Graph Embedding Framework *with* FPGA Acceleration *and* In-Storage Computing

CS 259 Final Project Report

Contributor: Hengda Shi , Gaohong Liu
{hengda.shi, cheimu}@cs.ucla.edu

Instructor: Prof. Jason Cong
Teaching Assistant: Dr. Zhe Chen

Department of Computer Science, UCLA

Abstract

Our project application called Graph Convolutional Neural Network lies in the field of machine learning and graph processing. It is a graph embedding network that can classify nodes in a graph into different classes. Analysing graph data such as social networks, citation networks, protein structures is a popular research topic because of its large potential practical interest. However, it has been a challenge for researchers to develop effective algorithm to correctly classify or cluster nodes or graphs.

With the success of GCN, we propose an architecture that utilizes graph coarsening technique to obtain a smaller representation of the graph while preserving almost all the spectral information, and near storage computing to bypass the memory for achieving higher throughput computation. The graph coarsening technique utilizes a graph coarsening framework called GraphZoom. It is a research project done by Professor Zhiru Zhang's group from Cornell University. The result of their experiment shows that this framework can obtain at least 5x speedup compared to the original implementation of the graph embedding framework and the same or even better accuracy. The near storage computing framework called Insider comes from Professor Jason Cong's group. It is a streaming batch processing framework that utilize FPGA as the computing unit to directly communicate with the disk for reading and writing files. More details would be discussed later in the following sections.

1. Related Work

Graph Convolutional Neural Network is inspired by the success of Convolutional Neural Network in the computer vision field. The Convolutional Neural Network is a type of neural network that aggregate a square area of the image into one scalar value and propagate the results into the typical neural network. Graph Convolutional Neural Network has a similar meaning in terms of aggregating neighbor information, but it is also different from CNN because images are Euclidean structure which means the distance between one pixel and its neighbors is always fixed to 1. Graph is non-Euclidean structure because the distance between one node and its neighbor can hardly be defined. Therefore, using kernel to aggregate condensed results from original image does not make sense in Graph Convolutional Neural Network. In this section, we would mainly discuss one of the most important work in Graph Convolutional Neural Network from Thomas N. Kipf, Max Welling ICLR 2017 paper "Semi-Supervised Classification with Graph Convolutional Networks".

The framework they proposed in the paper is called GCN. It consists of different layers including Dropout, Sparse Matrix Multiplication, ReLU, Dense Matrix Multiplication, Graph Convolution. The process can be demonstrated in the following figure, and more details would be added in the following sections. In high level, GCN would calculates the symmetric normalized Laplacian matrix from the edge index matrix which is a diagonal matrix that indicates the number of edges that each node has, and the adjacency matrix that indicates the connection relationship between each two nodes. It then decomposes

the Laplacian matrix to find its eigenvalues and eigenvectors. It uses first order Chebyshev polynomials to approximate the eigendecomposition process, which can largely reduce overhead caused by the decomposition.

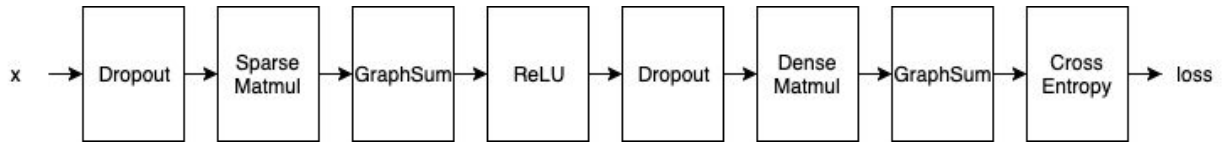

Figure 1.1 GCN framework workflow

GraphZoom is a multi-level graph coarsening framework that can largely improve the performance of graph embedding networks. It has four stages including graph fusion, graph coarsening, graph embedding, graph refinement.

