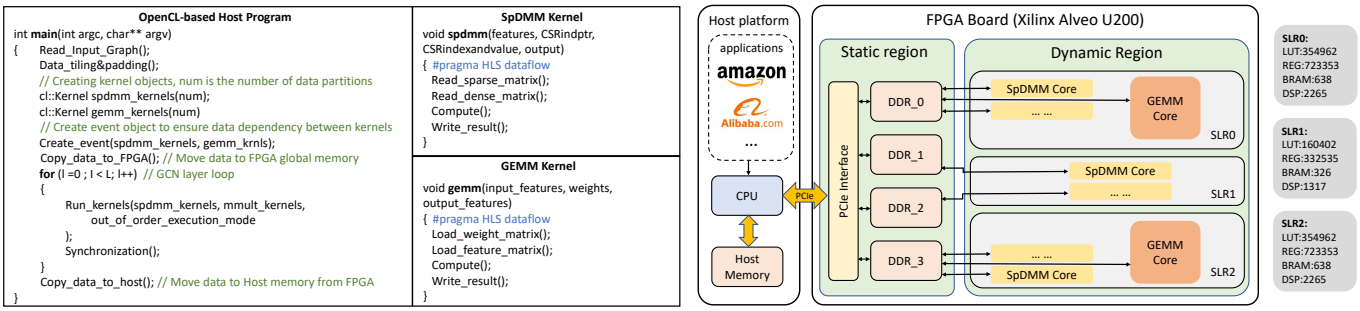

Fig. 1. End-to-end GCN inference using HLS programming model

graph, f_l and f_{l+1} are the feature dimensions of GCN layers. Basically, there are two main computational kernels in GCN models: (1) *Feature Aggregation (FA)-*$\boldsymbol{AX}$: $\boldsymbol{AX}$ is sparse-dense matrix-multiplication (SpDMM), where $\boldsymbol{A}$ is extremely sparse. For example, the adjacency matrix of PPI, Flickr, Reddit datasets have 0.5%, 0.2%, 0.2% density respectively. Such computation has poor data locality and large external memory traffic. Moreover, the arbitrary distribution of non-zero elements in $\boldsymbol{A}$ introduces irregular memory accesses, thus causes large memory access latency. (2) *Feature Transformation (FT)-*$(\boldsymbol{AX})\boldsymbol{W}$: Feature transformation requires general dense-dense matrix multiplication (GEMM) of $(\boldsymbol{AX})$ with $\boldsymbol{W}$. It has a regular computation pattern but suffers from large computation complexity.

III. Task Mapping and Kernel Optimizations

A. System Overview

The overview of our proposed acceleration system is shown on the right side of Fig. 1. GCN applications run on the host platform with a OpenCL-based host program to manage kernel execution. The FPGA board is deployed with the computation cores built from HLS.

B. Partition-centric Task Mapping and Scheduling

GCN inference contains consecutive L GCN-layer operations. Real-world input graphs are usually very large, consisting of tens of millions of nodes and edges [11]. Thus, we need to partition the execution task of GCN layers. Moreover, the two computation kernels of GCN layer operation have very different memory access pattern and computation pattern. We provide a task partition strategy that exploits both task parallelism and data parallelism. (1) *Task Parallelism:* We divide the GCN layer operation into two types of computation kernels–SpDMM and GEMM, to run feature aggregation and feature transformation respectively. (2) *Data Parallelism:* We partition the input graph. Each data partition V_i contains a set of vertices, which can perform FA and FT independently in a GCN layer. The proposed task mapping and coordination is performed by the host processor indicated in the left side of Fig. 1.

Using our proposed task mapping strategy, we create a pool of tasks, which are enqueued into a task/command queue managed by the host program. The host processor schedules the execution of the tasks on the SpDMM cores and GEMM cores that are deployed on the FPGA. To enable task-level parallelism and ensure data dependency across FA and FT kernels, we exploit the Bulk Synchronous Parallel model [12] which performs several supersteps consisting of local computation, global data communication and a barrier synchronization at the end of each superstep.

C. Kernel Optimizations

Based on our proposed partition-centric task mapping, we design SpDMM Core and GEMM Core to run the feature aggregation and feature transformation respectively. Feature Aggregation has low computation intensity but a large number of random memory accesses. Thus, it requires large memory bandwidth but a small number of computation units suffice. In contrast, Feature Transformation has a regular memory access pattern and requires large data parallelism. So, we need to perform optimizations for the two kernels differently. Since kernels are programmed in HLS, we can exploit the *pragma-driven* optimizations. The architecture of the two computation kernels are shown in Figure 2 and Figure 3 respectively.

