

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

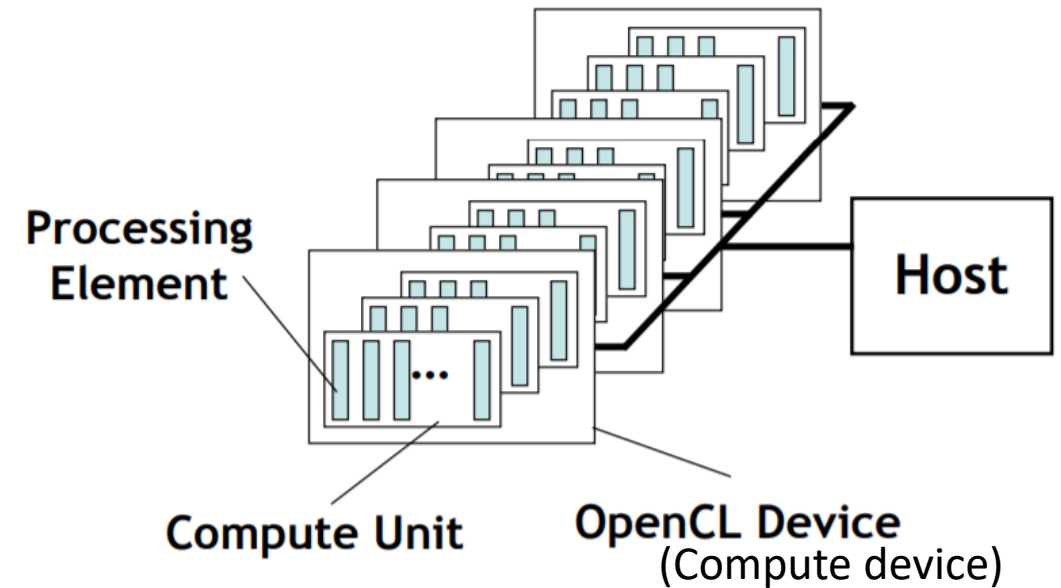
- ❑ More than one kind of processor cores
- ❑ Widely used in High Performance Computing(HPC)
- ❑ Applications involving Deep learning, Linear Algebra, Computational fluid mechanics and simulations in physics.
- ❑ Writing applications-OpenCL and CUDA frameworks

