Accelerating Graph Neural Networks in Pytorch With HLS and Deep Dataflows

⋆

Jose Nunez-Yanez

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

Abstract. Graph neural networks (GNNs) combine sparse and dense data compute requirements that are challenging to meet in resource-constrained embedded hardware. In this paper, we investigate a dataflow of dataflows architecture that optimizes data access and processing element utilization. The architecture is described with high-level synthesis and offers multiple configuration options including varying the number of independent hardware threads, the interface data width and the number of compute units per thread. Each hardware thread uses a fine-grained dataflow to stream words with a bit-width that depends on the network precision while a coarse-grained dataflow links the thread stages streaming 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. Results based on the citation networks show a performance gain of up to 140x with multi-threaded hardware configurations compared with the optimized software implementation available in Pytorch. The results also show competitive performance of the embedded hardware compared with other high-performance state-of-the-art hardware accelerators.

Keywords: neural network, FPGA, sparse, HLS, GNN, Pytorch

1 Introduction

GNNs perform tasks such as graph classification, node classification, link prediction or graph clustering and have been very successful in applications such as anomaly detection, bioinformatics and cybersecurity where data can be interpreted as graphs with a non-euclidean structure. Also in natural language processing GNNs have been shown to be a generalization of the Transformer [1] that achieves state-of-the-art performance in machine translation tasks. More recently , GNNs have started to appear in domains traditionally reserved to CNNs or RNNs such as video object detection [2]. GNN processing uses both dense

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

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. 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 results on creating a dataflow architecture for graph neural networks using high-level-synthesis and its integration into PYNQ and Pytorch. This 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 targeting resource-constrained SoC devices.
- We optimize the HLS description to handle the extreme sparsity found in graph neural networks and explore new HLS features that enable the creation of high-throughput and efficient dataflow of dataflows (DoD) architecture.
- We present performance results compared with other state-of-the-art hardware and discuss the integration flow as a Pytorch accelerator suitable for edge compute devices.
- We make our designs and framework open-source at: `https://github.com/eejlny/gemm_spmm`

This paper is organized as follows: section 2 reviews related work and presents the current motivation. Section 3 describes the proposed DoD architecture with data access and processing optimizations for high-performance dense and sparse tensor processing that are then further refine in section 4. Section 5 focuses on the details of the hardware multi-threaded extensions. Section 6 discusses the Pytorch integration while 7 performs a performance evaluation in a 2-layer GCN example network. Finally, section 8 concludes this paper and proposes future work.

2 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. The GNN type that has received more attention for

hardware acceleration is the GCN (Graph Convolutional Network) that can be expressed as a combination of dense and sparse matrix multiplications. Other GNN models such as GAT (graph attention networks) or GIN (Graph isomorphism network) add attention mechanisms that disrupt the matrix processing [3]. The authors of [3] also consider edge features to the node features to create an accelerator for general graph neural networks but this results in a lower throughput if applied to simpler GCNs. GCN 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 [4]. In [5] the authors indicate that in many applications 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 bypassed. The proposed systolic architecture allows the design to adapt the computing performance to the available input/output bandwidth. In GCNAX [6] the computation of aggregation and combination is done 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 [6] buffer the intermediate dense matrix resulting of the combination phase and pass it to the aggregation engine. The design employs 16 MAC array and targets an ASIC technology with performance and energy parameters estimated using a synthesised design. In GraphACT [7], 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 Alveo cards the hardware in [8] 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. A two mode strategy to change the order of the combination and aggregation stages (1) $(AH)W$ or (2) $A(HW)$ shows that (2) has lower computation when the 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. In [9] the authors also consider that feature processing consist of a dense and regular pattern and use a MLP (multi-layer-perceptron) for this stage. The hardware is described in Verilog and targets a large VCU128 equipped with HBM memory and several MBytes of device memory. They investigate an edge-level processing strategy that computes an edge a time and requires retraining and processing while our