Graph fusion would first perform kNN on the node feature matrix. Each row of the node feature matrix represents a feature vector of one node. Given all the features of all nodes, we could compute the l2 norm distance of all other nodes with respect to the current node. The current node would then have a list of distance sorted. kNN would take the first k nodes on the list and mark 1 on the corresponding location in a new matrix with dimension numNode * numNode. For example, if 2-NN is performed on node 0, node 4 and 6 are the closest two nodes in terms of l2-norm distance, then location (0, 4) and (0, 6) on the matrix would be marked 1. The output of the kNN would have the same dimensionality as the adjacency matrix. The adjacency matrix preserves the connection information among all nodes, and the new calculated matrix called attribute matrix preserves the feature information for all nodes. By adding them together, the final fused matrix would preserve both all these information and can be used in the coarsening step. The coarsening step utilizes a local spectral embedding scheme. The coarsening step would apply a low-pass graph filter on node feature matrix which would filter out eigenvectors corresponding to high eigenvalues. The resulting smoothed vector can be considered as a linear combination of the first eigenvectors. The smoothing function that they used is Gauss-Seidel iterations used for solving linear system of equations. The number of nodes would be shrunk to t-dimension based on the spectral similarity of their low dimensional embedding vectors. Given the coarsest graph, any type of graph embedding network can be performed such as DeepWalk, GCN, and GraphSage. The refinement process would then iteratively refine the embedding results from the coarsest graph to the original graph.

Insider is a near storage computing framework which utilizes FPGA as the near storage computing unit. it is able to saturate the high drive performance while retaining enough programmability. On the software side, INSIDER integrates with the existing system stack and provides effective abstractions. For the host programmer, INSIDER introduces virtual file abstraction to abstract ISC as file operations; this hides the existence of the drive processing unit and minimizes the host code modification to leverage the drive computing capability. INSIDER is designed for application which is 1) streaming 2) heavy computing tasks 3) high memory required. The programmers still write C++ code, but when there is a need to use FPGA and in-storage computing to reduce the workload of CPU and speedup the program, the programmers then invoke the INSIDER using virtual file system abstraction.

2. Algorithm

2.1 Algorithm of Sequential GCN Convolution Layer

As shown in the previous section, our convolution layer can be described by the following formula

$$h_{v_i}^{(l+1)} = \sigma\left(\sum_j \frac{1}{c_{ij}} h_{v_j}^{(l)} W^{(l)}\right)$$

Equation 2.1 GCN propagation rule

Where h is hidden layer and W is layer-specific trainable weight matrix, c is normalization and σ is activation function. Following this, the sequential version of convolution layer pseudocode works can be described as following in Algorithm 1.

Algorithm 1 Sequential Convolution Layer Algorithm

---

function Conv_Layer

```
    Load node features
    Load edge indices
    Compute node feature matrix * weight matrix
    Assign Index
    Assign Weight
    Accumulate
    Invert degree
    Normalization
    Update result matrix
```

end function

Firstly load the node features and edge indices, and then do the matrix multiplication of node feature and the corresponding weight. Then we assign index and weight and calculate current layer's result. Then do the normalization, update the result, and finally propagate to next layer.

2.2 Merlin Accelerated Convolution Layer Algorithm

To accelerate we use merlin compiler and corresponding pragma. We firstly use #pragma ACCEL kernel to indicate the kernel functions, then we use #pragma ACCEL pipeline, #pragma ACCEL parallel to describe the datapath on FPGA, which is described in Algorithm 2:

Algorithm 2 Merlin Accelerated Convolution Layer Algorithm

---

function Conv_Layer

#pragma ACCEL pipeline

Load node features

#pragma ACCEL pipeline
Load edge indices
#pragma ACCEL pipeline
/ Compute node feature matrix * weight matrix */*
Iterate by i dimension
#pragma ACCEL parallel
Iterate by j dimension
#pragma ACCEL parallel
Iterate by k dimension
/ Done */*
#pragma ACCEL pipeline
/ Assign edge index */*
Assign index by i dimension
#pragma ACCEL parallel
Assign index by j dimension
/ Done */*
#pragma ACCEL pipeline
Assign Weight
#pragma ACCEL pipeline
Invert degree
#pragma ACCEL pipeline
Normalization
#pragma ACCEL pipeline
/ Update result matrix */*
Compute result by i dimension
#pragma ACCEL parallel
Compute result by j dimension
/ Done */*
end function

We add pipeline between each step, and when there is matrix multiplication, we use loop unroll to break the matrix to make them compute parallelly so the matrix computation can be done element wisely. Besides the pragma we added, the other difference is the partition. Due to the requirement of merlin compiler, we then partition the input parameter and set the parallel factor to 128 to make compiler work. The algorithm itself is not modified.

