POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Aggregation engine for Graph Neural Networks

Supervisor	Candidate
Prof. Maurizio ZAMBONI	Giovanni CAPOCOTTA

April 2023

Abstract

Graph Neural Networks (GNNs) [1] are a class of deep learning methods intended to analyze graph data. GNNs include two different phases: the Aggregation phase, in which each node gathers information about its neighbors, and the Combination phase, which usually acts as a Neural Network on the output of the first phase. While the Combination possesses many of the same characteristics as other kinds of NNs with regard to the dataflow and can be optimized accordingly, the Aggregation phase presents some distinctive properties that prevent efficient mapping on traditional NN processors, and requires novel dedicated hardware and software schemes. In this work, an Aggregation Engine is designed based on a 2-D square mesh Network-on-Chip of SIMD cores. In order to have fast execution and efficient resource utilization, it is necessary to partition the input graph optimally among the different PEs at compile time. Such partitioning has been tested with different objectives, and the results have been compared for five distinct input graphs with different mesh sizes and other design parameters combinations. The best-suited partitioning objective proved to be weighted min-cut, and the most appropriate mesh size a 6-by-6 mesh. The experiments also highlighted the network congestion as the main factor limiting the scalability of the design, and degrading performance for high feature sizes. The 6-by-6 mesh has finally been implemented on an FPGA platform in order to report logic utilization and critical path. The resulting design achieves an average speedup of 74.38% over a sequential execution.

Table of Contents

List of Tables

List of Figures

Acronyms

NN

Neural Network

GNN

Graph Neural Network

NoC

Network on Chip

GeMM

Generic Matrix Multiplication

IPU

Intelligence Processing Unit

ACC

Accumulation unit

FLOP

Floating-Point Operation

FLOPS

Floating-Point Operations per Second

PE

Processing Element

SIMD

Single Instruction Multiple Data

MIMD

Multiple Instruction Multiple Data

MAC

Multiply And Accumulate

SIPO

Serial-In-Parallel-Out

PISO

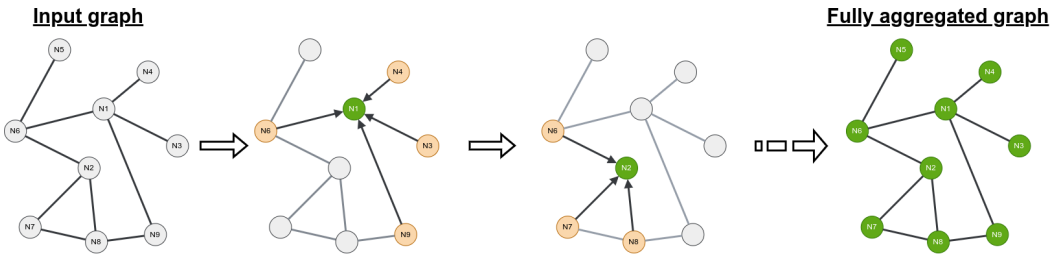
Parallel-In-Serial-Out

Chapter 1

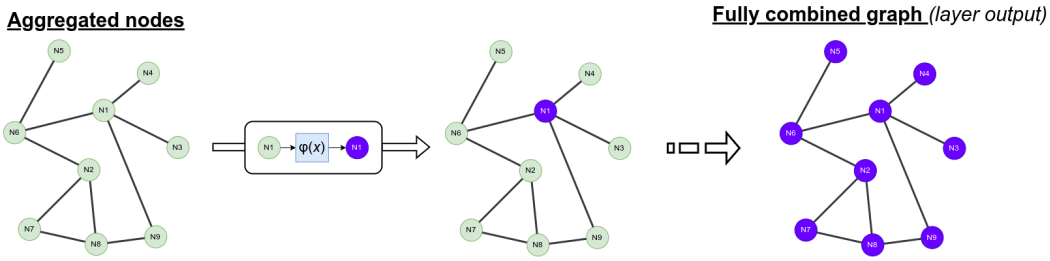
Introduction

1.1 What are Graph Neural Networks

While most Neural Networks are optimized to work on euclidean data, Graph Neural Networks (GNNs) are a class of deep learning methods specialized in processing data that is organized in a graph fashion. They are extensively used in various and diverse applications, such as accelerating drug discovery, predicting properties of molecules, forecasting traffic speed and targeted advertisement in e-commerce.
A general GNN layer is composed of two very different phases, as shown in algorithm 1 and illustrated in figure 1.1. The first is the aggregation phase, in which each node gathers information about its neighbors usually via a simple arithmetic operation. The second one is the combination phase, in which some more complex function (usually another kind of Neural Network) is applied to the aggregated node, producing the input to the next layer.



(a) Aggregation phase



(b) Combination phase

Figure 1.1: The two phases of a generic GNN algorithm. During the aggregation phase (a) the feature vector of each node is accumulated with those of its neighbours. During the combination phase (b) each aggregated node is taken as the input for a complex function, producing the output nodes of the layer. These operations are represented sequentially for simplicity, but intra-phase and inter-phase parallelization are possible.

1.2 Challenges of the GNN dataflow

Aggregation phase The aggregation phase presents a lot of challenges that are specific to GNNs, and sets them apart from the other kinds of NNs in regards to the dataflow. This phase can be modeled as a multiplication between the adjacency matrix of the underlying graph and the matrix of input features.

The matrix of input features is simply the matrix having the feature vector of the i-th node in the graph as its i-th row. The adjacency matrix is a square matrix used to represent the graph; if element (i, j) of the adjacency matrix is non-zero, then there's an edge going from node i to node j.

This adjacency matrix presents two distinctive properties [2]:

- it is extremely sparse, with over 99.9% of all entries being zeros for most graphs;
- the node degree usually follows a power law distribution;

both properties lead to execution problems in this phase. The high sparsity translates to highly irregular memory accesses, while the power-law degree distribution means that the workloads required to aggregate different nodes are very different.

Combination phase The combination phase follows a more traditional NN dataflow, as it can be modeled as a Generic Matrix Multiplication (GeMM) between the aggregated matrix and the weight matrix. The memory accesses are more regular and the dataflow can be optimized following traditional NN optimization techniques.

Algorithm 1 Generic GNN algorithm.

procedure GNN
    ▷ $h_v^{(l)}$ is the feature vector of node v at layer l
    ▷ $N(v)$ is the set of neighbors of node v
    ▷ $a_v^{(l)}$ is the aggregated feature vector of node v at layer l
    ▷ $\sigma(h_v^{(l)}, h_n^{(l)})$ is the aggregation function between node v and node n
    ▷ $\phi(a_v^{(l)})$ is the combination function

    $L \leftarrow$ number of layers
    $V \leftarrow$ set of nodes in the graph
    for $l = 1$ to L **do**
        for $v \in V$ **do**
            for $n \in N(v)$ **do**
                $a_v^{(l)} = \sigma(h_v^{(l-1)}, h_n^{(l-1)})$
            end for
            $h_v^{(l)} = \phi(a_v^{(l)})$
        end for
    end for
end procedure

1.3 State of the art

This section presents an overview of some of the current state-of-the-art GNN accelerators. Each of these deals with the problems brought forth by this particular dataflow in very different ways.

1.3.1 AWB-GCN

The first of these accelerators is Autotuning-Workload-Balancing GCN [2] (AWB-GCN). As the name suggests, this architecture focuses on adapting the workload at runtime, to redistribute it among a large number of PEs on the basis of the particular input graph it is working on without any preprocessing. In this way it aims at being efficient on a wide range of graphs with different applications and characteristics. In order to achieve this, three hardware techniques are implemented: dynamic distribution smoothing, remote switching and row remapping.

Dynamic distribution smoothing The architecture keeps track of the utilization of each PE by monitoring the number of pending task in their queues. During each round of computations, some of the tasks of the busier PEs are offloaded to their less busy neighbours, which then send back the results to the original PE. The result of this process is, indeed, a smoothing of the utilization profile, which mitigates uneven workloads between neighbor but doesn't solve the problem of local clustering, which is addressed by the remote switching technique.

Remote switching During each round of computation an autotuner keeps track of the most over-utilized and under-utilized PEs, and at the end of each round it swaps a fraction of their workloads. This fraction is different for each stage and computed at runtime. The switched PEs and the respective switch fraction are taken into consideration by the autotuner during the successive rounds in order to

eventually converge to an optimally balanced execution among all of the PEs.

Row remapping Due to the power-law degree distribution in most graphs, a small number of rows in the adjacency matrix will be very densely populated, giving rise to a huge spike in the utilization of the PE they are mapped to which cannot be adequately resolved by the switching hardware. To solve this problem these "evil rows" [2] are identified by the autotuner after a computation round and temporarily switched to a "Super-PE", which partitions the worst rows to a series of regular PEs that are managed by the Super-PE and are subject to both distribution smoothing and remote switching, since they could easily become the new peaks of the utilization profile otherwise.

Comparison with this thesis Overall, these schemes are definitely very effective in reducing workload imbalance, but the needed hardware is very complex as the architecture requires constant monitoring of the activity profile and even a controller to converge to a reasonable switching fraction. Since the graph structure is known a priori, adopting a run-time solution might not be the optimal choice: a good preprocessing of the input graph at compile time in order to try and load it onto the architecture in an optimal manner can prove equally effective, while avoiding the need for overly complex logic and thus reducing the area and energy consumption of the chip.

1.3.2 HyGCN

The HyGCN [3] (Hybrid GCN) architecture (shown in fig. 1.2) addresses the different demands of the accumulation and aggregation phases by having two separate engines coordinated by a communication interface, or *Coordinator*, which allows for them to work in parallel on the same layer.

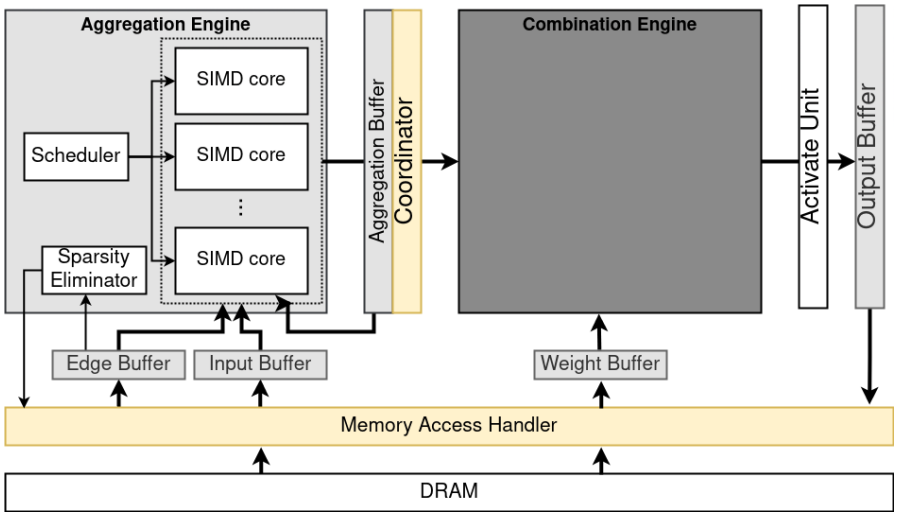


Figure 1.2: Block diagram of the HyGCN architecture, with details on the Combination Engine omitted.

Aggregation engine The aggregation engine presents a certain number of SIMD cores and a scheduler which dynamically assigns the workload to each core. Assigning all of the accumulations of a single node to each core would cause high latency and some heavy workload imbalance between the PEs, since the ones with less neighbours would need to wait for the more connected ones. To solve these problems and allow for intra-vertex parallelism (i.e. the accumulation of each feature in the feature vector can be done in parallel), HyGCN implement what the authors call "vertex-disperse processing" [3], which consists in assigning different parts of the feature vector to different cores. If the features of a given node cannot occupy all of the PEs, the free ones are assigned the features of a different node. Another big advantage of this loading scheme is that it allows for the accumulated node to be sent to the combination engine as soon as its accumulation is finished without waiting for the other nodes, thus enabling *inter-phase parallelism*.

Static graph partitioning In order to fit the input graph on the hardware, it is divided in subgraphs of equal sizes which are then processed one by one, where

the neighbours of each node are also accessed by traversing these subgraphs one at a time.
This has two benefits. In the first place, nodes in the same subgraph share a good number of neighbors, allowing for reuse of their feature vector without reloading it from the DRAM. Secondly, the partial accumulations for a given node are stored in the output buffer and will be reused until the execution moves to a different subgraph.

Sparsity elimination In order to reduce sparsity, a dynamic windowing approach is implemented on the adjacency matrix, with the aim of excluding the highest number of zero entries.

Combination engine The combination engine has multiple systolic arrays. These arrays are organized in groups, called *systolic modules*. These modules can either work independently (each module on a small group of nodes) or cooperatively (on a larger set of nodes, i.e. in burst mode). In independent working mode the each module operates on a small group of nodes. The benefit of using this mode is latency, since the modules need to wait for the aggregation of less nodes before starting the computation, and it works particularly well with the vertex-disperse processing.

In cooperative working mode many modules act as a single systolic array working on a larger group of nodes, meaning that the nodes are processed in burst mode, rather than in a vertex-by-vertex manner as in the previous mode. Though latency is lower, this greatly reduces energy consumption since the weights need to be fetched from the weight buffer only once and are then streamed to every node loaded to the array, thus enabling greater data reuse than the independent mode.

Comparison with this thesis The heterogeneous structure of the HyGCN processor is an example of the kind of system that the aggregation engine developed in this thesis could be a part of. Such a system opens up many possibilities for inter-phase optimization, and HyGCN takes full advantage of this by appropriately orchestrating the dataflow between its two engines.
Both HyGCN and the architecture proposed in this thesis make use of graph partitioning, but for two different reasons: while the partitioning implemented by HyGCN allows for the loading of a big graph on a hardware with limited memory, the one implemented in this thesis aims at reducing the amount of communication between parallel PEs.
While HyGCN needs to implement its windowing approach in order to deal with the matrix sparsity, the proposed architecture avoids this problem altogether by implementing a distributed memory instead of a centralized one. More details on the memory hierarchy are reported in chapter 2. HyGCN deals with the sparsity of the adjacency matrix through a windowing approach aiming at excluding the highest number of zero entries.

The architecture proposed in this thesis also utilizes graph partitioning, but for a different reason. While the partitioning implemented by HyGCN allows for the loading of a big graph on a hardware with limited memory, the one implemented in this thesis aims at reducing the amount of communication between parallel PEs.

1.3.3 Graphcore IPU

Graphcore introduced a new processing paradigm aimed specifically at Machine Learning workloads, the Intelligence Processing Unit (IPU) (shown in fig. 1.3).

IPU architecture The IPU processor is a MIMD (Multiple-Instruction-Multiple-Data) processor capable of working on a huge number of parallel and independent

threads with great granularity. It presents 1216 Processing Elements, called tiles, each of which contains a computing unit and a 256 KiB SRAM scratchpad [4]. Each tile can execute up to 6 threads at the same time through a technique similar to the Simultaneous MultiThreading (SMT) [5] found on CPUs, which helps in hiding instruction latencies. The PEs are interconnected by the IPU exchange. The tiles can reach "an impressive arithmetic throughput, up to 31.1 TFlops/s in single precision and 124.5 TFlops/s in mixed precision per chip"[4]. This is thanks to the *Accumulating Matrix Product* units (AMP), highly specialized and pipelined units capable of computing 64 mixed-precision or 16 single-precision FLOP/cycle, present in every tile.

Memory architecture The only memory available to the IPU aside from the register files is the distributed memory formed by the tiles' scratchpads, which adds up to 304 MiB in total. Having a distributed memory means that the threads running in each tile can access the necessary data efficiently even when the access patterns are irregular, and it allows for the processor to reach a very high nominal maximum bandwidth. The Graphcore Poplar [6] language and compiler takes care of scheduling the data transfers between the different tiles, sparing the programmer from having to do it explicitly.

Comparison with this thesis The architecture of the accelerator developed in this thesis is primarily inspired by the IPU processor, as both are a mesh NoC of SIMD cores working with a distributed memory architecture. The main differences are that the IPU cores support multi-threaded execution and the task of partitioning the graph among the cores is left to the software designer, while in the proposed aggregation engine every core runs a single thread, and graph partitioning is performed by the compiler in order to load the graph on the PE mesh optimally.

Figure 1.3: Block diagram of the IPU architecture.

1.4 Goal of this Thesis

As already stated, the combination phase can be optimized using traditional NN hardware and software techniques. Thus, the aim of this thesis will be to develop an hardware accelerator specific to the aggregation phase, and benchmark its performance on graphs with different characteristics. This accelerator could eventually either be improved to support the combination phase, or integrated

with a separate engine for the latter execution stage. This design has then been implemented targeting the "Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit" [7] in order to evaluate hardware metrics such as the critical path and the amount of logic utilised.

The architecture proposed in this thesis is a mesh Network-on-Chip (NoC), with each node of the mesh being a small SIMD core with a specialized instruction set. Figure 1.4 highlights the importance of using a good partitioning criterion to map the nodes onto each of the PEs, as the number of nodes that cross two different partitions is directly correlated to the amount of inter-PE communication that the network will have to sustain in order to complete the layer execution. More packets on the network at any given time mean a higher network congestion, which will inevitably degrade performance as the average transmission time of the packets increases. The main idea behind this design is to tackle this bottleneck by means of some preprocessing on the software side. Since the structure of the input graph is known at compile time, it can be optimally partitioned and each of the obtained subgraphs can then be loaded on a different core, reducing inter-core communication to a minimum. As discussed in section 4.1.2, a good partitioning criterion can also prove useful in balancing the workload between the different cores.

