Graph neural network hardware acceleration in Pytorch with streaming PYNQ overlays

Jose Nunez-Yanez
Department of Electrical Engineering
University of Linköping
Linköping, Sweden
jose.nunez-yanez@liu.se

***Abstract*—Graph neural networks (GNNs) show high learning accuracy when applied to non-euclidean data in which data elements do not fit into a regular structure. They combine sparse and dense data characteristics and this, in turn, results in a combination of compute and bandwidth intensive requirements challenging to meet with general purpose hardware. In this paper we investigate a dataflow of dataflows (DoD) hardware architecture using high-level synthesis that optimizes data access and processing element utilization. The architecture is highly configurable with both the number of hardware threads available for the aggregation and combination phases and the number of compute units per thread defined at compile time. The fine-grained dataflow in the compute units streams words with a bit-width that depends on the network precision while the coarse-grained dataflow that links the aggregation and combination stages streams partially computed matrix tiles. The accelerator is mapped to the programmable logic of a Zynq Ultrascale device whose processing system runs Pytorch extended with PYNQ overlays. Preliminary results on the citeseer citation network show a performance gain of 20x with multi-threaded hardware configurations compared with the multi-threaded CPU implementation available in Pytorch.**

***Index Terms*—neural network, FPGA, sparse, pruning, matrix multiplication acceleration, TensorFlow**

I. Introduction

GNNs perform tasks such as graph classification, node classification, link prediction or graph clustering and have a large number of applications in areas such as anomaly detection, bioinformatics, cybersecurity or natural language processing. GNN processing uses both dense and sparse data representations and the resulting irregular computing and data access means that both inference and training of GNNs are complex. Popular machine learning frameworks like Tensorflow and Pytorch support graph neural network development and, in this work, we focus on Pytorch and how its python interface can be integrated with accelerator overlays developed with Xilinx PYNQ for graph neural network processing. PYNQ is a Xilinx Python framework that runs on Ubuntu and provides a highly-productive development platform for Xilinx devices such as the Zynq family. In this paper, we present preliminary results on creating a dataflow architecture for graph neural networks using high-level-synthesis and its integration into PYNQ and Pytorch. The architecture is based on our previous work on the Tensorflow Lite accelerator FADES [1] and it has been named gFADES (graph FADES). This initial work focuses on a popular type of GNNs called graph convolutional networks (GCN) and the main contributions are as follows:

- We present gFADES as a graph neural network accelerator that uses a dataflow of dataflows (DoD) approach to compute the output features of the $l+1$ layer in a GCN: $H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{l}W^{l})$
  where W indicates the trainable weight matrix of layer l. H the input feature matrix for layer l and $\tilde{A}$ the normalize adjacency matrix. Each row of the input feature matrix H^{0} contains attributes or features for a node of the input graph. Each row of the output feature matrix $H^{(1)}$ is the embedding of the node in a lower dimension space.
- We demonstrate how gFADES performance can be scaled to adapt to the system bandwidth and compute availability with multiple hardware threads and multiple compute units.
- We explore new HLS features that enable the creation of high-throughput and efficient dataflow of dataflows (DoD) architectures.
- We present preliminary performance results and the integration flow as a high-performance Pytorch accelerator suitable for edge compute devices.

This paper is organized as follows: section 2 reviews related work. Section 3 describes the proposed DoD architecture for high-performance dense and sparse tensor operations. Section 4 focuses on the details of the hardware multi-threaded extensions. Section 5 performs a preliminary performance evaluation while section 6 discusses the Pytorch integration in a 2-layer CGN example network. Finally, section 7 concludes this paper and proposes future work.