SpDMM Core: SpDMM Cores perform multiplication of $\boldsymbol{A}$ and $\boldsymbol{X}$. Due to the high sparsity of $\boldsymbol{A}$, it is typically stored in compressed sparse row (CSR) format to save memory space. CSR format maintains an *indptr* array, an *index* array, and a *value* array. During SpDMM, matrix $\boldsymbol{A}$ is sequentially loaded from external memory; on the other hand, loading feature matrix $\boldsymbol{X}$ is dependent on the $index$ array, incurring many random memory accesses. As indicated in [13], HLS compiler has inefficiency in dealing with random memory accesses.

Therefore, to improve the bandwidth utilization, we perform several optimizations: (1) since the dimension of GCN layer feature ranges from $100 - 4000$ (usually greater than 16), we exploit the parallelism along the feature dimension. By bundling data into vectors of length 16 (each data is 32-bit floating point), the memory ports of SpDMM Core are arranged in $16 \times 32 = 512$-bit width, which maximize bandwidth efficiency of external memory access. As a result of vectorization, SpDMM computations are performed in a SIMD fashion. (2) Since loading data is the bottleneck of SpDMM, we want to keep the memory controller busy reading data required. To achieve this goal, we carefully pipelined our kernel using the *pipeline* pragma, and overlap computation and memory access using the *dataflow* pragma, so the SpDMM kernel can send out consecutive memory request. Furthermore,

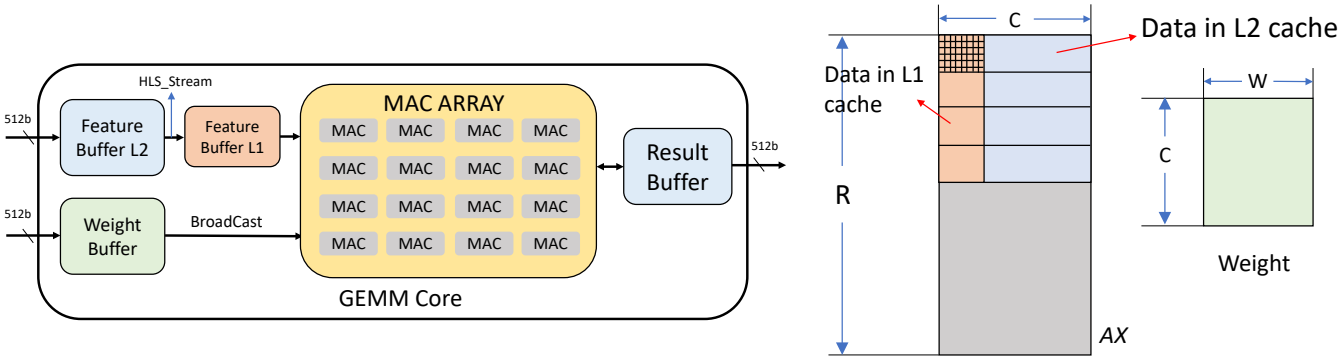

Fig. 2. GEMM Core

.

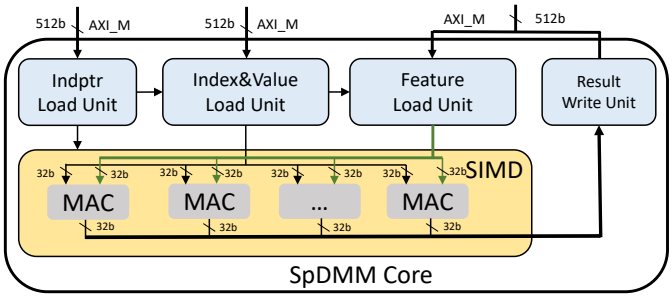

Fig. 3. SpDMM Core

memory accesses are buffered with FIFOs, which can avoid stalling of memory request. (3) To take full advantage of the external memory bandwidth, SpDMM Cores are connected to different DDR memory bank to enable parallel memory access. This can also reduce bank conflicts and improves bandwidth utilization.