strategy processes sub-graphs as used directly in by the reference Pytorch implementation [10]. The possible workload unbalance resulting from vertices with different number of edges is dealt with by assigning bigger chunks matrices with higher sparsity to the corresponding thread. In [11] the authors also use Verilog to target a large Stratix 10 SX FPGA. They indicate that GNNs have highly unbalanced non-zero data distributions in the adjacency matrix and propose a hardware accelerator with auto-tuning that monitors the workload and balances the distribution. Similarly to our work the processing of the feature matrix precedes the adjacency matrix and it is more computationally intensive. The authors in [12] explore the potential of the latest Versal family working with graph layers that combine subgraphs with different levels of sparsity. The subgraphs with lower levels of density are processed by the VLIW-Vector multiprocessor system while subgraphs with scattered nodes are processed the FPGA fabric. A retraining strategy is used to group and reorder vertices increasing the density of the graph structure. The lack of real hardware means that the evaluation is based on simulations that indicate good energy efficiency and performance although no complexity results are available. The research done in [13] proposes a sophisticated graph reordering technique to cluster graph nodes into islands with the objective of improving data locality and reused. The system is demonstrated in a large Stratix 10 SX FPGA with 4096 PEs (Processing Units) with floating-point MAC units. Dedicated BFS (Breadth-First Search) engines are used at run-time to locate the islands that are then forwarded to idle PEs for performing its combination and aggregation jobs. The previous state-of-the-art review indicates that there is significant efforts on accelerator design for large scale FPGAs and ASICs but less focus on edge devices with limited computing and bandwidth resources. Therefore, 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 to facilitate ease-of-use as a drop-in replacement for the sparse/dense computation libraries available in Pytorch. The design focuses on streaming data with independent dataflow stages to optimize the limited memory bandwidth available and keep the compute units busy. We consider fine-grained sparse adjacency matrices and sparse/dense feature matrices.

3 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 2 and it has been fully described C with Xilinx Vitis HLS (High-Level-Synthesis) tools. 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 creating a inter-dataflow between aggregation and

combination so that both engines can run in parallel with both single and multi-threaded hardware configurations. Internally the combination and aggregation engines use a fine-grained intra-dataflow that streams 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 2 we can see how multiple FIFOs, which total number depends on the number of compute units in each stage, join the different processing stages of the fine-grained dataflow. The coarse-grained inter-dataflow 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 2 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 5. The weight matrix is always considered to be dense while the adjacency matrix is always sparse and coded in CSR format. On the other hand, the feature matrix is available in CSR sparse mode for the first layer and in dense modes for the hidden layers. Figure 1 shows how the coarse-grained dataflow enables adjacency and feature tiles to be processed in parallel and that, for the considered datasets, adjacency tiles are processed faster than feature tiles. On the other hand, the last adjacency tile execution cannot not overlap and therefore adjacency tile processing needs to be also optimized to not impact overall performance.

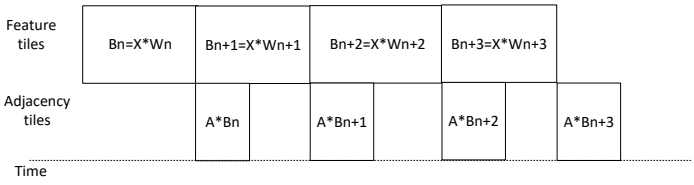


Fig. 1: Coarse-grained dataflow tiling

3.1 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 (or tiles) 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. We have verified that this is the case even taken into account the non-linear RELU function applied to the outputs from the first layer with observed

densities between 75% and 90% for the evaluated datasets. For this reason the combination engine is set to run in dense mode after the first layer. In this dense configuration the feature values port is used to read the dense input feature matrix values while the rowptr and column_index feature ports remain in idle mode.

3.2 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 [14] and convolutional (not graphs) neural networks. In this paper we present results with a 16-bit floating-point that does not require scaling so this stage is not enabled. We leave the evaluation of other data types and scaling strategies for future work. The write stage reads the FIFOs from each compute engine and writes the results to main memory.

4 Dataflow optimization

Fig. 3 illustrates how non-zero values are broadcast to all the compute units available in the compute stage. The figure shows zeros as empty squares intermixed with sample non-zero values. The compute stage processes all non-zero data of a row in the sparse matrix before moving to the next one. The compute stage loop that processes each row achieves a initialization interval of one clock cycle (so a new iteration of the loop can be started in each clock cycle) and has a latency of four clock cycles. This latency results from the need to flush the pipeline and restart the loop after each row. This latency does not represent a significant overhead as long as the number of iterations in the loop is large (i.e. iterations ¿¿ latency) but, in graph neural networks, sparsity means that a row with thousands of elements could contain just a few non-zeros (e.g. less than 10). This extreme sparsity means that the latency overhead is significant and can result in a significant performance degradation. In addition, in the targeted Zynq family the latency of a floating point add (FADD) is larger than one and the architecture deals with this by interleaving multiple accumulations in a single MAC unit with independent accumulation registers. The efficiency of this floating point configuration also degrades if this FADD pipeline only has a few non-zero values in the row before it needs to be restarted. To address this problem we group a few rows of the sparse matrix in a single sub-block to reduce the