Benchmarking The architecture is fully parametrized. It is possible to change the data width of a single feature, the length of the feature vector, the number of PEs in the mesh and many other dimensions. This allows to perform some exploration of the solution space in order to highlight the advantages and drawbacks of different combination of parameters.

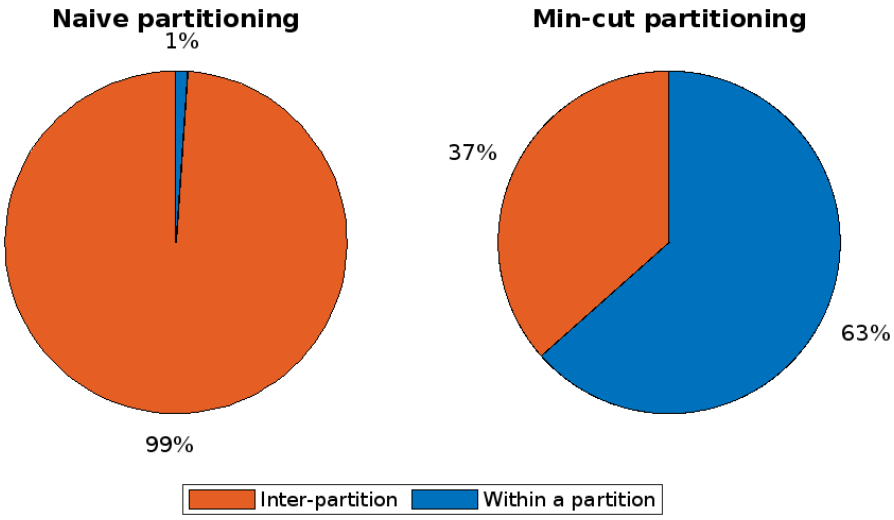


Figure 1.4: Distribution of the edges of graph cit-HepPh [8], with naive and min-cut partitioning.

Workflow The workflow to evaluate the performance on a given dataset is displayed in figure 3.1. First, the input graph is partitioned through the METIS partitioning tool using NetworkX-METIS [9]. Both the original and partitioned graph are processed via a Python script to generate the contents of the data and instruction memories of the PEs for each mesh size, as well as the reference values to check for the correctness of the simulation results. These contents are loaded onto the mesh during each RTL simulation, at the end of which the performance reports are generated.

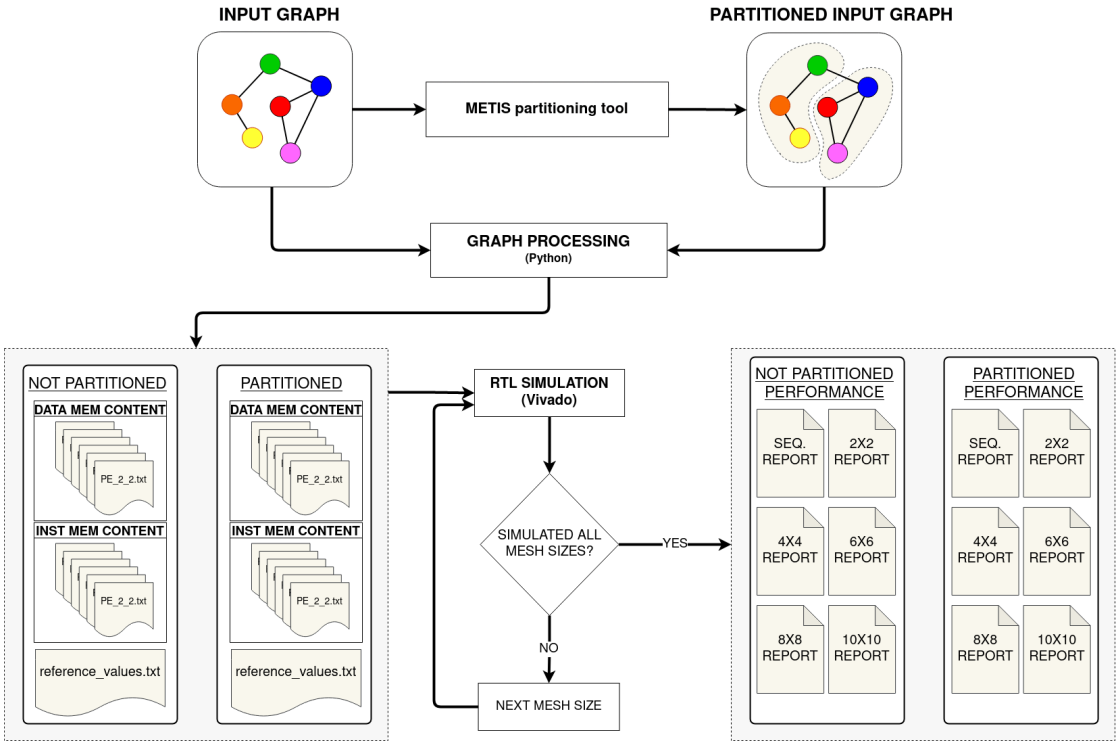


Figure 1.5: Workflow to generate the performance reports.

Chapter 2

Design of the architecture

The architecture consists of parallel processing elements (PEs) connected with each other through a 2D mesh network-on-chip (NoC). The PEs are responsible for the computation and the routers in the mesh NoC dictate the flow of the communication requests and responses between the different PEs. The architecture is designed as a parameterized template with design-time configurable mesh size and other PE and router parameters.

2.1 PE

The Processing Element is a core with two distinct data memories, an instruction memory and a array of Accumulation Units (ACC). The PE is made programmable with a custom instruction set as shown in table 2.1. Finite state machine (FSM) based control units are used to route the memory load requests to appropriate destinations (local SRAM or remote PE) and handle the data responses. The size of the feature vector is a design parameter, and the number of ACC units and scratchpads width are parameterized accordingly.

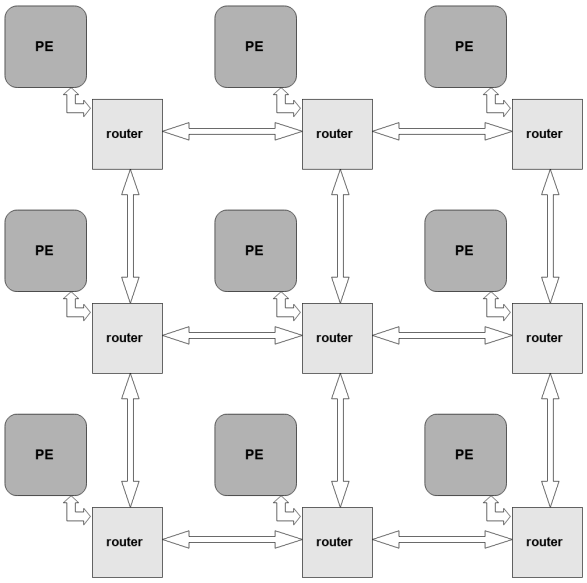


Figure 2.1: Example of a 3-by-3 mesh NoC

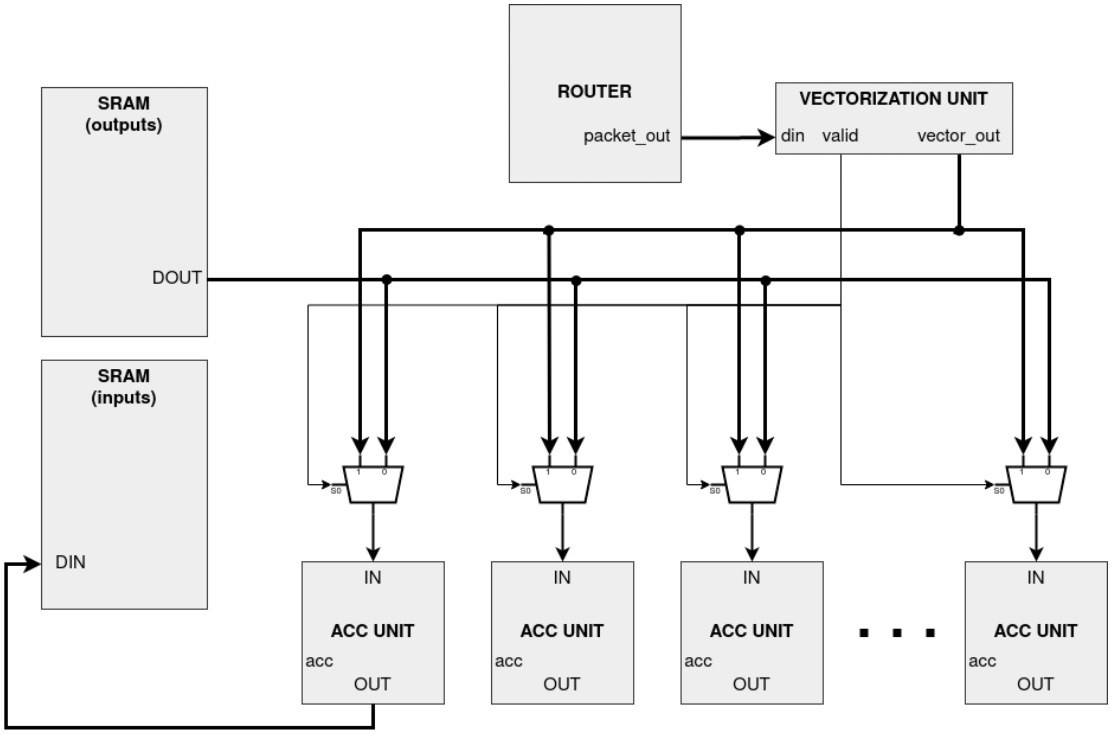


Figure 2.2: Block diagram of the access scheme to the ACC array.

Instruction set and decoder The custom instruction set is shown in table 2.1. An appropriate instruction decoder is used, which is a purely combinational unit with the job of interpreting the current instruction and generating the right control signals.

opcode	operand	description
000	`X`	**NOP**
001	[`X`][`X`][address]	**store** the ACC array output locally
010	$[X_{coord}][Y_{coord}]$[address]	**load** node to the ACC array
100	`X`	**accumulate with 0**
110	`X`	**accumulate internal**
111	`X`	**stop** current layer execution

Table 2.1: Instruction set. "`X`" can be any value.

Scratchpads The PE contains two dual-port SRAM scratchpads of parameterized height and width, which constitute a ping-pong memory buffer. One holds the input feature vectors of the current layer, and the other stores the outputs of the aggregation; every time a new layer begins, the role of the two memories is then swapped.

Data format In the PE's scratchpad, each feature vector is stored in a single memory location. During accumulation, each feature is sent to a different ACC unit and accumulated.

address	contents					
0x0000	F_0^0	F_1^0	F_2^0	F_3^0	...	F_N^0
0x0001	F_0^1	F_1^1	F_2^1	F_3^1	...	F_N^1
0x0002	F_0^2	F_1^2	F_2^2	F_3^2	...	F_N^2
...	...					

Table 2.2: Data structure in the scratchpad, where F_n^i is the n-th feature of the i-th node.

Load unit When the instruction decoder receives a *load* instruction, the load unit decides whether in order to load the requested vector it needs to access the local scratchpad or forward the request to the fetch request unit.

Fetch request unit The fetch unit extracts the X_{coord} and Y_{coord} from the load memory address and forms the fetch request packet according to the format in figure 2.9, and sends it to the local router. When a fetch request is sent, the PE is stalled until the request is satisfied, that is to say until all of the features of the requested feature vector reach the PE. This is necessary in order to correctly accumulate all of the neighbors to the base node, since if the PE were kept running it could store the partial results and load the next base node before accumulating the nodes that are not stored in its local scratchpad and didn't reach the PE in time after their fetch requests.

Fetch response unit The fetch response unit accepts the fetch request from the local router, accesses the right feature vector through the dedicated port of the SRAM, and sends it back with one packet per feature. The formation of the packets is shown in figure 2.4: when a valid request is detected by this unit, the coordinates of the PE that issued the request are stored in two registers; the requested vector is then stored in a parallel-in-serial-out (PISO) buffer, This buffer outputs one feature per clock cycle, which is concatenated with the stored coordinates and the '0' bit (indicating that the packet is part of a fetch response), and sent to the router.

Vectorization unit Once the packet containing a feature reaches the requesting PE, it is read by the vectorization unit, which holds a serial-in-parallel-out (SIPO) buffer. Once the buffer is full (i.e. all of the features of the requested feature vector

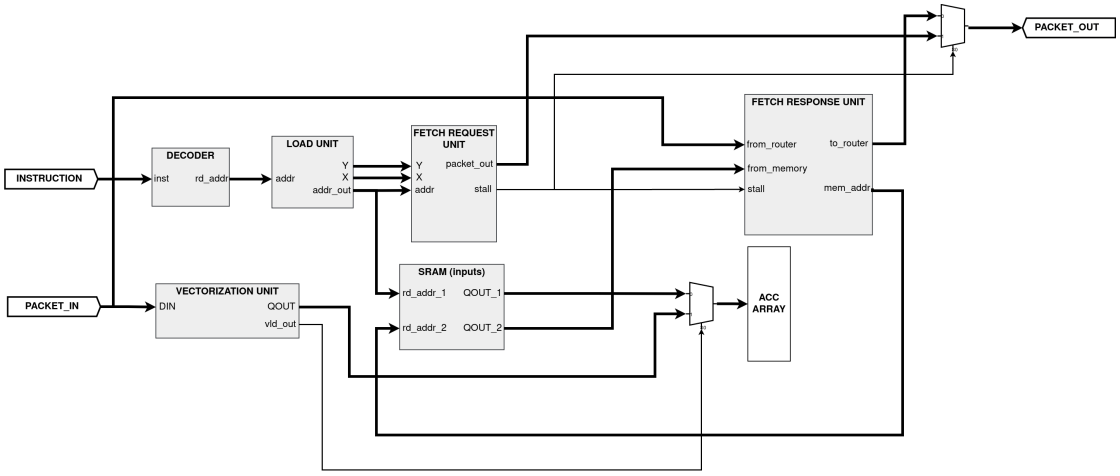


Figure 2.3: Schematic of the units responsible of the fetch requests, fetch responses and loading of the nodes to the ACC array.

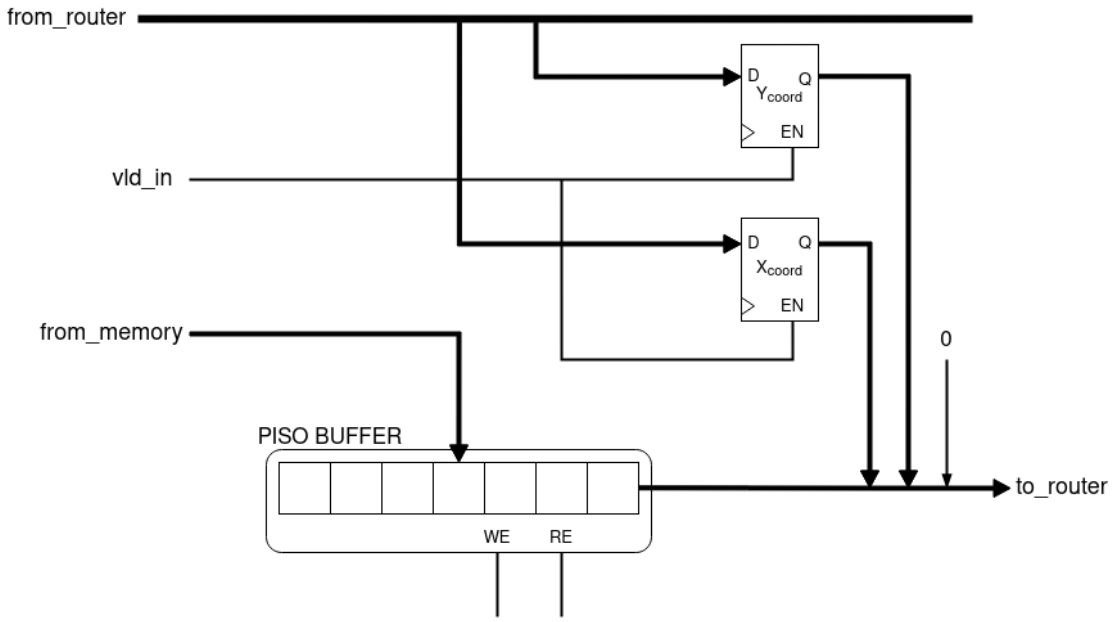


Figure 2.4: Diagram of the packets formation inside the fetch response unit.

have reached the PE), its contents are loaded to the ACC array and ready for accumulation.

2.2 Router

The routers have five ports, one for each neighboring routers and one for communication with the associated PE. In order to prevent deadlocks [10] and avoid needing some kind of deadlock detection scheme, X-Y (or dimension-ordered) routing is implemented, with a store-and-forward switching technique. The RTL description for the router can be found in appendix A.