GEMM Core: The architecture of GEMM Core is shown in Fig. 2. The computation kernel is a 2-D MAC (multiply-and-accumulate) array. GEMM Core performs block matrix multiplication of $(\boldsymbol{AX})$ with $\boldsymbol{W}$. In GCN, $\boldsymbol{W}$ is small; it can be fully stored in a on-chip Weight Buffer to prevent redundant access to external memory. The feature matrix is loaded from the external memory. To maximize the on-chip data reuse, we design a two-level cache hierarchy for $(\boldsymbol{AX})$. As shown in the right side of Fig.2, a blue block of matrix is loaded on-chip into the L2 buffer, the loaded size depends on the available on-chip memory (such as URAM or BRAM). High-end FPGAs usually has different types of on-chip memories, we load the orange part of the matrix to smaller but faster on-chip memory such as LUTRAM, which is the L1 buffer. The orange matrix blocks are data that will be computed soon. By implementing the two-level memory structure, we can maximize data reuse and reduce memory access latency. We also re-design the datapath of block matrix multiplication as shown in Fig. 4. The straight-forward implementation of block matrix multiplication reads and writes to the same address repeatedly , which would result in large initial interval of pipeline to resolve memory dependency. Our re-design datapath avoids stalling by adapting a different computation order that writes to different block of

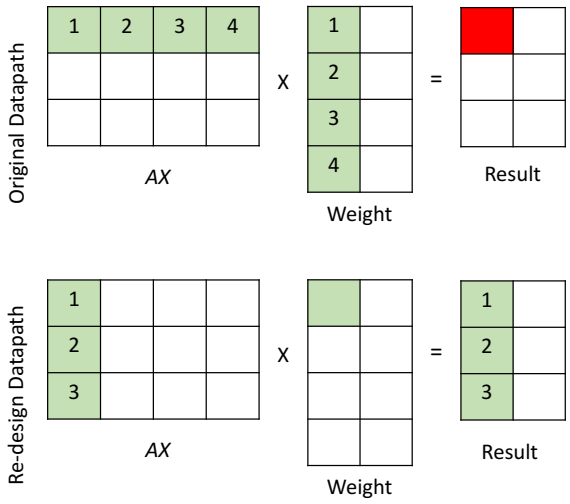

Fig. 4. Original Datapath and Re-designed Datapath

matrix every iteration. Note that though we compute different blocks every iteration, these blocks are all stored in the L1 buffer, thus would not incur extra overhead to read different blocks. With the new datapath and the two-level cache deploy, we achieved $3\times$ speedup within the GEMM core compared to the original straight-forward design.

IV. Design Space Exploration

The SpDMM core and GEMM core work as a producer-consumer model, where the input of GEMM core comes from the output of SpDMM core. To avoid idling, the execution time of the two core should be balanced. Thus, we can decide the number of SpDMM and GEMM to be instantiated using Equation 2. To estimate the execution time of the two core, we set the numerators as the amount of FLOPs performed by SpDMM and GEMM and the denominators are the performance of the two core. N is the number of vertices, deg is the average degree of the graph and f_l is the feature dimension of layer l, the product of the three is the amount of FLOPS performed by SpDMM. Similarly, $N \times f_l \times f_{l+1}$

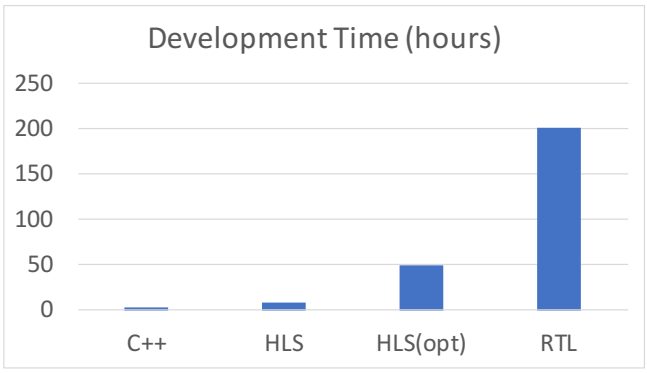

Fig. 5. Development time of two-layer Vanilla-GCN model using C++, HLS, HLS(optimized) and RTL.