OpenCL



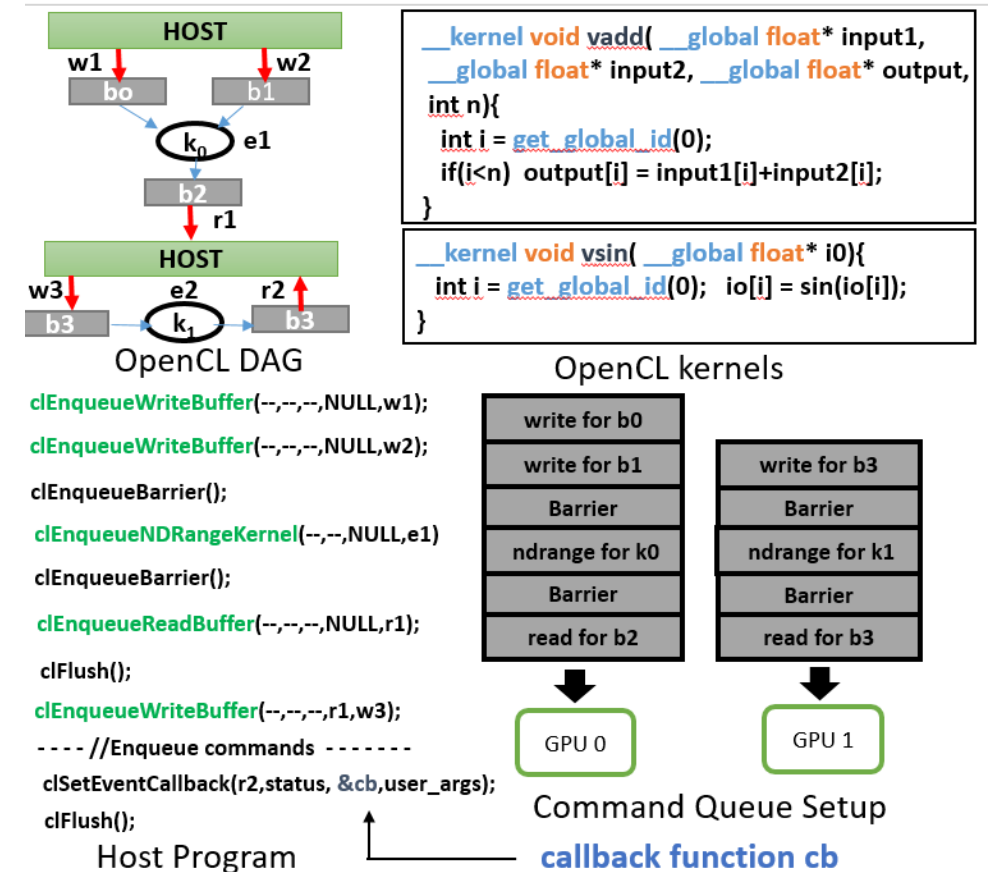
OpenCL

- ❑ 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)



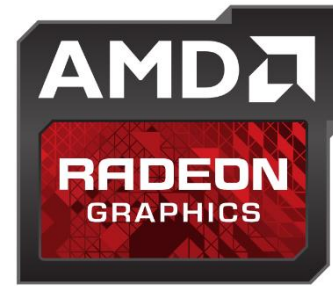
OpenCL Execution model

- ❑ 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

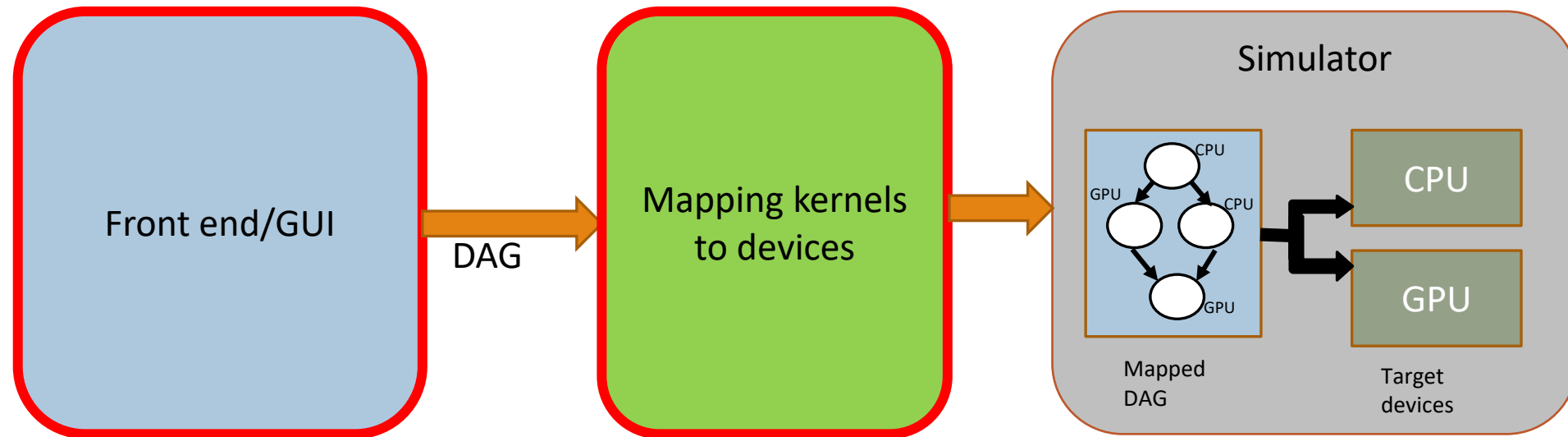


Designer Overhead

- ❑ 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

Select kernels from the list below

(OR)