II. Related work and Motivation

Over the last few years the interest on graph neural networks applications has increased significantly and the topic of hardware acceleration has started to receive widespread attention. These accelerators typically consider that the aggregation phase consists of sparse x dense matrix operation with an sparse adjacency matrix while the combination phase is a dense x dense operation with a dense feature matrix [2]. In [3] the authors indicate that in many applications

This research was partially funded by the Wallenberg AI autonomous autonomous systems and software (WASP) program funded by the Knut and Alice Wallenberg Foundation

the input features contain significant levels of sparsity and propose a sparse block strategy for these cases. The input feature matrix is encoded in CSR (Compressed Sparse Row) format with coarse-grained blocks of zeros that can be by-passed. The proposed systolic architecture allows the design to adapt the computing performance to the available input/output bandwidth. The design targets an ASIC technology and uses internal memory to preload the weight matrix in the systolic array which is not possible in embedded FPGA devices due to the limited amount of internal BRAM memory. The hardware also loads rows of the feature matrix selectively when they correspond to non-zero elements in the adjacency. This implies that to obtain high hardware utilization once the sparse value is detected as a non-zero, the loading of multiple feature values must be performed in one clock cycle to avoid stalling the compute engine. Instead this work uses streaming techniques to stream sparse and dense feature values processing multiple tensor columns in parallel. We have observed that although the adjacency matrix is very sparse, each row uses different elements of the feature column and eventually all adjacency data is involved in the computation. The recoding in [3] of input features into blocks on non-zeros will loose fine-grained detail in the input features. Our hardware treats the sparsity of the feature matrix in the same way as the adjacency matrix so no blocking is needed. It can switch to dense processing in a single clock cycle when processing the dense hidden feature layers of the graph neural network. Other recent work for GNN processing includes GCNAX [4] that computes aggregation and combination in two separate phases to take advantage of the sparseness of the adjacency matrix and the possible sparseness of the input features of the first layer of the GCN. The authors in [4] buffer the intermediate dense matrix resulting of the combination phase and pass it to the aggregation engine. The design employs 16 MAC array and also targets an ASIC technology with performance and energy parameters estimated using a synthesised design. In GraphACT [5], a hybrid CPU-FPGA platform targeting large scale Xilinx Alveo cards equipped with HBM memory is presented focusing on the acceleration of large graph training. The hardware is based on a systolic array and the training algorithm is optimized to fit the constraints of the hardware. The paper does not provide details on logic complexity or design methodology focusing on a graph theory to improve hardware mapping. The paper reports a DSP utilization of 5632 cores and it is shown to outperform an NVIDIA tesla GPU by 10% to 30% for different datasets. Also using large scale Alveo cards and large scale graph acceleration the hardware in [6] proposes a pre-processing stage that performs graph sparsification to reduce the number of edge connections and node reordering to increase data locality. This reduces the required size of on-chip memory. The research treats the feature matrix as a dense matrix for all the layers in the network and introduces a two mode strategy to change the order of the combination and aggregation stages (1) $(AH)W$ or (2) $A(HW)$. Their analysis shows that (2) has lower computation when next layer feature vector is shorter than the current layer. Our hardware always uses mode (2) because both A and H can be sparse and therefore both the aggregation and combination stages have the opportunity to work in sparse mode. Also, in the 2-layer GCN evaluated we have observed so far that the feature vector size decreases after the first layer.

Compared with previous work, we focus on resource constrained devices operating at the edge such as the Zynq and Zynq ultrascale family that lack high-bandwidth memory features. We also aim at integrating the accelerator as part of the Pytorch framework so it can be used as a drop-in replacement for the sparse/dense computation libraries currently available in Pytorch. The design focuses on streaming data with independent dataflow stages to optimize the limited memory bandwidth available and keep the compute engines busy. We consider fine-grained sparse adjacency matrices and sparse/dense feature matrices.

III. Dataflow description

The dataflow combines hardware engines for aggregation and combination stages that correspond to adjacency and feature matrix processing respectively. Each of these engines can instantiate a variable number of hardware threads and compute units depending on the required level of performance and available bandwidth. The dataflow of a single thread is shown in Figure 1 and it has been fully described using Xilinx Vitis HLS (High-Level-Synthesis) toolset. In the HLS description a dataflow of dataflows (DoD) is created with a new HLS 2022 feature called $streamofblocks$ that enables the selective read_lock and write_lock of the PIPO (Ping-Pong) buffers that join the combination and aggregation stages and ensure that both engines can run in parallel with both single and multi-threaded hardware configurations. The combination and aggregation engines use fine-grained intra-dataflow stages that are triggered by data words with bit-widths that depend on the selected data type (e.g. 16-bit half, 16-bit fixed, 32-bit float etc). In Figure 1 we can see how multiple FIFOs whose number depend on the number of compute units in each stage join the different processing stages of the fine-grained dataflow. The coarse-grain inter-dataflow between both engines has a data granularity that consists of tiles with dimensions that depend on the number of compute units and the number of hardware threads. These tiles ensure high-throughput by keeping all stages in the dataflow active. In Figure 1 we can see how a PIPO joins the different processing stages of the coarse-grained dataflow. Multiple PIPOs are needed in multi-threaded configurations as seen in Figure 2. The weight matrix is always considered to be dense while the adjacency matrix is always sparse in CSR format. The feature matrix is available in sparse mode for the first layer and in dense modes for the hidden layers.

