A Simulator for intelligently scheduling data parallel workloads

BTP presentation

JORU SAIKUMAR-17ME33037

ADVISOR : PROF. SOUMYAJIT DEY

MENTORED BY ANIRBAN GHOSE

Parallel Computing

- Computation divided into subcodes and run across multiple devices parallely.

Two ways of achieving Parallelism:-

- Data level Parallelism-Same function, data divided across devices.

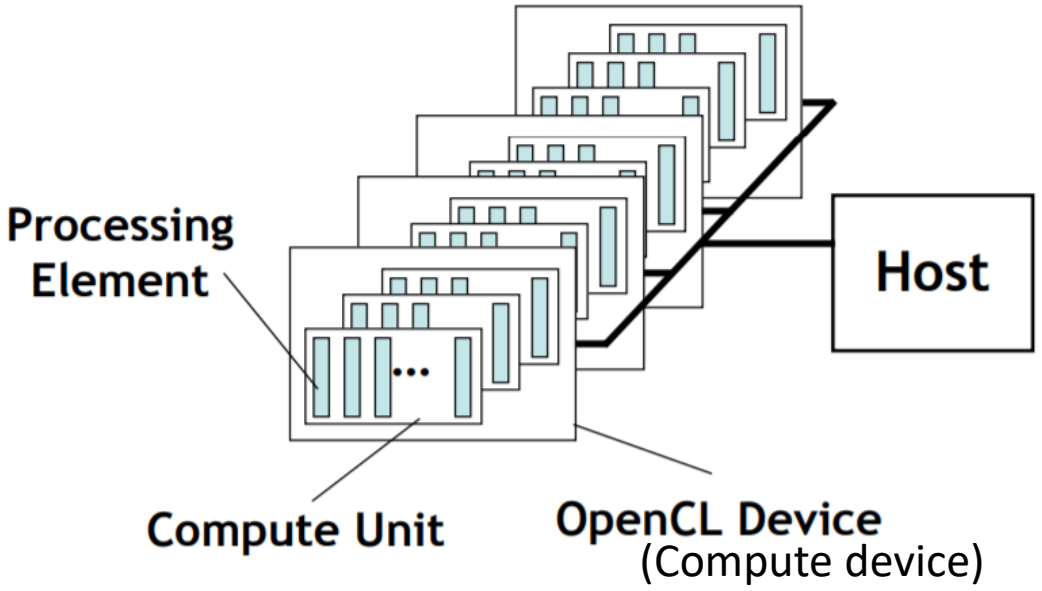
- Task level Parallelism-Entire data used in multiple functions.

Heterogeneous Programming

- More than one kind of processor cores
- Widely used in High Performance Computing(HPC)
- Applications involving Deep learning, Linear Algebra, Computational fluid mechanics and simulations in physics.
- Writing applications-OpenCL and CUDA frameworks

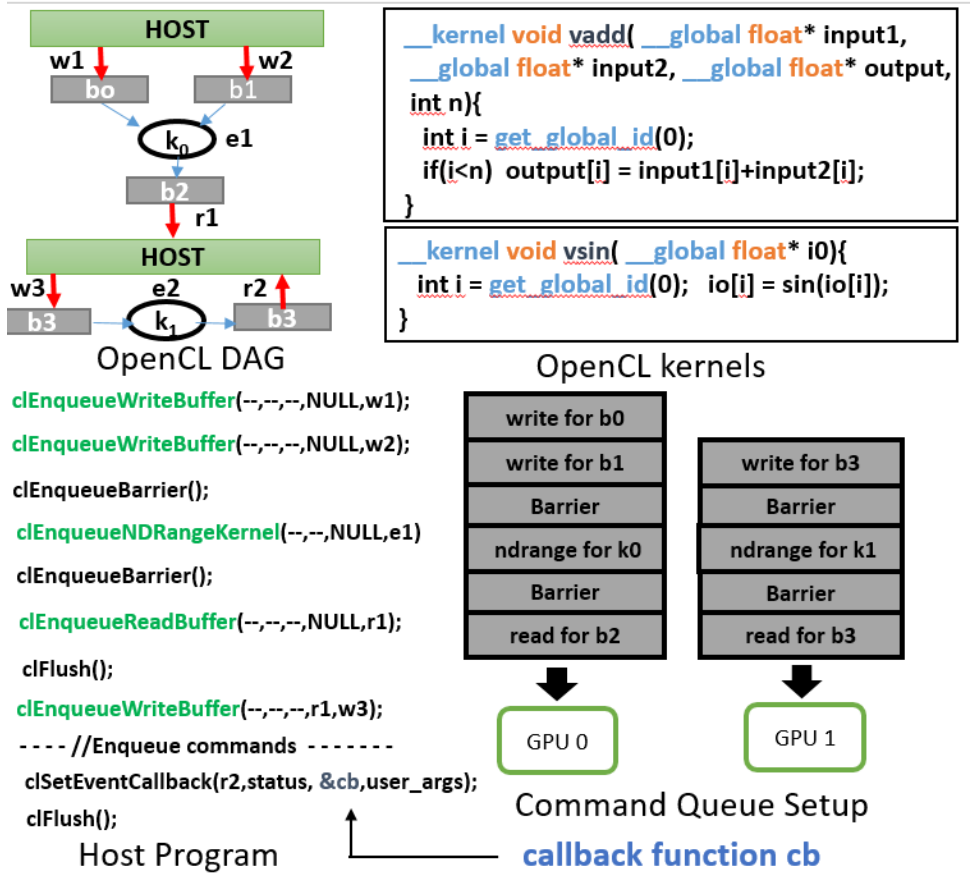
OpenCL

- Portability to run same program on different types of devices
- Kernel-functions executed on OpenCL device
- OpenCL device ➡ Compute units(work-groups) ➡ Processing elements(work-items)



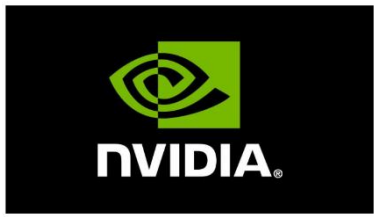
OpenCL Execution model

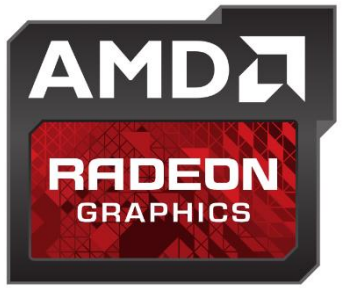
- An OpenCL application DAG comprises of kernels with dependencies between input and output buffers.
- Kernels-run on accelerator devices, Host program-runs on single core CPU device.
- Host program – responsible for scheduling the kernels following the dependency constraints