2.3 Insider Accelerated Convolution Layer Algorithm

The INSIDER accelerated convolution layer algorithm is shown in algorithm 3:

Algorithm 3 Host Code of INSIDER Accelerated Convolution Layer Algorithm

```
function Conv_Layer
    Load node features
    Load edge indices
    vread(node feature matrix *
weight matrix)
    Assign edge index
    Assign Weight
    Invert degree
    Normalization
    Update result matrix
end function
```

Algorithm 3 Device Code of INSIDER Accelerated Convolution Layer Algorithm

```
function Matrix_Multiplication
    Load node feature matrix
    Load weight matrix
    Result Matrix <- node feature
matrix * weight matrix
    Send Result Matrix
end function
```

The algorithm is same to sequential version of convolution layer, and the only difference is that host code should issue matrix multiplication to INSIDER partitions continuously and wait until INSIDER sends the result matrix back. Then host program goes on computing the rest of convolution layer.

3. Architecture and Implementation

As discussed in previous section, we mainly have two types of acceleration. One is using FPGA only, and program FPGA to accelerate the whole GCN only, and use host program to control it; and the other one is in streaming style using INSIDER where host program offloads the matrix multiplication to INSIDER, and processing the GCN in streaming style whenever INSIDER sends partial matrix multiplication results back to the host program.

3.1 GCN acceleration using FPGA only

The architecture of using FPGA only is shown in figure 1, host program works as a control and invokes the kernel of GCN on FPGA to compute the GCN. We firstly implemented the GCN FPGA image using merlin compiler by adding Merlin compiler's pragma such as #pragma ACCEL pipeline and #pragma ACCEL parallel. Then using Merling compile to generate the RTL, then generate the FPGA image.

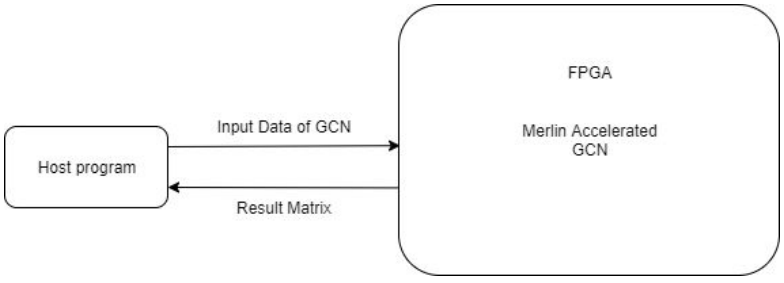

Figure 3.1 of acceleration using FPGA only.

3.2 GCN acceleration using INSIDER

After we implemented the datapath on FPGA, we then begin to implement the INSIDER's FPGA which is implemented as device program of INSIDER. The architecture of GCN acceleration using INSIDER is shown in figure 2. The users still write host programs but totally different. Host program is the entire GCN program without matrix multiplication. The matrix multiplication is offload to INSIDER to compute. The INSIDER will read the three input parameter matrices partitions required by GCN. Then after the processing of this partition of nodes features is finished, the matrix multiplication result will be sent back to the host program and INSIDER will receive the new partition of nodes features. The host program do the computation of GCN on CPU as normal but it offloads the matrix multiplication to INSIDER and read the results directly from INSIDER without computing them again to achieve the speed up.

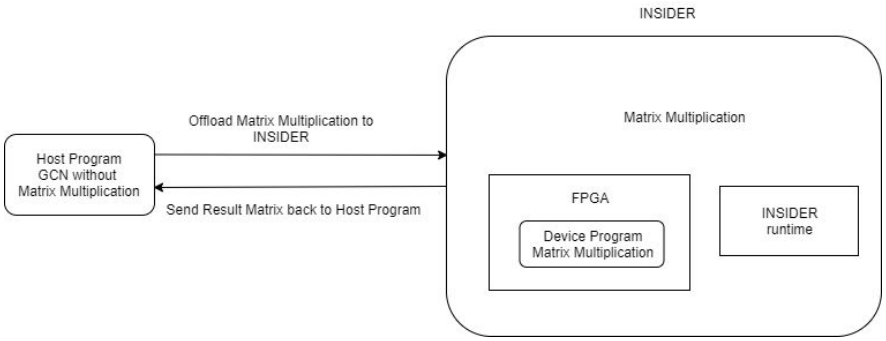

Figure 3.2 of acceleration using INSIDER.

The datapath of FPGA in insider is the same but the coding style is different. INSIDER now only support HLS now. The workflow to implement INSIDER device program is shown in figure 3.