A. Combination engine

The combination engine computes $FEA * W$ where FEA represents the feature matrix and W the weight matrix and generates a dense matrix output B in chunks for the aggregation engine that computes $ADJ * B$ where ADJ represents

the adjacency matrix. This engine consists of two main stages that read sparse or dense feature data and compute the dot product. Sparse mode in the combination engine is generally used in the first graph layer where the size of the feature matrix is large and significantly sparse. In the second or subsequent layers the feature matrix is the output from the previous layer and dense. Notice that performing a conversion from the dense output matrix into a sparse CSR matrix at run-time would represent a significant software overhead for no significant gain. To avoid these issues the combination engine is set to run in dense mode and the feature values port is used to read the dense input feature matrix while the rowptr and column_index feature ports remain in idle mode. The read stage is activated after loading a w matrix tile with a column count that equals the number of compute units and row count that equals the number of rows present in the w matrix. It then starts streaming elements of the FEA matrix in CSR (Compressed Sparse Row) format with *column_index* and *non-zero* values in sparse mode. This means that in sparse mode, the accelerator performs a variable number of reads of feature values and column indices per matrix. This does not represent a performance limiting factor in the dataflow architecture because the READ stage is independent of the other stages, and it will run at full speed over all the data elements present in the FEA matrix as long as there are no output FIFO overflows.

The compute stage is the main computing loop that instantiates the bulk of the DSP blocks. It aims at activating all compute units (CUs) in parallel in each clock cycle. This is the case for all w tiles with a number of columns equal to the number of CUs. Typically, the last tile contains a number of columns lower than the number of compute units, and in this scenario, some of the CUs do not write their output FIFOs. This enables support for arbitrary matrix shapes that are not a multiple of the tile size.

B. Aggregation engine

The aggregation engine is used always in sparse mode since the same adjacency values are used for the first and second graph layers. The aggregation engine contains read,compute, scale and write stages. The read stage is very similar to the read stage of the combination engine but it lacks the logic needed to compute in dense mode. It streams values and column indices while it internally computes the number of non-zeros present in each row using the rowptr data. All this information is streamed into FIFOs that are used by the compute stage. The compute stage has as inputs the PIPOs that are used by the combination engine to write its results and the FIFOs with the adjacency matrix data. The scaling stage is needed for low bit widths as presented in our previous work targeting Tensorflow Lite [1] and convolutional neural networks. In this initial analysis for graph neural networks the data types are 16-bit floating-point and do not require scaling so this stage is not enabled. The write stage reads the FIFOs from each compute engine and writes the results to main memory.

C. Multi-threaded extensions

To exploit the additional bandwidth and compute performance available in the Zynq ultrascale device the number of working hardware threads is configurable at compile time. Each hardware thread is assigned a number of rows of the adjacency and feature matrices while having access to the same weight data. Each hardware thread has independent ports connected to the multiple high-performance AXI ports available in the Zynq device.