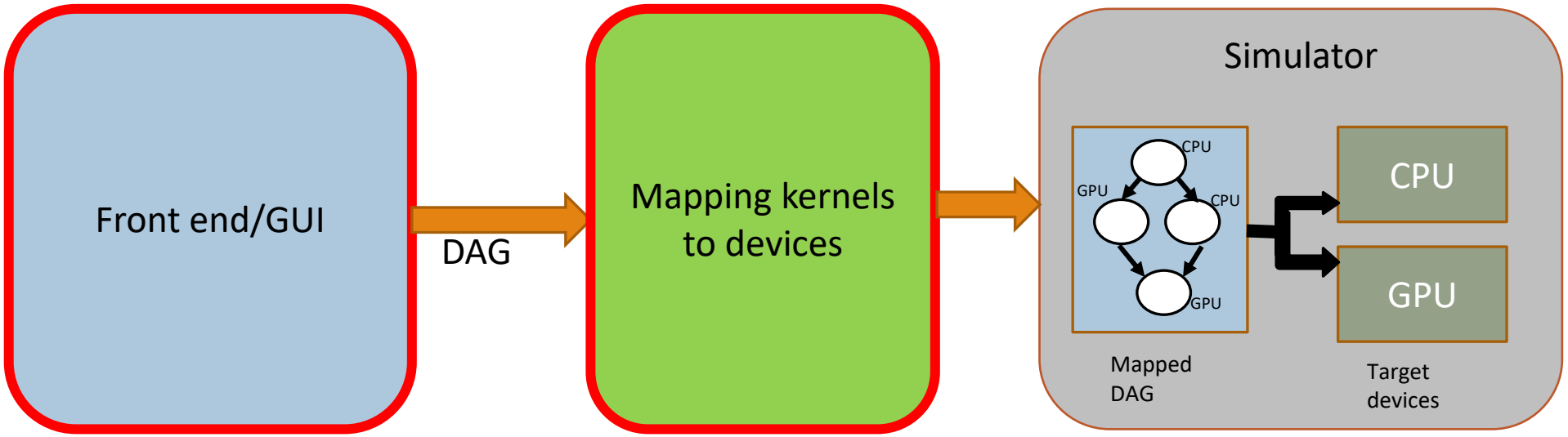
Designer Overhead

- Steep learning curve for heterogeneous programming
- Complex host side development
- User must be familiar with scheduling methodologies for optimum application to architecture mapping(for minimum makespan).





Proposed solution



Contributions

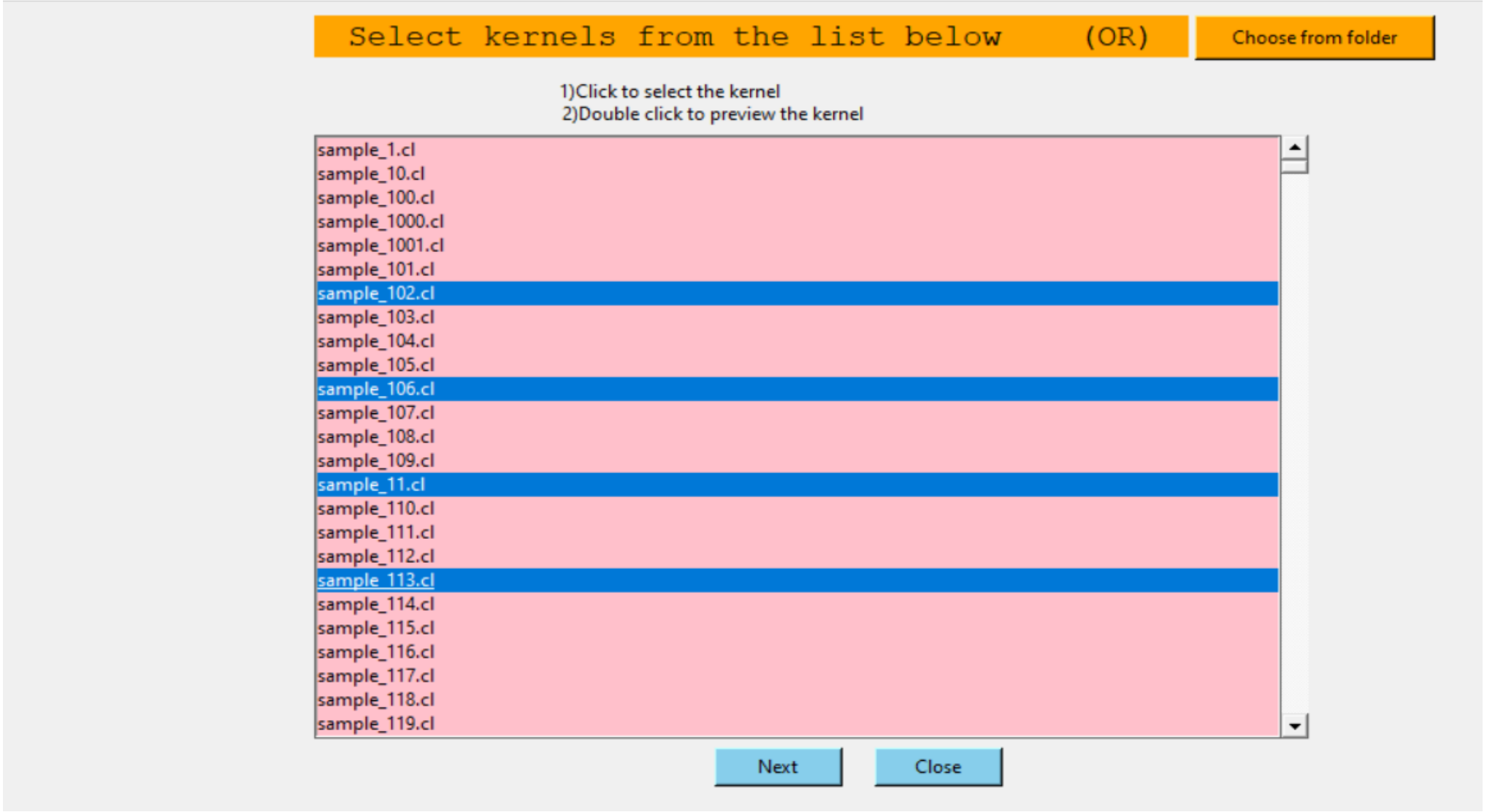
- A GUI enabled frontend providing a higher level abstraction thus enabling users to specify application dependencies with ease.

- Machine learning based implementations for ascertaining task device mapping decisions(topology oblivious and topology aware)

