2024 34th International Conference on Field-Programmable Logic and Applications (FPL) | 979-8-3315-3007-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/FPL64840.2024.00034

A Software-Programmable Neural Processing Unit for Graph Neural Network Inference on FPGAs

Taikun Zhang*†, Andrew Boutros*, Sergey Gribok†, Kwadwo Boateng†, Vaughn Betz*
*Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada
†Programmable Solutions Group, Intel Corporation
{taikun.zhang, andrew.boutros}@mail.utoronto.ca, {sergey.gribok, kwadwo.boateng}@intel.com, vaughn@eecg.utoronto.ca

***Abstract*—Graph neural networks (GNNs) are a widely-used class of deep learning (DL) models for learning latent representations of graph-structured data for a variety of node/graph-level prediction tasks, some of which require real-time low latency inference. Most existing GNN accelerators rely on preprocessing input graphs on a host/embedded CPU to parallelize computations on different sub-graphs, making them unsuitable for real-time use cases. Others are extremely specialized streaming pipelines for only a specific type of GNN and therefore suffer from long FPGA bitstream compile times when the model is updated and cannot be used in applications that combine GNNs with other classes of DL models. In this work, we enhance the neural processing unit (NPU) FPGA overlay architecture, instruction set, and software stack to support a variety of GNN models. We achieve this without sacrificing the NPU flexibility; our enhanced NPU can be programmed purely through software to accelerate different GNNs or any of its originally supported DL workloads (e.g. MLPs, RNNs, GRUs, LSTMs). In addition, this flexibility enables our NPU software compiler to generate GNN kernels with different performance targets (throughput-optimized vs. latency-optimized) by exploiting different dimensions of compute parallelism on the same overlay architecture. Besides the flexibility benefits, our NPU implemented on an Intel Stratix 10 NX (14nm) FPGA can process 7.8× more graphs per second at a similar latency on average compared to a state-of-the-art model-specific FPGA accelerator targeting real-time applications on an AMD Ultrascale+ same-generation FPGA. It also achieves 5.8× higher throughput compared to an Nvidia RTX A6000 GPU (8nm) and 2.6× lower latency than a state-of-the-art accelerator that combines CPU-based graph preprocessing with AMD Versal (7nm) fabric and AI engine compute. Finally, we present a case study for using our enhanced NPU in real-time GNN-based multi-input multi-output (MIMO) antenna scheduling, highlighting that it meets the latency requirements of this task in 5G communication networks.**

I. Introduction

Many of the applications that we use daily are now powered by deep learning (DL) such as voice-controlled assistants, home surveillance systems, and automated customer service. Therefore, specialized accelerators are being widely deployed both in datacenters and at the edge to improve inference latency and energy efficiency, and keep up with the ever-increasing compute demands of DL models. Graph neural networks (GNNs) are DL models that can extract patterns and infer high-level trends in graph-structured data [1]. For example, they can classify a graph based on the structure and connections between its nodes, predict the likelihood of forming future connections between nodes, or label new graph nodes based on a pre-labeled subset of nodes. Graphs are ubiquitous in many applications; graph nodes can represent users in a social network or customers/products in an online retail, while graph edges can describe the relations or interactions between them. GNNs are already widely used in production for travel time prediction in Google Maps [2], customer response prediction in DiDi [3], user engagement prediction in Snapchat [4], and many other use cases.

Other applications of GNNs are evolving in various domains such as detecting fraudulent financial transactions [5], 3D object detection from LiDAR point clouds in autonomous vehicles [6], classifying collected data in particle physics colliders [7], and multiple-input multiple-output (MIMO) scheduling in wireless communication networks [8]. Typically, these use cases have stringent latency constraints. In addition, the GNN inference is only a part of a bigger system that interfaces with diverse sensors for data acquisition and executes different post-inference operations based on the prediction outcome. In such applications, field-programmable gate arrays (FPGAs) can offer significant benefits. The flexibility of FPGAs enables customization of the compute pipeline and memory sub-system to minimize the end-to-end processing latency, and their diverse IOs allow direct interfacing with a variety of sensors and actuators in a complete system [9].

However, most prior FPGA-based GNN accelerators are either not optimized for low latency real-time applications [10] or require heavy pre-processing of the input graphs which is not feasible for such use cases [11]. Alternatively, other FPGA solutions targeted at low latency GNN inference [12] rely on extreme specialization by generating custom streaming hardware that can only accelerate a specific GNN and process a single input at a time (i.e. batch-1). These solutions suffer from long compile times to generate a new bitstream every time the GNN is modified. They also cannot accelerate other types of DL models, prohibiting their standalone use for applications that combine spatial GNNs with other temporal models such as recurrent neural networks (RNNs), gated recurrent units (GRUs), or long short-term memories (LSTMs) [13], [14]. In addition, batch-1 inference is not always the optimal solution in practice, especially in cases where an action is taken based on predictions on multiple inputs from different sensors (e.g. processing inputs from multiple LiDARs in an autonomous vehicle [15], or scheduling users to multiple sub-bands of massive MIMO antennas in 5G networks [8]). In such cases, processing a small number of inputs simultaneously can achieve lower end-to-end latency than multiple batch-1 inferences in sequence.

In this work, we extend the architecture, instruction set, and software stack of the neural processing unit (NPU) to support all the operations required in a variety of GNN types. The NPU was originally designed as a software-programmable specialized processor (i.e. *overlay*) for low latency inference of memory-bound sequence models (e.g. RNNs, GRUs, LSTMs) and multi-layer perceptrons (MLPs) [16]. It was later re-architected to exploit the new artificial intelligence (AI) tensor blocks in the DL-optimized Intel Stratix 10 NX devices, achieving best-in-class results for these models with an order of magnitude higher performance than same-generation DL-optimized Nvidia GPUs [17]. We demonstrate that it is possible to maintain the flexibility of the NPU overlay while achieving similar or lower latency GNN inference than prior works specialized only for GNN acceleration. Our enhanced NPU can be programmed purely through software to accelerate not only different widely-used types of GNNs, such as graph convolutional networks (GCNs), graph attention networks (GATs), and graph isomorphism networks (GINs), but also the models it was originally designed for such as RNNs, GRUs, LSTMs, and MLPs. Furthermore, this flexibility

979-8-3315-3007-5/24/$31.00 ©2024 IEEE
DOI 10.1109/FPL64840.2024.00034

allows for accelerating GNNs with different performance objectives (i.e. latency-optimized or throughput-optimized) depending on the target application. Finally, we present a case study for using our enhanced NPU in real-time acceleration of massive MIMO antenna scheduling using GNNs. This paper's contributions include:

- NPU architecture, compiler, and toolchain enhancements to accelerate GNNs without sacrificing the overlay's flexibility.
- Compiler enhancements to exploit different parallelism levels in GNNs for latency- vs. throughput-oriented use cases.
- Performance comparisons of our enhanced NPU to state-of-the-art FPGA solutions and GPUs across various GNNs.
- A case study on real-time GNNs for MIMO antenna scheduling in 5G communication networks.

II. Background & Related Work

A. GNN Models

GNNs are a class of neural networks that can extract patterns and perform predictions on graph-structured data. A graph $G = (V, E)$ consists of a set of $|V|$ nodes and $|E|$ edges. Each node and edge in the graph has a *feature vector*, which represents different properties of this node/edge depending on the target application. A GNN takes as an input the feature vectors of all nodes and edges as well as the structural information of the graph, and learns a new more expressive representation of the graph nodes known as *node embeddings*. These embedding vectors capture information about the node itself, its neighbors, connections, and context within the graph. After that, the embedding vectors generated by the GNN can be used to perform different node-level tasks such as node labeling and classification, or can be all combined to generate a *graph embedding* to perform graph-level predictions. Using a given GNN, nodes with similar features and connections will have similar node embeddings and similar graphs will have similar graph embeddings.