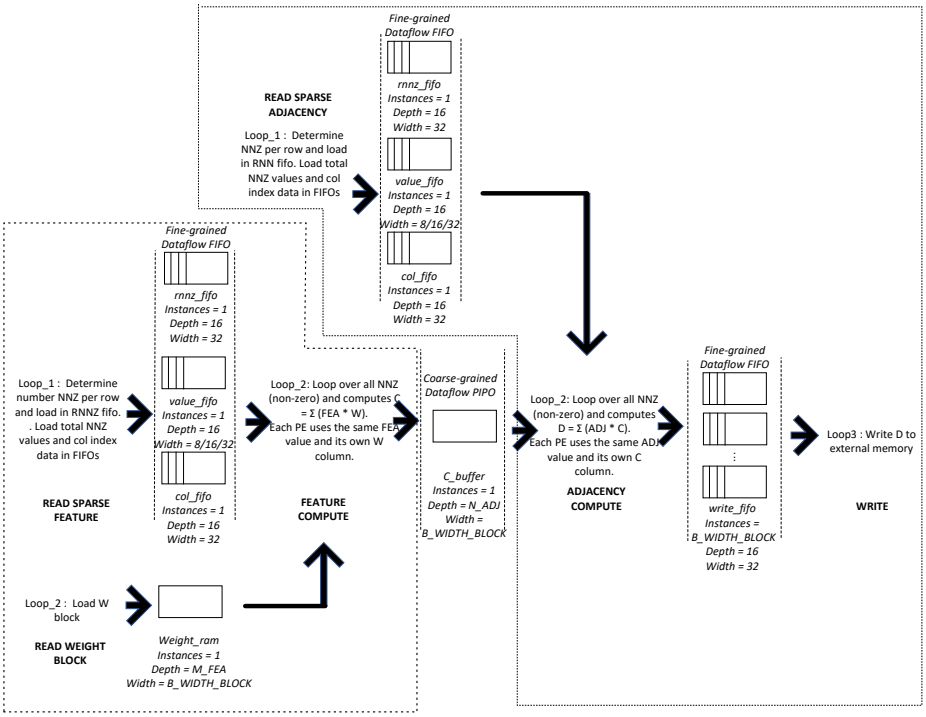


Fig. 2: Single-threaded dataflow of dataflows description

number of row restarts. The challenge is how to distribute the results obtained from the block to the corresponding individual rows. In order to that we create intervals accumulating the number of non-zeros in each of the rows of the sub-block. Then, the hardware can use these intervals to assign the compute loop result to the corresponding row. The FIFOs supplying the number of non-zeros in each row to the compute stage are re-organized to store these accumulations that create the intervals. The compute stage obtains multiple row results in parallel that are then assigned to the correct row result using the current iteration of the loop as an index that points to a single non-zero interval as illustrated in Fig. 4 with an example of sub-blocks formed by four rows.

Dataset	sub-block1	sub-block2	sub-block4	sub-block8
citeseer	5.95	4.26	4.07	3.98
cora	4.7	2.21	2.06	1.99
pubmead	40.7	35.2	34.1	33.5