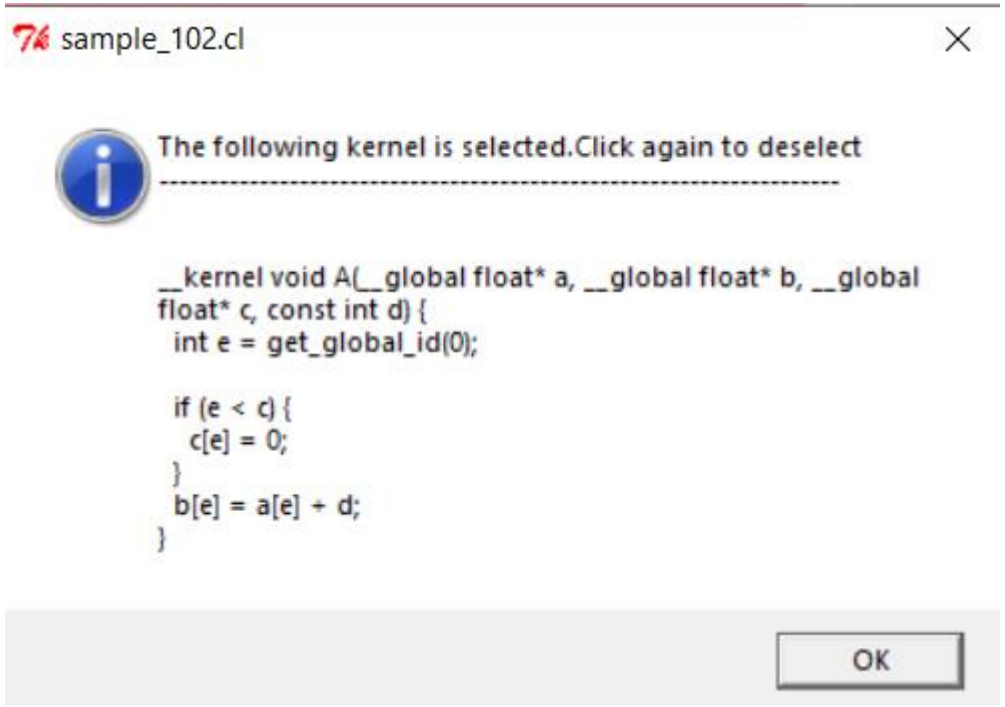
Graphical User Interface

- Allows users to create their own application DAGs without writing any complex codes and input formats.
- Built using Tkinter in python
- Select list of readily available database kernels from drop down menu

DAG Generation

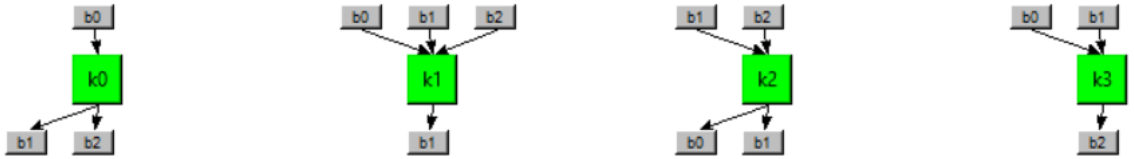


- Double click on the kernel's entry for preview.
- Use ***Choose from folder*** button to select kernels from a custom folder.
- The nodes for kernels and buffers are automatically added in the canvas