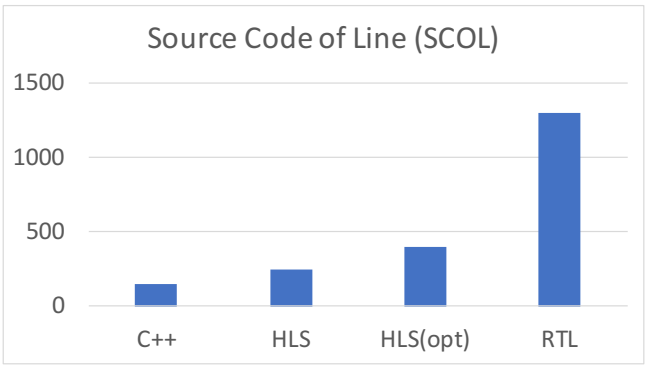

Fig. 6. Source code of lines of two-layer Vanilla-GCN using C++, HLS, HLS(optimized) and RTL.

is the amount of FLOPS performed by GEMM. x and y are the amount of SpDMM core and GEMM core instantiated. As mentioned in III-B, data are vectorized and cores executes in SIMD fashion, l_{vector} indicated the length of data vector.

$$\frac{N \times deg \times f_l}{x \times l_{vector} \times freq} = \frac{N \times f_l \times f_{l+1}}{y \times {l_{vector}}^2 \times freq} \tag{2}$$

Equation 2 can be further simplified as:

$$\frac{x}{y} = \frac{deg \times l_{vector}}{f_{l+1}} \tag{3}$$

The average degree of Reddit, Yelp and Amazon-2M ranges from 40 to 50, and f_1 is 128. The vector length l_{vector} is set to 16. Thus, to balance the execution time of two core, x/y should be set around 5 to 6.25.

We can instantiate many computation cores concurrently to achieve task-level parallelism. However, directly instantiating several computation cores may lead to many PnR failures and poor timing performance because Cloud-based FPGAs usually consists of multiple-dies, and the hardware resources are unevenly distributed in different dies (or Super Logic Regions (SLRs) in Xilinx's terminology). As shown in the rightmost part of Figure 1. For example, in Xilinx Alveo U200, SLR1 has less BRAMs and DSPs than SLR0 and SLR2. Moreover, these SLRs are connected through the Stacked Silicon Interconnect (SSI) technology, which leads to two issues: (1) there are limited number of wires across SLRs, and (2) the wires across SLRs have long delay. Therefore, the design without proper pre-placement can easily lead to (1) designs running out of resources in the SLRs, (2) routing failures due to the limited cross-die wires, and (3) low design frequency so the timing requirements can be met.

Thus, in addition to deciding the amount of SpDMM and GEMM, we enforce two types of constraints to find a feasible pre-placement for our design:

- **Placement constraint:** We place each SpDMM core and GEMM core into a specific SLR. The placement of computation core crossing SLRs is not allowed, and so as the cross-die DDR memory connections. This is done by explicitly specifying the mapping of kernels onto SLRs, and can prevent PnR failures.
- **Resource constraint:** While there are several computation cores in an SLR, the total usages of the on-chip resources, such as memory interface, BRAMs, UltraRAMs, and DSPs, should not exceed the available resources. This is done by explicitly specifying the mapping of hardware resources and memory interface in the GEMM core and SpDMM core .

V. Experiments

A. Experimental Setting

We implement our kernel designs in Vitis-HLS and host program in OpenCL. We used Xilinx Alveo U200 accelerator card as the platform for evaluation. It has 77 GB/s external memory bandwidth with 4 memory banks, $1182k$ Look-up tables, 6840 DSPs, 75.9 Mb of BRAM and 270 Mb URAM. We use Float32 as the data format. Vitis 2020.1 is used for IP generation, hardware linkage and Vitis Analyzer is used to obtain experimental data including resource utilization, power consumption and execution time. We use three large-scale graph datasets Reddit, Yelp and Amazon-2M [14] for evaluation. These datasets are extracted from the cloud-based GCN applications and their details are shown in Table I. We use the two-layer Vanilla-GCN model [5] for evaluation. The GCN-layer operation and GCN-layer dimensions (f_0, f_1, f_2) are specified in equation (1) and Table I, respectively.