Table 1: Sparse sub-block execution in ms

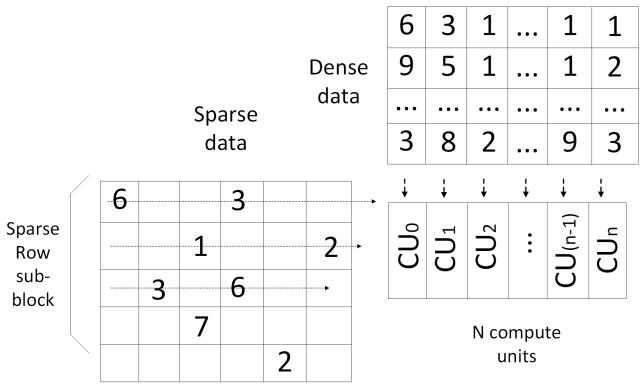


Fig. 3: Sparse sub-block throughput optimization

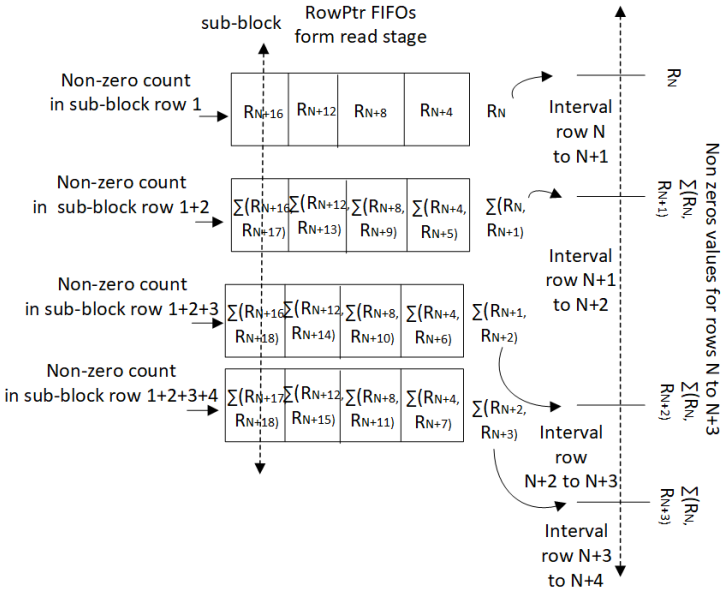


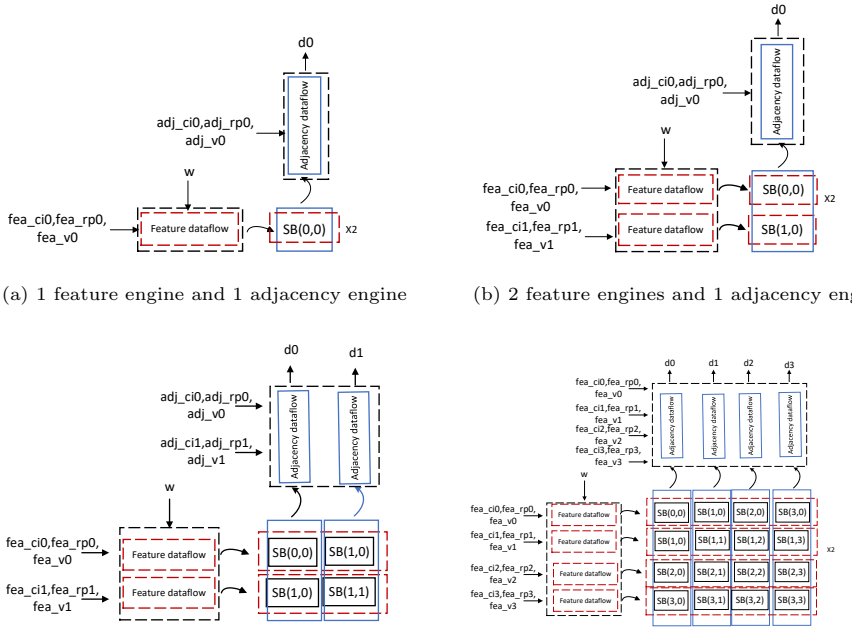
Fig. 4: Sparse sub-block non-zero intervals

Table 1 evaluates the effect of the proposed sparse matrix sub-blocking in execution time for the considered datasets. The dataflow is configured with 2 compute units in the feature and adjacency compute stages. Sub-block1 corresponds to a single row in the sub-block (i.e. original approach with no sub-blocking). It is clear that for all the data sets sub-blocking reduces execution time significantly and for the rest of this paper a sub-block size of 4 is selected to balance performance and complexity. Larger sub-blocks group more sparse rows together but this results in more complex interval logic needed to extract individual row results from the sub-block computation.

5 Multi-threaded extension

To exploit the memory 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 device. Figure 5 compares a base configuration with one thread for adjacency and feature processing with multi-threaded configurations with up to 4 threads per engine. The ports *column_index*, *row_pointer* and *values* are used to stream the CSR matrices, w carries dense weight data and d carries the output to main memory. As seen in previous work, feature processing tends to be more compute intensive than adjacency processing so a configuration with a higher number of threads for the combination engine compared with the aggregation engine is supported as an option to better distribute the available hardware resources. The number of PIPO buffers that connect aggregation and combination stages is determined by the number of threads with a power of two relation. 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 process all the rows involved in 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 are used to create the PIPOs. Notice that in addition the each thread contains a number of compute units that is always a multiple of 2 to utilize efficiently the double read/write ports available in Xilinx BRAMs. Therefore the number of BRAMs present in each PIPO depends on the tile size and also the number of compute units present in each hardware thread.