DAG Generation

Choose dependencies



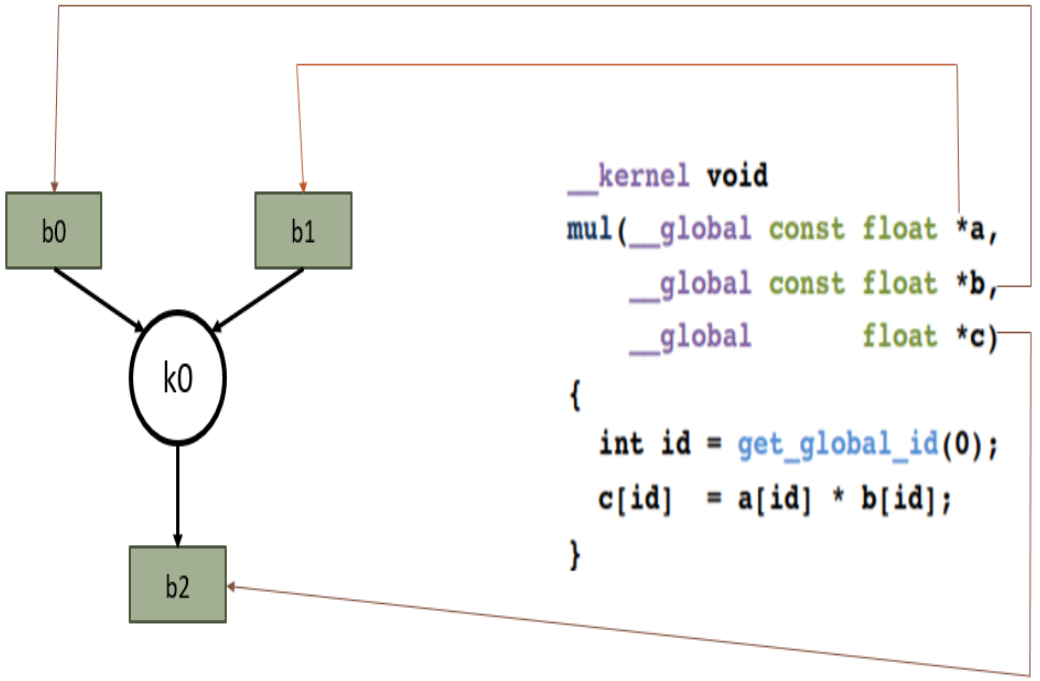
edge | Esc | Move

NOTATIONS

k0 : sample_102.cl
k1 : sample_106.cl
k2 : sample_11.cl
k3 : sample_113.cl

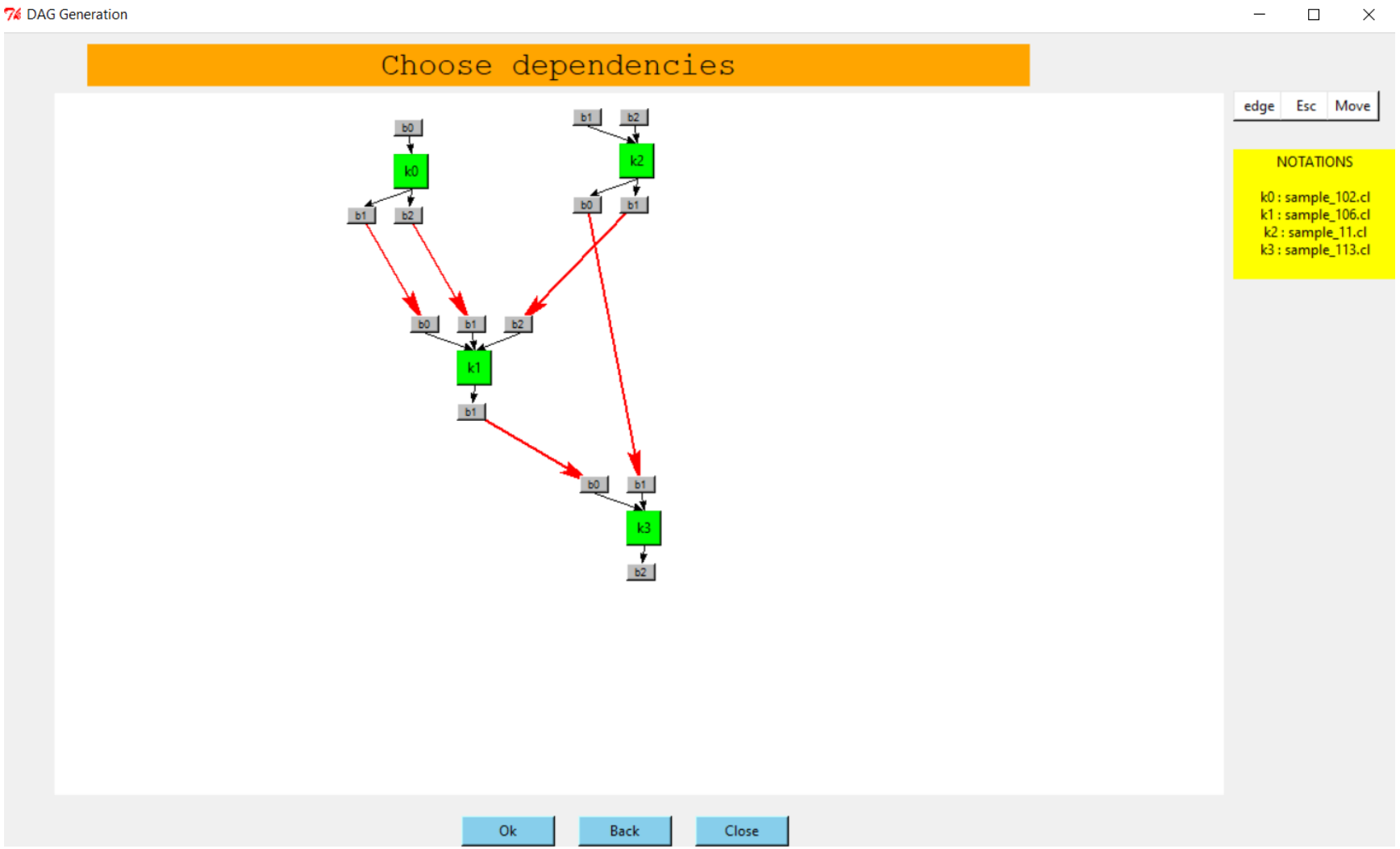
Ok | Back | Close

- Buffer nodes are added by parsing through the kernel.

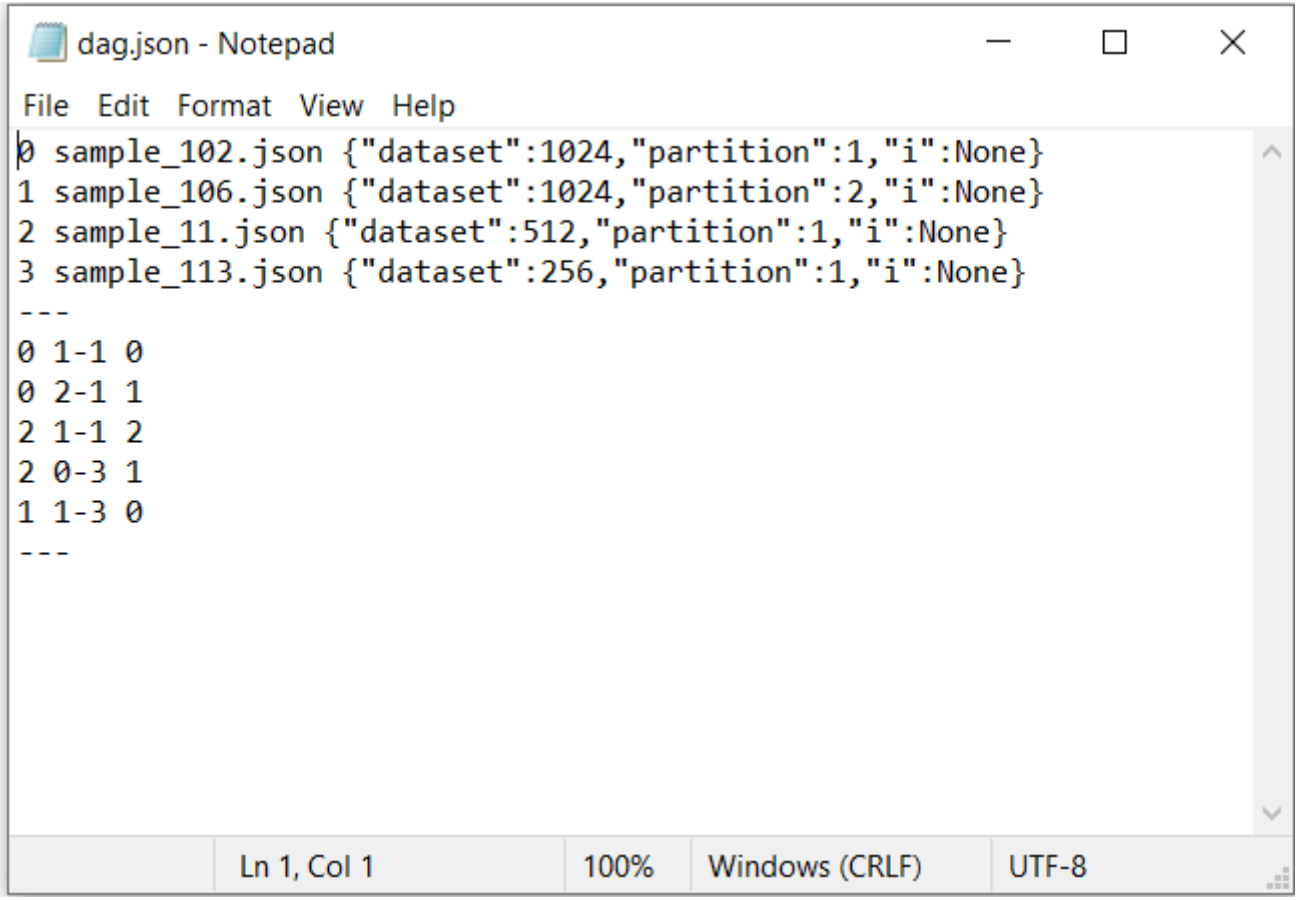


- Parameters for each kernel – edit Json text in ***Parameters*** window
- Symbolic variables – entry widgets at the bottom.

- Specify the dependencies by drawing in the canvas.



- JSON Specification file created when submitting the DAG.
- Used by the backend of the simulator for creating DAGs using Networkx.



Kernel mapping stage

- The DAG generated through GUI-Each node mapped to run on GPU or CPU device.

- Basically, generate a command queue for each device

- Two types of mapping:

  A)Topology oblivious mapping

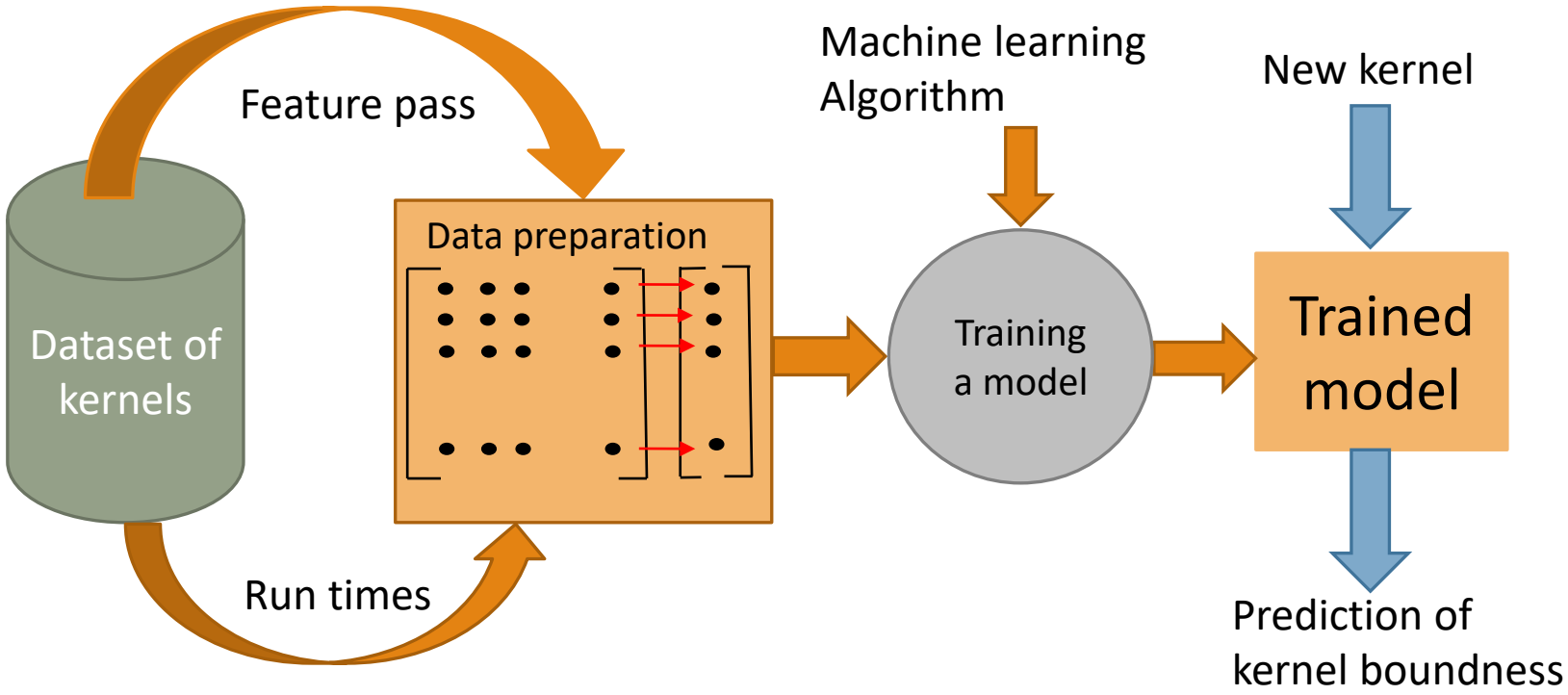
  B)Topology aware mapping

A)Topology oblivious mapping

- Kernels mapped to devices individually

Table 4: Kernel features

Basic Blocks	#Basic Blocks
Branches	#Branches
DivInsts	#Divergent Instructions
DivRegionInsts	#Instructions in Divergent Regions
DivRegionInstsRatio	Ratio between #instructions inside Divergent Regions and the Total instructions
DivRegions	#Divergent Regions
TotInsts	#Instructions
FPInsts	#Floating point Instructions
Int/FP Inst Ratio	Ration between Integer and Floating Point Instructions
IntInsts	#Integer Instructions
MathFunctions	#Math Builtin Functions
Loads	#Loads
Stores	#Stores
Barriers	#Barriers



- Trained logistic regression and XGBoost classification models.
- Imbalanced dataset – used SMOTE sampling technique.
- Feature selection methods for choosing only best features for the model.
- RandomizedSearch – for tuning hyper-parameters.
- Evaluation metrics used: 20% split test data, 10-fold cross-validation

- XGBoost over-performed logistic regression model with 10-fold cross validation accuracy of 93.4%. For logistic regression it is 84.4%.

TABLE 2: Logistic Regression

Evaluation method	Metric	K=10	K=20	K=30	K=40	K=50	K=56(all)
20% split	accuracy	59.45	60.9	57.6	56.13	58	59.1
	precision	79.4	80.8	76	84	83	86.5
	recall	57.5	56.8	55	52.7	55.4	55.5
10-fold cross validation	accuracy	81.4	84.4	78.8	84	79.6	82.4
	precision	56	55.6	56.4	55	55.4	55.6
	recall	58.7	58.4	59	58	57.7	58.4

TABLE 3: XGBoost

Evaluation method	Metric	K=10	K=20	K=30	K=40	K=50	K=56(all)
20% split	accuracy	84.4	87.6	89.23	87.4	85.34	86.6
	precision	96.8	95.6	92.7	89.9	91.2	89.7
	recall	77.6	83.1	86.2	85.14	79.5	84.5
10-fold cross validation	accuracy	96	93.4	92.8	92.11	91.2	90.5
	precision	76.8	84	84.23	84.1	84.3	84.4
	recall	83.5	88	87.7	87.3	87.14	86.9

- Also trained linear regression, XGBoost regression, SVM for predicting the speedup ratios.

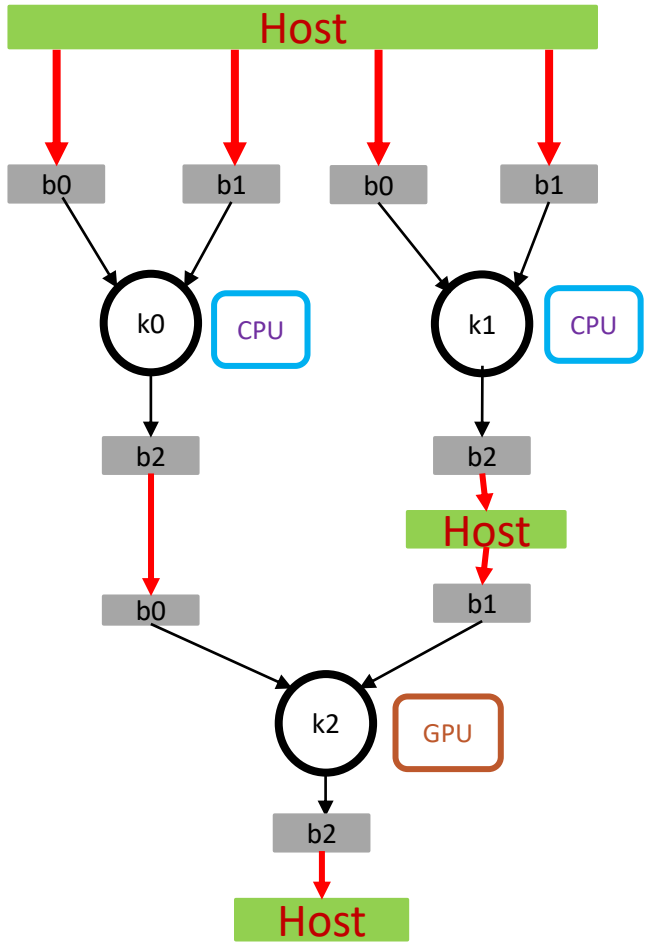
- Evaluated using RMSE, MAE and R^2 score.