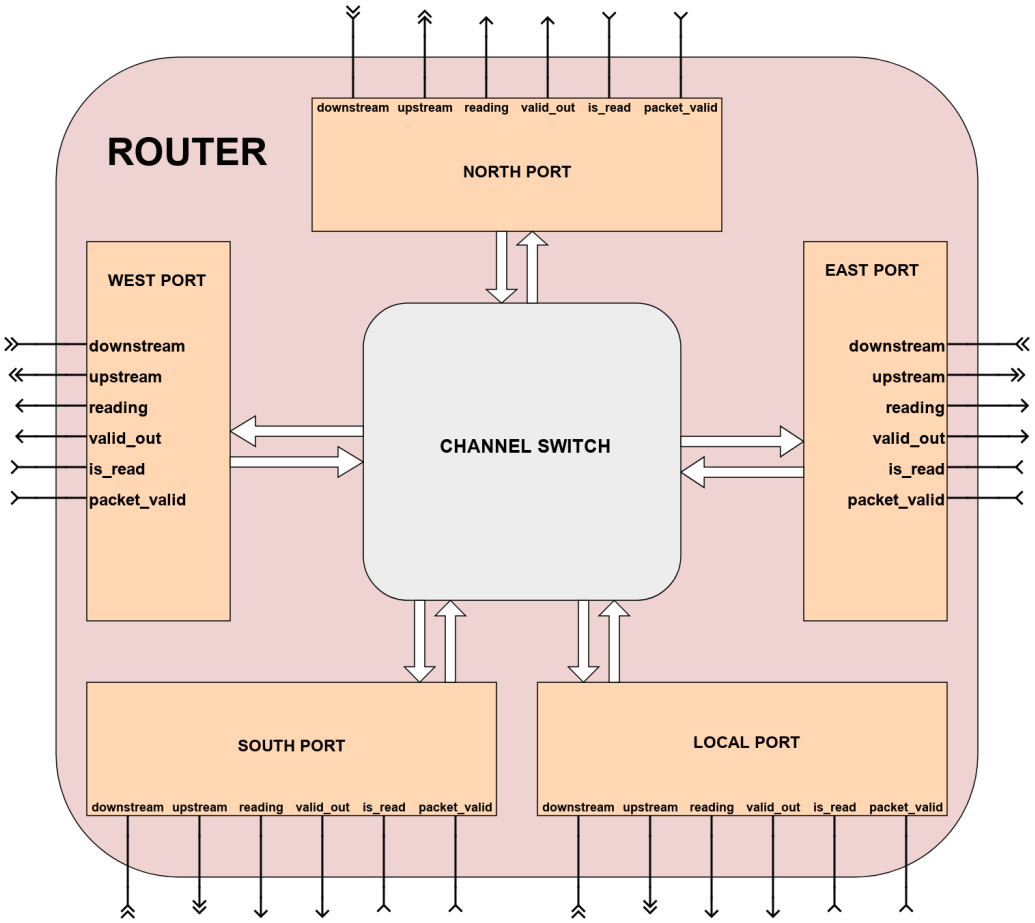


Figure 2.5: Router's internal structure and interface.

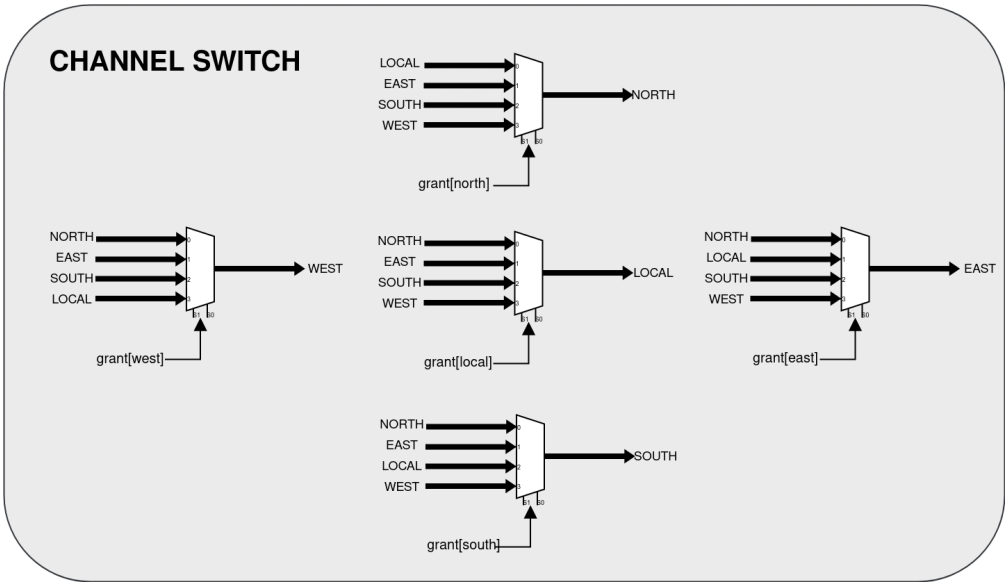


Figure 2.6: Channel switch internal structure. It multiplexes the output packet and the status and control signals from the input ports to a given output port, based on the grant for that port.

2.2.1 Input ports

Each router presents five *input ports* modules. Each input port implements the receiver and transmitter protocols with an output port, with which they are connected by a channel switch depending on the grant given by an arbiter. Each input has a FIFO buffer of parameterized depth, whose contents are routed to the next router or to the local PE when the respective port is free.

Each input port has the following interface:

- `upstream`: output to send the packets;
- `reading`: asserted when the port is reading data;
- `valid_out`: asserted when data is ready to be sent to the next destination;
- `downstream`: input to receive the packets;

- `is_read`: asserted by a neighboring router when it is reading data from this port;

Transmission protocol Whenever its FIFO buffer is not empty, the port asserts the `packet_valid` output. The `is_read` input of the port corresponds to the `read_en` signal of the FIFO, thus after one clock cycle from the assertion of this signal by the receiver, a packet is sent.

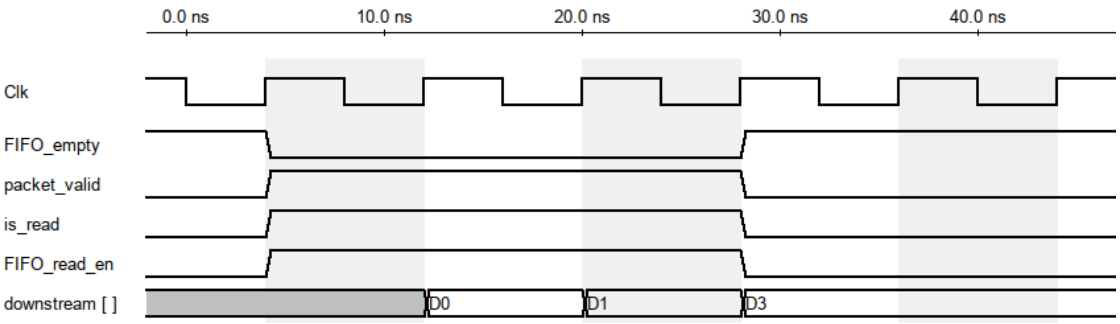


Figure 2.7: Timing diagram of the transmission protocol

Reception protocol When the `packet_valid` input is high, and the FIFO is not full, the `reading` output and the `write_en` signal of the FIFO are asserted.

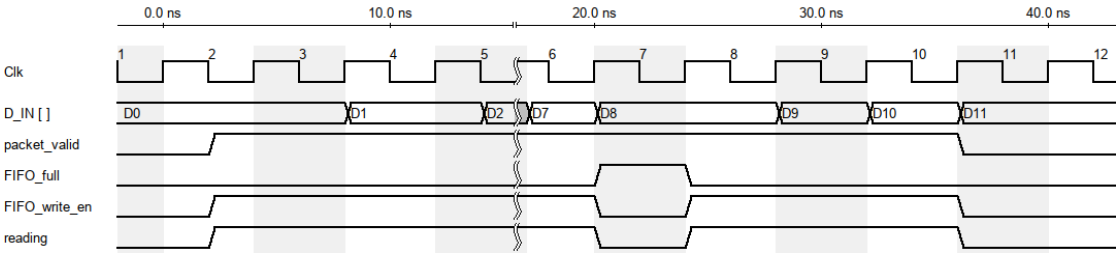


Figure 2.8: Timing diagram of the receiver protocol

Packet format When a PE issues a request for the fetch of a remote feature vector, this is fed to the respective router according to the data structure in figure 2.9: X_{dest} and Y_{dest} are the coordinates in the mesh of the PE that is the target of

the fetch request, X_{src} and Y_{src} are the coordinates the PE issuing the request, and $ADDR$ is the address of the feature vector in the memory of the destination PE.

1	X_{dest}	Y_{dest}	X_{src}	Y_{src}	$ADDRESS$

Figure 2.9: Data structure of a fetch request.

Once the destination PE receives a fetch request, it sends the requested feature vector to the requesting PE as a sequence of packets, each containing one feature, in the format shown in fig. 2.10. Note that X_{dest} and Y_{dest} in these packets are the coordinates of the requesting PE.

0	X_{dest}	Y_{dest}	$FEATURE$

Figure 2.10: Data structure of a fetch response.

2.2.2 Output arbitration

Each port has a respective arbiter. The arbiter implements a Round-Robin arbitration scheme, which has been chosen for its simplicity and for a fair bandwidth allocation across the channels. The RTL description for the arbiter can be found in appendix A.

Round-robin arbiter The round-robin arbitration is a good scheme in those cases when there isn't any requester that should have a higher priority over the others, and it makes sure that the access to the resources is unbiased and starvation free.

As shown in figure 2.11, it works in a similar way as fixed-priority arbitration, but every time an access is granted the priority of each requester shifts in a circular manner, so that the requester that comes immediately after the one that just got access to the resource becomes the one with the higher priority.

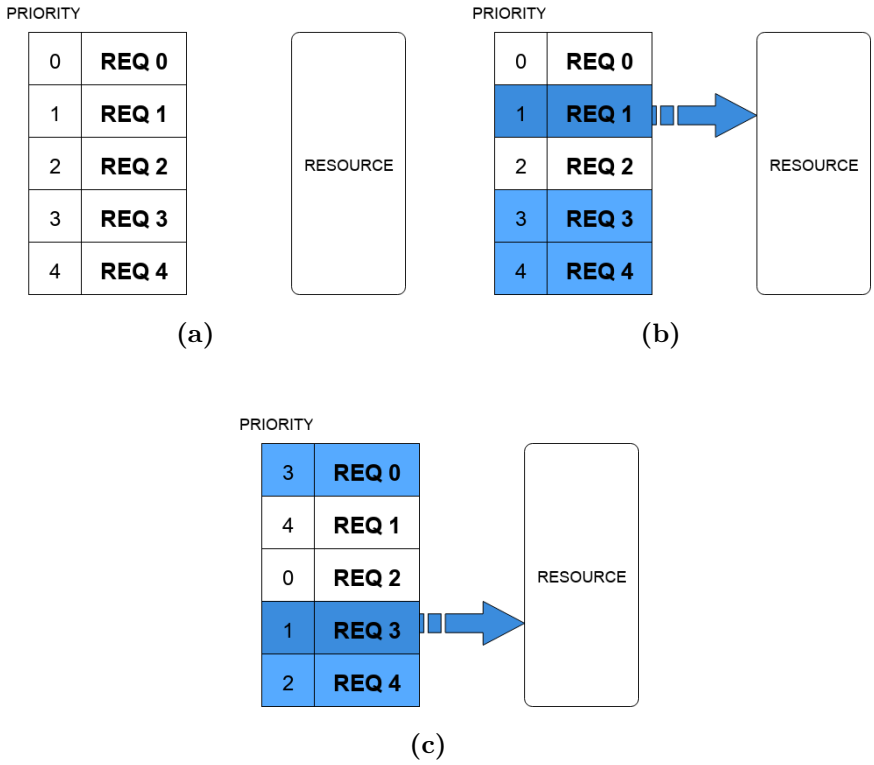


Figure 2.11: Round-Robin arbitration: each requester is given a priority (a). The highest priority request is granted (b) (the active requests are in blue). After each grant, the priorities are rotated (c). The priority pointer indicates the requester with highest priority.

In summary, the accelerator consists of a mesh of cores, the scratchpads are the only form of memory available to the engine and constitute a distributed memory architecture. Each PE holds a subset of nodes of the input graph in its local memory. When a node needs to be loaded to the ACC array for accumulation, the load unit detects whether or not it is stored in the local memory. If it is, the node is loaded, otherwise a fetch request is issued by the fetch request unit. The fetch request is transmitted by the routers to the destination PE. This PE then sends back the requested node, one feature at a time, through the fetch response unit. Each incoming feature is stored in the vectorization unit of the requesting

PE, until the last one arrives and the whole feature vector is finally loaded to the ACC array. Most of the dimensions in the architecture are parameterized in order to be able to experiment with different combinations and observe in which cases the performance increases and degrades.

Chapter 3

Testing

3.1 Memory contents

Before proceeding with the testing of the architecture, the data and instruction memories of the different cores need to be populated with the right input nodes and instructions respectively.

3.1.1 Graph preprocessing

All of the graph preprocessing is done in Python. This includes the partitioning of the graph through the NetworkX-METIS [9] partitioning tool and the generation of the text files containing the memory contents and the correct values of the graph nodes after one aggregation layer.
The scripts used for this can be found in appendix C. The input graph is stored in a text file containing every couple of nodes connected by an edge; every graph is processed as undirected. After specifying the parameters for the current test (i.e. data width, feature size and number of cores), the graph is partitioned following the chosen partitioning criterion, while each node's ID is used as an address in the

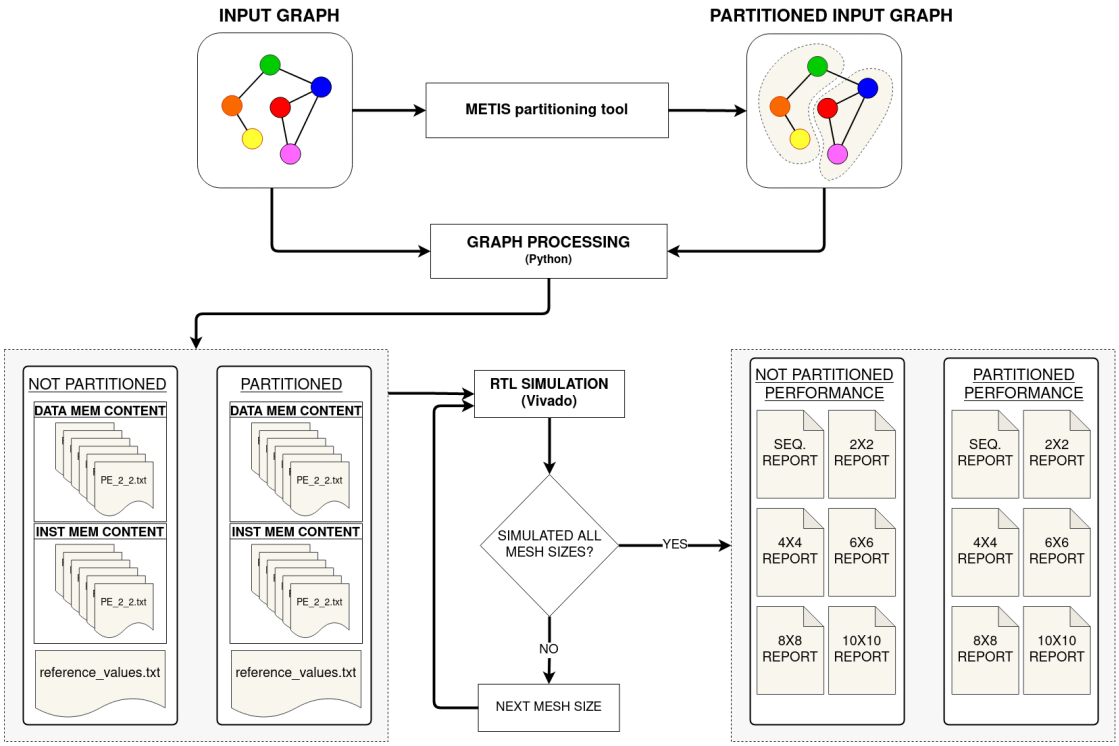


Figure 3.1: Workflow to generate the performance reports.

distributed memory for the naive partitioning. The script subsequently generates the data memory contents for the two cases based on the results of the partitioning and the node IDs. The instruction memory contents are generated by traversing the graph and adding an aggregation instruction for each neighbour of every node. At the same time, the reference values are generated by aggregating neighbouring nodes in the script itself; since the nodes are stored in a different order in the partitioned and naive approach, two different reference values files are also needed, otherwise in at least one of the two cases there wouldn't be any correspondence between the address of an input node and the one of the corresponding output.

3.2 Testbench

The goal of the testbench is not only to verify the correctness of the layer outputs, but also to generate the files reporting some interesting metrics:

- the number of clock cycles needed to finish the layer;
- for each core, the amount of cycles in which it is stalled (i.e. waiting for a fetch request to be answered);
- for each router, the amount of cycles in which there is a conflict at at least one of its output ports;

The testbench can be found in appendix B. It takes the memory content files generated during the graph preprocessing and uses them to populate the instruction and data memories of each PE. The architecture is then simulated until the testbench detects that the aggregation layer is finished through the apposite `done` signal coming from the cores. During simulation, it evaluates and updates the value of the aforementioned metrics. Once the simulation ends, it inspects the memory contents of the core and checks them against the reference values stored in the corresponding file to make sure that the results are correct.
The simulations for the partitioned and "unpartitioned" cases for each graph were then run in batch via a .tcl script.