TABLE I
Statistics of the Dataset and GCN-layer dimensions

Dataset	#Nodes	#Edges	f_0	f_1	f_2
Reddit	232965	11606919	602	128	41
Yelp	716847	27907940	300	128	107
Amazon-2M	2449029	123718152	100	128	47

B. Development Effort

Using HLS, we are able to develop FPGA kernels in a short amount of time. We first implement a two-layer Vanilla-GCN model in C++, and then modify the C++ code into synthesizable HLS code by adding some HLS pragmas for parallelization and port connections. Finally, we perform kernel optimizations as described in III-C to improve the

TABLE II
COMPARISON OF EXECUTION TIME

	Execution Time (sec)								
	PyG-CPU	PyG-GPU	DGL-CPU	DGL-GPU	1s1m	2s1m	4s2m	8s2m	16s2m
Amazon-2M	OoM‡	OoM	36	OoM	73.63	37.34	18.79	9.57	5.63
Reddit	81	OoM	3.2	0.65	16.56	8.46	4.26	2.18	1.10
Yelp	56	OoM	3.5	0.47	9.74	5.44	2.61	1.74	1.21

‡ Cannot run due to limited memory.

TABLE III
RESOURCE UTILIZATION FOR VARIOUS CONFIGURATIONS

Design	Resource Utilization			
	LUTs	**BRAMs**	**URAMs**	**DSPs**
1s1m	126k (10.63%)	53 (2.45%)	304 (31.7%)	1383 (20.22%)
2s1m	147k (12.24%)	83 (3.84%)	304 (32.7%)	1470 (21.49%)
4s2m	289k (24.48%)	166 (7.68%)	608 (63.3%)	2940 (42.98%)
8s2m	366k (30.39%)	286 (13.34%)	608 (63.3%)	3288 (48.06%)
16s2m	520k (43.88%)	526 (24.53%)	608 (63.3%)	3984 (58.16%)

TABLE IV
COMPARISON OF ENERGY EFFICIENCY (KJ/INFERENCE)

	Our design (16s2m)	**PyG-CPU**	**DGL-CPU**	**DGL-GPU**
Amazon-2M	0.47	OoM‡	3.78	OoM
Reddit	0.1	8.4	0.34	0.16
Yelp	0.06	5.9	0.37	0.12

‡ Cannot run due to limited memory.

overall performance. As shown in Figure 5, implementing a Vanilla-GCN takes about 4 hours using C++; modification into synthesizable HLS code takes another 4 hours, thus implementing Vanilla-GCN using HLS takes 8 hours in total. The optimizations for HLS takes approximately 50 hours. We also compare development effort with another two-layer Vanilla-GCN accelerator implemented in RTL [15]. Though the RTL implementation is not exactly the same as our design, design of SpDMM and GEMM kernel are similar. Result shows that it takes $4\times$ development time to implement a similar design in RTL compared with HLS.

HLS kernel development features C style coding, so functions can be expressed using less lines of code compared with RTL style coding. We compared the source code of lines (SCOL) for implementing a two-layer vanilla-GCN using C++, HLS and RTL. As shown in Figure 6, C++ and HLS takes similar amount of codes to implement this design, and is $3.3\times$ less than RTL.

C. Resource utilization

The hardware resource utilization is reported in Table III. We set various design configurations for evaluation. Using xsym as the notation to represent a design configuration, x denotes number of SpDMM Cores and y denotes number of GEMM cores. A design footprint is shown in Figure 7.

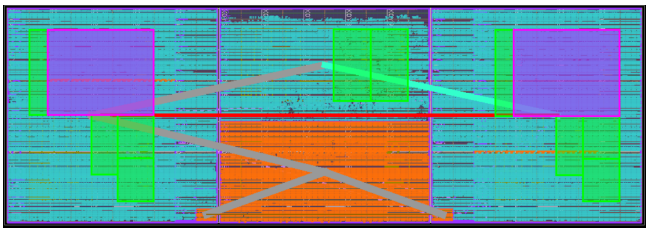

Fig. 7. FPGA footprint of our design. GEMM Cores are placed in the purple regions and SpDMM Cores are placed in the fluorescent green regions. The orange region is reserved as the FPGA shell.

D. Scalability

In order to evaluate the scalability of our design, we compare the execution time under various design configurations. Given the resource constraint, we instantiate two GEMM cores. Based on Equation 3, suggested amount of SpDMM would be $2 \times 6.25 = 12.5$. However, there might be performance degrade for SpDMM due to the uncertain latency of irregular memory access. Thus, we instantiate more than 12 cores of SpDMM in our design. The results are reported in Table II, it shows strong scaling as more compute units are deployed.