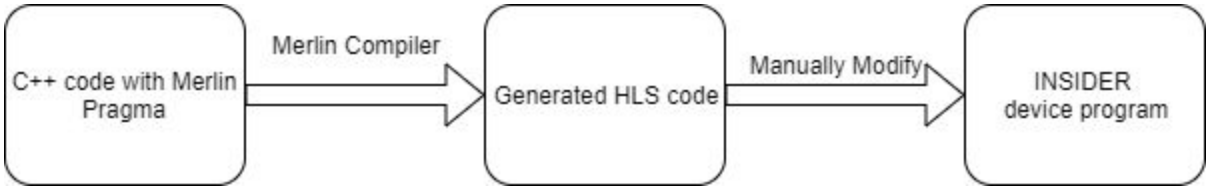

Figure 3.3 of workflow to generate INSIDER device program.

We highly optimized the GCN using merlin compiler. Then we compile it using merlin compiler and generated a series of synthesized HLS codes. Then to integrate with INSIDER, we manually modify these generated HLS code with INSIDER's device programming rules. Then we got the INSIDER's device program.

The host code is similar to normal C++ GCN code. The major difference is how to offload computing to INSIDER. When host code need to compute the matrix multiplication, it uses INSIDER's API vread()to invoke computing kernels. Then read results and compute the rest of GCN work.

4. Accelerator Design

Since we first accelerated GCN convolution layer on FPGA directly using merlin compiler, we implemented the corresponding GCN convolution layer accelerator and generated the bitstream and FPGA image. The accelerator data path on FPGA is shown in the following figure where the matrix multiplication and the result computation are computed in parallel and each phase of GCN is computed in pipeline to speed up.

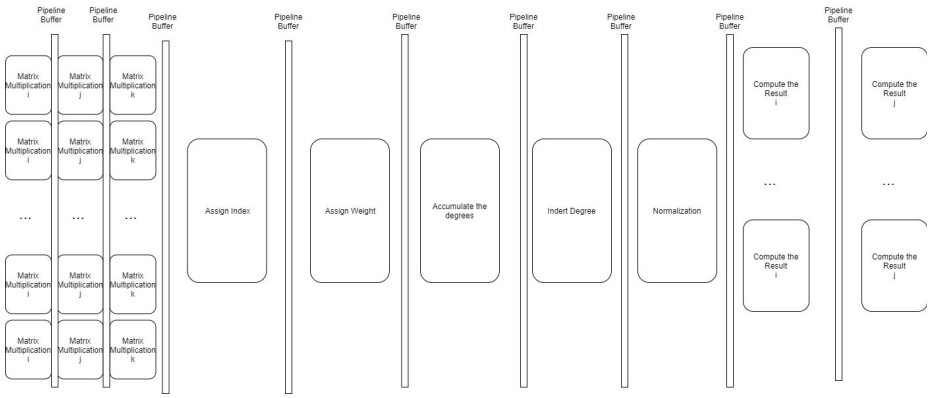

Figure 4.1 datapath of accelerator on FPGA

The whole computing process is working exactly the same as the algorithm described in Algorithm (Section 2).
Without using any program, the baseline resources report is following,

Kernel	Cycles	LUT	FF	BRAM	DSP	URAM	Detail
gcnconv_kernel (gcnconv.cpp:8)	440170421 (1467.235ms)	7393 (~0%)	8857 (~0%)	578 (13%)	52 (~0%)	0 (~0%)	-

Table 4.1 baseline resource report

and it is clear that only BRAMs are used and other resources are wasted 100%. After adding merlin-pragma, we got following resources report.

Kernel	Cycles	LUT	FF	BRAM	DSP	URAM	Detail
gcnconv_kernel (gcnconv.cpp:8)	14311508 (47.705ms)	168990 (14%)	215194 (9%)	382 (8%)	221 (3%)	0 (~0%)	-

Table 4.2 accelerated design resource report

And all kinds of resources are used where we used 14% LUT, 9% Flip-Flop, 8% Bram and 3% DSP. The performance is presented in the next section.