Chapter 4

Results

4.1 Performance

The performance of this architecture has been tested via behavioural simulation on five distinct input graphs and different feature sizes, with the goal of exposing the bottlenecks and identifying the most efficient mesh size in terms of latency, and finding out whether the optimal size is the same for all graphs or if it depends on its characteristics.

4.1.1 Choice of graphs

The input graphs chosen for the performance evaluation are listed in table 4.1 and come from different applications.

In order to be as exhaustive as possible in simulating the various execution scenarios of the engine, they all present a different number of nodes and edges. The number of edges is a particularly important figure, since it determines the number of cycles needed to accumulate the average node but also the amount of inter-PE communication necessary to process the graph and the effectiveness of the

Dataset	Node no.	Edge no.	Feature length	Type
email-Eu-core [11]	1 005	16 705	5	Email network
cit-HepPh [8]	34 546	421 578	20	Citation network
soc-Slashdot0922 [12]	82 168	948 464	10	Social network
soc-sign-epinions [13]	131 828	711 783	1	Review site
web-NotreDame [14]	325 729	1 117 563	5	Web pages

Table 4.1: Graphs used for the performance evaluation

pre-partitioning, which might not prove too useful for densely connected graphs. For the purposes of benchmarking, the feature vectors are randomly generated and graph edges are always treated as undirected (although the architecture can also support directed edges).

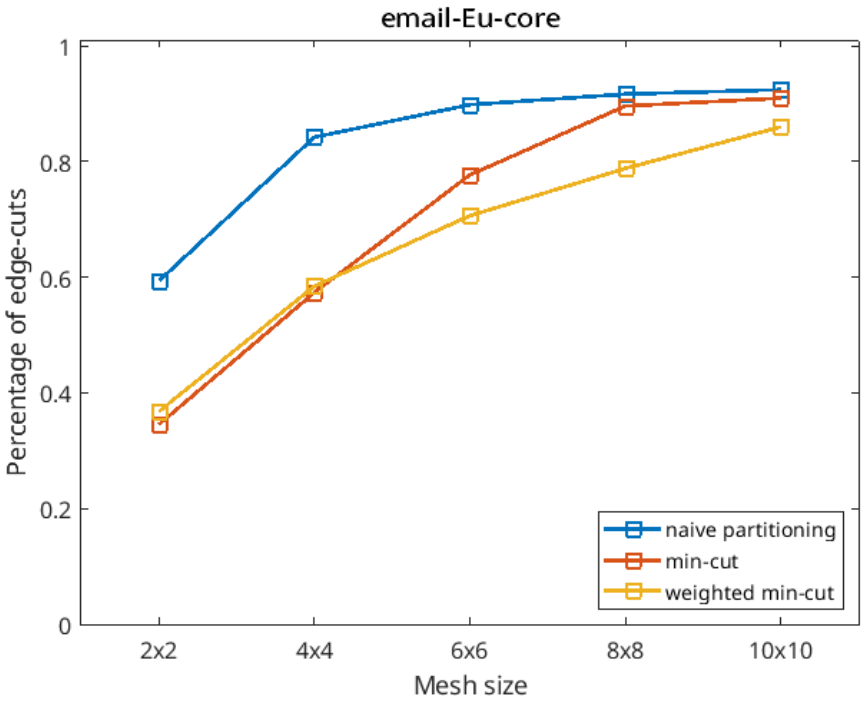
4.1.2 Partitioning criterion

Pre-partitioning the input graph can have two main goals: (1) minimizing the inter-subgraph communication volume in order to reduce network congestion, and (2) balancing the workload among the PEs. These objectives are not necessarily compatible, since the partitioning with the minimum number of edge-cuts (i.e. edges going from one subgraph to another) isn't necessarily well-balanced in terms of workload, while one with a very evenly balanced workload among every core might cause some of the subgraphs to be very connected with the others. For these reasons, two different partitioning objectives have been tested:

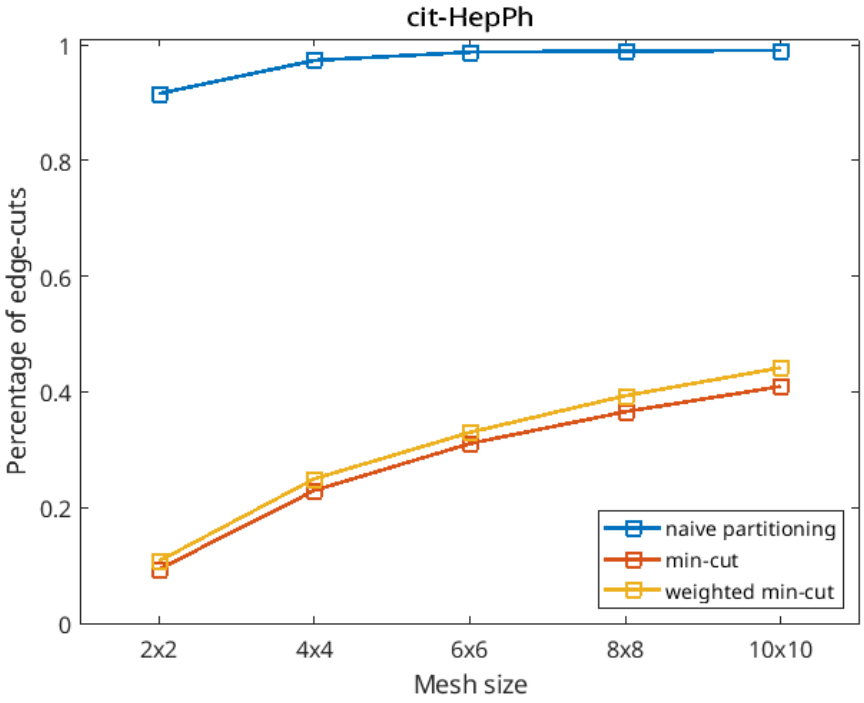
- **min-cut**: partitioning with the sole the goal of minimizing the edge-cut number;
- **weighted min-cut**: each node is assigned a weight equal to the amount of accumulation operations required for its aggregation, which is same as the number of edges connected to it. The partitioner tries to minimize the edge-cut number while keeping a similar sum of the node weights for every subgraph.

The different graphs have been partitioned with both methods, and a "naive partitioning" (i.e. assigning nodes to the PEs in order of node ID) has been used as a baseline. Figure 4.-1 shows the ratio of the number of edges crossing two different partitions to the total number of edges for the different graphs: the lower this number, the less communication there is between the PEs, meaning that the accelerator should perform better. For most of the tested cases, both proved equally useful in reducing the communication volume, with the "min-cut only" objective generally being only slightly more effective. One exception is the case of "email-Eu-core" 4.1a, where the weighted partitioning actually converged to a better result.

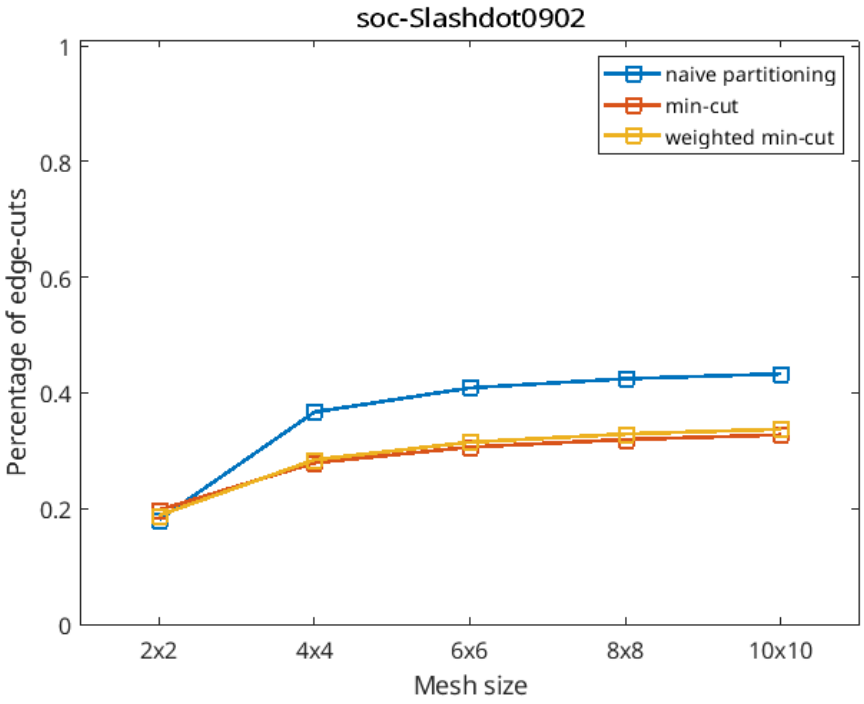
Since the degree of a node represents the workload required to fully accumulate it, the weighted min-cut partitioning is optimizing for both network traffic minimization and workload re-balancing. Taking this and fig. 4.-1 into account, this partitioning should prove more efficient than min-cut alone. This claim is supported by the benchmarking of the architecture discussed in the next section.



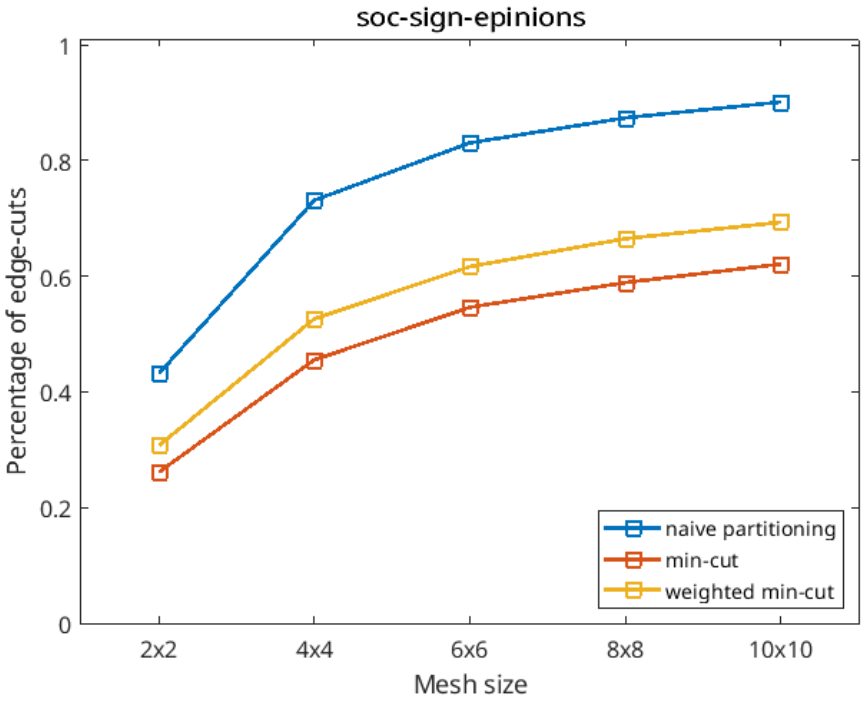
(a) email-Eu-core



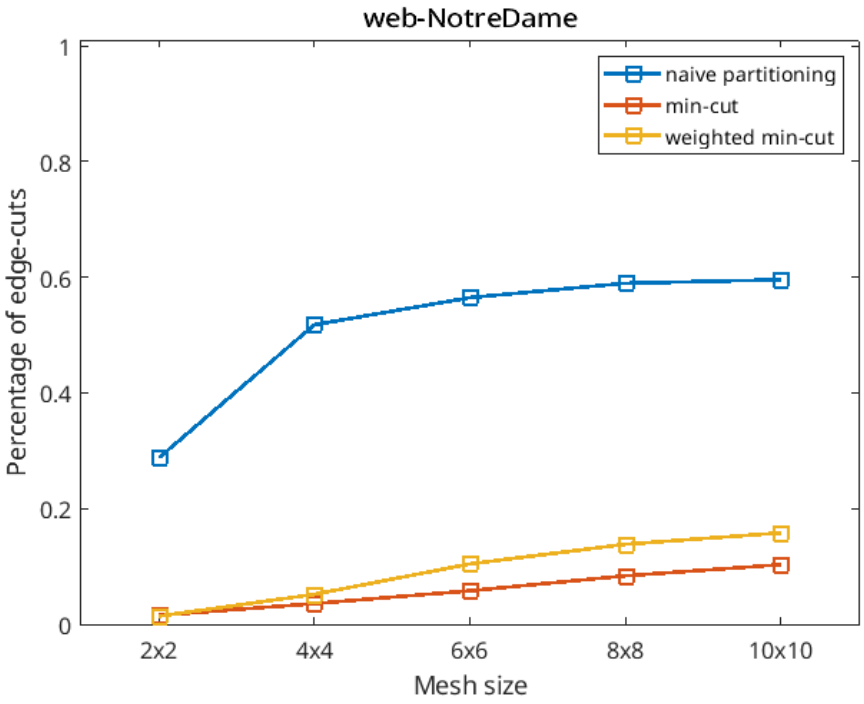
(b) cit-HepPh



(c) soc-Slashdot0902



(d) soc-sign-epinions



(e) web-NotreDame

Figure 4.-1: Ratio of edge-cuts to total edges, with the three different partitioning schemes.

4.1.3 Performance evaluation

The aggregation of each graph has been run sequentially (i.e. entirely on a single PE) as a baseline, and on a square mesh of increasing size, up to 100 PEs. On every mesh size, the latency has been evaluated for all of the three partitioning schemes.

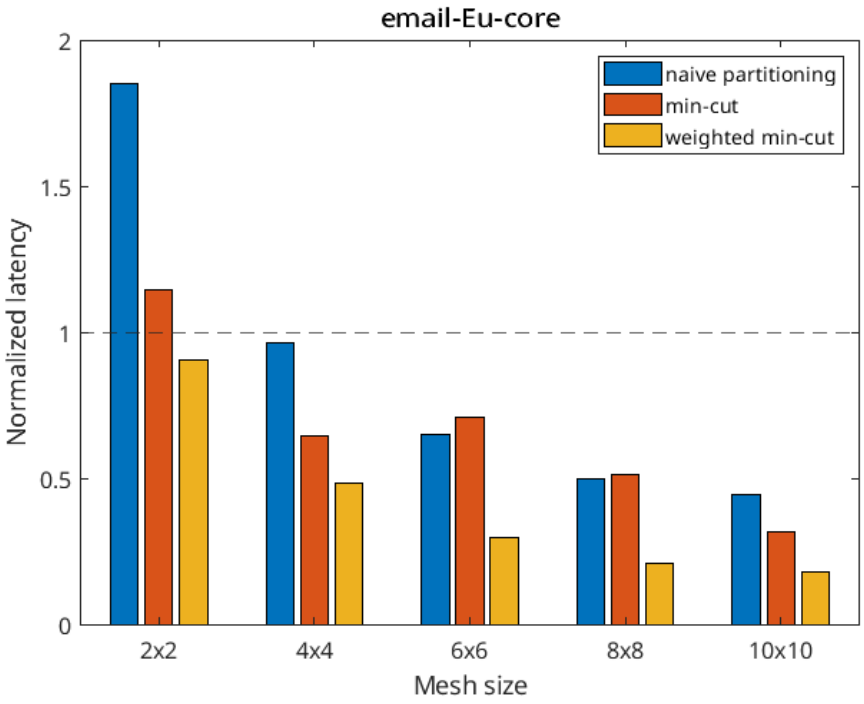
The results presented in figure 4.-2 demonstrate how prepartitioning the graph with the weighted min-cut method consistently brings a considerable latency decrease in the layer processing, in some cases reaching even over a 90% improvement with respect to the naive partitioning. Other than that, there are two interesting

trends in the results of these experiments.

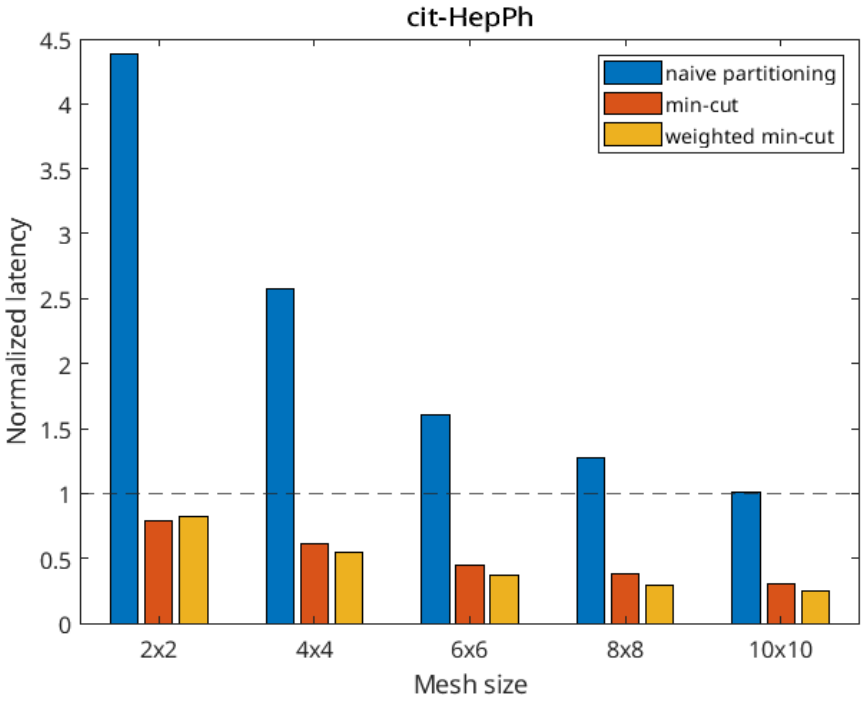
- First, having more PEs in parallel doesn't guarantee a lower latency that the sequential execution, as graph partitioning proves to be essential in order to utilize this architecture effectively.