TABLE 4: Regression models

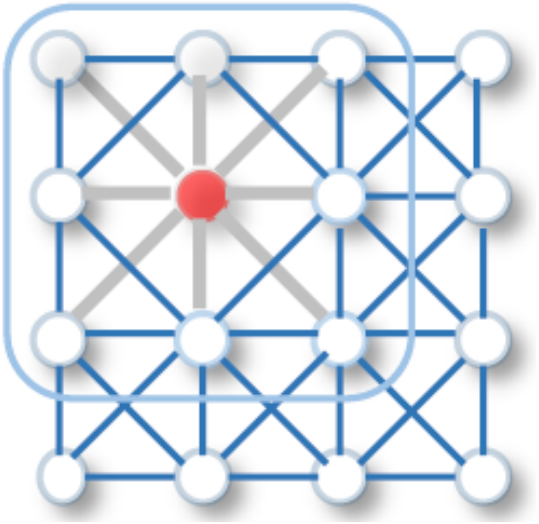
Model	RMSE	MAE	R^2 score
Linear Regression	1.2316	1.0056	0.078
XGBoost	1.1189	0.9013	0.241
Support Vector Machine(SVM)	1.1789	0.9334	0.136

B)Topology aware scheduling

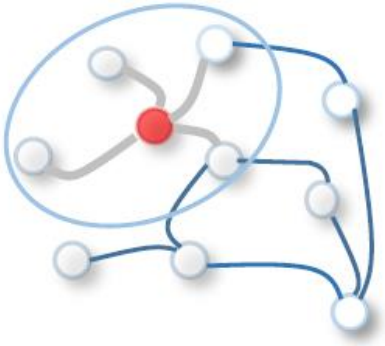
- Kernel mapping depends on state information of neighborhood kernels as well.
- Collection of kernels mapped to the same device affect synchronization and data transfer overheads.



- Graph neural networks-capture the dependence of graphs via message passing between nodes.
- CNNs involves convolution and pooling operations using a 2D or 3D filter of weights.
- GCNNs carry out message passing using filter of essential nodes and edges.

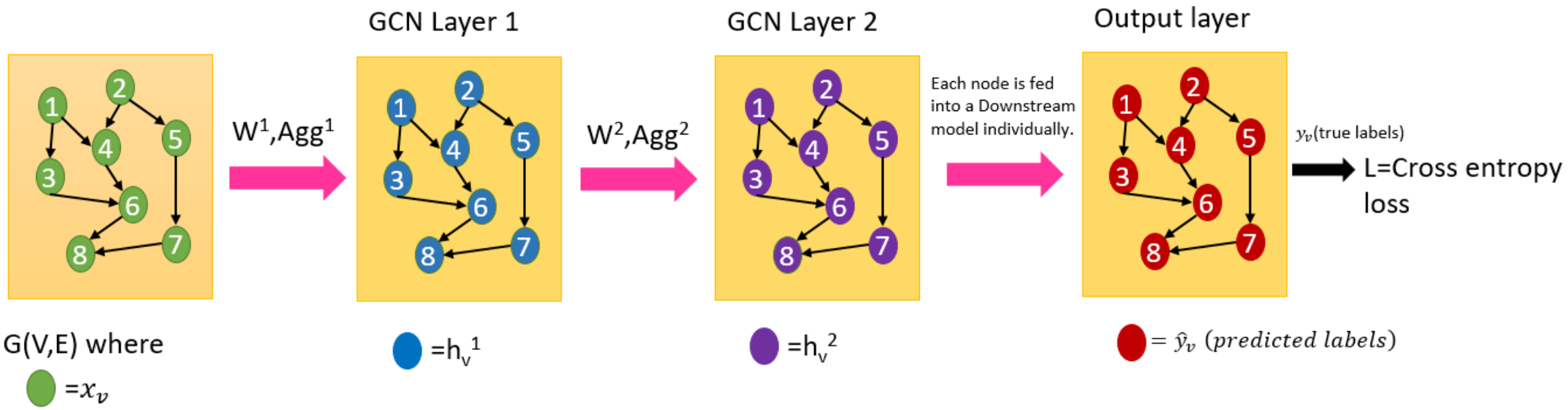


Euclidean space



non Euclidean space

GraphSAGE – A variant of GCNNs



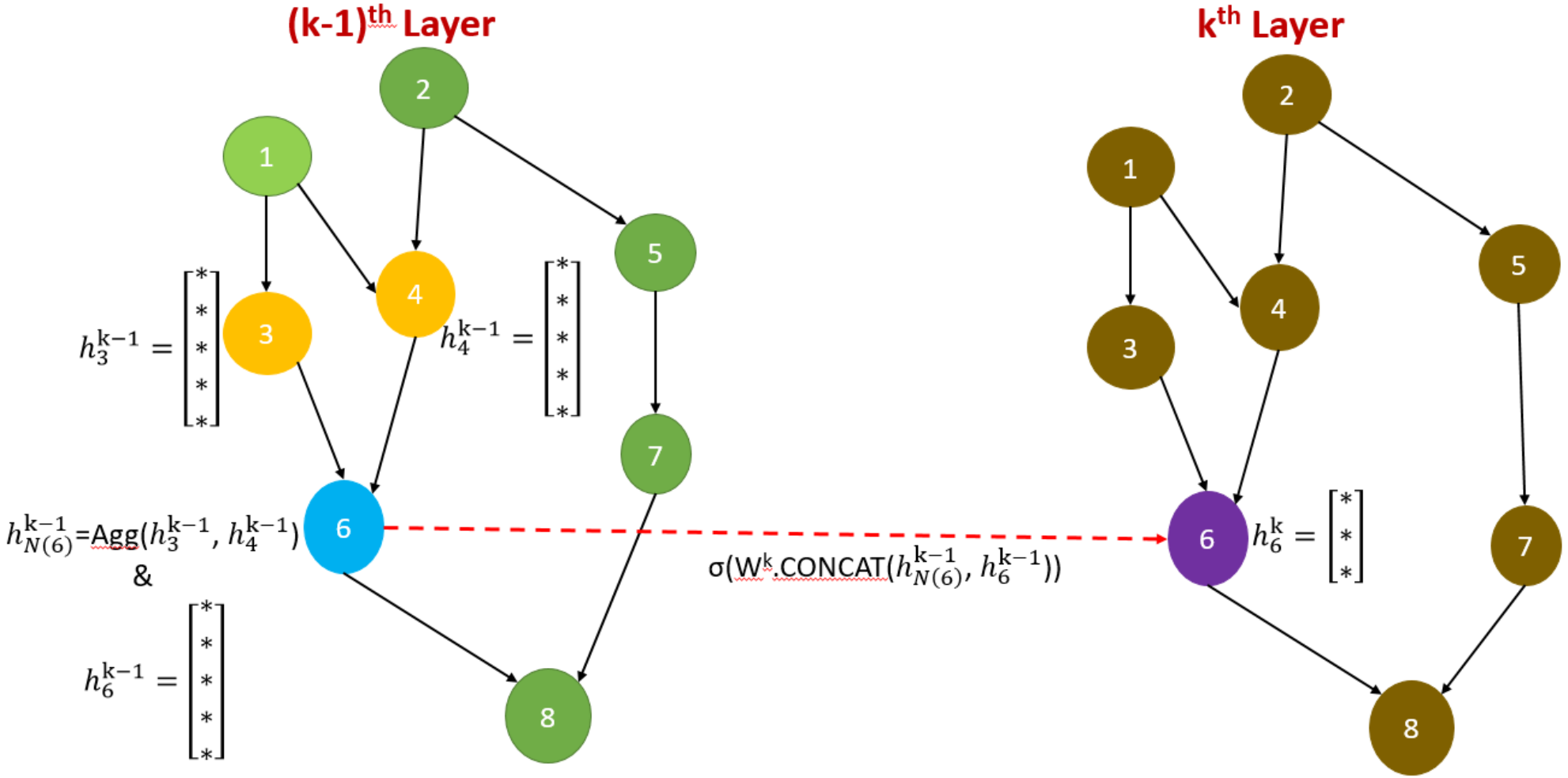
Convolution operation on a node at each layer