5. Expected Performance

By wrapping our GCN framework with GraphZoom, we can obtain the speedup of GraphZoom similar to the graph below:

Please discuss the expected performance of your accelerator, and identify the performance bottlenecks of the overall design (computation or communication? Which level of communication, at PCI-e or DRAM?, etc)

	PPI		Reddit	
Method	Micro-F1	Time(mins)	Micro-F1	Time(hours)
GraphSAGE-GCN	0.601	9.6	0.908	10.1
GZoom(GSAGE-GCN, *l*=1)	**0.621**	4.8(2.0×)	**0.923**	3.4(3.0×)
GZoom(GSAGE-GCN, *l*=2)	0.612	**1.8(5.2×)**	0.917	**1.6(6.3×)**
GraphSAGE-mean	0.598	11.1	0.897	8.1
GZoom(GSAGE-mean, *l*=1)	0.614	5.2(2.2×)	**0.925**	2.6(3.1×)
GZoom(GSAGE-mean, *l*=2)	**0.617**	**1.8(6.2×)**	0.919	**1.2(6.8×)**
GraphSAGE-LSTM	0.596	387.3	0.907	92.2
GZoom(GSAGE-LSTM, *l*=1)	0.614	151.8(2.6×)	**0.920**	39.8(2.3×)
GZoom(GSAGE-LSTM, *l*=2)	**0.615**	**52.5(7.4×)**	0.917	**14.5(6.4×)**
GraphSAGE-pool	0.602	144.9	0.892	84.3
GZoom(GSAGE-pool, *l*=1)	0.611	66.0(2.2×)	**0.921**	27.0(3.1×)
GZoom(GSAGE-pool, *l*=2)	**0.614**	**23.4(6.2×)**	0.912	**12.4(6.8×)**

Table 5.1 GraphZoom speedup comparison

As we can see in the graph, GraphZoom can obtain at least 5x speedup compared to the GraphSage algorithm. GraghSage algorithm is a memory optimized implementation of GCN, which reduces the memory footprint by sampling neighbors of the nodes while aggregating information. We can expect the performance of our framework would improve 5x compared to the original.

The following reports are from the merlin compiler. The first table is the hardware simulation result for the baseline GCN convolution layer. The second table is the hardware simulation result for the accelerated merlin design. As we can see in the tables, the accelerated version of the kernel achieves about 30x speedup compared to the baseline. It is mostly due to the acceleration on the matrix multiplication which contains three for loops.

Hierarchy	TC	AC	CPC	Detail
gcnconv_kernel (gcnconv.cpp:8)		440170421 (100.0%)	440170421	-
loop i (gcnconv.cpp:21)	2708	435533056 (98.9%)	435533056	flattened with loop j (gcnconv.cpp:22)
loop j (gcnconv.cpp:22)	16	-	-	flattened
loop k (gcnconv.cpp:24)	1433	27201860 (6.2%)	10045	pipeline II=7
loop i (gcnconv.cpp:30)	2	5436 (0.0%)	5436	flattened with loop j (gcnconv.cpp:31), pipeline II=1
loop j (gcnconv.cpp:31)	2708	-	-	flattened
loop i (gcnconv.cpp:36)	13264	13264 (0.0%)	13264	-
loop i (gcnconv.cpp:40)	13264	132641 (0.0%)	132641	pipeline II=10[1]
loop i (gcnconv.cpp:44)	2708	2733 (0.0%)	2733	pipeline II=1
loop i (gcnconv.cpp:48)	13264	26546 (0.0%)	26546	pipeline II=2
loop i (gcnconv.cpp:52)	13264	13271 (0.0%)	13271	pipeline II=1
loop j (gcnconv.cpp:53)	16	-	-	parallel factor=16x
loop i (gcnconv.cpp:58)	16	4456713 (1.0%)	4456713	flattened with loop j (gcnconv.cpp:59), pipeline II=21
loop j (gcnconv.cpp:59)	13264	-	-	flattened