Choose from folder

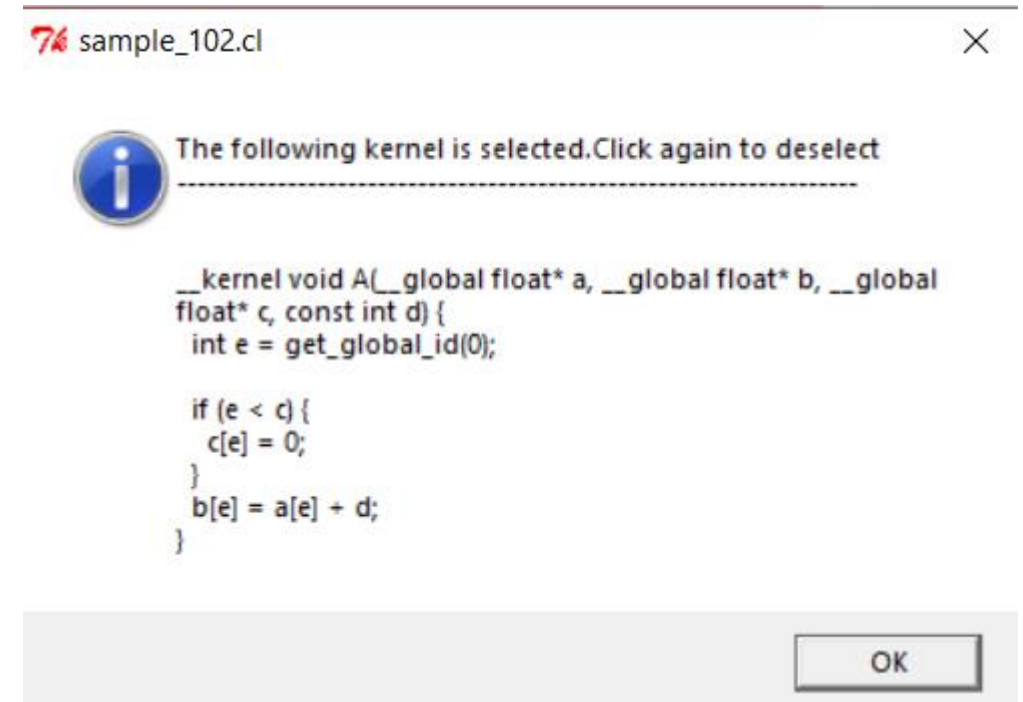
- 1)Click to select the kernel
- 2)Double click to preview the kernel

sample_1.cl
sample_10.cl
sample_100.cl
sample_1000.cl
sample_1001.cl
sample_101.cl
sample_102.cl
sample_103.cl
sample_104.cl
sample_105.cl
sample_106.cl
sample_107.cl
sample_108.cl
sample_109.cl
sample_11.cl
sample_110.cl
sample_111.cl
sample_112.cl
sample_113.cl
sample_114.cl
sample_115.cl
sample_116.cl
sample_117.cl
sample_118.cl
sample_119.cl

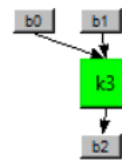
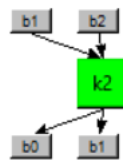
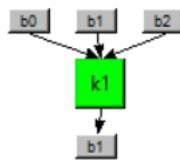
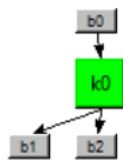
Next