- Secondly, the performance improvement for the weighted min-cut slows down as mesh size increases, while the other two partitioning methods can even become less efficient in some cases.

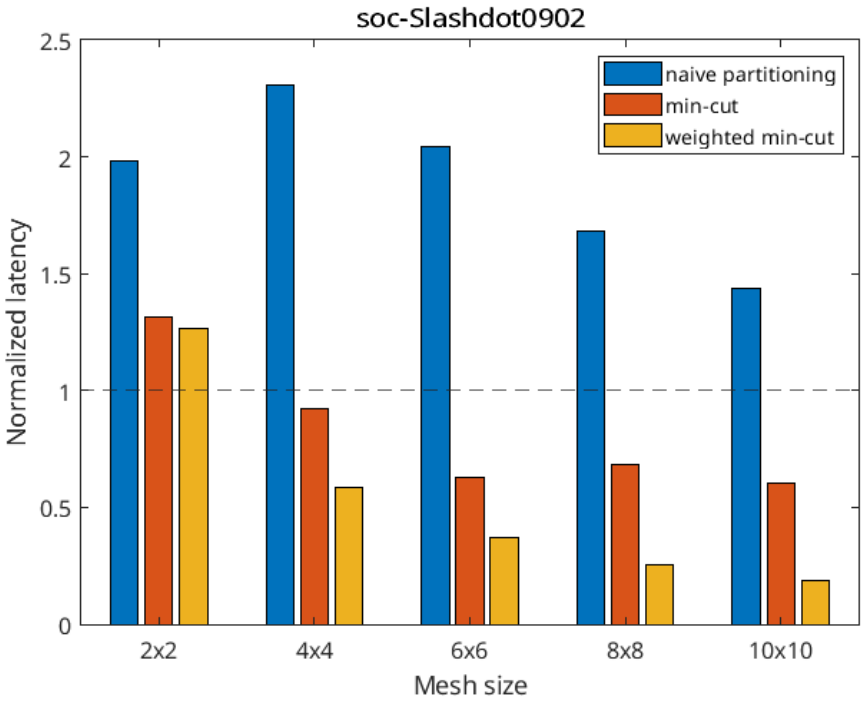
The reason behind both of these tendencies lies in the volume of network traffic. The inefficiency of naive partitioning highlights the fact that inter-PE communication poses a big latency overhead, and thus represents a crucial bottleneck for the performance of the accelerator. The non-monotonous results for naive and min-cut partitioning, on the other hand, display the weakness of the mesh network topology: as the mesh size increases, the PEs will have to exchange packets with nodes in the mesh that are further and further away. Every router the packet has to cross in order to reach the destination PE stores it inside one of its FIFO buffers until every other packet that reached it sooner is granted access to the respective output port; this adds a significant delay to the transmission time. For weighted min-cut partitioning the latency consistently decreases for bigger mesh sizes because this effect is somewhat compensated by the workload re-balancing, but it is still visible as the performance gain in using bigger and bigger meshes reaches a plateau.
Overall, these trade-offs between parallelization and network congestion result in the most appropriate size being a 6-by-6 mesh with weighted min-cut partitioning, since the latency decrease between testing the same graph on a 6-by-6 and a 10-by-10 mesh ranged from 2.6% to 18.6%, which hardly justifies the 278% increase in logic usage. This configuration presents an average performance improvement of 74.38% over the sequential execution.



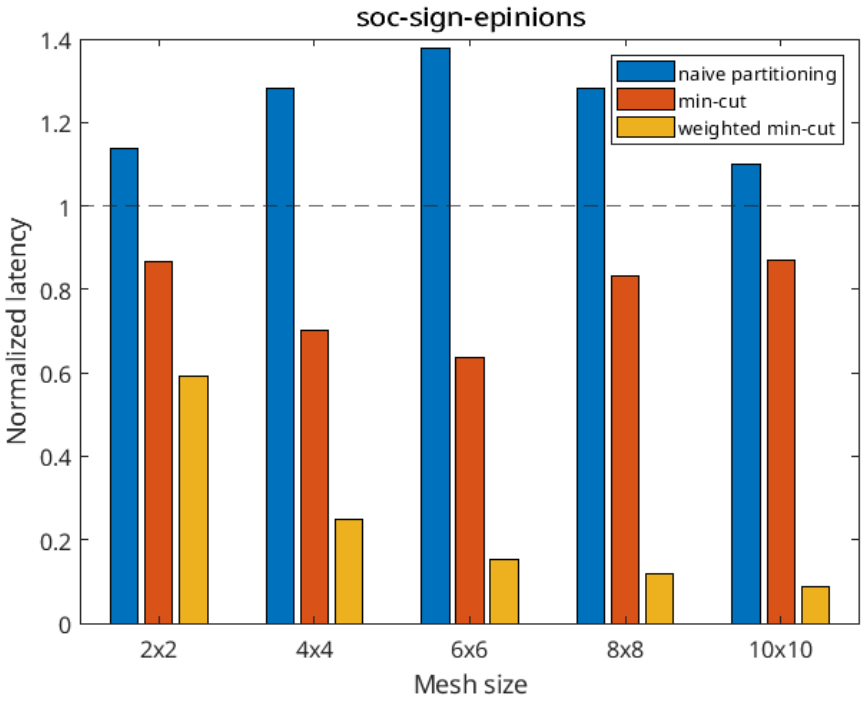
(a) email-Eu-core



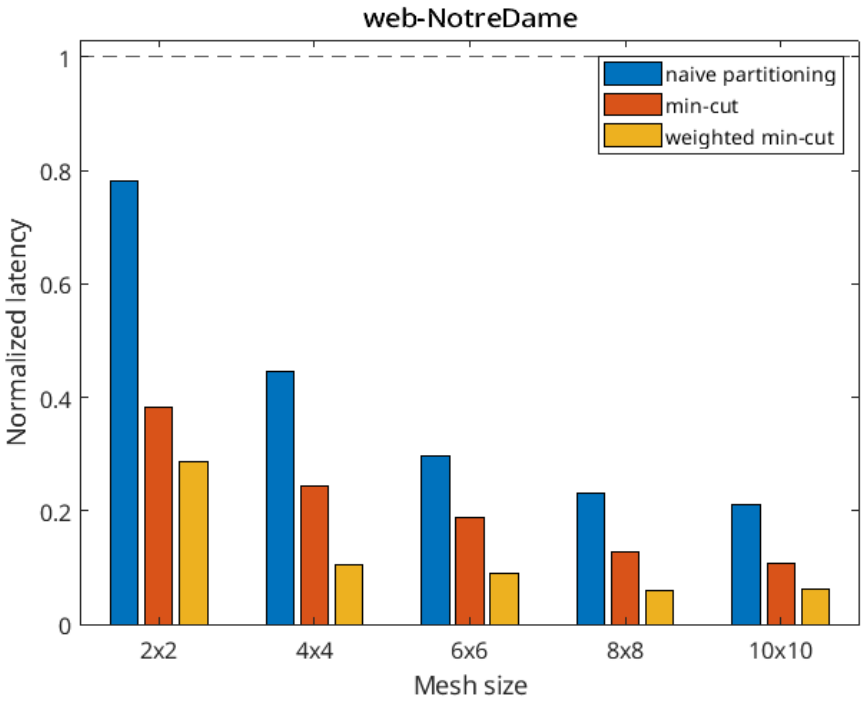
(b) cit-HepPh



(c) soc-Slashdot0902



(d) soc-sign-epinions



(e) web-NotreDame

Figure 4.-2: Latency for a single layer, normalized with respect to the sequential processing of the layer (i.e. a single PE)

Impact of the feature size The experiments in figure 4.-2 have highlighted how excessively increasing the mesh size will lead to network congestion, leading to little to no benefit with respect to a smaller network. For this same reason, another factor with a big impact on latency is the graph's feature size. Ideally the feature size shouldn't impact latency at all, since the number of ACC units in the PEs matched the feature length for every experiment, but due to the fact that the feature vectors are sent over the network after being serialized, having a higher feature size means that more packets will be sent for each fetch response, which contribute in overcrowding the router's FIFO buffers. Figure 4.-1 shows the normalized execution time of a layer of the "cit-HepPh" dataset on a 6-by-6 mesh with different feature lengths: the latency increases almost linearly with the feature

size, demonstrating the bottleneck due to packet serialization in the network.

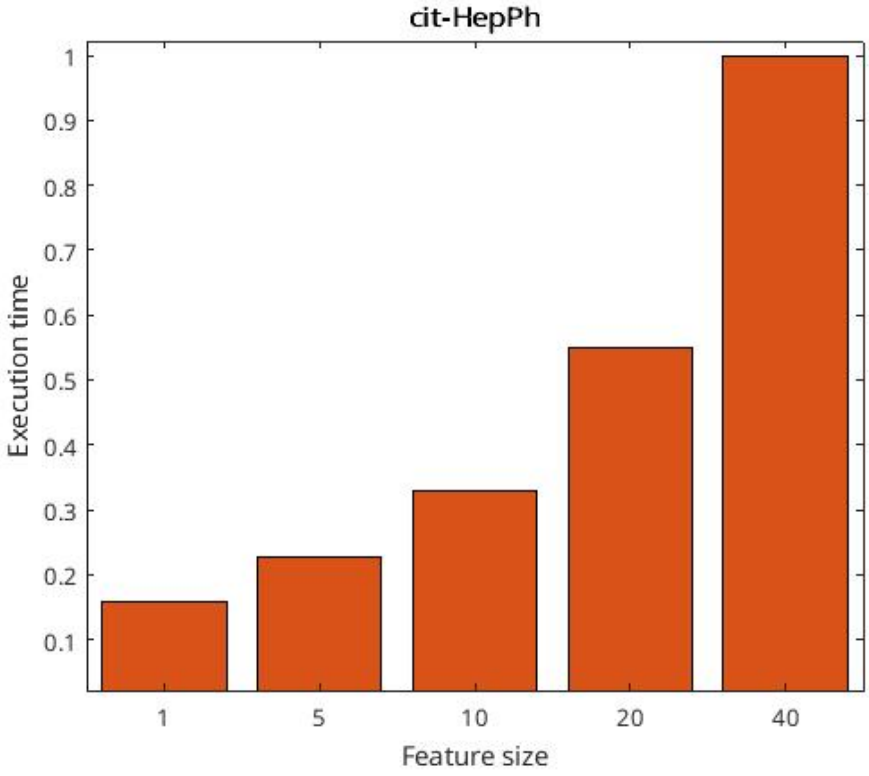
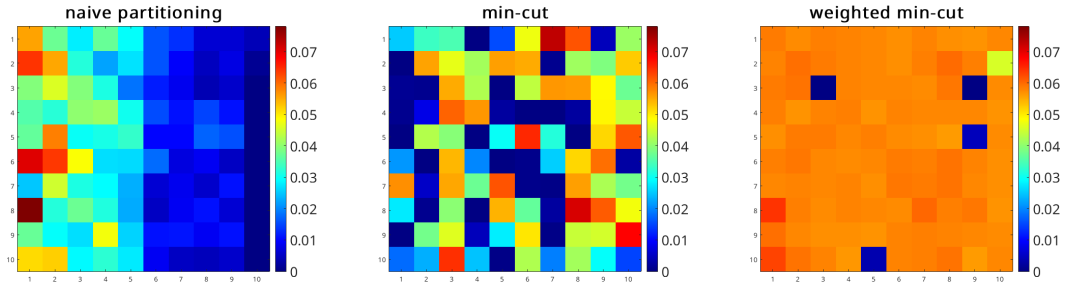


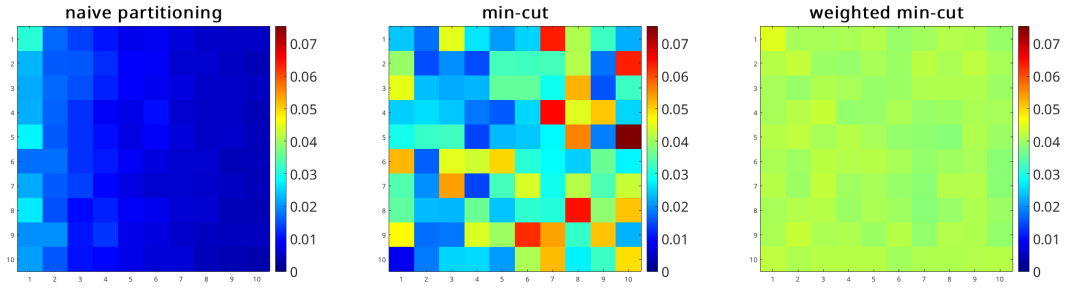
Figure 4.-1: Execution time for cit-HepPh on a 6x6 mesh with different feature sizes, normalized.

Utilization As already stated, partitioning the input graph with weighted min-cut has the added benefit of balancing the workload among the different PEs. Figure 4.0 shows the utilization profile of a 10-by-10 mesh with different input graphs, with the three partitioning schemes. While it is still low because of the packets transmission overhead, the utilization profile for weighted min-cut partitioning is higher on average and more uniform than min-cut alone, as the workload re-balancing implemented by this partitioning criterion leaves less PEs underutilized. As a result, even though in most cases the communication volume for the two methods is comparable (see fig. 4.-1), weighted min-cut partitioning consistently

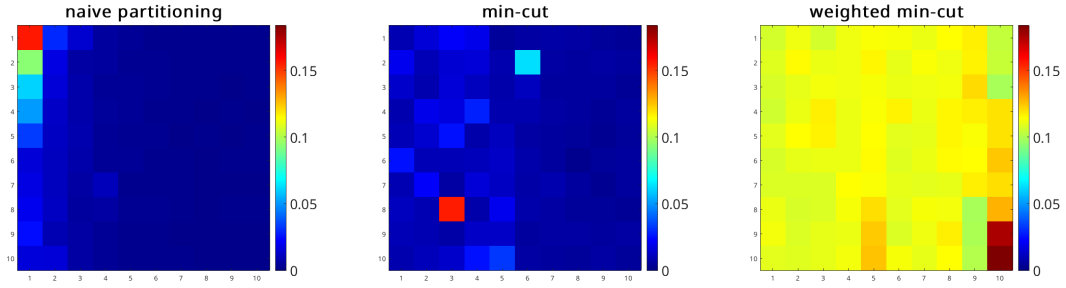
achieves better results (fig. 4.-2). It is important to note that this effect only addresses the workload imbalance between the clusters of nodes loaded onto the PEs, and not the very connected nodes which require a much higher number of accumulations than average, since every node is fully accumulated on a single PE.



(a) email-Eu-core



(b) cit-HepPh



(c) soc-sign-epinions

Figure 4.0: Heat maps showing the utilization profile for a 10-by-10 mesh.

In summary, performing these experiments pointed out the various trade-offs the architecture is subject to and the parameter combination best suited for it (i.e.

6-by-6 mesh), while also highlighting some limitations:

- optimally partitioning the graph is crucial to obtain a good efficiency, as having too much inter-PE communication and workload imbalance have a serious impact on latency;
- there is a trade-off between the number of PEs working in parallel and the transmission time of the packets, which causes the latency decrease with respect to mesh size to reach a good compromise between latency and logic utilization at 36 parallel PEs (6-by-6 mesh).
- graphs with a high feature length inherently incur in higher latencies due to packet serialization in the NoC routers. This can be possibly alleviated with wider connections between routers at the expense of higher physical area for routing.

4.2 Synthesis

In order to obtain a metric of the critical path and area occupied by the architecture, the design has been synthesized through Xilinx Vivado ML Editions™v2021.2 [15], targeting the FPGA "Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit" [7].

Resource utilization The architecture has been synthesized with the parameters in table 4.2. There are 1.34 MiB of memory for each PE, adding up to 48.2 MiB of total on-chip memory. The resource utilization for the target platform is reported in figures 4.1 and 4.2. The distinction between memory LUTs and logic LUTs in figure 4.2 stems from the fact that only around 44% of the total LUTs on the target platform are able to be utilized as memory.

Parameter	Value
Data width	8
Feature size	40
Data memory cells	16384
Instruction memory cells	32768
Mesh size	6×6

Table 4.2: Synthesis parameters.

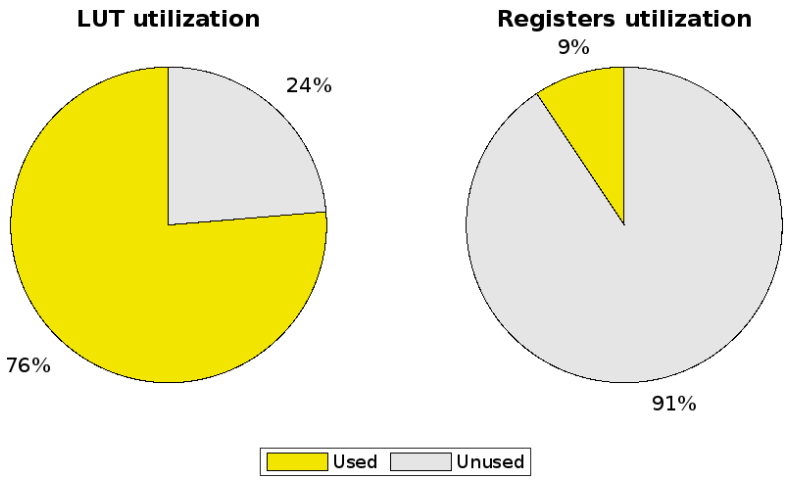


Figure 4.1: Utilization percentage of LUTs and registers for the 6-by-6 mesh.