Table 5.2 GCN convolutional layer baseline performance table

Hierarchy	TC	AC	CPC	Detail
gcnconv_kernel (gcnconv.cpp:8)		14311508 (100.0%)	14311508	-
auto memory burst for 'weight'(read)		1433 (0.0%)	1433	cache size=91712B
auto memory burst for 'edge_index'(read)		1658 (0.0%)	1658	cache size=106112B
loop matrix_mul_i (gcnconv.cpp:21)	2708	14010948 (97.9%)	14010948	-
loop matrix_mul_j (gcnconv.cpp:24)	16	13994944 (97.8%)	5168	-
loop matrix_mul_k (gcnconv.cpp:27)	1433	11611904 (81.1%)	268	loop tiled, parallel factor=128x, pipeline II=7
loop assign_index_i (gcnconv.cpp:35)	2708	2708 (0.0%)	2708	pipeline II=1
loop assign_index_j (gcnconv.cpp:37)	2	-	-	parallel factor=2x
loop assign_weight (gcnconv.cpp:43)	13264	13264 (0.1%)	13264	-
loop inc_degree (gcnconv.cpp:48)	13264	132640 (0.9%)	132640	pipeline II=10[2]
loop invert (gcnconv.cpp:53)	2708	2722 (0.0%)	2722	pipeline II=1
loop assign_norm (gcnconv.cpp:58)	13264	13274 (0.1%)	13274	pipeline II=1
auto memory burst for 'result'(read)		2708 (0.0%)	2708	cache size=173312B
auto memory burst for 'result'(read)		2708 (0.0%)	2708	cache size=173312B
loop calc_result_i (gcnconv.cpp:63)	13264	132651 (0.9%)	132651	pipeline II=10[3]
loop calc_result_j (gcnconv.cpp:65)	16	-	-	parallel factor=16x
auto memory burst for 'result'(write)		2708 (0.0%)	2708	cache size=173312B
auto memory burst for 'edge_index'(write)		998 (0.0%)	998	cache size=63888B

Table 5.3 GCN convolutional layer merlin performance table

6. Verification Flow

The development procedure is working as following (figure 6.1).

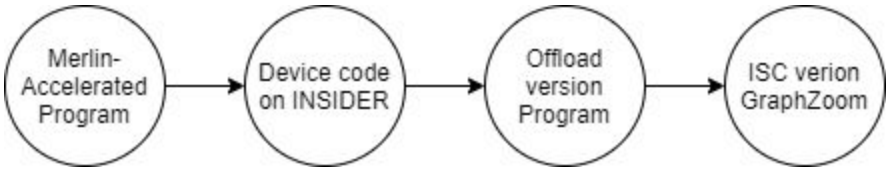

Figure 6.1 Development Process

We will first implement a merlin-accelerated program, and then modify it as device code on INSIDER, and then implement offload version GCN program, and then integrate this part of the GCN back into GraphZoom, so the verification flow verifies each part of intermediate programs.

6.1 Step 1: Verification of Merlin-Accelerated Program

The Merlin-Accelerated version of GCN convolution layer is working as following (figure 6.2).

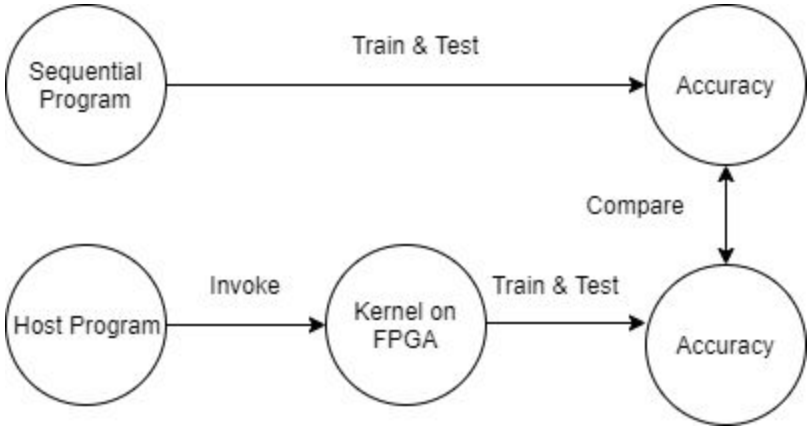

Figure 6.2 Merlin-accelerated program verification process

We trained the sequential version of GCN as baseline and test it to get the accuracy of the whole algorithm. Then we use accelerate the whole program on FPGA and compare the accuracy to verify the correctness of the whole program.

6.2 Step 2: Verification of INSIDER device program

We then modify the generated HLS program from Merlin-Compiler to INSIDER's device program. The workflow of INSIDER is shown as following (figure 6.3)