Close

- ❑ 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



Choose dependencies



edge Esc Move

NOTATIONS

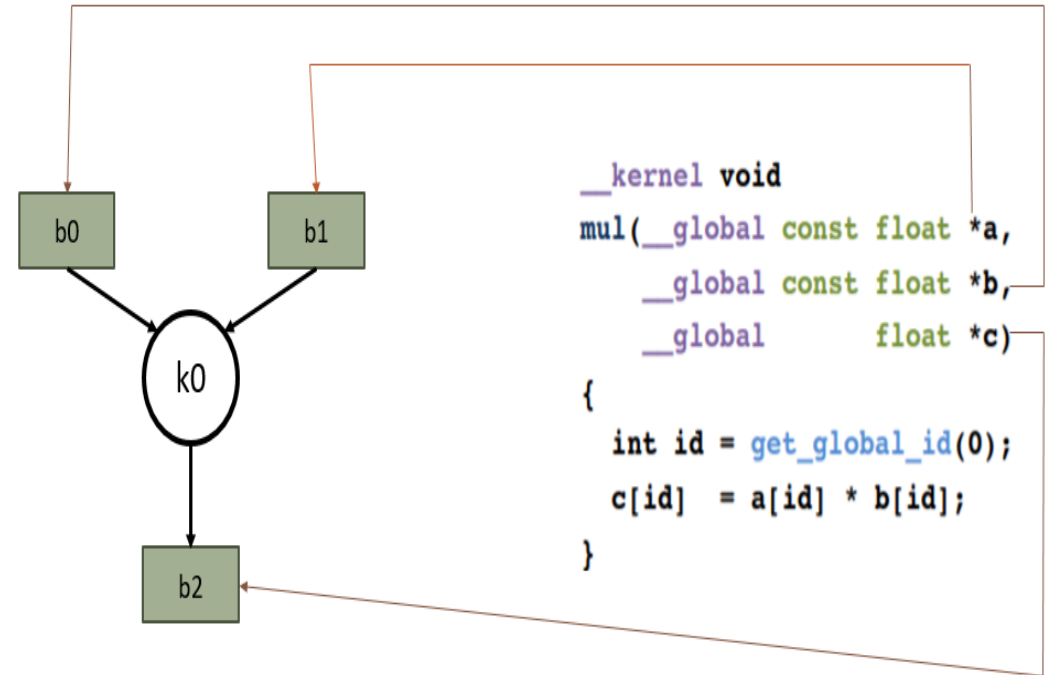
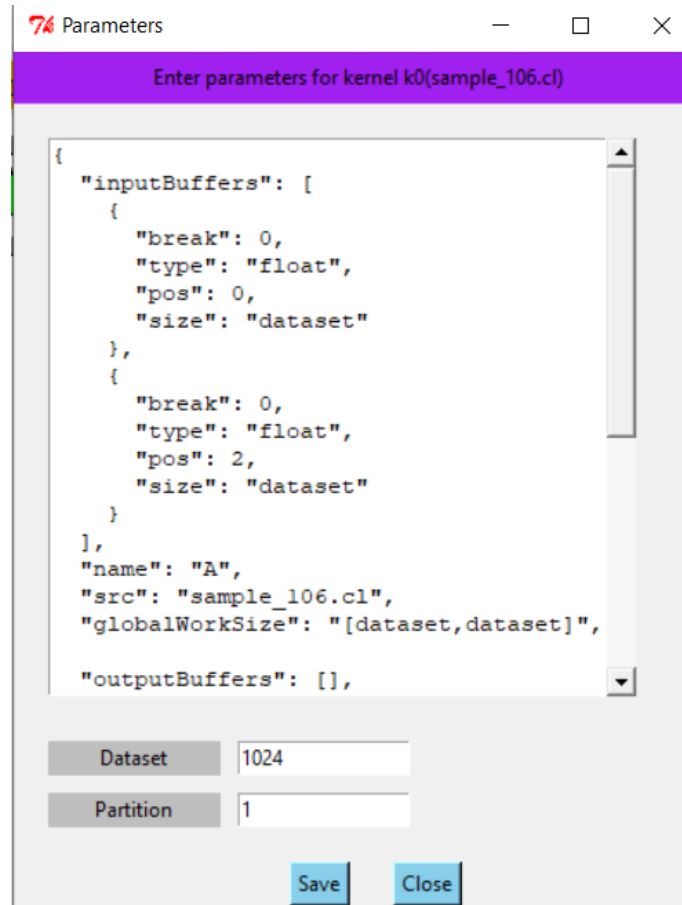
k0 : sample_102.cl
k1 : sample_106.cl
k2 : sample_11.cl
k3 : sample_113.cl