$h_{N(v)}^k$= Aggᵏ({h_u^{k-1}, ∀u∈ $N(v)$}) //Aggregation

h_v^k = σ(Wᵏ.[h_v^{k-1}, $h_{N(v)}^k$]) //Transformation

$h_v^k = h_v^k / \|h_v^k\|$ //Normalization

where $N(v)$={u: (u,v) ∈ E} and h_v^0=x_v


(k-1)th Layer
kth Layer
1
2
3
4
5
6
7
8
$h_3^{k-1} = \begin{bmatrix} * \\ * \\ * \\ * \\ * \end{bmatrix}$
$h_4^{k-1} = \begin{bmatrix} * \\ * \\ * \\ * \\ * \end{bmatrix}$
$h_{N(6)}^{k-1}$=Agg(h_3^{k-1}, h_4^{k-1})
&
$h_6^{k-1} = \begin{bmatrix} * \\ * \\ * \\ * \\ * \end{bmatrix}$
σ(W^k.CONCAT($h_{N(6)}^{k-1}$, h_6^{k-1}))
$h_6^k = \begin{bmatrix} * \\ * \\ * \end{bmatrix}$

Algorithm 2 Forward propagation algorithm

Input: Graph G(V,E); input features $\{x_v, \forall v \in V\}$; number of layers K; Aggregator functions(Agg^k) and weight matrices(W^k), $\forall k \in \{1, ..., K\}$; non linearity σ ; Downstream Model D.

Output: Binary labels $\hat{y}_v, \forall v \in V$. 0 represents CPU device and 1 represents GPU device.

$h_v^0 \leftarrow x_v$
for $k \in \{1, 2, 3, ..., K\}$ **do**
  for $v \in V$ **do**
    $h_{N(v)}^k \leftarrow Agg^k(\{h_u^{k-1}, \forall u \in N(v)\})$
    $h_v^k \leftarrow \sigma(W^k.CONCAT(h_{N(v)}^k, h_v^{k-1}))$
  $h_v^k \leftarrow h_v^k / ||h_v^k||^2, \forall v \in V$
for $v \in V$ **do**
  $z_v \leftarrow D(h_v^K)$
$\hat{y}_v \leftarrow z_v$

- W^k => Weight matrices for transformation.
- Agg^k => Aggregator function:

1)Mean aggregator-Element wise mean of the neighborhood node vectors(No trainable parameters)

2)LSTM aggregator-Based on LSTM architecture –has trainable parameters but permutation dependent

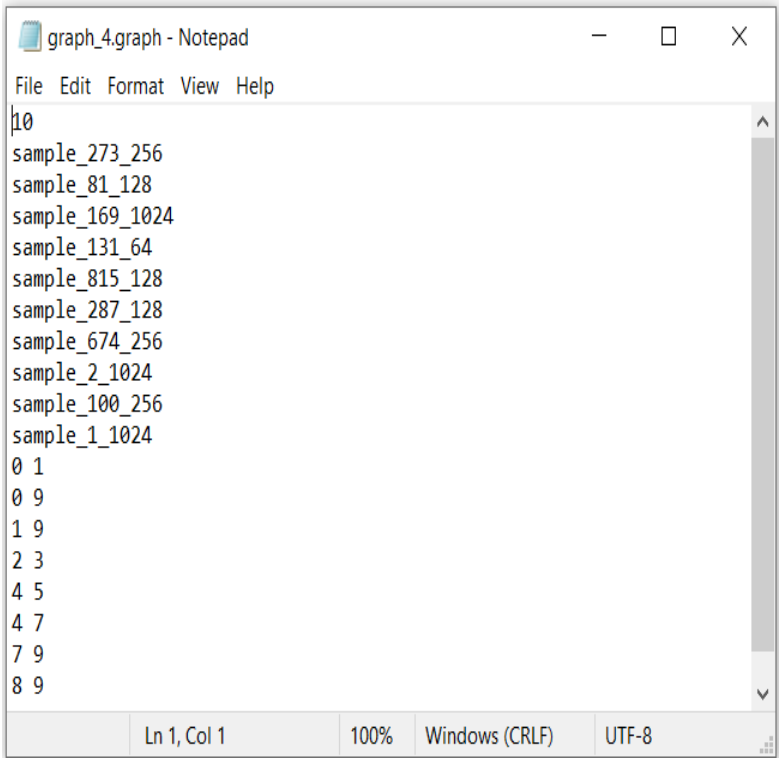
3)Pooling aggregator-element wise max pooling after transforming each neighbor vector with a fully connected layer(trainable parameters)

Training data – Random DAG Generator

Algorithm 1 DAG Generating Method

Input: n-number of kernels, p-probability measure

```
dagkernels ← Sample n kernels randomly from database kernels
path ←Create a '.graph' file and write list of dagkernels into it.
for i ∈ 1...n do
    for j ∈ i + 1...n do
        if Random() <p then
            write the indices(i,j) seperated by white space into the graph file.
nCPU ← 2, mGPU ← 2
timings ←dict()
for each of 2^n enumerations do
    DC ← DAGCreator()
    dag ← DC.create_dag_from_file(path,device_enumeration)
    SA ← SchedulingAlgorithm(dag,nCPU,mGPU)
    SA.list_scheduling()
    timings[device_enumeration] ← SA.makespan()
Copy the timings dictionary to a 'Json' file
```



- Loss function:

$$J(\theta) = \frac{\sum_{m=1}^{M} w_m * L_m(\theta)}{\sum_{m=1}^{M} w_m}$$

$$L_m(\theta) = -\frac{1}{n} \sum_{v \in V_m} [y_v log(\hat{y}_v) + (1 - y_v) log(1 - \hat{y}_v)]$$