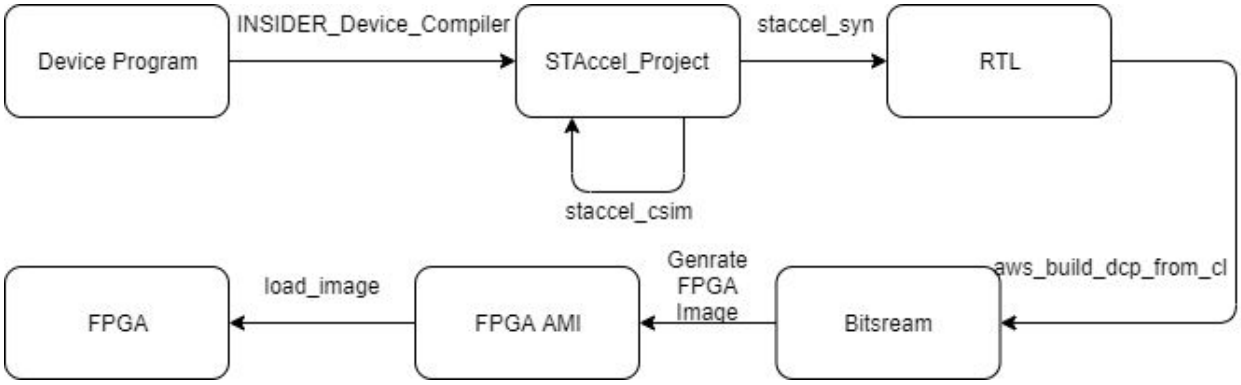

Figure 6.3 Merlin-accelerated program verification process

Because during the INSIDER workflow there is a process using ST_Accel co-simulation, we directly use it to simulate our device program. Specifically, since we have not integrated GCN convolution layer on INSIDER, we only implemented an easy experimental application on INSIDER when we first get through the INSIDER, so we verify it using ST_Accel co-simulation. We modified the program `csim/src/interconnect.cpp` and invoke `user_simulation_function` to simulate it.

6.3 Step 3: Verification of Offload version GCN program

If possible and we integrate the INSIDER with GCN convolution layer, we could verify the offload version program by the following process shown in figure 6.4

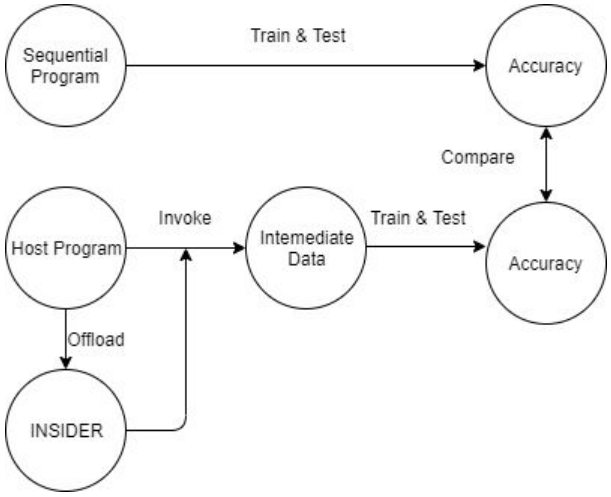

Figure 6.4 offload version program verification process

The idea is the same: Train and test the sequential program; then train and test the offload version of program; Finally compare the accuracy of two version to verify the correctness.

6.4 Step 4: Verification of In-Storage computing version GraphZoom

With the offload version of GCN program, we can add it into GraphZoom finally and the verification flow is shown in figure 6.5.

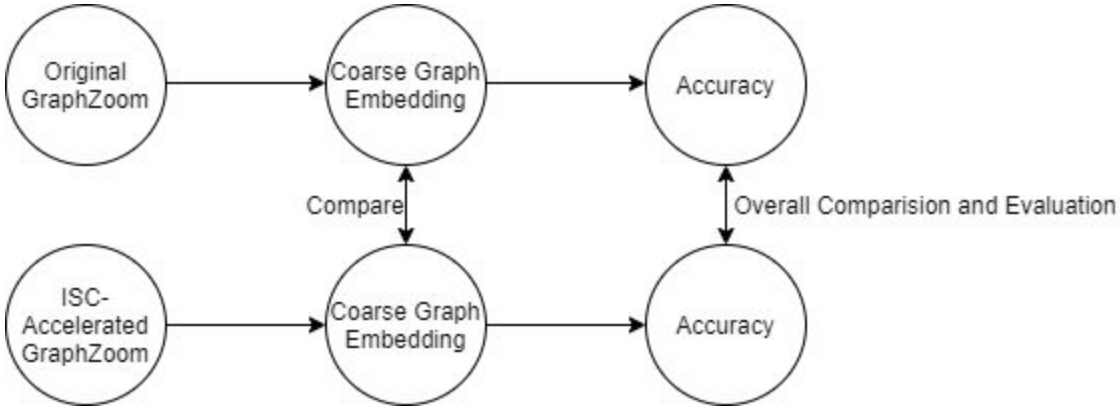

Figure 6.5 Verification of ISC-enabled GraphZoom

Same process. Compare the Coarse Graph Embedding between pure python version GraphZoom and ISC-enabled GraphZoom program; Then compare the final accuracy of two, and evaluate.