Table 2 shows the memory and logic complexity for different configurations for a weight matrix size with up to 20480 rows. The configuration name is shown as $XtYtZc$ with X/Y indicating the number of threads in feature and adjacency respectively and Z the number of compute units per thread. The target device is the Zynq Ultrascale+ XCZU28DR available in the RFSoC 2x2 board and the design uses a clock frequency of 250 MHz. Notice that the maximum row count



(a) 1 feature engine and 1 adjacency engine

(b) 2 feature engines and 1 adjacency engine

(c) 2 feature engines and 2 adjacency engines

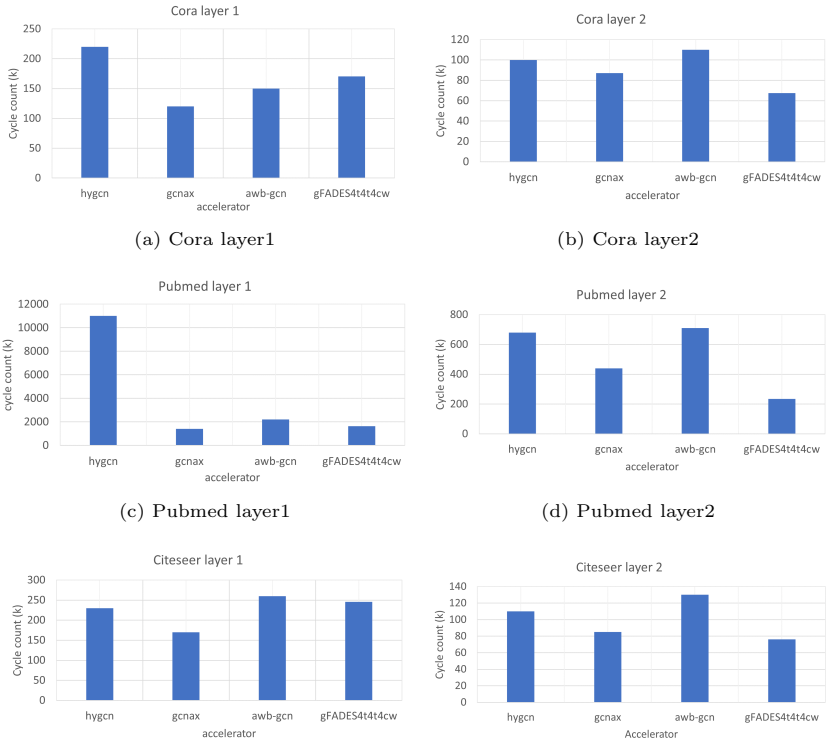
(d) 4 feature engines and 4 adjacency engines

Fig. 5: Examples of hardware multi-threaded configurations

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 large arrays using the processor to invoke the accelerator multiple times.

Configuration	LUTs(K)	FFs(K)	BRAM_18Ks	DSP48Es
(1t1t2c)	22.6	29.4	109	22
(1t1c8c)	27.2	31.2	421	34
(1t1t16c)	33.7	33.6	813	50
(2t2t4c)	40.0	47.4	425	48
(2t2t8c)	48.1	50.4	841	64
(4t4t4c)	79.7	84.03	848	92
(4t2t8c)	71.5	74.2	1036	334

Table 2: configuration complexity comparison



(a) Cora layer1 (b) Cora layer2

(c) Pubmed layer1 (d) Pubmed layer2

(e) Citeseer layer1 (f) citeseer layer2

Fig. 6: Performance comparison

6 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 runing 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 and it 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 one 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 1.1: 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 tests of the accelerator with a GCN consisting of two layers as presented in [10] with a first layer with a fixed number of 16 hidden units and a second layer whose hidden units depend on the classification classes of the data set. 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 after a *RELU* function. This second feature matrix is now dense and the accelerator uses a hybrid mode where the adjacency matrix is still sparse but the feature matrix is dense.

Listing 1.2: 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
```

```
            self.run_kernel()
            output_acc = D_buffer
            output_acc = torch.from_numpy(output_acc)
            output_acc = torch.tensor(output_acc,dtype=
                ↪ torch.float32)
            output = output_acc
        else:
            print("Running CPU")
            support = torch.mm(input, self.weight)
            output_cpu = torch.spmm(adj, support)
            output = output_cpu
```

7 Performance evaluation

To evaluate the performance of the design we select the popular Cora, Citeseer and Pubmed datasets. These datasets are designed for node classification tasks and the objective is to predict the category of unknown publications. Table 3 summarizes the statistics of the datasets and the density of the adjacency and feature matrix. The second layer receives as input the features output from the first layer and the density shown in Table 3 has been measured after the application of the RELU function.

Dataset	Node count	density adjacency	Feature count	density features layer1	density features layer2
cora	2708	0.18%	1433	1.27%	76.7%
citeseer	3327	0.11%	3703	0.85%	90.5%
pubmed	19717	0.028%	500	10%	84.7%

Table 3: Dataset characteristics

Table 4 shows the performance obtained on these dataset for the hardware configurations presented in Table 2. The Cortex A53 CPU results are based on Pytorch *torch.spmm/mm* sparse and general matrix as originally presented in [10] and shown in the listing below.

Listing 1.3: GCN layer on CPU