A GNN performs this representation learning by processing the input graph using a number of *message passing layers*, each of which executes two main steps: aggregation and transformation. In the aggregation stage, the nodes of the graph exchange messages; each node receives the states (i.e. intermediate embeddings) of all its direct neighboring nodes and combines them using an aggregation function. Then, the aggregated state is used to update the state of the node itself before progressing to the next layer. This can be formulated mathematically as:

$$\mathbf{h}_u^{(l+1)} = \mathcal{U}^{(l)}\left(\mathbf{h}_u^{(l)}, \mathbf{m}_{\mathcal{N}(u)}^{(l)}\right) \tag{1}$$

$$\mathbf{m}_{\mathcal{N}(u)}^{(l)} = \mathcal{A}^{(l)}\left(\{\mathbf{h}_v^{(l)}, \forall v \in \mathcal{N}(u)\}\right) \tag{2}$$

such that for layer l, $\mathbf{h}_u^{(l)}$ is the state of node u, the set $\mathcal{N}(u)$ contains the direct neighbors of node u, $\mathbf{m}_{\mathcal{N}(u)}^{(l)}$ is the aggregated message from the neighbors of node u, and $\mathcal{U}$ and $\mathcal{A}$ are differentiable update and aggregate functions respectively. This means that after the first layer, the state of each node captures information about itself and its immediate neighbors. Then, after the second layer, it also captures information about its neighbors' neighbors, and so on. Finally, the node states calculated after L layers are the learned embedding vectors. Different variants of GNNs follow the same formulation and overall execution but use different update and aggregation functions.

Graph convolutional networks (GCNs) extend the concept of convolutions from vision networks to graphs [18]. In this type of GNN, the state of each node in the graph is calculated by aggregating the states of neighboring nodes multiplied by edge weights, in a manner analagous to a convolution kernel aggregating values of neighboring pixels in an image. Thus, its aggregate function $\mathcal{A}$ is:

$$\mathbf{m}_{\mathcal{N}(u)}^{(l)} = \sum_{v \in \mathcal{N}(u)} \frac{e_{u,v}}{\sqrt{\hat{d}_u \hat{d}_v}} \ \mathbf{h}_v^{(l)} \tag{3}$$

where $e_{u,v}$ is the edge weight between nodes (u, v) and $\hat{d}_u$ is the summation of all edge weights of node u plus one to avoid division by zero ($\hat{d}_u = 1 + \sum_{v \in \mathcal{N}(u)} e_{u,v}$). Another flavor of GCNs uses the **GraphSAGE** operator from [13], which calculates the aggregated message as the mean of the neighboring node states:

$$\mathbf{m}_{\mathcal{N}(u)}^{(l)} = \text{mean}(\{\mathbf{h}_v^{(l)}, \forall v \in \mathcal{N}(u)\}) \tag{4}$$

Graph attention networks (GATs) use a self-attention mechanism, similar to that commonly used in transformer networks [19], to assign different importance to the states of neighboring nodes during aggregation [20]:

$$\mathbf{m}_{\mathcal{N}(u)}^{(l)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v^{(l)} \tag{5}$$

$$\alpha_{i,j} = \frac{\exp\left(\text{LReLU}\left(\mathbf{a}_s^\top \mathbf{W}\mathbf{h}_i^{(l)} + \mathbf{a}_t^\top \mathbf{W}\mathbf{h}_j^{(l)}\right)\right)}{\sum_{j' \in \mathcal{N}(i)} \exp\left(\text{LReLU}\left(\mathbf{a}_s^\top \mathbf{W}\mathbf{h}_i^{(l)} + \mathbf{a}_t^\top \mathbf{W}\mathbf{h}_{j'}^{(l)}\right)\right)} \tag{6}$$

where $\alpha_{i,j}$ is the softmax-normalized attention weight between nodes (i, j), $\mathbf{a}_s/\mathbf{a}_t$ are learnable attention vectors for the source/destination node features, $\mathbf{W}$ is a learnable weight matrix for linear transformation of the node states, and LeakyReLU is the leaky rectified linear unit non-linear activation function.

Graph isomorphism networks (GINs) are used to generate an embedding for the whole graph. It combines the states of all graph nodes in a layer using a permutation invariant function $\mathcal{C}^{(l)}$ (e.g. summation or pooling) and then concatenates the results of all layers to generate the final graph embedding $\mathbf{h}_G$ [21]. Its node-level aggregate function combines the state of a neighboring node with the edge feature vector connecting to it as follows, where $\mathbf{e}_{i,j}$ is the feature vector of the edge between nodes (i, j):

$$\mathbf{m}_{\mathcal{N}(u)}^{(l+1)} = \sum_{v \in \mathcal{N}(u)} \text{ReLU}\left(\mathbf{h}_v^{(l)} + \mathbf{e}_{u,v}\right) \tag{7}$$

$$\mathbf{h}_u^{(l+1)} = \text{MLP}^{(l)}\left((1 + \epsilon^{(l)}) \ . \ \mathbf{h}_u^{(l)} + \mathbf{m}_{\mathcal{N}(u)}^{(l+1)}\right) \tag{8}$$

$$\mathbf{h}_G = \text{CONCAT}\left(\left\{\mathcal{C}^{(l)}(\{\mathbf{h}_v^{(l)}, \forall v \in G\})\right\}\right) \tag{9}$$

Across these GNN variants with different aggregation functions, the update function $\mathcal{U}$ can also be a simple function such as mean or max, a linear transformation (i.e. matrix-vector multiplication), a feed-forward MLP network, or even a temporal GRU network [22]. Our enhanced NPU overlay can flexibly accelerate all the GNN variants briefly explained in this section (in addition to other DL models such as RNNs, GRUs, LSTMs and MLPs) by executing different sequences of instructions programmed purely through software.

B. GNN Acceleration

Many prior works have introduced specialized hardware for inference acceleration of different types of GNNs in recent years [23]. However, there are several limitations in existing solutions, especially for low latency real-time use cases.

Firstly, many accelerators such as AWB-GCN [24], GCNAX [25], and I-GCN [26] focus only on GCNs where computations can be condensed into large sparse matrix multiplication (SpMM) and general matrix multiplication (GEMM) operations. Others such as GraphACT [27], HyGCN [28], H-GCN [29], and VersaGNN [30] apply graph preprocessing to exploit optimizations such as data locality or graph partitioning to reduce the computational complexity. The data layout transformations or preprocessing techniques used in these works are not suitable for real-time applications in which there is a strict constraint on the latency between receiving an input graph and obtaining a prediction result. In contrast, our enhanced NPU does not perform any pre-processing operations and operates

directly on graph node/edge embedding vectors without the need for any data layout transformation.

Secondly, these accelerators do not support edge embeddings which capture additional information about connections between nodes and have been shown to increase the model accuracy on several tasks [31]. Supporting edge embeddings requires executing edge-specific operations in each GNN layer which cannot be mapped to SpMM and GEMM operations, increasing the complexity of the accelerator architecture. Our enhanced NPU can process edge embeddings at no additional hardware cost since it does not rely on refactoring GNN computations as SpMM and GEMM operations.

Finally, most of these accelerators do not have the flexibility to support non-trivial aggregation functions that cannot be formulated as matrix multiplications such as max, min, mean, and standard deviation. In GATs for example, calculating the aggregated message is dependent on scaling the features of a given node and those of its neighbors (see Eq. 5). These aggregation weights are calculated dynamically on-the-fly at each layer and thus cannot be expressed as matrix multiplication operations. Our enhanced NPU provides such flexibility and enables accelerating more complex GNNs.