Figure 2 shows examples of multi-threaded configurations where multiple data ports and hardware threads are used to stream and process the CSR matrices column_index, row_pointer and values. As seen in previous work, feature processing tends to be more compute intensive than adjacency processing so in many applications a configuration with a higher number of threads for the combination engine compared with the aggregation engine is beneficial as shown in Figure 2c. The number of PIPO buffers connecting aggregation and combination stages is determined by the number of threads. The figure shows how each combination thread writes the same output to a number of PIPOs equal to the number of aggregation threads. Then, each aggregation thread reads from a number of PIPOs equal to the number of combination engines. Each of these PIPOs contains different data and are needed to complete all the rows needed for the adjacency tensor processing. This organization ensures that all the compute units can write and read data in parallel without dataflow stalls and overcomes the limited number of read/write ports available in the BRAMs that build the PIPOs. Notice the each thread contains a number of CUs that is always a multiple of 2 to utilize efficiently the double read/write ports available in Xilinx BRAMs. Table I shows the memory and logic complexity for the different configurations for a weight matrix size with up to 20480 rows. The target device is the Zynq Ultrascale+ XCZU28DR available in the RFSoC 2x2 board with a programmable logic clock frequency of 250 MHz. Notice that the maximum row count of the weight matrix is in this case limited to 20480 but this is also limited by the hardware need to allocate physically contiguous memory in main memory to hold the matrix data. A way to overcome this size limitation is to introduce a new level of software tiling for very large arrays using the processor to invoke the accelerator multiple times using an approach as indicated the listing below:

Listing 1: Float compute kernel example

```
1
2
3  for (int c = 0; c < C; c++)
4    for (int m = 0; m < M; m++)
5      for (int n = 0; n < N; n++)
6        for (int k = 0; k < K; k++)
7          D(m,c)+=A(m,n)*H(n,k)*W(k,c);
```

Where D(x,y), A(x,y), H(x,y), W(x,y) represent tiles of compatible dimensions of the original matrices.