E. Cross-platform Comparison

Execution time: We compare the execution time to perform a full-batch GCN inference using a two-layer Vanilla-GCN model on different platforms. For GPU platform, We assume the graphs can fit in the GPU memory so full-batch inference can be performed. We compare our FPGA implementation with the baseline implementations using PyTorch Geometric (PyG) [16] and the highly-optimized framework Deep Graph Library (DGL) [17]. The CPU baselines are executed on an Intel Xeon Gold 5120 CPU platform, which has 28 cores with 56 threads, and 19.25 MB L3 cache. The GPU baselines are executed on Nvidia Titan Xp, a high-end GPU which consists of 3840 CUDA cores running at 1405 MHz, and memory bandwidth up to 547.7 GB/sec.

As shown in Table II, our design using 16s2m achieves $73.63\times$, $46.28\times$ speedup on Reddit, Yelp compared with PyG [16]. The speedup comes from our optimization for memory bandwidth utilization and storing data on-chip for data reuse. On the other hand, PyG frequently store and load the intermediate results which causes large memory access overhead. Our design achieves $2.9 \times - 6.4\times$ speedup compared with the CPU version of DGL. For GPU version, latency of our

design is $1.67\times$ and $2.5\times$ on Reddit and Yelp; notice that the peak performance of Nvidia Titan Xp is 9.3 TFLOPs/sec, while our design is only 153.6 GFLOPs/sec, and the GPU has 547 GB/sec of memory bandwidth, which is $7.1\times$ more than FPGA's 77 GB/sec bandwidth.

Energy efficiency: We define energy efficiency as energy consumed per inference (KJ/Inference). We compare the energy efficiency of our design with PyG. We measure the energy consumption of FPGA by enabling power-profile option in Xilinx XRT [18] to perform fine-grained energy profiling. We use the Thermal Design Power (TDP) [19] to estimate the CPU and GPU power consumption. TDP indicates the power consumption of a CPU or GPU under maximum theoretical load, which is suitable in our case during GCN inference. We derive the energy consumption by multiplying the TDP with time for inference. The results are shown in Table IV. The results show that our FPGA implementation is $84\times$, $97.9\times$ more energy efficient using Reddit, Yelp than PyG-CPU. For DGL, our design is $5.87\times$ more energy efficient on average compared with CPU version, and $1.8\times$ more energy efficient on average compared with GPU version.

VI. Conclusion

In this paper, we performed a detailed case-study of GCN inference acceleration on a state-of-the-art FPGA using HLS. We explored HLS-based optimizations for our kernels and performed design space exploration. We evaluated our design on three commonly used datasets and achieved average speedup of $59.95\times, 4.40\times$ and $0.5\times$ compared with PyG-CPU, DGL-CPU and DGL-GPU respectively.

For the future, we plan to build Application Programming Interfaces (APIs) so a general N-layer GCN model can be easily implemented. We also plan to exploit the High Bandwidth Memory to achieve more speedup for GCN inference.

VII. Acknowledgement

This work is sponsored by the U.S. National Science Foundation under grant numbers OAC-1911229 and CCF-1919289. Equipment and support by Xilinx are greatly appreciated.

References

[1] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 3165–3166.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[3] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE HPCA*, 2020, pp. 15–29.

[4] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "Neugraph: parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference.*

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[6] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," 2017.

[7] H. Gao, Z. Wang, and S. Ji, "Large-scale learnable graph convolutional networks," *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2018.

[8] F. Wu, T. Zhang, A. H. de Souza Jr., C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," 2019.

[9] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale gcn inference," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 61–68.

[10] Xilinx, "Alveo u200 and u250 data center accelerator cards data sheet," https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf, 2020.

[11] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *International Conference on Learning Representations*, 2020.

[12] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, Aug. 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[13] Y.-k. Choi, Y. Chi, J. Wang, L. Guo, and J. Cong, "When hls meets fpga hbm: Benchmarking and bandwidth optimization," *arXiv preprint arXiv:2010.06075*, 2020.

[14] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn," *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2019. [Online]. Available: http://dx.doi.org/10.1145/3292500.3330925

[15] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale gcn inference," in *2020 IEEE 31st ASAP*. IEEE, 2020, pp. 61–68.

[16] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[17] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[18] Xilinx, "Xilinx runtime," https://github.com/Xilinx/XRT, 2020.

[19] Intel, "Measuring processor power: Tdp vs. acp," 2011.