The current state-of-the-art FPGA-based GNN acceleration targeting low latency workloads is FlowGNN [12]. It is an open-source framework that can generate specialized dataflow architectures for a wide variety of GNNs and also addresses the limitations of prior GNN accelerators described previously. Although it can achieve low latency inference suitable for real-time applications, it sacrifices the flexibility of the accelerator architecture. To accelerate a different type of GNN, it requires compiling a new bitstream and programming the FPGA with a specialized processing pipeline that can only accelerate this GNN type. This approach is only suitable for use cases where only a specific type of GNN is needed with very infrequent updates. Their extreme specialization also limits the FlowGNN architecture to processing only one input at a time, possibly making it sub-optimal for applications in which multiple inference results on different inputs from different sensors are needed for executing a specific action. For instance, applying the brakes in an autonomous vehicle based on inferences of several point clouds from several LiDARs [15] requires processing of multiple inputs simultaneously. In this work, we take a different approach by building a software-programmable overlay architecture that can achieve similar or better performance without sacrificing flexibility. Our enhanced NPU can execute different GNN types as well as other DL models (e.g. RNNs, GRUs, LSTMs, MLPs) on the same architecture just by executing different instruction sequences. This flexibility also enables us to map computations to the in-fabric tensor blocks in different ways for latency-optimized or throughput-optimized GNNs by exploiting compute parallelism across nodes from one or multiple graphs, respectively.

GraphAgile [11] is another FPGA-based overlay that is software programmable for the GNN domain-specific workloads. Several techniques are used by the GraphAgile compiler to optimize the computational complexity, operation scheduling, and external memory communication. However, it does not support edge embeddings and many of its compiler optimizations involve preprocessing operations such as graph partitioning and computation reordering, making it unsuitable for low-latency use cases. In addition, it is a GNN-specific overlay and does not support other types of DL workloads.

C. The Neural Processing Unit (NPU) Overlay

The NPU is a state-of-the-art FPGA-based very-long-instruction-word (VLIW) processor architecture designed for low-latency DL inference workloads using pre-trained and quantized `int8` models, specifically RNNs, GRUs, LSTMs, and MLPs [16]. It keeps all model weights *persistent* in the on-chip memories of the FPGA and exploits the massive parallelism of DL models to initiate the execution of thousands of operations with a single instruction, significantly reducing the per-operation energy and area overhead of software programmability. As shown in Fig. 1, its pipeline consists of a chain of coarse grained units. Each unit has its own instruction decoder that decodes a field within the VLIW instruction and issues a series of SIMD micro-instructions. The matrix-vector multiplication unit (MVU) consists of T compute tiles, each of which has D groups of C dot-product engines (DPEs) of size L multiplication lanes. The rest of the NPU pipeline also has L SIMD lanes. Vector operands are broadcast from C vector register files (VRFs) to all DPE groups in a tile, while persistent model weights are fed from the matrix register files (MRFs) shared between C DPEs in a group (i.e. C cores). The MVU is followed by an external VRF (eVRF) that is used to skip the MVU in case an instruction does not start with a matrix-vector operation. In the original NPU architecture, the eVRF feeds two identical multi-function unit (MFU) blocks that implement vector elementwise operations commonly used in DL models, such as activation functions (e.g. sigmoid, tanh, ReLU), addition/subtraction, and multiplication. The final stage is the loader which can write back results to any of the processor architecture states (i.e. VRFs) for further processing, and also can communicate with external components (e.g. other FPGA modules or a network interface) through input/output FIFOs.

The most recent version of the NPU [17] targets the DL-optimized tensor blocks that replace the conventional DSP blocks in the Intel Stratix 10 NX FPGAs [32]. In their highest compute density mode, each one of these tensor blocks implements three dot-10 operations with restrictions on the inputs of these operations; one input vector is broadcast to all three dot products in the tensor block and the other set of three operands are fed from three internal data reuse registers. These data reuse registers are also double-buffered to enable loading a new set of vectors while using the current set for computation. Thus, to make the best use of the compute capabilities of the tensor blocks, each set of MVU DPEs and corresponding core process three inputs simultaneously (i.e. batch-3 per core), for an overall batch size of $3\times C$. To program the NPU, DL application developers do not need to know any of these architecture details. They can write their models using a subset of the Tensorflow Keras API [33] and then use the NPU compiler to generate the VLIW instructions to be executed on the FPGA overlay.

III. NPU Enhancements for GNNs

In this work, we enhance the NPU architecture to support all the operations needed for the different types of GNNs described in Section II-A. In this section, we will describe the details of our NPU enhancements (highlighted in red in Fig. 1).

A. Numerical Precision

In the baseline NPU implementation, the MVU used `int8` precision which is natively supported in the tensor mode of operation of the Stratix 10 NX tensor blocks, while the rest of the NPU pipeline used a higher `int32` precision. Prior work has shown that `int4/int8` quantization of GNN models reduces memory footprint and speeds up inference while having comparable accuracy to `fp32` [34]. However, many GNN aggregation operators and other operations such as softmax in GATs require using higher-precision numerical representations to maintain accuracy. Therefore, we keep the MVU `int8` precision and modify the rest of the NPU pipeline to use Google's 16-bit Brain floating-point precision (`bfloat16`) which was shown to provide similar inference accuracy to single-precision floating-point precision [35]. In addition, each tensor block in the Stratix 10 NX device can be configured to implement three independent `bfloat16` additions/subtractions and with additional soft logic, it can also implement three `bfloat16` multiplications.

The inputs/outputs of our enhanced NPU are streamed in/out in `bfloat16` precision. The vectors written by the loader to the MVU VRFs are converted from `bfloat16` to `int8` using a precision adapter on the VRF write path. Then, we implement another precision adapter in the eVRF to convert the `int32` MVU outputs to

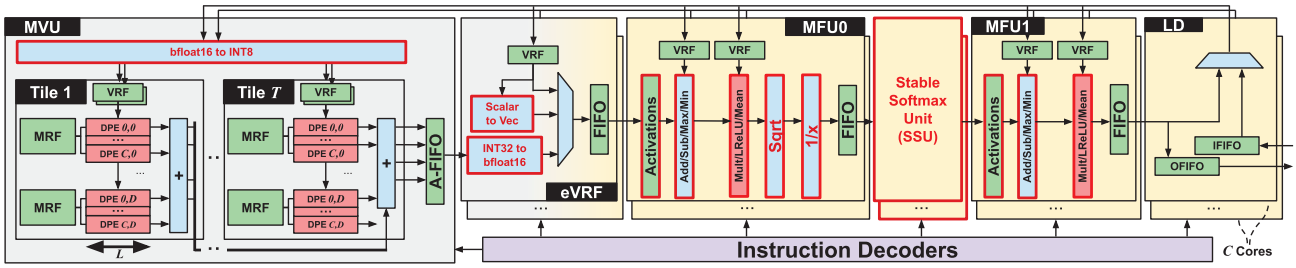

Fig. 1: The enhanced NPU overlay architecture with the newly added functionalities highlighted by a red border. The units highlighted in yellow are modified to use `bfloat16` precision, while the grey units use `int8`.

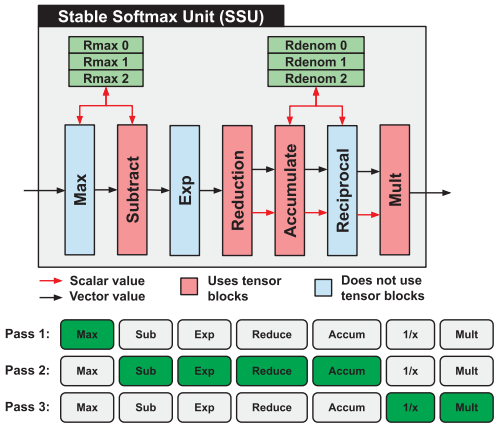

Fig. 2: Stable softmax unit internal architecture. Since each NPU core operates at batch-3, sets of three scalar registers are used to store the running maximum of the input vectors (`Rmax 0-2`) and the reduced and accumulated softmax denominator values (`Rdenom 0-2`).