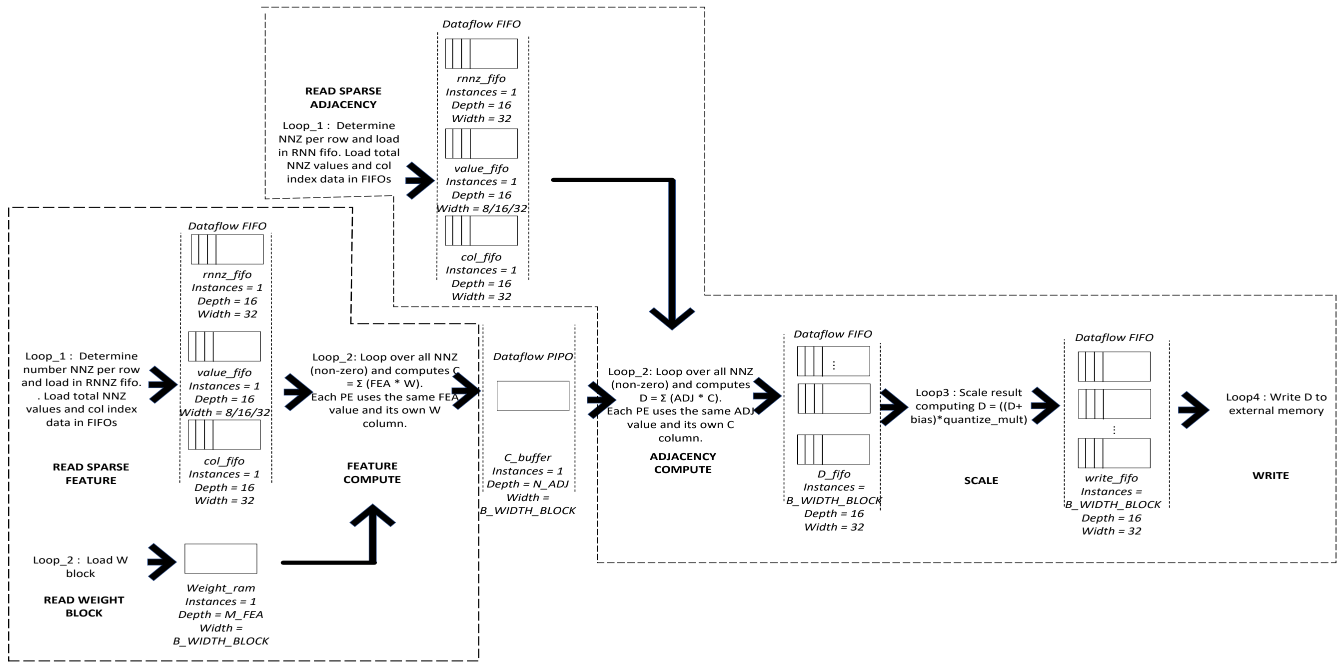
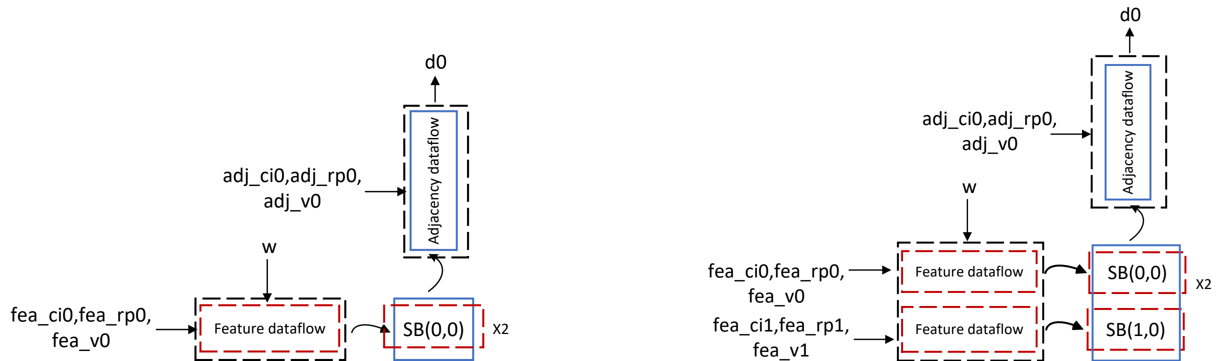
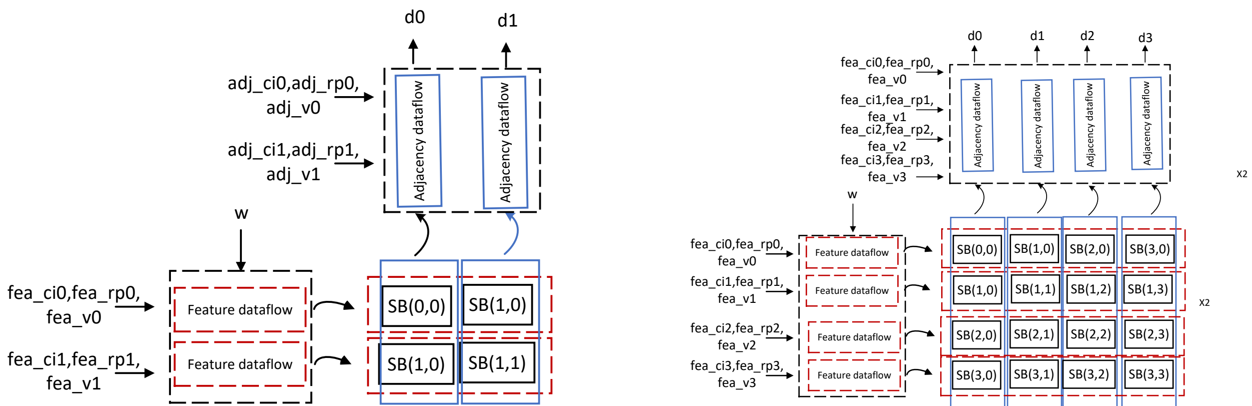


Fig. 1: Single-threaded dataflow of dataflows description



(a) 1 feature engine and 1 adjacency engine

(b) 2 feature engines and 1 adjacency engine



(c) 2 feature engines and 2 adjacency engines

(d) 4 feature engines and 4 adjacency engines

Fig. 2: Examples of hardware multi-threaded configurations

Configuration	LUTs(K)	FFs(K)	BRAM_18Ks	DSP48Es
(1t1t2c,half)	21.7	29.7	58	24
(2t2t4c,half)	38.0	50.1	345	76
(2t2t8c,half)	46.4	58.2	537	140
(1t1t16c,half)	35.7	44.15	645	136
(2t1c16c,half)	49.43	57.62	999	200
(4t2t8c,half)	65.71	77.56	1068	204

TABLE I: configuration complexity comparison

IV. Preliminary performance evaluation

To evaluate the performance of the design we initially select the citeseer dataset. The CiteSeer dataset consists of 3327 nodes where each node means a scientific publication and edges that mean citation relationships. Each node has a predefined feature vector with 3703 values that indicate the absence or presence of a corresponding dictionary word in the publication. In total there are 12431 non-zeros in the adjacency matrix indicating links between nodes and 105165 non-zeros in the feature matrix indicating the presence of words in the nodes. The dataset is designed for node classification tasks and the objective is to predict the category of unknown publications. Table II summarizes the statistics of the dataset and the density of the adjacency and feature matrix. In this preliminary performance evaluation we focus on a single GCN layer with 21 hidden features that represent the features generated by the layer for each node and also the output of the node that will be propagated to the next layer.