Critical path The post-synthesis critical path for the architecture is 4.579 ns, resulting in the layer latencies for the simulated graphs with optimal partitioning indicated in table 4.3

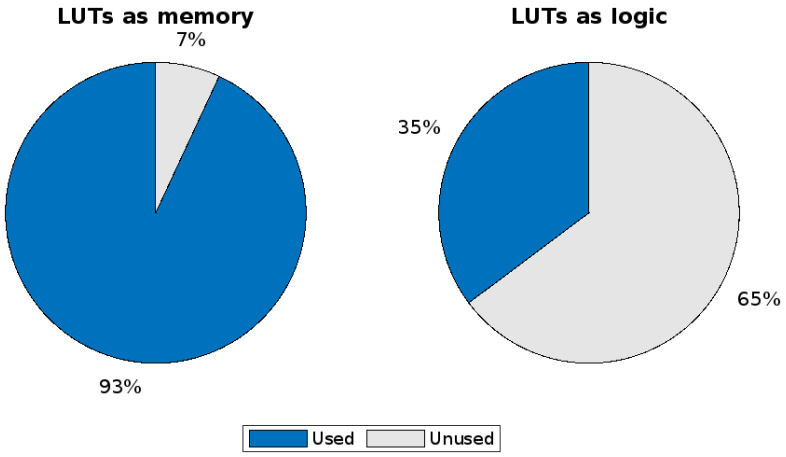


Figure 4.2: Percentage of LUTs used for memory and logic for the 6-by-6 mesh.

Graph	Latency
email-Eu-core	54 µs
cit-HepPh	636 µs
soc-Slashdot0902	1.6 ms
soc-sign-epinions	2 ms
web-NotreDame	533 µs

Table 4.3: Minimum layer latencies for different input graphs.

Chapter 5

Conclusions

The goal of the thesis was to develop an aggregation engine for GNN algorithms that would address the peculiarities of this dataflow. Though irregular memory accesses are still present, they have been reduced by optimally partitioning the input graph at compile time, which has been made possible in an straight-forward way by approaching the aggregation phase as an operation on a graph structure instead of a sparse-dense matrix multiplication. The problem of workload imbalance between cores has also been solved by partitioning the graph with weighted min-cut as objective, but imbalances due to individual highly connected nodes has not been addressed, since the aggregation of each node takes place entirely on a single PE.

Advantages and drawbacks Preprocessing the graph to optimally partition it among the PEs has proven to be not only very efficient but decisive in obtaining low latencies, both because it keeps the network activity to a minimum and it rebalances the workload among the mesh.
The main hindrance to the latency of the architecture has proven to be the high transmission time of the packets, due to the way the network gets easily congested, especially for high feature sizes. Additionally, this problem gets worse with bigger

mesh sizes.

Possible future development The current state of the accelerator opens up the possibility for different future developments. On a system level, it could either be integrated with a combination engine or expanded in order to support combination itself. Due to the fact that the architecture has been tailored around the peculiarities of the aggregation phase, using a different engine for the aggregation would probably prove to be the most reasonable choice, similarly to what has been implemented by HyGCN [3].
The most pressing bottleneck of the architecture that needs to be addressed is the network congestion, which limits scalability and degrades performance for high feature sizes. In the future, the optimal bandwidth for the NoC mesh depending on the feature size could be studied. Alternatively, changing the NoC topology from a square mesh to one where the communication between far away routers is less penalized (such as a binary tree) would be effective in reducing the communication latency, allowing for larger efficient networks which would reduce execution time with a higher level of parallelization.
Ultimately, there is the opportunity for further workload balancing. One possible way to achieve this would be to identify the most connected nodes and divide their workload between different PEs at compile time, obtaining a result similar to the "evil-row remapping" seen in AWB-GCN [2] but without the need of a real-time monitor of each PE's activity. This would of course require some sort of synchronization scheme between the PEs that share the same nodes.

In conclusion, this thesis takes a step forward in understanding the GNN bottlenecks and implements promising solutions to address them.

Appendix A

RTL

content/Code/RTL/NoC.sv

```
1  import my_pkg::*;
2
3  module NoC
4      (
5      input logic arst_n, clk,
6      input logic dummy_sel, //synth only
7      input logic [VECTOR_LENGTH-1:0] dummy_in, //for synth. purp.
8      output [PACKET_LENGTH-1 : 0] dummy_out, //for synthesis purposes
9      output [PACKET_LENGTH-1:0] dummy_mem_out,
10     output logic done_out
11     );
12
13     logic signed [PACKET_LENGTH-1:0] router_in [1 : MESH_SIZE][1 : MESH_SIZE];
14     logic signed [PACKET_LENGTH-1:0] router_out [1 : MESH_SIZE][1 : MESH_SIZE];
15     //logic [OPCODE_LENGTH + ADDR_LENGTH - 1 :0] instruction [1 : MESH_SIZE][1 :
       MESH_SIZE];
16     logic router_valid [1 : MESH_SIZE][1 : MESH_SIZE];
17     logic reading [1 : MESH_SIZE][1 : MESH_SIZE];
18     logic read [0 : (MESH_SIZE + 1)][0 : (MESH_SIZE + 1)][0:4];
19     logic signed [PACKET_LENGTH-1:0] mac_in [1 : MESH_SIZE][1 : MESH_SIZE];
20     logic signed [PACKET_LENGTH-1:0] downstream[0 : (MESH_SIZE + 1)][0 : (
       MESH_SIZE + 1)][0:4];
21     logic local_reading[1 : MESH_SIZE][1 : MESH_SIZE];
```

```
22
23      logic packet_valid [0 : (MESH_SIZE + 1)][0 : (MESH_SIZE + 1)][0:4];
24      logic valid_out [1 : MESH_SIZE][1 : MESH_SIZE][0:4];
25      logic [1 : MESH_SIZE][1 : MESH_SIZE] done;
26      logic layer_finished;
27
28      assign layer_finished = &done; //if all the done signals are asserted
29      assign done_out = layer_finished;
30
31      assign dummy_out = router_in[1][1];
32
33      genvar x,y;
34      generate
35          for(y=1; y <= MESH_SIZE; y++) begin
36              for(x=1; x <= MESH_SIZE; x++) begin
37
38              localparam  bit [4:0] ACTIVE_PORTS  = get_active_ports(y-1, x-1);
39
40              router #(
41                        .X_COORD(x-1),
42                        .Y_COORD(y-1)
43              )
44              router(
45                        .clk(clk),
46                        .arst_n(arst_n),
47                        .packet_valid(packet_valid[x][y]), //input
48                        .n_reading(read[x][y+1][SOUTH]),
49                        .e_reading(read[x+1][y][WEST]),
50                        .w_reading(read[x-1][y][EAST]),
51                        .s_reading(read[x][y-1][NORTH]),
52                        .local_reading(reading[x][y]),
53                        .is_read(read[x][y]), //input
54                        .n_valid_out(packet_valid[x][y+1][SOUTH]),
55                        .e_valid_out(packet_valid[x+1][y][WEST]),
56                        .s_valid_out(packet_valid[x][y-1][NORTH]),
57                        .w_valid_out(packet_valid[x-1][y][EAST]),
58                        .local_valid_out(router_valid[x][y]),
59                        .downstream(downstream[x][y]), //input
60
61                        .n_up(downstream[x][y+1][SOUTH]),
62                        .e_up(downstream[x+1][y][WEST]),
```

```
63                  .s_up(downstream[x][y-1][NORTH]),
64                  .w_up(downstream[x-1][y][EAST]),
65                  .local_up(router_out[x][y])
66              );
67
68              assign downstream[x][y][LOCAL] = router_in[x][y];
69
70              if (!ACTIVE_PORTS[EAST]) begin
71                  dummy_node
72                  dummy(
73                      .w_up(downstream[x][y][EAST]),
74                      .w_valid_out(packet_valid[x][y][EAST]),
75                      .is_reading_w(read[x][y][EAST])
76                  );
77              end
78
79              if (!ACTIVE_PORTS[NORTH]) begin
80                  dummy_node
81                  dummy(
82                      .s_up(downstream[x][y][NORTH]),
83                      .s_valid_out(packet_valid[x][y][NORTH]),
84                      .is_reading_s(read[x][y][NORTH])
85                  );
86              end
87
88              if (!ACTIVE_PORTS[WEST]) begin
89                  dummy_node
90                  dummy(
91                      .e_up(downstream[x][y][WEST]),
92                      .e_valid_out(packet_valid[x][y][WEST]),
93                      .is_reading_e(read[x][y][WEST])
94                  );
95              end
96
97              if (!ACTIVE_PORTS[SOUTH]) begin
98                  dummy_node
99                  dummy(
100                     .n_up(downstream[x][y][SOUTH]),
101                     .n_valid_out(packet_valid[x][y][SOUTH]),
102                     .is_reading_n(read[x][y][SOUTH])
103                 );
```

```
104             end
105
106
107
108             PE #(   .X_COORD(x-1),
109                     .Y_COORD(y-1)
110                     )
111             PE(
112                      .clk(clk),
113                      .arst_n(arst_n),
114                      .read(reading[x][y]),
115                      .dummy_sel(dummy_sel),
116                      .dummy_in(dummy_in),
117                      .dummy_out(dummy_mem_out),
118                      .reading(read[x][y][LOCAL]),
119                      .vld_in(router_valid[x][y]),
120                      .vld_out(packet_valid[x][y][LOCAL]),
121                      .din(router_out[x][y]),
122                      .dout(router_in[x][y]),
123                      .done(done[x][y])
124                      );
125             end
126         end
127
128     endgenerate
129
130     function bit [4:0] get_active_ports(int y, int x);
131         bit [4:0] active_ports;
132         active_ports[LOCAL] = 1;
133         active_ports[EAST]  = (x < MESH_SIZE-1) ? 1 : 0;
134         active_ports[NORTH] = (y < MESH_SIZE-1) ? 1 : 0;
135         active_ports[WEST]  = (x > 0           ) ? 1 : 0;
136         active_ports[SOUTH] = (y > 0           ) ? 1 : 0;
137         return active_ports;
138   endfunction
139
140 endmodule
```

content/Code/RTL/RR_arbiter.sv

```
1 import my_pkg::*;
```

```
2
3  module RR_arbiter
4      (
5          input logic clk, arst_n,
6          //input update_priority,
7          input logic [4:0] requests,
8          input logic req_satisfied,
9          output logic [4:0] grant,
10         output logic conflict //only to gather metrics via TB
11     );
12
13     logic[2:0] pointer_next = 0;
14     logic[4:0] req_shifted;
15     logic[4:0] grant_shifted;
16     logic[9:0] req_shifted_double, gr_shifted_double;
17     logic conflict_q;
18     assign conflict = conflict_q;
19
20     always_comb begin
21         case (requests)
22             5'b00000: conflict_q = 1'b0;
23             5'b00001: conflict_q = 1'b0;
24             5'b00010: conflict_q = 1'b0;
25             5'b00100: conflict_q = 1'b0;
26             5'b01000: conflict_q = 1'b0;
27             5'b10000: conflict_q = 1'b0;
28             default: conflict_q = 1'b1;
29         endcase
30     end
31
32     always @(posedge clk or negedge arst_n) begin
33         if(arst_n == 0 || pointer_next >= 5)
34             pointer_next = 0;
35         else if(grant != {1'b0, 1'b0, 1'b0, 1'b0, 1'b0} /*&& req_satisfied == 1*/)
       begin
36             if(grant == 5'b00001) pointer_next = 3'b001;
37             if(grant == 5'b00010) pointer_next = 3'b010;
38             if(grant == 5'b00100) pointer_next = 3'b011;
39             if(grant == 5'b01000) pointer_next = 3'b100;
40             if(grant == 5'b10000) pointer_next = 3'b000;
41             end
```

```
42
43      end
44
45      //rotating the requesters:
46      assign req_shifted_double = {requests, requests} >> pointer_next;
47      assign req_shifted = req_shifted_double[4:0];
48
49      //assigning the grant:
50      assign grant_shifted[0] = req_shifted[0];
51      assign grant_shifted[1] = ~req_shifted[0] & req_shifted [1];
52      assign grant_shifted[2] = ~req_shifted[0] & ~req_shifted[1] & req_shifted[2];
53      assign grant_shifted[3] = ~req_shifted[0] & ~req_shifted[1] & ~req_shifted[2]
        & req_shifted[3];
54      assign grant_shifted[4] = ~req_shifted[0] & ~req_shifted[1] & ~req_shifted[2]
        & ~req_shifted[3] & req_shifted[4];
55
56      assign gr_shifted_double = {grant_shifted, grant_shifted} << pointer_next;
57      assign grant = gr_shifted_double[9:5];
58
59  endmodule
```