`bfloat16` for further processing. These precision adapters have negligible impact on overall performance since they only introduce 5-6 cycles of latency to the processing pipeline, and increase the utilized soft logic resources by less than 5%. Most of these cycles can also be hidden by other operations such as reading from the register file in the eVRF block. For the elementwise operations in the MFU, we implement additions and multiplications using a mix of tensor blocks and soft logic resources, and use `bfloat16` implementations of tanh and sigmoid generated by Intel's DSP builder tool [36].

B. Stable Softmax Unit (SSU)

As shown in Eq. 5, GATs require a softmax operation to normalize the attention weights (α) across neighbouring nodes. Therefore, we implement a numerically stable softmax unit (SSU) and integrate it into the NPU pipeline after the first MFU. The SSU implements Eq. 10, which is commonly used in DL applications to maintain numerical stability [37]. It can process a vector $\mathbf{x}$ of any length N and supports interleaved batch-3 execution to match the rest of the NPU core pipeline. This softmax formulation is considered numerically stable since we subtract the maximum value in the vector from all its elements, producing a vector of only non-positive elements that will be used as exponents. This avoids overflow for large vector elements and guarantees that at least one element exponentiation will evaluate to $e^0 = 1$ to avoid a vanishing denominator.

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i - \max(\mathbf{x})}}{\sum_{j=1}^{N} e^{x_j - \max(\mathbf{x})}} \quad \text{for } \mathbf{x} = (x_1, ..., x_N) \qquad (10)$$

The internal architecture of the SSU is illustrated in Fig. 2. It consists of 7 compute units that implement the following functionalities: (1) *Max:* finds the maximum values in the interleaved batch-3 input vectors and stores them in the 3 `Rmax` scalar registers, (2) *Subtract:* subtracts the maximum values stored in the `Rmax` registers from the corresponding input vectors, (3) *Exp:* performs a vector elementwise exponentiation operation, (4) *Reduce:* sums up vector elements into a scalar, (5) *Accumulate:* adds reduction outputs over multiple time steps and stores the results in the `Rdenom` scalar registers, (6) *Reciprocal:* calculates the inverse of the accumulation results stored in the `Rdenom` registers, and (7) *Mult:* performs elementwise multiplication between a vector and the reciprocal scalar value. All compute units are controlled by a micro-instruction word to either execute their operations on their inputs or pass their inputs directly to their outputs (i.e. bypass).

In case the length of the input vector of the softmax function is higher than the native dimension (i.e. SIMD width) of the NPU pipeline, the vector is split into chunks that are fed sequentially to the pipeline blocks. Since the processing pipeline operates at batch-3, these chunks are interleaved such that the first chunk of all 3 inputs are received first before progressing to the next chunk (i.e. $A_0, B_0, C_0, A_1, B_1, C_1, ...$). This means that the softmax is calculated over multiple passes since it involves operations that require all the vector elements, such as max and denominator summation. Thus, the SSU includes 3 internal registers to store the partial max and sum values of the 3 interleaved inputs in a batch. The bottom part of Fig. 2 highlights the used compute units in each pass. In the first pass, all vector chunks flow through the SSU and are written back to the eVRF by the loader to calculate the maximum value in each vector. In the second pass, the same vector chunks again flow through the SSU to subtract the calculated maximum value from all vector elements and perform the elementwise exponentiation, followed by reduction and accumulation of the exponentials for calculating the denominator. In this pass, the accumulation scalar results are stored in the internal `Rdenom` registers and the exponentiation vector results are also passed to the output of the SSU to be again written back to the eVRF by the loader. In the third and final pass, the accumulation results stored in the internal registers are reciprocated and then the vectors flow through the SSU to be multiplied by the reciprocal values, producing the final results of the softmax operation. These passes are automatically scheduled by the compiler for any vector size with no explicit programming from the user.

Unlike other FPGA GNN accelerators such as [38] and [39], we do not approximate the exponentiation operation by using lookup tables or changing the exponentiation base to 2 instead of e as these changes can degrade inference accuracy. The exponentiation compute unit in our SSU provides accuracy within three units in last place (ULPs) of the `bfloat16` baseline software implementation.

C. Enhanced MFUs

Compared to the NPU's original target workloads, GNNs require many new elementwise vector operations. For example, GATs use the LeakyReLU activation function (see Eq. 5) and GCNs [40] involve operators such as division and square root when edge weights are not unity (see Eq. 3). Other types of GNNs contain multiple

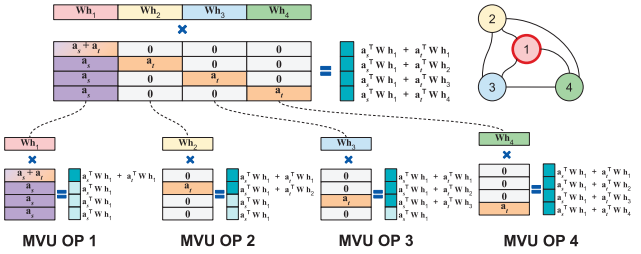

Fig. 3: Attention weight calculations using the MVU and instruction-controlled BRAM accumulator to accumulate results across multiple MVU operations.

aggregation methods such as mean, min, max, and standard deviation [41]. Therefore, to have a general overlay that can support commonly-used GNN aggregation functions, we extended the MFU functionality to support these operations with minimal effect on performance. We added min and max operations to the add/subtract stage and maximized hardware reuse by adding multiplexers in front of one of the two inputs of the elementwise multiplication stage. These multiplexers enable executing elementwise multiplication, LeakyReLU, or mean operations by selecting between inputs from the original VRF, the LeakyReLU constant, or a read-only memory storing pre-calculated values of $1/n$ where $n \in \{1, ..., N_{max}\}$ and N_{max} is the maximum number of nodes supported by the NPU, respectively. We also add square root and reciprocal compute units generated by Intel's DSP builder [36] to only one of the two MFUs as they are less commonly used, and adding them to both MFUs would result in underutilized hardware and an unnecessary increase in the processing pipeline latency.

D. Instruction-Controlled BRAM Accumulators

As shown in Eq. 5, to calculate the attention weights in GATs, the embedding vectors of all nodes ($\mathbf{h}$ vectors) go through a linear transformation using the weight matrix $\mathbf{W}$. Then, the next step in the computation is to perform a dot product between the attention vectors ($\mathbf{a}_s/\mathbf{a}_t$) and the transformed embedding vectors of each pair of neighboring nodes ($\mathbf{Wh}_i/\mathbf{Wh_j}$). This step can be formulated as a matrix vector operation as illustrated at the top of Fig. 3 for node 1 with three neighbors. However, mapping this computation as a single matrix-vector multiplication operation would require replicating the input vector in many different permutations for calculating the attention weights when nodes 2, 3, and 4 are the source nodes, which can be prohibitive due to limited on-chip memory capacity.

Instead, we split this operation into multiple smaller matrix-vector operations as shown in the bottom of Fig. 3. This mapping allows us to store the transformed embedding vectors once and flexibly reuse them with different matrix column blocks depending on which source node attention weight we are calculating. However, this split requires accumulating the results of multiple MVU operations, which could only be performed by the MFU addition unit in the baseline NPU architecture. This requires the result vector of each MVU operation to go through the whole NPU pipeline and be written back by the loader to the VRF of the MFU add/subtract unit to be accumulated to the next result vector, creating a major performance bottleneck. The MVU of the baseline NPU architecture already included a BRAM-based accumulator (i.e. scratchpad) to interleave the accumulation of batch-3 partial results within a single large matrix-vector multiplication [17]. However, this BRAM accumulator was not exposed to the NPU ISA and compiler, and thus cannot be used for accumulating results across different matrix-vector multiplications mapped to the MVU. To alleviate this performance bottleneck, we change the NPU microarchitecture and ISA to expose the BRAM accumulator to the NPU compiler. This enables the compiler to control when and how to accumulate different MVU outputs depending on the workload.