Dataset	nodes	dens. adj	inp features	dens. fea	hidden features
Citeseer	3327	0.11%	3703	0.85%	21

TABLE II: configuration complexity comparison

The statistics show that both adjacency and feature matrix are significantly sparse and this can be effectively exploited by the gFADES accelerator. Figure 3 shows the performance obtained on this dataset for the hardware configurations presented in Table I. The reference implementation running on the CPU uses Pytorch as originally presented in [7] and the code is illustrated below.

Listing 2: GCN pytorch layer in CPU

```
1 support = torch.spmm(input, self.weight
      ↪ )
2 output_cpu = torch.spmm(adj, support)
```

We can see that in this initial analysis all the hardware configurations show promising performance improvements but additional verification is required with other datasets. Also, additional work is required to test the performance of the accelerator against other devices such as embedded GPUs.

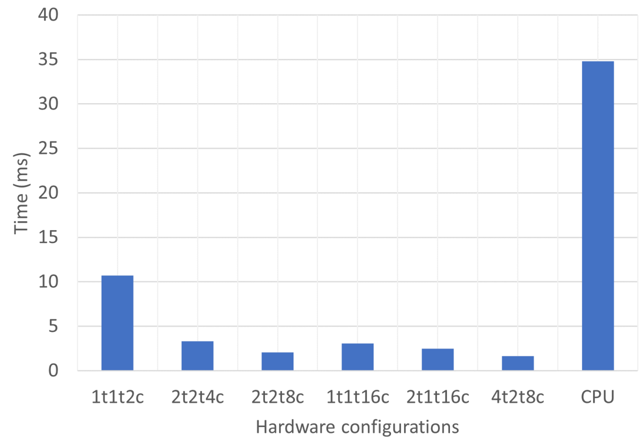


Fig. 3: Performance evaluation on citeseer dataset

V. Pytorch integration

The gFADES accelerator is implemented as a PYNQ overlay integrated in the Pytorch machine learning framework. We use a PYNQ 2.7 image that runs Ubuntu 22.04 and install Pytorch 1.9 on the RFSoC 2x2 board from the original sources. PYNQ enables full control of the accelerator from a python environment. PYNQ uses numpy arrays as the data buffers for the accelerator. Numpy arrays of contiguous memory are allocated in the python script to store the input and output data for the accelerator. The accelerator is then configured with the addresses of these buffers. Any additional IP control registers are also written such as those indicating dense or sparse mode and matrix sizes. Finally, a run kernel script starts the IP block by setting the AP_START bit to 1 and checks when the accelerator completes reading the AP_DONE bit. Pytorch uses torch tensors to store its data that in addition to the values store additional information such as requires_grad used to compute derivates automatically during the backward pass. These torch tensors are similar to numpy arrays and conversion between both data types is possible reusing the same memory without explicit data copying. This simplifies and optimizes the integration of PYNQ and Pytorch. For example:

Listing 3: Torch tensor and acceleerator PYNQ numpy arrays integration

```
1
2
3 from pynq import allocate
4 import torch
5 #allocate numpy array suitable for
      ↪ accelerator calls
6 B_buffer = allocate(shape=(P_w,M_fea),
7 dtype=np.float16)
8 #Obtain Torch tensor
9 torch_B_buffer=torch.from_numpy(
      ↪ B_buffer)
10 #configure IP register with numpy
      ↪ pointer address
11 my_ip.register_map.B_offset_1 =
      ↪ B_buffer.physical_address
12 # other register configuration and
      ↪ memory allocation omitted for
      ↪ clarify.
13 #run hardware kernel
14 run_kernel()
```