7. Challenges

7.1. Unfamiliarity with frameworks used in this project

We are not familiar with the GCN framework and coarsening techniques because it contains many mathematical background including graph spectral theory and approximation with polynomials. The

unfamiliarity made us hard to understand the concepts presented in the papers. The mathematical formulas presented in the paper are extremely different from what we would actually implement in the code, careful evaluation is also needed to identify the procedures of the algorithm. We compared the formulas and the code structure to identify the structure of the program and understand the details of the framework

7.2. Lack of documentation for the Insider program

The Insider program is extremely undocumented. Even though it uses the hls pragmas in the device code, the workflow and implementation is largely different from the hls context. In the device code, there is always a while(1) that wraps all the operations in between which is confusing because this loop would never exit. The design of the data structure in Queue is also confusing because there is no documentation descripting what should the data structure be and what are the limitations of this data structure. It seems like the data is always less than 512 bits but there is no clear documentation describing it. There are only three APIs (vread, vwrite, vclose) for the host to use and it seems like all the example code would take part of the data to process and then continue with the rest which is basically streaming in nature. In contrast, GCN has many sparse computations which cannot be streamed in nature. It becomes the biggest bottleneck.

7.3 Difficulty in tiling graph for computation

Another bottleneck lies in the difficulty of tiling graph for computation. Tiling has to be done on graph because Insider is streaming in nature. However, we cannot jump to the next line of the file and jump back after reading. It makes the tiling hard to achieve as the size would easily go beyond the limit of the on-chip memory. We did not actually implement tiling as it would take time to rewrite the accelerated merlin code to insider hls and configure queue data properly.

8. Possible Extensions

8.1 Streaming Nature of INSIDER

Since the INSIDER is designed for streaming applications, directly deploy the whole GCN on it is impossible. The extension is firstly using edge-based graph processing and partition the graph. Then each time send a partition of graph to INSIDER to stream the GCN convolution layer, or at least matrix multiplication.

8.2 Communication between Virtual & Real file and between Python & C++

Current version of INSIDER uses virtual file where it reads input from real files and writes output to virtual file, a memory region. Because our GraphZoom program is written in python, we try to rewrite the output to real file and let python to read the real file to get the coarse graph embedding. However, this could reduce the performance the acceleration since the intermediate data should write back to disk and read from it.

The extensions could be follow. Since the host program written in C++, which is naturally use POSIX functions to write virtual file in shareable region. Then when use python to read it, try to use python to

control the process and use inter-process communication to read this shared memory region to directly read the output from INSIDER to achieve speedup.

8.3 Support Merlin Compiler or Other Platform

Current INSIDER can only support HLS and C++, which is hard for software programmers to use. So another extension is to develop INSIDER using HeteroCL, which is easier and more powerful than the current one.

9. Contributions

Evaluate Baseline	Hengda Shi
Implement Merlin-Accelerated Convolution Layer	Hengda Shi
Setup and debug INSIDER tool flow	Gaohong Liu
Write experimental INSIDER device and host program	Gaohong Liu
Integrate INSIDER with GCN	Both
Others (such as presentation and report,etc.)	Both

10. Reference

[1] C. Deng, Z. Zhao, Y. Wang, Z. Zhang, and Z. Feng, **GraphZoom: A Multi-Level Spectral Approach for Accurate and Scalable Graph Embedding**, *arXiv e-print*, arXiv:1910.02370, Oct. 2019.

[2] Z. Ruan, T. He, J, Cong. **INSIDER: Designing In-Storage Computing System for Emerging High-Performance** Drive. *2019 USENIX Annual Technical Conference (USENIX ATC 19). USENIX Association, 2019*

[3] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, **HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing**, *International Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb. 2019.

[4] T. N. Kipf and M. Welling. **Semi-supervised classification with graph convolutional networks**. *International Conference on Learning Representations (ICLR), 2017.*

[5] Wu, Zonghan, et al. **A comprehensive survey on graph neural networks**. *arXiv preprint arXiv:1901.00596* (2019).