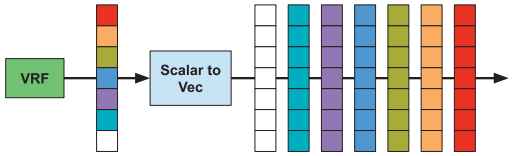

Fig. 4: Broadcast of scalar values to vectors in the NPU eVRF.

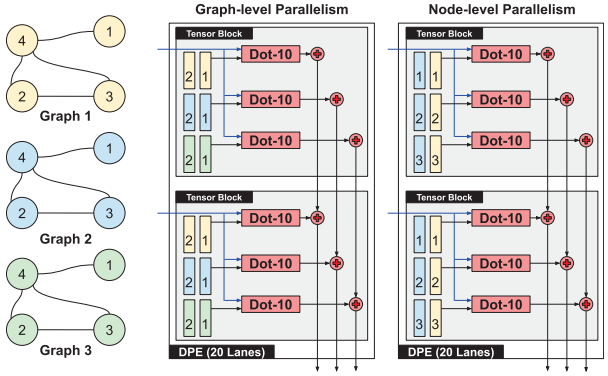

Fig. 5: Exploiting graph-level or node-level parallelism in the NPU.

E. Scalar Vector Support

The baseline NPU pipeline only supported vector elementwise operations. However, GNNs require scalar-vector operations such as vector scaling. These scalars are also not constant values that can be pre-stored in the NPU VRFs in vector format. They are dynamically calculated in the GNN compute graph as a vector, and then each element of the vector becomes the scaling factor of other vectors in subsequent operations. Therefore, we add a new scalar-to-vector transformation block to the NPU eVRF that operates as illustrated in Fig. 4. For a vector read from the VRF, this block can select an element and broadcast it as a vector for subsequent elementwise operations. The specific element of the input vector and the number of duplicated broadcast vectors (for scaling vectors that are longer than the pipeline SIMD width) is controlled by the eVRF instruction.

F. Compiler and Toolchain Enhancements

We extend the NPU ISA to introduce new instructions for controlling the newly introduced hardware blocks. We also enhance the compiler frontend to enable describing GNN workloads purely in software using PyTorch Geometric [42] or an NPU domain-specific language. Although these enhancements are an essential component of our work, we leave out their details for space constraints.

IV. Exploiting Parallelism in GNNs

In the workloads supported by the baseline NPU of [17], there is a single input vector per MLP inference or RNN/GRU/LSTM time step. Since each tensor block in the Stratix 10 NX fabric has 3 dot-product units that share an operand, it was necessary to increase the processing batch size to 3 inputs per core to best utilize the tensor block compute capabilities. Unlike these conventional workloads, a single GNN input is a complete graph consisting of many node/edge feature vectors, which presents different parallelization opportunities to be exploited for different performance targets.

A. Graph-level Parallelism

The same form of parallelism used by the baseline NPU can still be exploited by simultaneously processing nodes from 3 different

input graphs per NPU core. This would result in a **throughput-optimized** implementation which is useful for applications that can tolerate batching a few inputs or when the final prediction outcome depends on processing multiple inputs from multiple sources at the same time. In this case, the compiler maps embedding vectors of nodes/edges from different input graphs (that have the same topology) to be multiplied by the same weight matrix in GNN aggregate and update functions that can utilize the MVU as illustrated on the left side of Fig. 5. The transformed node/edge embedding vectors are then processed by the rest of the NPU pipeline in a sequential interleaved manner. Since the NPU only supports persistent inference where all the node/edge feature vectors and model weights are stored in the on-chip memory (similar to FlowGNN [12]), graph-level parallelism can limit the size of processed graphs as it requires storing 3 different graphs during inference.

B. Node/Edge-level Parallelism

Unlike the conventional NPU workloads, GNNs also contain many independent operations that can be performed in parallel on different nodes or edges of the same graph. For example, each node can perform the linear transformation of the embedding vectors of its neighboring nodes during aggregation or the update operation of its own embedding vector independently from other nodes in the graph. Therefore, the compiler can exploit this parallelism by mapping the embedding vectors of different nodes/edges from the same graph to be multiplied by the same linear transformation matrix as shown in the right side of Fig. 5. This node/edge-level parallelism results in a more **latency-optimized** implementation compared to graph-level parallelism, since all the compute capabilities of the tensor blocks are dedicated to process a single graph instead of three. It also enables the NPU to process larger graphs since the register file memory resources are not split between multiple graphs.

These different approaches of exploiting GNN parallelism highlight a unique benefit of the NPU overlay flexibility; it allows users to not only implement different DL models on the same architecture purely through software but also to instruct the compiler to exploit different parallelism approaches and optimize for different performance targets (throughput-optimized vs. latency-optimized) depending on the application requirements.

V. Experimental Results

A. Methodology and Experimental Setup

1) Performance Comparisons: We evaluate our enhanced NPU using 3 common GNN types: GATs, GINs, and GCNs. We perform inference on the same models used in [12], to allow a direct comparison to FlowGNN, the current state-of-the-art GNN accelerator targeted at low latency real-time applications (using `int16` precision) on AMD Ultrascale+ (TSMC 16nm) FPGAs. We also compare to the GPU implementation results presented in [12] for an Nvidia RTX A6000 GPU (TSMC 8nm) which is one process technology node ahead of the Stratix 10 NX FPGA (Intel 14nm) we target. The GNN workloads running on the GPU are implemented using PyTorch Geometric and the results are averaged across 5 inference runs to reduce measurement noise. For throughput comparisons, we use the batch size that achieved the best throughput for the GPU and for latency comparisons, we use batch-1 GPU results. Finally, we compare the NPU to other GNN accelerators [29], [43] that utilize the AI engines in the AMD Versal architecture (TSMC 7nm) [44]. These accelerators do not support edge embeddings and also rely on preprocessing the input graphs to exploit sparsity and map sub-graphs to different SpMM/GEMM units in parallel.

2) Workloads and Datasets: For GCNs and GINs, we run 5 layers with node/edge embedding dimensions of 100, global mean pooling, and one linear output layer with an output size of 10. For GATs, we run 5 layers with a node embedding dimension of 16 and no edge embeddings (to match FlowGNN), 4 attention heads, global mean pooling, and one linear output layer with an output size of 10. Table I shows the 6 graph datasets that we used for evaluation. We use MolHIV and MolPCBA which are two molecular datasets from the Open Graph Benchmark [45] and the Hep10k dataset generated by [12] from the high-energy physics top quark tagging reference dataset [46]. These 3 datasets include edge features and contain small graphs on the order of 10-200 nodes which are suitable for the real-time GNN inference applications targeted by both our NPU and FlowGNN. We also evaluate the NPU on 3 larger publication datasets, Cora, Citseer, and PubMed [47], to showcase the high performance of the NPU on standard datasets that other GNN accelerators typically use to evaluate performance.

TABLE I: The graph datasets used for evaluation.