```
support = torch.mm(input, self.weight)
output = torch.spmm(adj, support)
```

Table 4 shows that the higher hardware acceleration is obtained with the 4t4t4c configurations that outperforms the configuration 4t2c8c that sacrifices 2 adjacency threads to increase the number of compute units from 4 to 8. The hardware acceleration over the CPU version ranges from 140x to 26x for the different

configuration	citeseer l1/l2	cora l1/l2	pubmed l1/l2
1t1t2c	4.07/0.97	2.06/0.87	34.1/3.6
2t2t2c	2.23/0.54	1.21/0.49	17.9/1.87
2t2t4c	1.43/0.53	0.92/0.48	10.06/1.83
4t2t8c	1.27/0.51	0.96/0.48	6.87/1.58
4t4t4c	0.98/0.30	0.68/0.27	6.56/0.93
CPU	158/4.29	53.5/5.1	184.6/31.8

Table 4: Layer 1/2 execution time in milliseconds

data sets. As seen in Table 2 4t4t4c uses fewer memory and DSP blocks but it requires more logic than 4t2c8c. In principle, the overlapping between adjacency and feature processing and the lower computational complexity of the adjacency means that having fewer adjacency threads and more compute units could be beneficial. In the considered data sets this does not compensate the additional time required to process the final adjacency tile that does not overlap as seen in 1. Finally, Figure 6 compares the performance of the gFADES architecture with state-of-the-art FPGA-based GNN accelerators reviewed in section 2. These accelerators target high-end FPGA devices or ASICs and deploy HBM (high-bandwidth memory) that is not available in our resource constrained device. To make the comparison fairer we use the accelerator performance data taking verbatim from [6]. In [6] the architectures are normalized by using a 16 MAC array which will be equivalent to a 4t4t4c configuration or 4 hardware threads with 4 PE cores per thread stage. The results show that although gFADES targets an embedded and simpler architecture, it offers competitive performance. The explanation is that the simple and deep dataflow can exploit the limited bandwidth available in the Zynq device and that the pipeline optimizations enable up to 90% of performance of a theoretical ideal configuration where the DSP blocks never have to wait for data. In addition and compared to other architectures, the hybrid processing mode can combine dense-sparse processing in layer 2 with full fine-grained sparsity in layer1 optimally adapting to the level of sparsity present in the feature matrix.

8 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. 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.

References

1. C. Chen, Y. Wu, Q. Dai, H.-Y. Zhou, M. Xu, S. Yang, X. Han, and Y. Yu, "A survey on graph neural networks and graph transformers in computer vision: A task-oriented perspective," 2022.
2. K. Han, Y. Wang, J. Guo, Y. Tang, and E. Wu, "Vision gnn: An image is worth graph of nodes," 2022.
3. R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, (Los Alamitos, CA, USA), pp. 1099–1112, IEEE Computer Society, mar 2023.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. W. Yuan, T. Tian, Q. Wu, and X. Jin, "Qegcn: An fpga-based accelerator for quantized gcns with edge-level parallelism," *Journal of Systems Architecture*, vol. 129, p. 102596, 2022.
10. T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016.
11. T. Geng, A. Li, R. Shi, C. Wu, W. Tianqi, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. Herbordt, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," pp. 922–936, 10 2020.
12. C. Zhang, T. Geng, A. Guo, J. Tian, M. Herbordt, A. Li, and D. Tao, "H-gcn: A graph convolutional network accelerator on versal acap architecture," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, (Los Alamitos, CA, USA), pp. 200–208, IEEE Computer Society, sep 2022.
13. T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin, and A. Li, "I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 1051–1063, Association for Computing Machinery, 2021.
14. J. Nunez-Yanez, "Fused architecture for dense and sparse matrix processing in tensorflow lite," *IEEE Micro*, vol. 42, no. 6, pp. 55–66, 2022.