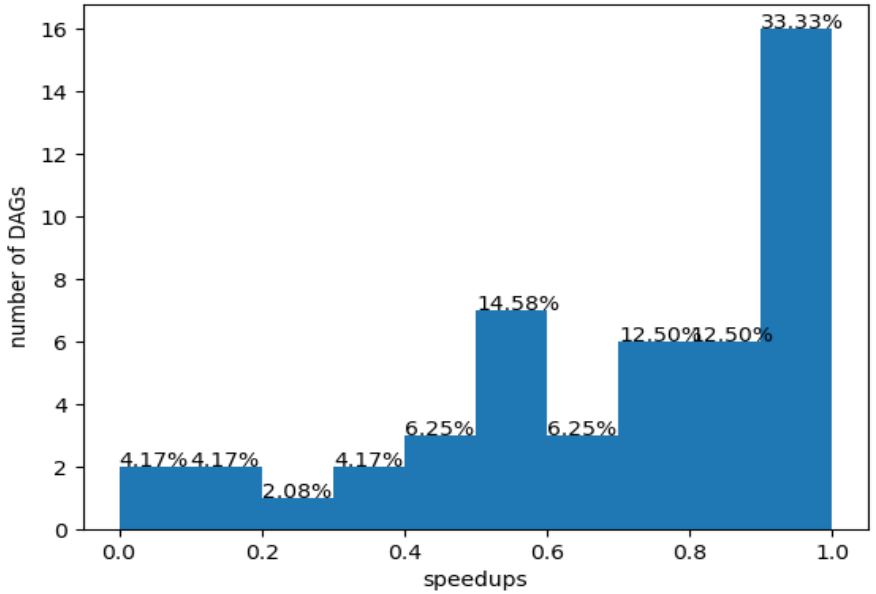
- Trained model with 3 GCN layers – dimension of state vectors at each layer = 16, downstream model = Logistic Regression.

- Highly unbalanced model – Predicting single class always.

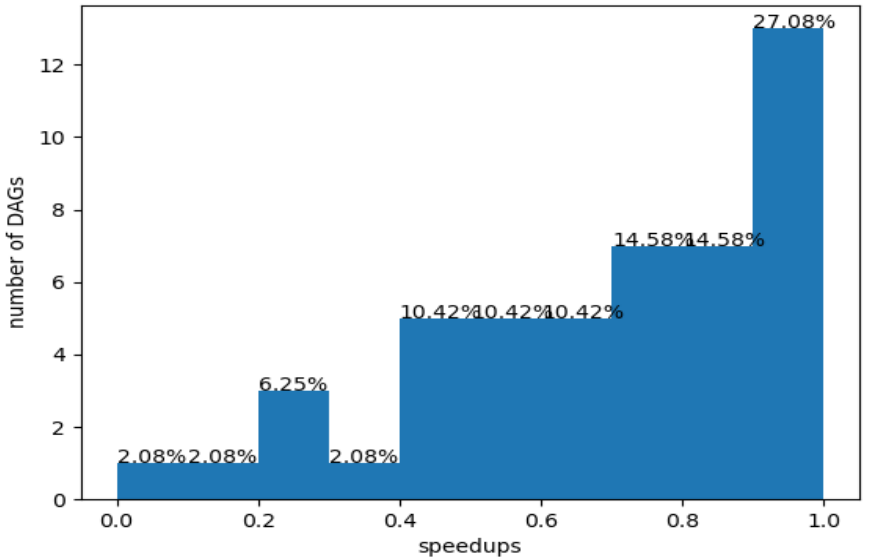
Aggregator function	Test loss	Accuracy	Average Speedup
Mean Aggregator	0.6083	77.33%	0.83913
MeanPool Aggregator	0.5385	77.33%	0.83913
MaxPool Aggregator	0.6575	77.33%	0.83913
LSTM Aggregator	0.5935	77.33%	0.83913

- Introduced new parameter b ∈ [0,1] in DAG Generator – obtained a balanced dataset.

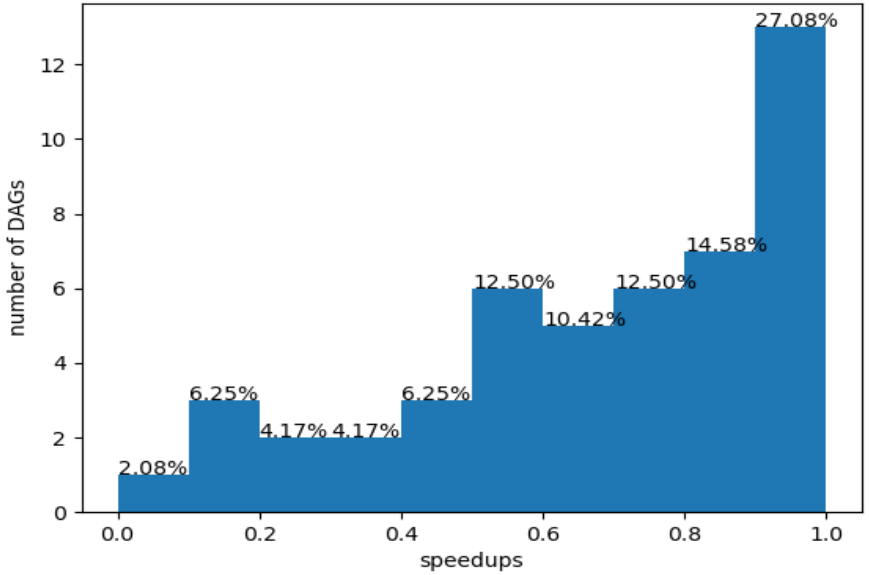
Aggregator function	Test loss	Accuracy	Average Speedup
Mean Aggregator	0.69275	61.042%	0.6955
MeanPool Aggregator	0.69532	61.25%	0.6957
MaxPool Aggregator	0.7349	59.16%	0.6874
LSTM Aggregator	0.7304	56.875%	0.6779



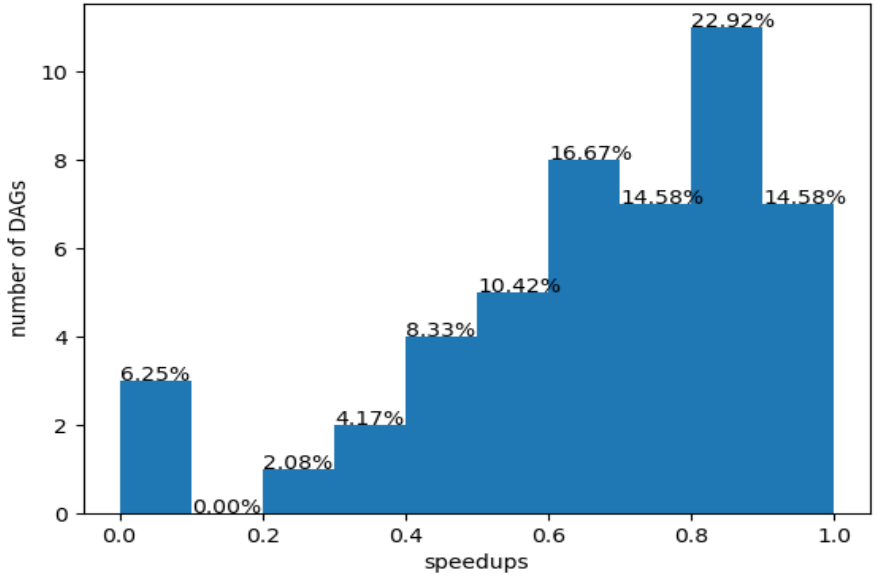
a)Mean aggregator



b)Mean Pool aggregator



a)Max Pool aggregator



a)LSTM aggregator

Thank you