Dataset	Graphs	Avg. #Nodes	Avg. #Edges	Edge Embed.
MolHIV	41,127	25.5	55.6	Yes
MolPCBA	437,929	26	59.3	Yes
Hep10K	10,000	49.1	785.3	Yes
Cora	1	2,708	5,429	No
CiteSeer	1	3,327	4,732	No
PubMed	1	19,717	44,348	No

For the MolHIV and MolPCBA datasets, the graphs are small enough to store all the node/edge embedding feature vectors for a batch size of 3 inputs per NPU core. Thus, we use them to study the performance tradeoff between our throughput-optimized and latency-optimized GNN kernels explained in Section IV. The NPU operates at batch-9 (3 cores $\times$ batch-3 per core) and batch-3 (3 cores $\times$ batch-1 per core) for the throughput-optimized and latency-optimized modes, respectively. For the other four datasets, the graphs have significantly more nodes/edges, as shown in Table I. This requires operating the NPU at only batch-1 where the compute units of only one core are enabled and the VRF on-chip memory resources of the other cores are re-purposed as an extension to the enabled core's VRF to accommodate these large graphs.

3) NPU Performance and Power Measurements: We measure the NPU compute latency on a Stratix 10 NX development kit using hardware performance counters that record the number of cycles starting from popping the first input vector from the input FIFO until pushing the last output vector to the output FIFO. We also add the latency for transferring the input graph and sending back the inference output, assuming that the NPU receives/sends inputs/outputs directly over 100 Gbps Ethernet. The network transfer latency results we used are based on real measurements for different network payload sizes from [17]. This is a more constrained input/output bandwidth compared to the 128 Gbps PCIe interface used by FlowGNN on the AMD Alveo U50 card. For throughput results, we can overlap the processing of one input graph with the loading of the next. Therefore, we use the higher latency value between the input/output transfer and compute when calculating the NPU throughput (i.e. data-transfer-bound vs. compute-bound). In all the cases we experimented with, the input/output loading was never the throughput bottleneck. The NPU power consumption results are obtained by measuring the power of the Stratix 10 NX development kit using a high resolution power meter in room temperature ambient without any special cooling solutions (only air-cooled heat sink).

B. FPGA Implementation Results

The RNN, GRU, and LSTM workloads used to evaluate the baseline NPU had input vectors with sizes ranging from 512 to 1792 elements. However, the GNN workloads that we evaluate here have much smaller node/edge embedding vector sizes (100 for GCNs and GINs, 16 for GATs). As a result, the original NPU configuration with 2 cores, 7 MVU tiles, 40 DPEs and 40 lanes (2C-7T-40D-40L) would result in significant hardware underutilization since an input vector would be padded by zeros to fit the architecture's native input dimension (7 tiles $\times$ 40 lanes = 280). In addition, the inputs to these GNN workloads (entire graphs) have a higher memory footprint

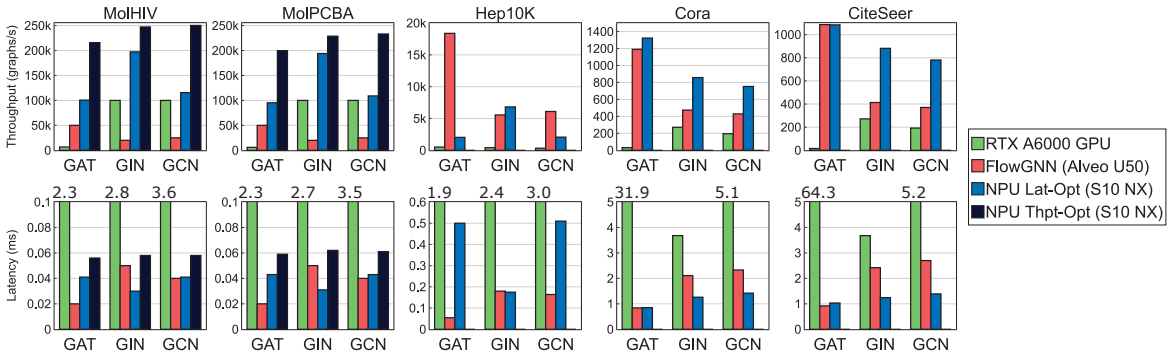

Fig. 6: Throughput and latency results comparison between Nvidia RTX A6000, FlowGNN on AMD Alveo U50, and our enhanced throughput/latency-optimized NPU on Intel Stratix 10 NX.

TABLE II: Resource utilization of our enhanced NPU on the Stratix 10 NX device (TBs: tensor blocks, ALMs: adaptive logic modules).

	TBs	ALMs	BRAMs
Available	3,960 (100%)	702,720 (100%)	6,847 (100%)
Total	2,061 (52%)	343,620 (49%)	5,438 (79%)
|– **MVU**	1,800 (45%)	112,705 (16%)	2,930 (42.8%)
|– **eVRF**	-	28,141 (4%)	435 (6.4%)
|– **MFU0**	99 (2.5%)	66,815 (9.5%)	753 (11%)
|– **SSU**	63 (2%)	75,031 (10.7%)	246 (3.6%)
|– **MFU1**	99 (2.5%)	54,351 (7.7%)	750 (11%)
|– **Loader**	-	4,867 (0.7%)	150 (2.2%)
|– **Inst.Decoders**	-	1,710 (0.2%)	174 (2.5%)

and require deeper VRFs compared to those of the conventional workloads (sequence vectors). Therefore, we perform a design space exploration by sweeping different values for the NPU architecture parameters and we find that an architecture with 3 cores, 3 tiles, 40 DPEs and 40 lanes (3C-3T-40D-40L) achieves the best tradeoff between hardware utilization, processing latency, and FPGA resource utilization across all workloads and datasets in our evaluation. By reducing the number of tiles, it also reduces the amount of on-chip memory resources allocated to MVU MRFs and allows us to implement deeper VRFs to store the larger GNN inputs. This new NPU configuration with smaller but more cores in addition to all the hardware features added for GNN support achieves 36% higher latency but improves throughput by 6% on average for the original MLP/RNN/GRU/LSTM NPU workloads. If a user is interested in only a subset of workloads, this latency overhead can be completely eliminated by customizing the NPU configuration for the workloads of interest [48]. We use Intel Quartus 22.4 to synthesize, place and route our enhanced NPU on the Intel Stratix 10 NX device [32]. Our NPU achieves a maximum operating frequency of 300 MHz and its resource utilization results are presented in Table II.

C. Comparison Results vs. FlowGNN and GPU

Fig. 6 shows the throughput and latency comparison results between our enhanced NPU and both FlowGNN and the Nvidia RTX A6000 GPU across 5 datasets and 3 different GNN models. Our NPU running throughput-optimized GNN kernels for the MolHIV and MolPCBA dataset achieves the highest throughput; it can process 7.8× and 5.8× more graphs per second on average compared to FlowGNN and the GPU, respectively. On these datasets, the latency-optimized kernels running on our enhanced NPU are 1.6× faster for GINs, but 50% and 5% slower for GATs and GCNs compared to FlowGNN. The latency results of both FlowGNN and our NPU are significantly better than that of the RTX A6000, highlighting that FPGAs are a better match for real-time GNN inference. The results on MolHIV and MolPCBA also highlight a unique benefit of the NPU overlay architecture; it can programmed purely through

TABLE III: Energy efficiency (in million graphs/kJ) of our NPU, FlowGNN, and RTX A6000 GPU on the MolHIV dataset. The NPU results are for latency-optimized / throughput-optimized kernels.

Workload	NPU	FlowGNN	RTX A6000
GAT	2.4 / 5.1	2.3	0.0054
GIN	4.4 / 5.5	0.73	0.0045
GCN	2.8 / 5.9	0.88	0.0035

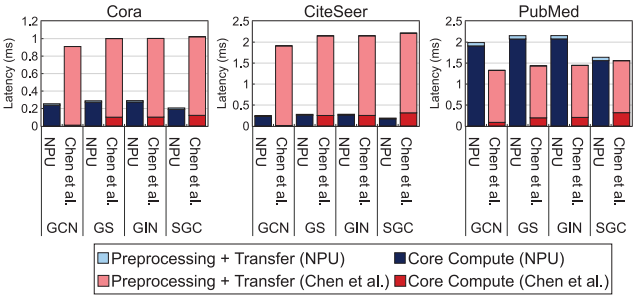

Fig. 7: Batch-1 latency of the NPU on an Intel Stratix 10 NX (14nm) and the Chen et al. accelerator [43] on an AMD Versal (7nm) for GCN, GraphSage (GS), GIN, and simplified graph convolution (SGC) models on the Cora, CiteSeer, and PubMed datasets.

software to run different sequences of instructions executing the same GNN model to target either a latency- or throughput-optimized implementation depending on the target application. On average, the latency-optimized GNN kernels can achieve 1.6× lower latency than the throughput-optimized ones. On the other hand, throughput-optimized kernels have 1.8× higher throughput on average.