The obtained torch_B_buffer tensor and B_buffer numpy array can then be used in the rest of the algorithm. We perform an initial test of the accelerator with a GCN consisting of two layers with a first layer with 32 hidden units and a second layer with 16 hidden units feeding a final fully connected layer that outputs 7 possible classes. The hardware configuration consists of 1 aggregation thread, 1 combination tread and 16 compute units. The following listing shows and excerpt from the integration of the accelerator in the layersṗy script from [7]. When the accelerator is active (i.e. acc=1) we initially set the hardware registers to point to the layer parameters. Then, we activate the accelerator with run_kernel() and obtained the output from D_buffer. The first layer runs the hardware in full sparse mode and both feature and adjacency matrix are processed in sparse mode. On the other hand, the second layer has as input the output feature matrix generated by the first layer. This second feature matrix is now dense and the accelerator uses a dense mode that means that the adjacency matrix is still sparse but the feature matrix is dense.

Listing 4: Pytorch acceleration with gFADES

```
1        if (acc==1):
2            print("Running gFADES")
3            self.my_ip.register_map.
               ↪ M_fea=self.
               ↪ in_features
4            self.my_ip.register_map.P_w
               ↪ =self.out_features
5            self.my_ip.register_map.
               ↪ gemm_mode=dense
6            self.my_ip.register_map.
               ↪ D1_offset_1 =
               ↪ D_buffer.
               ↪ physical_address
7            self.my_ip.register_map.
               ↪ values_fea1_offset_1
               ↪ = values_fea_buffer.
               ↪ physical_address
8            self.my_ip.register_map.
               ↪ B_offset_1 = B_buffer
               ↪ .physical_address
9            self.run_kernel()
10           output_acc = D_buffer
11           output_acc = torch.
               ↪ from_numpy(output_acc
               ↪ )
12           output_acc = torch.tensor(
               ↪ output_acc,dtype=
               ↪ torch.float32)
13           output = output_acc
14       else:
15           print("Running CPU")
16           support = torch.mm(input,
               ↪ self.weight)
17           output_cpu = torch.spmm(adj
               ↪ , support)
18           output = output_cpu
```

The CPU uses all 4 CPU cores available to compute but we observe a performance improvement in hardware for both layers. The pure sparse mode reduces processing from 41.58 ms to 2.34 ms while the hybrid sparse/dense mode reduces processing from 6.63 ms to 1.63 ms.

VI. Conclusions

In this paper we have presented the gFADES dataflow hardware architecture for graph convolutional networks. The gFADES architecture is highly configurable in terms of logic and bandwidth requirements and suitable for edge FPGA devices. Preliminary performance and functional results following the integration into the Pytorch machine learning framework show good acceleration both in pure sparse and in hybrid sparse-dense mode. Future work includes testing additional data sets, hardware configurations and quantization strategies for low bit-width data types. Also, we intend to streamline the integration process with Pytorch/Tensorflow and investigate how gFADES can be used in the backward pass to accelerate training in addition to the forward pass. Finally, investigating how gFADES can be scaled up to non-Zynq devices equipped with HBM (high bandwidth memory) is also a worthwhile avenue of research.

References

[1] J. Nunez-Yanez, "Fused architecture for dense and sparse matrix processing in tensorflow lite," *IEEE Micro*, vol. 42, no. 6, pp. 55–66, 2022.

[2] R. Garg, E. Qin, F. Muñoz-Martínez, R. Guirado, A. Jain, S. Abadal, J. L. Abellán, M. E. Acacio, E. Alarcón, S. Rajamanickam, and T. Krishna, "Understanding the design-space of sparse/dense multiphase gnn dataflows on spatial accelerators," 2021.

[3] C. Peltekis, D. Filippas, C. Nicopoulos, and G. Dimitrakopoulos, "Fusedgcn: A systolic three-matrix multiplication architecture for graph convolutional networks," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 93–97, 2022.

[4] J. Li, A. Louri, A. Karanth, and R. Bunescu, "Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 775–788, 2021.

[5] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, (New York, NY, USA), p. 255–265, Association for Computing Machinery, 2020.

[6] 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)*, pp. 61–68, 2020.

[7] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016.