content/Code/RTL/router.sv

```
1   import my_pkg::*;
2
3   module router #(
4           parameter int X_COORD = 0,
5           parameter int Y_COORD = 0
6       )
7       (
8       input logic clk, arst_n,
9       input logic is_read [0:4],
10      input logic packet_valid [0:4],
11
12      output signed [PACKET_LENGTH - 1:0] n_up,
13      output signed [PACKET_LENGTH - 1:0] s_up,
14      output signed [PACKET_LENGTH - 1:0] w_up,
15      output signed [PACKET_LENGTH - 1:0] e_up,
16      output signed [PACKET_LENGTH - 1:0] local_up,
17
18      input signed [PACKET_LENGTH - 1:0] downstream [0:4],
```

```
19
20      output logic n_valid_out, w_valid_out, s_valid_out, e_valid_out,
        local_valid_out,
21      output logic n_reading, w_reading, s_reading, e_reading, local_reading
22      );
23
24      logic is_reading[0:4];
25      assign local_reading = is_reading[LOCAL];
26      assign e_reading = is_reading[EAST];
27      assign n_reading = is_reading[NORTH];
28      assign w_reading = is_reading[WEST];
29      assign s_reading = is_reading[SOUTH];
30
31      logic signed [PACKET_LENGTH - 1:0] n_out;
32      logic signed [PACKET_LENGTH - 1:0] s_out;
33      logic signed [PACKET_LENGTH - 1:0] w_out;
34      logic signed [PACKET_LENGTH - 1:0] e_out;
35      logic signed [PACKET_LENGTH - 1:0] local_out;
36
37      logic valid_out [0:4];
38      logic valid_input_block [0:4];
39      logic is_read_reg [0:4];
40      logic is_read_input_block [0:4];
41
42      logic [4:0] req_source [0:4]; //requests of each input port
43      logic [4:0] req_dest [0:4]; //requests to the arbiter of each output port
44      logic [4:0] grant [0:4];
45      logic arb_conflict [0:4];
46      logic router_conflict;
47      logic signed [PACKET_LENGTH - 1 : 0] to_switch [0:4];
48
49      assign router_conflict = arb_conflict[0] || arb_conflict[1] || arb_conflict[2]
         || arb_conflict[3] || arb_conflict[4];
50
51      router_input #(.X_COORD(X_COORD), .Y_COORD(Y_COORD))
52      local_input(
53          .clk(clk),
54          .arst_n(arst_n),
55          .to_switch(to_switch[LOCAL]),
56          .packet_valid(packet_valid[LOCAL]),
57          .is_reading(is_reading[LOCAL]),
```

```
58          .is_read(is_read_input_block[LOCAL]),
59          .valid_out(valid_input_block[LOCAL]),
60          .down(downstream[LOCAL]),
61          .port_request(req_source[LOCAL])
62      );
63
64      router_input #(.X_COORD(X_COORD), .Y_COORD(Y_COORD))
65      north_input(
66          .clk(clk),
67          .arst_n(arst_n),
68          .packet_valid(packet_valid[NORTH]),
69          .is_reading(is_reading[NORTH]),
70          .is_read(is_read_input_block[NORTH]),
71          .to_switch(to_switch[NORTH]),
72          .valid_out(valid_input_block[NORTH]),
73          .down(downstream[NORTH]),
74          .port_request(req_source[NORTH])
75      );
76
77      router_input #(.X_COORD(X_COORD), .Y_COORD(Y_COORD))
78      west_input(
79          .clk(clk),
80          .arst_n(arst_n),
81          .to_switch(to_switch[WEST]),
82          .packet_valid(packet_valid[WEST]),
83          .is_reading(is_reading[WEST]),
84          .is_read(is_read_input_block[WEST]),
85          .valid_out(valid_input_block[WEST]),
86          .down(downstream[WEST]),
87          .port_request(req_source[WEST])
88      );
89
90      router_input #(.X_COORD(X_COORD), .Y_COORD(Y_COORD))
91      south_input(
92          .clk(clk),
93          .arst_n(arst_n),
94          .to_switch(to_switch[SOUTH]),
95          .packet_valid(packet_valid[SOUTH]),
96          .is_reading(is_reading[SOUTH]),
97          .is_read(is_read_input_block[SOUTH]),
98          .valid_out(valid_input_block[SOUTH]),
```

```
99          .down(downstream[SOUTH]),
100         .port_request(req_source[SOUTH])
101     );
102
103     router_input #( .X_COORD(X_COORD), .Y_COORD(Y_COORD))
104     east_input(
105         .clk(clk),
106         .arst_n(arst_n),
107         .to_switch(to_switch[EAST]),
108         .packet_valid(packet_valid[EAST]),
109         .is_reading(is_reading[EAST]),
110         .is_read(is_read_input_block[EAST]),
111         .valid_out(valid_input_block[EAST]),
112         .down(downstream[EAST]),
113         .port_request(req_source[EAST])
114     );
115
116     always_comb begin
117         for (integer i = 0; i < 5; i++) begin
118             req_dest[i] = {req_source[SOUTH][i], req_source[WEST][i], req_source[
    NORTH][i], req_source[EAST][i], req_source[LOCAL][i]};
119         end
120     end
121
122     RR_arbiter local_arb(
123         .clk(clk),
124         .arst_n(arst_n),
125         .requests(req_dest[LOCAL]),
126         .req_satisfied(is_read[LOCAL]),
127         .conflict(arb_conflict[LOCAL]),
128         .grant(grant[LOCAL]));
129
130     RR_arbiter east_arb(
131         .clk(clk),
132         .arst_n(arst_n),
133         .requests(req_dest[EAST]),
134         .req_satisfied(is_read[EAST]),
135         .conflict(arb_conflict[EAST]),
136         .grant(grant[EAST]));
137
138     RR_arbiter north_arb(
```

```
139         .clk(clk),
140         .arst_n(arst_n),
141         .requests(req_dest[NORTH]),
142         .req_satisfied(is_read[NORTH]),
143         .conflict(arb_conflict[NORTH]),
144         .grant(grant[NORTH]));
145
146     RR_arbiter south_arb(
147         .clk(clk),
148         .arst_n(arst_n),
149         .requests(req_dest[SOUTH]),
150         .req_satisfied(is_read[SOUTH]),
151         .conflict(arb_conflict[SOUTH]),
152         .grant(grant[SOUTH]));
153
154     RR_arbiter west_arb(
155         .clk(clk),
156         .arst_n(arst_n),
157         .requests(req_dest[WEST]),
158         .req_satisfied(is_read[WEST]),
159         .conflict(arb_conflict[WEST]),
160         .grant(grant[WEST]));
161
162     assign n_valid_out = valid_out[NORTH];
163     assign s_valid_out = valid_out[SOUTH];
164     assign w_valid_out = valid_out[WEST];
165     assign e_valid_out = valid_out[EAST];
166     assign local_valid_out = valid_out[LOCAL];
167
168
169
170     always_comb begin
171         is_read_input_block = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0};
172         local_out = 0;
173         n_out = 0;
174         e_out = 0;
175         w_out = 0;
176         s_out = 0;
177         valid_out = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0};
178
179         //local output:
```

```
180         case (grant[LOCAL])
181         NORTH_REQ: begin
182             local_out = to_switch[NORTH];
183             valid_out[LOCAL] = valid_input_block[NORTH];
184             is_read_input_block[NORTH] = is_read[LOCAL];
185         end
186         EAST_REQ: begin
187             local_out = to_switch[EAST];
188             valid_out[LOCAL] = valid_input_block[EAST];
189             is_read_input_block[EAST] = is_read[LOCAL];
190         end
191         SOUTH_REQ: begin
192             local_out = to_switch[SOUTH];
193             valid_out[LOCAL] = valid_input_block[SOUTH];
194             is_read_input_block[SOUTH] = is_read[LOCAL];
195         end
196         WEST_REQ: begin
197             local_out = to_switch[WEST];
198             valid_out[LOCAL] = valid_input_block[WEST];
199             is_read_input_block[WEST] = is_read[LOCAL];
200         end
201         NO_REQ: begin
202             valid_out[LOCAL] = 0;
203         end
204         endcase
205
206         //north:
207         case (grant[NORTH])
208         LOCAL_REQ: begin
209             n_out = to_switch[LOCAL];
210             valid_out[NORTH] = valid_input_block[LOCAL];
211             is_read_input_block[LOCAL] = is_read[NORTH];
212         end
213         EAST_REQ: begin
214             n_out = to_switch[EAST];
215             valid_out[NORTH] = valid_input_block[EAST];
216             is_read_input_block[EAST] = is_read[NORTH];
217         end
218         SOUTH_REQ: begin
219             n_out = to_switch[SOUTH];
220             valid_out[NORTH] = valid_input_block[SOUTH];
```

```
221             is_read_input_block[SOUTH] = is_read[NORTH];
222         end
223         WEST_REQ: begin
224             n_out = to_switch[WEST];
225             valid_out[NORTH] = valid_input_block[WEST];
226             is_read_input_block[WEST] = is_read[NORTH];
227         end
228         NO_REQ: begin
229             valid_out[NORTH] = 0;
230         end
231         endcase
232 
233         //east
234         case (grant[EAST])
235         LOCAL_REQ: begin
236             e_out = to_switch[LOCAL];
237             valid_out[EAST] = valid_input_block[LOCAL];
238             is_read_input_block[LOCAL] = is_read[EAST];
239             end
240         NORTH_REQ: begin
241             e_out = to_switch[NORTH];
242             valid_out[EAST] = valid_input_block[NORTH];
243             is_read_input_block[NORTH] = is_read[EAST];
244             end
245         SOUTH_REQ: begin
246             e_out = to_switch[SOUTH];
247             valid_out[EAST] = valid_input_block[SOUTH];
248             is_read_input_block[SOUTH] = is_read[EAST];
249             end
250         WEST_REQ:  begin
251             e_out = to_switch[WEST];
252             valid_out[EAST] = valid_input_block[WEST];
253             is_read_input_block[WEST] = is_read[EAST];
254             end
255         NO_REQ: begin
256             valid_out[EAST] = 0;
257             end
258         endcase
259 
260         //south
261         case (grant[SOUTH])
```

```
262         LOCAL_REQ: begin
263             s_out = to_switch[LOCAL];
264             valid_out[SOUTH] = valid_input_block[LOCAL];
265             is_read_input_block[LOCAL] = is_read[SOUTH];
266             end
267         NORTH_REQ: begin
268             s_out = to_switch[NORTH];
269             valid_out[SOUTH] = valid_input_block[NORTH];
270             is_read_input_block[NORTH] = is_read[SOUTH];
271             end
272         EAST_REQ: begin
273             s_out = to_switch[EAST];
274             valid_out[SOUTH] = valid_input_block[EAST];
275             is_read_input_block[EAST] = is_read[SOUTH];
276             end
277         WEST_REQ: begin
278             s_out = to_switch[WEST];
279             valid_out[SOUTH] = valid_input_block[WEST];
280             is_read_input_block[WEST] = is_read[SOUTH];
281             end
282         NO_REQ: begin
283             valid_out[SOUTH] = 0;
284         end
285         endcase
286 
287         //west
288         case (grant[WEST])
289         LOCAL_REQ: begin
290             w_out = to_switch[LOCAL];
291             valid_out[WEST] = valid_input_block[LOCAL];
292             is_read_input_block[LOCAL] = is_read[WEST];
293             end
294         NORTH_REQ: begin
295             w_out = to_switch[NORTH];
296             valid_out[WEST] = valid_input_block[NORTH];
297             is_read_input_block[NORTH] = is_read[WEST];
298             end
299         SOUTH_REQ: begin
300             w_out = to_switch[SOUTH];
301             valid_out[WEST] = valid_input_block[SOUTH];
302             is_read_input_block[SOUTH] = is_read[WEST];
```

```
303             end
304         EAST_REQ: begin
305             w_out = to_switch[EAST];
306             valid_out[WEST] = valid_input_block[EAST];
307             is_read_input_block[EAST] = is_read[WEST];
308             end
309         NO_REQ: begin
310             valid_out[WEST] = 0;
311         end
312         endcase
313     end
314 
315     assign local_up = local_out;
316     assign n_up = n_out;
317     assign e_up = e_out;
318     assign w_up = w_out;
319     assign s_up = s_out;
320    // assign is_read_reg = is_read;
321 
322 endmodule
```

Appendix B

Simulation

content/Code/SIM/NoC_tb.sv

```
import my_pkg::*;

module NoC_tb #(
    parameter string INS_PATH = "/users/students/r0875167/Dataset/soc-Slashdot/10_10/imem_content/",
    parameter string DATA_PATH = "/users/students/r0875167/Dataset/soc-Slashdot/10_10/dmem_content/",
    parameter string REF_FILE = "/users/students/r0875167/Dataset/soc-Slashdot/10_10/reference_values.txt",
    parameter string REPORT_FILE = "/users/students/r0875167/GNN-on-server/GNN-accelerator/generated_reports/SIM/wslashdot-w/10_10_unpart.txt"
)
    ();

    logic arst_n, clk;

    NoC
    DUT(
        .arst_n(arst_n),
        .clk(clk)
    );

    //logic [VECTOR_LENGTH-1:0] reference [0 : NODE_NO-1];
```

```
20      //logic [VECTOR_LENGTH-1:0] mem_content [0 : PE_NUMBER*MEM_HEIGHT-1];
21      //logic [VECTOR_LENGTH-1:0] results [0 : NODE_NO-1];
22      logic [0:PE_NUMBER-1] pe_stall;
23      logic [0:PE_NUMBER-1] conflicts;
24      logic done_signal [1:MESH_SIZE][1:MESH_SIZE];
25
26      int index;
27      //initial $readmemb(REF_FILE, reference);
28
29      //assign results vector:
30      genvar x,y,i;
31      /*
32      generate
33          for(y=1; y<=MESH_SIZE; y++) begin
34              for(x=1; x<=MESH_SIZE; x++) begin
35                  for(i=0; i<MEM_HEIGHT; i++) begin
36                      localparam index = i + MEM_HEIGHT*(x-1) + MESH_SIZE*MEM_HEIGHT
    *(y-1);
37                      assign mem_content[index] = DUT.genblk1[y].genblk1[x].PE.
    memory_2.data[i];
38                      //initial $display("index=%d, x=%d, y=%d", index, x, y);
39                  end
40              end
41          end
42      endgenerate
43      */
44
45      generate
46          for(y=1; y<=MESH_SIZE; y++) begin
47              for(x=1; x<=MESH_SIZE; x++) begin
48                  assign done_signal[x][y] = DUT.done[x][y];
49                  localparam index = x-1 + MESH_SIZE*(y-1);
50                  assign pe_stall[index] = DUT.genblk1[y].genblk1[x].PE.pc_stall;
51                  assign conflicts[index] = DUT.genblk1[y].genblk1[x].router.
    router_conflict;
52              end
53          end
54      endgenerate
55
56      //populate data memories:
57      generate
```

```
58          for(y=1; y<=MESH_SIZE; y++) begin
59              for(x=1; x<=MESH_SIZE; x++) begin
60                  string DATA_FILE, INS_FILE;
61
62                  initial begin
63                      automatic string x_str = "";
64                      automatic string y_str = "";
65                      x_str.itoa(x-1);
66                      y_str.itoa(y-1);
67                      DATA_FILE = {DATA_PATH, "PE_", x_str, "_", y_str, ".txt"};
68                      INS_FILE = {INS_PATH, "PE_", x_str, "_", y_str, ".txt"};
69                  end
70
71                  initial begin
72                      $readmemb(DATA_FILE, DUT.genblk1[y].genblk1[x].PE.memory_1.
    data);
73                      $readmemb(INS_FILE, DUT.genblk1[y].genblk1[x].PE.
    instruction_mem.data);
74                  end
75              end
76          end
77      endgenerate
78
79      int cycles_no = 0;
80      real stall_perc [0:PE_NUMBER-1] = '{PE_NUMBER{0}};
81      int stall_cc [0:PE_NUMBER-1] = '{PE_NUMBER{0}};
82      int conflicts_cc [0:PE_NUMBER-1] = '{PE_NUMBER{0}};
83      int errors = 0;
84
85      initial begin
86          clk = 1;
87          wait_and_check();
88          generate_reports();
89          $finish;
90      end
91
92      initial begin
93          arst_n = 0;
94          #3
95          arst_n = 1;
96      end
```

```
97
98
99      int exec_finished[1:MESH_SIZE][1:MESH_SIZE];
100     always begin
101         #5
102         clk = !clk;
103         cycles_no++;
104         foreach(pe_stall[i]) begin
105             if(pe_stall[i]) begin
106                 stall_cc[i] = stall_cc[i] + 1;
107             end
108         end
109         foreach(conflicts[i]) begin
110             if(conflicts[i]) begin
111                 conflicts_cc[i] = conflicts_cc[i] + 1;
112             end
113         end
114         for(int y=1; y<=MESH_SIZE; y++) begin
115             for(int x=1; x<=MESH_SIZE; x++) begin
116                 if(done_signal[x][y]==1'b1 && exec_finished[x][y]==0) begin
117                     exec_finished[x][y] = cycles_no;
118                 end
119             end
120         end
121     end
122
123
124
125     task wait_and_check();
126         automatic int index = 0;
127         automatic int j = 0;
128         begin
129             wait(DUT.layer_finished);
130
131             for(int y=1; y<=MESH_SIZE; y++) begin
132                 for(int x=1; x<=MESH_SIZE; x++) begin
133                     if(exec_finished[x][y]==0) begin
134                         exec_finished[x][y] = cycles_no;
135                     end
136                 end
137             end
```

```
138
139             foreach(stall_perc[i]) begin
140                 stall_perc[i] = real'(stall_cc[i])/real'(cycles_no);
141             end
142         /*
143             foreach(mem_content[index]) begin
144                 if(mem_content[index] !== 'x) begin
145                    results[j] = mem_content[index];
146                    j++;
147                    //$display("index %d",index);
148                    //$display("j %d", j);
149                 end
150             end
151
152
153             for(int i=0; i<NODE_NO; i++) begin
154                 if(reference[i] !== results[i] || results[i] === 'x) begin
155                     //$display("Error at PE[%d][%d], address %d", x, y, i);
156                     errors = errors + 1;
157                 end
158             end
159         */
160             $display("Layer finished in %d cycles", cycles_no);
161             $display("%d errors found", errors);
162         end
163     endtask
164
165     task generate_reports();
166         int fd;
167         begin
168             fd = $fopen(REPORT_FILE, "w");
169             $fdisplay(fd, "PARAMETERS:\nNODE_NO = %6d\nFEATURE_SIZE = %3d\nMESH =
    %3d X%3d\nNOT PARTITIONED", NODE_NO, FEATURES, MESH_SIZE, MESH_SIZE);
170             $fdisplay(fd, "Error #: %d", errors);
171             $fdisplay(fd, "Execution CC #: %d", cycles_no);
172             $fdisplay(fd, "Stall perc. for each PE:");
173             for(int y = 0; y < MESH_SIZE; y++) begin
174                 for(int x = 0; x < MESH_SIZE; x++) begin
175                     $fdisplay(fd, "[%2d][%2d]: %.3f", x, y, stall_perc[x+MESH_SIZE
    *y]);
176                 end
```

```
177             end
178             $fdisplay(fd, "\nConflicts for each router:");
179             for(int y = 0; y < MESH_SIZE; y++) begin
180                 for(int x = 0; x < MESH_SIZE; x++) begin
181                     $fdisplay(fd, "[%2d][%2d]: %d", x, y, conflicts_cc[x+MESH_SIZE
    *y]);
182                 end
183             end
184             $fdisplay(fd, "\nExecution cycles:");
185             for(int y = 1; y <= MESH_SIZE; y++) begin
186                 for(int x = 1; x <= MESH_SIZE; x++) begin
187                     $fdisplay(fd, "[%2d][%2d]: %d", x, y, exec_finished[x][y]);
188                 end
189             end
190             $fclose(fd);
191         end
192     endtask
193
194 endmodule
```