For the three other datasets (Hep10K, Cora, and CiteSeer), our NPU can only run the latency-optimized kernels with batch-1 to accommodate the larger input graph sizes. For both Cora and CiteSeer, even the latency-optimized kernels running on our enhanced NPU can achieve 1.6× and 8.8× higher throughput than FlowGNN and the GPU, respectively. In addition, it can achieve similar GAT latency and 1.8× lower GIN and GCN latency compared to FlowGNN. On the Hep10K dataset, our NPU performs slightly better on GINs but falls behind on GAT and GCN workloads compared to FlowGNN. This can be attributed to the nature of this dataset, in which each node has 16 neighbors. In such cases, the NPU MFUs are the bottleneck when executing the aggregation functions.

Table III compares the energy efficiency of our NPU to FlowGNN and the RTX A6000 GPU on the MolHIV dataset as an example. The NPU processes 3.6× and more than 900× more graphs/kJ compared to FlowGNN and the GPU, respectively.

D. Comparison Results vs. Versal-based GNN Accelerator

Chen et al. [43] develop a high-performance accelerator that formulates GNNs as GEMM/SpMM operations and exploits the heterogeneous nature of the AMD Versal architecture [44] which combines an FPGA fabric with general-purpose ARM CPU cores and an array of vector processors with programmable bus-based interconnect (i.e. AI engines). The host CPU first preprocesses the input graphs by partitioning them into smaller submatrices for parallel execution. During runtime, an analyzer and a scheduler running on the on-chip ARM Cortex-A72 CPU maps the computation kernels of GNNs to GEMM or SpMM primitives (based on their data sparsity), which are then computed using the AI engines or FPGA fabric, respectively.

Although the AMD Versal architecture has a process technology advantage over the Intel Stratix 10 NX (7nm vs. 14nm), we directly compare the end-to-end performance of an instance of our NPU overlay with 1 core, 1 tile, 20 DPEs, and 20 lanes to the GNN accelerator from [43]. Fig. 7 presents the latency results of both accelerators for GCN, GIN, GraphSage and simplified graph convolution (SGC) models on the Cora, CiteSeer, and PubMed datasets. For the Chen et al. [43] accelerator, we add the preprocessing time and the actual compute time for a fair comparison to our NPU that does not require any preprocessing. For the Cora and CiteSeer datasets across all four models, the NPU achieves 3.4-11.7× lower latency due to the significant preprocessing overhead in [43]. The PubMed dataset contains graphs with 7× more nodes than Cora and CiteSeer, which benefits more from the parallelization across different SpMM and GEMM engines in the Chen et al. accelerator. Despite the process technology gap, our NPU can still achieve 0.66-0.95× the performance of [43] on the PubMed dataset. This comparison shows the advantage the NPU accrues from not requiring any preprocessing operations, and how it makes the NPU more suitable for embedded real-time applications, especially on smaller graphs. In addition, the accelerator from [43] does not support GNNs with complex aggregation functions or edge embeddings due to formulating workloads as a series of GEMM/SpMM operations. Although the I-GCN accelerator [26] achieves the highest performance for GCN workloads, we were not able to directly compare the NPU to it since I-GCN's preprocessing overhead (which is key to its high performance) is not disclosed in the paper.

VI. Case Study: GNN-based MIMO Antenna Scheduling in 5G Communication Networks

With advances in wireless communication networks, there is an ever-increasing demand for communication infrastructure that can serve more user equipments (UEs) with higher bandwidths and lower latencies. Multi-antenna transmission, commonly referred to as massive multi-input multi-output (MIMO), is used in modern 5G networks to transmit different data streams between a base station and multiple UEs simultaneously [49]. This can be achieved using a large antenna array that can be split into multiple sub-arrays to direct multiple beams over a range of horizontal and vertical angles, and thus spatially multiplex frequency sub-bands over a given time slot between multiple UEs. In such massive MIMO systems, the base station needs to determine the best subset of UEs to group together for transmission within a given time slot (i.e. MIMO antenna scheduling) based on the current channel state and set of active UEs. These transmission time slots are typically ∼1ms in 5G networks [50], requiring extremely low latency scheduling.

Classic approaches used greedy scheduling algorithms, which achieve sub-optimal results and do not scale well with the number of active UEs. Therefore, recent approaches deploy DL models to perform the MIMO scheduling task, and more specifically GNNs [8]. In this approach, a given network is represented as 3 graphs, where the nodes are a single sub-band and multiple UEs (or data streams within UEs) with weighted edges between them. One graph is fully-connected with edge weights representing inter-UE interference. The other two are star graphs with edges between the sub-band nodes and all the UE nodes where the edge weights capture UE fairness scaling and UE selection scores. The MIMO antenna schedule is inferred using a 4-layer GNN, where each layer is the combination of a GAT and two GCN layers applied to the three graphs. This approach was shown to achieve high-quality schedules while reducing complexity by about 90% compared to traditional greedy algorithms [8].

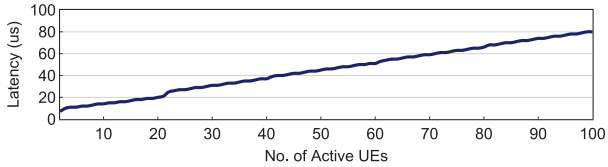

Fig. 8: GNN-based MIMO antenna scheduling latency using an NPU with 8 cores, 1 tile, 20 lanes, and 20 DPEs for 24 sub-band inferences and a varying number of active UEs (i.e. graph nodes).

This represents a realistic application for GNNs which requires: (1) low latency and high-throughput real-time GNN inference due to batched graph processing for scheduling multiple sub-bands simultaneously, (2) flexibility to execute multiple GNN layer types in the same workload, and (3) energy-efficient embedded deployment in a wireless base station without the opportunity to preprocess input graphs beforehand. Unlike all existing GNN acceleration solutions, our enhanced NPU overlay can satisfy all three requirements for this use case. Fig. 8 presents the latency results for an 8C-1T-20D-20L NPU instance running inference of the MIMO antenna scheduling GNN from [8] on network graphs with varying number of active UEs (i.e. nodes). It shows that our NPU can infer the schedule for 24 sub-bands (8 cores × batch-3) simultaneously with 100 UEs/nodes each in only 80μs, which is well below the 1ms transmission time slot in 5G networks.

VII. Conclusion

In this work, we enhance the architecture, instruction set, and software stack of the NPU overlay to support low latency GNN inference.We modify the architecture of the NPU to use higher numerical precision (`bfloat16`) for the vector operations, include a stable softmax unit, and add vector functional units for common operations in GNNs (e.g. mean/max/min, square root, reciprocal). We also upgrade the software stack to enable users to describe their GNNs in software and compile it to NPU instructions. The software programmability of the NPU not only allows users to run different DL models and GNN types on the same accelerator, but also enables the compiler to generate GNN codes for different performance targets (throughput-optimized vs. latency-optimized) by exploiting different parallelism dimensions on the same underlying hardware.