Appendix C

Scripts

content/Code/SCRIPT/partition_graph.py

```
import networkx as nx
import sys
import nxmetis as mts

def partition(edge_f, partition_no):
    #graph creation
    G = nx.read_edgelist(edge_f)
    opt = mts.MetisOptions(contig=0, ccorder=1, compress=1)
    (edge_cuts, subgraphs) = mts.partition(G, partition_no, options=opt)
    return(edge_cuts, subgraphs)

def find_2d(target, lst_2d):
    pos = 0
    for i, lst in enumerate(lst_2d):
        for j, element in enumerate(lst):
            if int(element) == int(target):
                return(i, j, pos)
            pos = pos+1
    return(None, None, None)

def create_dict(lst):
    pos = 0
    ret_dic = {}
```

```
24      for i, subg in enumerate(lst):
25          for j, element in enumerate(subg):
26              ret_dic[element] = (i, j, pos)
27              pos = pos+1
28      return ret_dic
```

content/Code/SCRIPT/generate_mem_content.py

```
1  import sys
2  import snap
3  import random
4  import math
5  import os
6  import time
7  import partition_graph as part
8
9  if(len(sys.argv) < 5):
10     print("Need arguments: file name, data width, feature size, PE number")
11     exit()
12
13 filename = sys.argv[1]
14 data_width = int(sys.argv[2])
15 feat_size = int(sys.argv[3])
16 mesh_size = math.sqrt(int(sys.argv[4]))
17 pe_no = int(sys.argv[4])
18 run_name = str(int(mesh_size)) + "_" + str(int(mesh_size))
19
20 if(pe_no == 1):
21     coord_bits = 1
22 else:
23     coord_bits = math.ceil(math.log(mesh_size,2))
24
25 if(len(sys.argv) > 5 and sys.argv[5] == "-v"):
26     verbose = 1
27 else:
28     verbose = 0
29
30 if(verbose):
31     print(str(mem_size) + " -> memory size\n")
32     print(str(mesh_size) + " -> mesh size\n")
33     print(str(coord_bits) + " -> coord bits\n")
```

```
34
35
36
37 print("Acquiring graph...")
38 graph = snap.LoadEdgeList(snap.TUNGraph, filename, 0, 1)
39 print("Graph acquired.")
40 print("Partitioning graph...")
41 (cut_no, subgraphs) = part.partition(filename, pe_no)
42 print("Graph partitioning done")
43
44 mem_size = math.ceil(graph.GetNodes()/pe_no)
45
46 max_size = 0
47 for i in range(len(subgraphs)):
48     if(max_size < len(subgraphs[i])):
49         max_size = len(subgraphs[i])
50
51 print("Max subgraph size is " + str(max_size))
52 addr_bits = math.ceil(math.log(max_size,2))
53
54 graph_dict = part.create_dict(subgraphs)
55 #print(graph_dict)
56
57 global_neigh = []
58 node_Ids = []
59
60 feat_file = open("features.txt", "w")
61 addr_file = open("addresses.txt", "w")
62
63 index = 0
64 x_coord = 0
65 y_coord = 0
66
67 #os.system("rm ./dmem_content/*.txt")
68 #file = open("./dmem_content/PE[0][0].txt")
69
70
71 id_list = []
72 addr_list = []
73 feature_list = []
74
```

```
75 feat_file.write("*0,0\n")
76 for NI in graph.Nodes():
77     #generate random feature vector:
78     feature_vector = "";
79     for i in range(feat_size*data_width):
80         temp = str(random.randint(0,1))
81         feature_vector += temp
82     feat_file.write(str(hex(int(feature_vector,2)))[2:] + "\n")
83     feature_list.append(feature_vector)
84 
85     #generate address for the frature vector:
86     if(index == mem_size):
87         #print("x = " + str(x_coord))
88         #print("y = " + str(y_coord))
89         index = 0
90         if(x_coord < mesh_size-1):
91             x_coord = x_coord + 1
92             feat_file.write("*" + str(x_coord) + "," + str(y_coord) + "\n")
93         elif(y_coord < mesh_size-1):
94             x_coord = 0
95             y_coord = y_coord + 1
96             feat_file.write("*" + str(x_coord) + "," + str(y_coord) + "\n")
97         else:
98             print("Not enough memory")
99             addr_file.close()
100            feat_file.close()
101            exit()
102
103    x_string = str(format(x_coord, "b").zfill(coord_bits))
104    y_string = str(format(y_coord, "b").zfill(coord_bits))
105    address = x_string + y_string + str(format(index,"b").zfill(addr_bits))
106    addr_file.write(address + "\n")
107
108    id_list.append(NI.GetId())
109    addr_list.append(address)
110
111    index = index + 1
112
113
114
115 os.system("rm " + run_name + "/imem_content/*.txt")
```

```
116 os.system("rm "  + run_name + "/dmem_content/*.txt")
117 os.system("rm " + run_name + "/imem_partitioned/*.txt")
118 os.system("rm " + run_name + "/dmem_partitioned/*.txt")
119 os.system("rm " + run_name + "/reference_partitioned.txt")
120 os.system("rm " + run_name + "/reference_values.txt")
121 inst_file = open(run_name + "/imem_content/PE_0_0.txt", "w")
122 f_file = open(run_name + "/dmem_content/PE_0_0.txt", "w")
123 ref_file = open(run_name + "/reference_values.txt", "w")
124 part_file = open(run_name + "/dmem_partitioned/PE_0_0.txt", "w")
125 ins_part_file = open(run_name + "/imem_partitioned/PE_0_0.txt", "w")
126 ref_part_file = open(run_name + "/reference_partitioned.txt", "w")
127 x_coord = 0
128 y_coord = 0
129 index = 0
130 aggr_idx = 0
131 sub_order = []
132 reference_list = []
133 node_counter = 0
134 start_time = time.time()
135 unpart_edge_cut = 0
136
137 for NI in graph.Nodes():
138 #for NI in nodes:
139     #generate address for the feature vector and open right instruction file:
140     if(index == mem_size):
141         index = 0
142         if(x_coord < mesh_size-1):
143             x_coord = x_coord + 1
144             #print("x = " + str(x_coord))
145
146             inst_file.close()
147             f_file.close()
148             inst_file = open(run_name + "/imem_content/PE_" + str(x_coord) + "_"↘
     + str(y_coord) + ".txt", "w")
149             f_file = open(run_name + "/dmem_content/PE_" + str(x_coord) + "_" + ↘
    str(y_coord) + ".txt", "w")
150         elif(y_coord < mesh_size-1):
151             x_coord = 0
152             y_coord = y_coord + 1
153             #print("y = " + str(y_coord))
154             #stop instruction
```

```
155             inst_file.write("111")
156             for l in range(addr_bits + 2*coord_bits):
157                 inst_file.write("0")
158             inst_file.write(" //STOP\n")
159             inst_file.close()
160             f_file.close()
161             inst_file = open(run_name + "/imem_content/PE_" + str(x_coord) + "_"↘
     + str(y_coord) + ".txt", "w")
162             f_file = open(run_name + "/dmem_content/PE_" + str(x_coord) + "_" + ↘
    str(y_coord) + ".txt", "w")
163
164
165     x_string = str(format(x_coord, "b").zfill(coord_bits))
166     y_string = str(format(y_coord, "b").zfill(coord_bits))
167     address = x_string + y_string + str(format(index,"b").zfill(addr_bits))
168
169     #output the features to the file
170     feat_index = int(index + mem_size*x_coord + mesh_size*mem_size*y_coord)
171     f_file.write(str(feature_list[feat_index]) + "\n")
172
173     #load node to PE
174     inst_file.write("010" + address + " //load node\n")
175
176     #write node into correct partitioned files:
177     part_file.close()
178     ins_part_file.close()
179     (sub_no, el_no, position) = graph_dict[str(NI.GetId())]
180     part_x = int(sub_no % mesh_size)
181     part_y = math.floor(sub_no/mesh_size)
182
183     sub_order.append(position) #order the nodes are loaded to the PEs (needed to↘
      generate reference values)
184
185     #DEBUG
186     #print(str(NI.GetId()) + " -> pos: " + str(position))
187     #print("pos = " + str(position))
188     #print(sub_order)
189     #print("sub: " + str(sub_no) + "\tn: " + str(el_no))
190     #print("x: " + str(part_x) + "\ty: " + str(part_y))
191
```

```
192     part_file = open(run_name + "/dmem_partitioned/PE_" + str(part_x) + "_" + ↘
        str(part_y) + ".txt", "a")
193     ins_part_file = open(run_name + "/imem_partitioned/PE_" + str(part_x) + "_" ↘
        + str(part_y) + ".txt", "a")
194     part_file.write(str(feature_list[feat_index] + "\n"))
195     x_string = str(format(part_x, "b").zfill(coord_bits))
196     y_string = str(format(part_y, "b").zfill(coord_bits))
197     addr_part_base = x_string + y_string + str(format(el_no,"b").zfill(addr_bits↘
        ))
198     ins_part_file.write("010" + addr_part_base + " //load node\n")
199
200     index = index + 1
201
202     #create a list with each feature as elements
203     base_node = []
204     for i in range(0, feat_size*data_width, data_width):
205         base_node.append(int(feature_list[feat_index][i:i+data_width],2))
206
207     #generate reference values
208     if(verbose):
209         print("Now aggregating: ")
210         print(str(feature_list[aggr_idx]) + " ")
211         print(str(base_node) + "\n")
212
213     #accumulate with 0
214     inst_file.write("100")
215     for l in range(addr_bits + 2*coord_bits):
216         inst_file.write("0")
217     inst_file.write(" //accumulate 0\n")
218
219     ins_part_file.write("100")
220     for l in range(addr_bits + 2*coord_bits):
221         ins_part_file.write("0")
222     ins_part_file.write(" //accumulate 0\n")
223
224
225     aggregated = base_node
226     for Id in NI.GetOutEdges():
227         id_index = id_list.index(Id) #for unpartitioned
228
229         (sub_no, el_no, neigh_pos) = graph_dict[str(Id)]
```

```
230         part_x = int(sub_no % mesh_size)
231         part_y = math.floor(sub_no/mesh_size)
232         x_string_part = str(format(part_x, "b").zfill(coord_bits))
233         y_string_part = str(format(part_y, "b").zfill(coord_bits))
234         addr_part = x_string_part + y_string_part + str(format(el_no,"b").zfill(↘
    addr_bits))
235 
236         ins_part_file.write("010" + addr_part + " //load node (neighbour)\n")
237 
238         neigh_address = addr_list[id_index]
239         inst_file.write("010" + neigh_address + " //load node (neighbour)\n") #↘
    load neighbour to PE
240 
241         #create the neighbour list with each feature as elements
242         neigh_node = []
243         for i in range(0, feat_size*data_width, data_width):
244             neigh_node.append(int(feature_list[id_index][i:i+data_width],2))
245 
246         if(verbose):
247             print(str(feature_list[id_index]) + " ")
248             print(str(neigh_node) + "\n")
249 
250         #accumulate internal
251         inst_file.write("110")
252         for l in range(addr_bits + 2*coord_bits):
253             inst_file.write("0")
254         inst_file.write(" //accumulate internal\n")
255 
256         ins_part_file.write("110")
257         for l in range(addr_bits + 2*coord_bits):
258             ins_part_file.write("0")
259         ins_part_file.write(" //accumulate internal\n")
260 
261         #update reference values
262         #print("aggregating f.v.:" + str(aggregated) + " + " + str(int(↘
    feature_list[id_index],2)) + " = ")
263         for i in range(feat_size):
264             aggregated[i] = aggregated[i] + neigh_node[i]
265 
266         #DEBUG
267         #if(NI.GetId() == 0):
```

```
268         #     st = ""
269         #     print(Id)
270         #     print(neigh_node)
271         #     for i in range(feat_size):
272         #         st = str(bin(aggregated[i]))[2:]
273         #         print(hex(aggregated[i]))
274         #         if len(st) < data_width:
275         #             st = st + st.zfill(data_width)
276         #         else:
277         #             st = st + st[-data_width:]
278             #print(Id)
279             #print(hex(int(st,2)))
280 
281         #print(str(feature_list[id_index]) + "\n")
282 
283     #print aggregation results to reference file
284     node_counter = node_counter + 1
285     if(node_counter%500 == 0):
286         end_time = time.time()
287         elapsed = end_time-start_time
288         start_time = time.time()
289         print("[" + str(int(mesh_size)) + "x" + str(int(mesh_size)) + "]: " + ↘
    str(node_counter) + " nodes processed... (last 500 nodes in " + str(int(↘
    elapsed)) + " s)")
290 
291     ref_string = ""
292     result_string = ""
293     for i in range(feat_size):
294         result_string = str(bin(aggregated[i]))[2:]
295         if len(result_string) < data_width:
296             ref_string = ref_string + result_string.zfill(data_width)
297         else:
298             ref_string = ref_string + result_string[-data_width:]
299 
300     ref_file.write(ref_string + "\n")
301     reference_list.append(ref_string)
302 
303     if(verbose):
304         print("Result: " + ref_string + " ")
305 
306     inst_file.write("001" + address + " //store result\n") #store
```

```
307     ins_part_file.write("001" + addr_part_base + " //store result\n") #store
308
309
310     aggr_idx = aggr_idx + 1
311
312 #for i in sub_order:
313 #    ref_part_file.write(reference_list[i] + "\n")
314
315
316 sorted_zipped = sorted(zip(sub_order, reference_list)) #reorder ref.values
317 for tup in sorted_zipped:
318     ref_part_file.write(tup[1] + "\n") #write to ref_part file only the ref. ↘
    value
319
320
321 inst_file.close()
322 for x in range(int(mesh_size)):
323     for y in range(int(mesh_size)):
324         inst_file = open(run_name + "/imem_content/PE_" + str(x) + "_" + str(y) ↘
    + ".txt", "a")
325         ins_part_file = open(run_name + "/imem_partitioned/PE_" + str(x) + "_" +↘
     str(y) + ".txt", "a")
326        #stop instruction
327         inst_file.write("111")
328         ins_part_file.write("111")
329         for l in range(addr_bits + 2*coord_bits):
330             inst_file.write("0")
331             ins_part_file.write("0")
332         inst_file.write(" //STOP\n")
333         ins_part_file.write(" //STOP\n")
334
335 max_lines = 0
336 for x in range(int(mesh_size)):
337     for y in range(int(mesh_size)):
338         inst_file.close()
339         ins_part_file.close()
340         num_lines_normal = sum(1 for line in open(run_name + "/imem_content/PE_"↘
     + str(x) + "_" + str(y) + ".txt"))
341         num_lines_part = sum(1 for line in open(run_name + "/imem_partitioned/↘
    PE_" + str(x) + "_" + str(y) + ".txt"))
342         if(num_lines_normal > max_lines):
```

```
343             max_lines = num_lines_normal
344         if(num_lines_part > max_lines):
345             max_lines = num_lines_part
346 
347 edges = 0
348 unpart_edge_cut = 0
349 for EI in graph.Edges():
350     N1 = EI.GetSrcNId()
351     N2 = EI.GetDstNId()
352     edges = edges+1
353     if(math.floor(N1/mem_size) != math.floor(N2/mem_size)):
354         unpart_edge_cut = unpart_edge_cut + 1
355 
356 print("DONE: " + str(len(id_list)) + " nodes")
357 dim_file=open(run_name + "/dimensions.txt", "w")
358 dim_file.write("Total edge-cut for the UNPARTITIONED graph is\t" + str(↘
    unpart_edge_cut) + "\n")
359 dim_file.write("Total edge-cut for the PARTITIONED graph is\t" + str(cut_no) + "↘
    \n")
360 dim_file.write(str(len(id_list)) + " nodes\n")
361 dim_file.write(str(edges) + " edges\n")
362 dim_file.write("Instruction memory height must be at least " + str(max_lines) + ↘
    "\n")
363 dim_file.write("DMEM must be " + str(max_size) + "\n")
364 for i in range(len(subgraphs)):
365     dim_file.write("Subgraph " + str(i) + " has size " + str(len(subgraphs[i])) ↘
    + "\n")
366 print("Instruction memory height must be at least " + str(max_lines))
367 
368 #DEBUG
369 #print(subgraphs)
370 #print(sub_order)
371 #print(sorted(zip(sub_order, list(range(17)))))
372 #print(sorted(zip(sub_order, reference_list)))
373 #(n, elno, test_pos) = part.find_2d(944,subgraphs)
374 #print("n = " + str(n) + "    el# = " + str(elno))
375 
376 
377 dim_file.close()
378 ins_part_file.close()
379 ref_part_file.close()
```

```
380 part_file.close()
381 ref_file.close()
382 inst_file.close()
383 addr_file.close()
384 feat_file.close()
```

Bibliography

[1] Thomas N Kipf. «Deep learning with graph-structured representations». In: (2020) (cit. on p. i).

[2] Tong Geng et al. «AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing». In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 922–936. DOI: 10.1109/MICRO50266.2020.00079 (cit. on pp. 2, 4, 5, 44).

[3] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. *HyGCN: A GCN Accelerator with Hybrid Architecture*. 2020. DOI: 10.48550/ARXIV.2001.02514. URL: https://arxiv.org/abs/2001.02514 (cit. on pp. 5, 6, 44).

[4] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. *Dissecting the Graphcore IPU Architecture via Microbenchmarking*. 2019. arXiv: 1912.03413 [cs.DC] (cit. on p. 9).

[5] D.M. Tullsen, S.J. Eggers, and H.M. Levy. «Simultaneous multithreading: Maximizing on-chip parallelism». In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 392–403 (cit. on p. 9).

[6] *Poplar Graph Framework Software*. URL: https://www.graphcore.ai/products/poplar (visited on 06/01/2022) (cit. on p. 9).

[7] *Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit product information*. URL: https://www.xilinx.com/products/boards-and-kits/zcu104.html#information (visited on 05/23/2022) (cit. on pp. 11, 40).

[8] *cit-HepPh dataset*. URL: https://snap.stanford.edu/data/cit-HepPh.html (visited on 06/01/2022) (cit. on pp. 12, 29).

[9] *NetworX-METIS*. URL: https://networkx-metis.readthedocs.io/en/latest/overview.html (visited on 06/01/2022) (cit. on pp. 12, 25).

[10] C.J. Glass and L.M. Ni. «The Turn Model for Adaptive Routing». In: *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. 1992, pp. 278–287. DOI: 10.1109/ISCA.1992.753324 (cit. on p. 19).

[11] *email-Eu-core dataset*. URL: https://snap.stanford.edu/data/email-Eu-core.html (visited on 06/01/2022) (cit. on p. 29).

[12] *soc-Slashdot0902 dataset*. URL: https://snap.stanford.edu/data/soc-Slashdot0902.html (visited on 06/01/2022) (cit. on p. 29).

[13] *soc-sign-epinions dataset*. URL: https://snap.stanford.edu/data/soc-sign-epinions.html (visited on 06/01/2022) (cit. on p. 29).

[14] *web-NotreDame dataset*. URL: https://snap.stanford.edu/data/web-NotreDame.html (visited on 06/01/2022) (cit. on p. 29).

[15] *Vivado ML Editions*. URL: https://www.xilinx.com/products/design-tools/vivado.html (visited on 05/23/2022) (cit. on p. 40).