Without sacrificing the overlay flexibility, our enhanced NPU implemented on a 14nm Intel Stratix 10 NX FPGA can achieve 7.8× higher throughput and similar latency on average compared to the state-of-the-art model-specific streaming hardware on same-generation FPGAs. It also does not require any input graph preprocessing and thus achieves 2.6× lower latency on average compared to GNN overlays that rely on preprocessing and target a newer-generation 7nm AMD Versal architecture with AI engines. Compared to an 8nm Nvidia RTX A6000 GPU, our NPU can achieve 5.8× higher throughput,19× lower latency, and 900× more energy efficient while being software-programmable. Finally, we presented a case study for using our enhanced NPU in real-time GNN-based multi-input multi-output (MIMO) antenna scheduling in 5G networks. This use case showed that, unlike other existing GNN acceleration solutions on FPGAs, our NPU can run on practical models that combine multiple GNN types and also meet real-time latency requirements.

ACKNOWLEDGMENTS

This work was funded by the Intel/VMWare Crossroads 3D-FPGA Academic Research Centre and NSERC. The authors thank Rishov Sarkar from the FlowGNN team for helpful answers about its design and performance.

REFERENCES

[1] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph Neural Networks: A Review of Methods and Applications," *AI open*, vol. 1, pp. 57–81, 2020.

[2] A. Derrow-Pinion, J. She, D. Wong, O. Lange, T. Hester, L. Perez, M. Nunkesser, S. Lee, X. Guo, B. Wiltshire *et al.*, "ETA Prediction with Graph Neural Networks in Google Maps," in *ACM International Conference on Information & Knowledge Management (CIKM)*, 2021.

[3] W. Luo, H. Zhang, X. Yang, L. Bo, X. Yang, Z. Li, X. Qie, and J. Ye, "Dynamic Heterogeneous Graph Neural Network for Real-time Event Prediction," in *ACM International Conference on Knowledge Discovery & Data Mining (KDD)*, 2020.

[4] Y. Liu, X. Shi, L. Pierce, and X. Ren, "Characterizing and Forecasting User Engagement with In-App Action Graph: A Case Study of Snapchat," in *ACM International Conference on Knowledge Discovery & Data Mining (KDD)*, 2019.

[5] M. Lu, Z. Han, S. X. Rao, Z. Zhang, Y. Zhao, Y. Shan, R. Raghunathan, C. Zhang, and J. Jiang, "BRIGHT: Graph Neural Networks in Real-Time Fraud Detection," in *ACM International Conference on Information & Knowledge Management (CIKM)*, 2022.

[6] W. Shi and R. Rajkumar, "Point-GNN: Graph Neural Network for 3D Object Detection in a Point Cloud," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[7] J. Shlomi, P. Battaglia, and J.-R. Vlimant, "Graph Neural Networks in Particle Physics," *Machine Learning: Science and Technology*, vol. 2, no. 2, 2020.

[8] O. Orhan, V. N. Swamy, M. Rahman, H. Nikopour, and S. Talwar, "Graph Neural Networks to Enable Scalable MAC for Massive MIMO Wireless Infrastructure," in *IEEE International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 2023.

[9] A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.

[10] Z. Tao, C. Wu, Y. Liang, K. Wang, and L. He, "LW-GCN: A Lightweight FPGA-based Graph Convolutional Network Accelerator," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 16, no. 1, pp. 1–19, 2022.

[11] B. Zhang, H. Zeng, and V. Prasanna, "GraphAGILE: An FPGA-based Overlay Accelerator for Low-latency GNN Inference," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2023.

[12] 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 *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[13] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[14] D. Buterez, J. P. Janet, S. J. Kiddle, D. Oglic, and P. Liò, "Graph Neural Networks with Adaptive Readouts," *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[15] T.-H. Kim and T.-H. Park, "Placement Optimization of Multiple LiDAR Sensors for Autonomous Vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 5, pp. 2139–2145, 2019.

[16] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar *et al.*, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[17] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz, and M. Langhammer, "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs," in *IEEE International Conference on Field-Programmable Technology (FPT)*, 2020.

[18] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional Networks on Graphs for Learning Molecular Fingerprints," *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All You Need," *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[20] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[21] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?" in *International Conference on Learning Representations (ICLR)*, 2018.

[22] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated Graph Sequence Neural Networks," in *International Conference on Learning Representations (ICLR)*, 2016.

[23] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing Graph Neural Networks: A Survey from Algorithms to Accelerators," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–38, 2021.

[24] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. Herbordt, "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[25] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A Flexible and Energy-Efficient Accelerator for Graph Convolutional Neural Networks," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[26] 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 *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[27] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[28] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN Accelerator with Hybrid Architecture," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[29] C. Zhang, T. Geng, A. Guo, J. Tian, M. Herbordt, A. Lit, and D. Tao, "H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture," in *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[30] F. Shi, A. Y. Jin, and S.-C. Zhu, "VersaGNN: A Versatile Accelerator for Graph Neural Networks," *arXiv:2105.01280*, 2021.

[31] L. Gong and Q. Cheng, "Exploiting Edge Features for Graph Neural Networks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[32] M. Langhammer, E. Nurvitadhi, B. Pasca, and S. Gribok, "Stratix 10 NX Architecture and Applications," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021.

[33] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[34] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," *arXiv preprint arXiv:2008.05000*, 2020.

[35] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A Study of Bfloat16 for Deep Learning Training," *arXiv:1905.12322*, 2019.

[36] Intel Corp., "DSP Builder for Intel FPGAs Advanced Blockset Handbook (HB_DSPB_ADV)," 2023.

[37] E. Kloberdanz, K. G. Kloberdanz, and W. Le, "Deepstability: A study of unstable numerical methods and their solutions in deep learning," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 586–597. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3510095

[38] W. Yan, W. Tong, and X. Zhi, "FPGAN: An FPGA Accelerator for Graph Attention Networks with Software and Hardware Co-optimization," *IEEE Access*, vol. 8, pp. 171 608–171 620, 2020.

[39] T. Yang, L. Hu, C. Shi, H. Ji, X. Li, and L. Nie, "HGAT: Heterogeneous Graph Attention Networks for Semi-Supervised Short Text Classification," *ACM Transactions on Information Systems (TOIS)*, vol. 39, no. 3, pp. 1–29, 2021.

[40] T. N. Kipf and M. Welling, "Semi-supervised Classification with Graph Convolutional Networks," *arXiv:1609.02907*, 2016.

[41] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković, "Principal Neighbourhood Aggregation for Graph Nets," *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
[42] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric," *arXiv:1903.02428*, 2019.
[43] P. Chen, P. Manjunath, S. Wijeratne, B. Zhang, and V. Prasanna, "Exploiting On-chip Heterogeneity of Versal Architecture for GNN Inference Acceleration," in *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, 2023.
[44] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx Adaptive Compute Acceleration Platform: Versal Architecture," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.
[45] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open Graph Benchmark: Datasets for Machine Learning on Graphs," *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
[46] G. Kasieczka, T. Plehn, J. Thompson, and M. Russel, "Top Quark Tagging Reference Dataset," 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2603256
[47] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective Classification in Network Data," *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.
[48] A. Boutros, E. Nurvitadhi, and V. Betz, "Specializing for Efficiency: Customizing AI inference processors on FPGAs," in *IEEE International Conference on Microelectronics (ICM)*, 2021.
[49] P. von Butovitsch, D. Astely, A. Furuskär, B. Göransson, B. Hogan, J. Karlsson, and E. Larsson, "Massive MIMO for 5G networks (Ericsson White Paper BNEW-23:004809UEN)," 2023.
[50] Y. Chen, Y. Wu, Y. T. Hou, and W. Lou, "mCore: Achieving Sub-Millisecond Scheduling for 5G MU-MIMO Systems," in *IEEE Conference on Computer Communications (INFOCOM)*, 2021.