Ok

Back

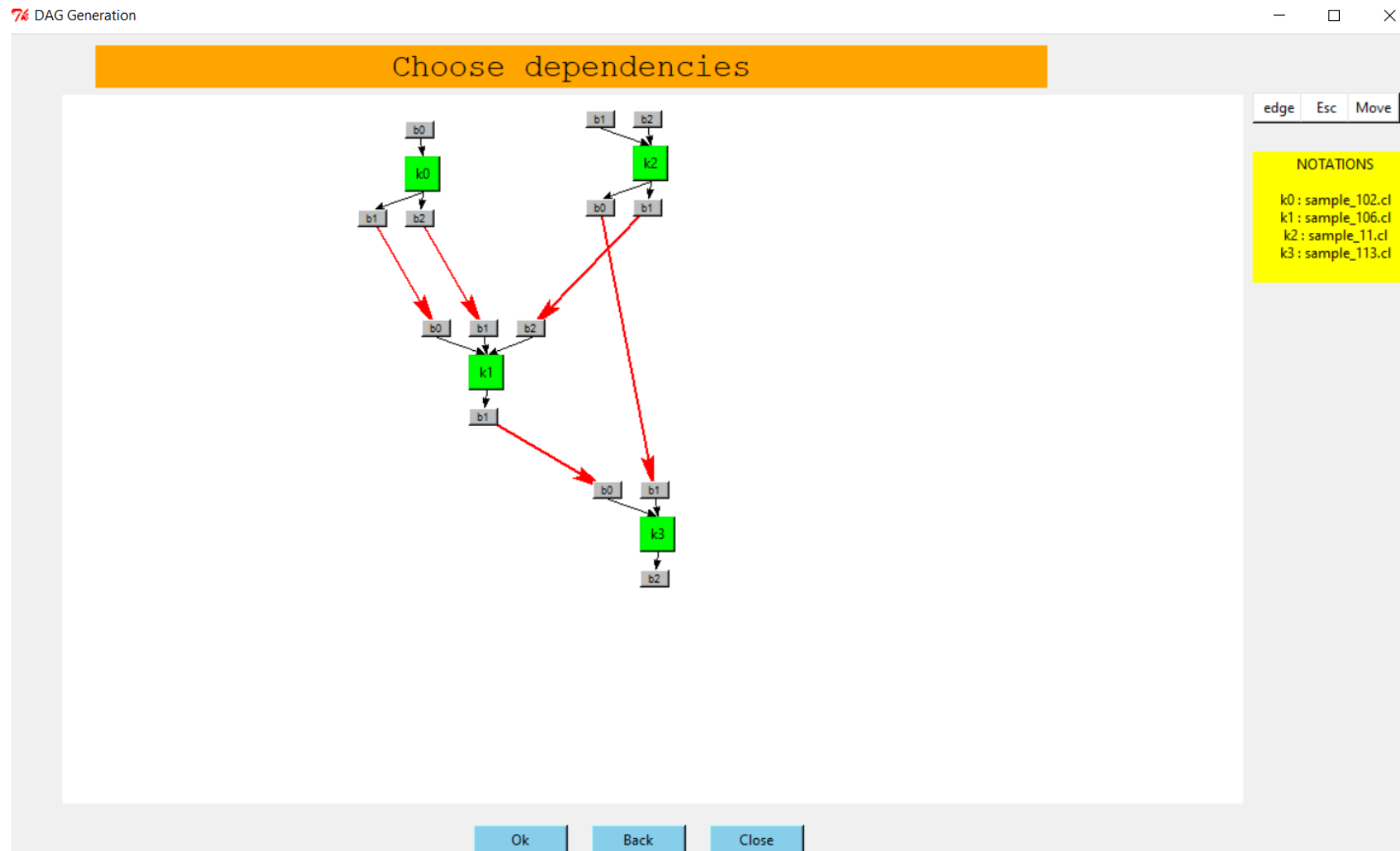
Close

- Buffer nodes are added by parsing through the kernel.

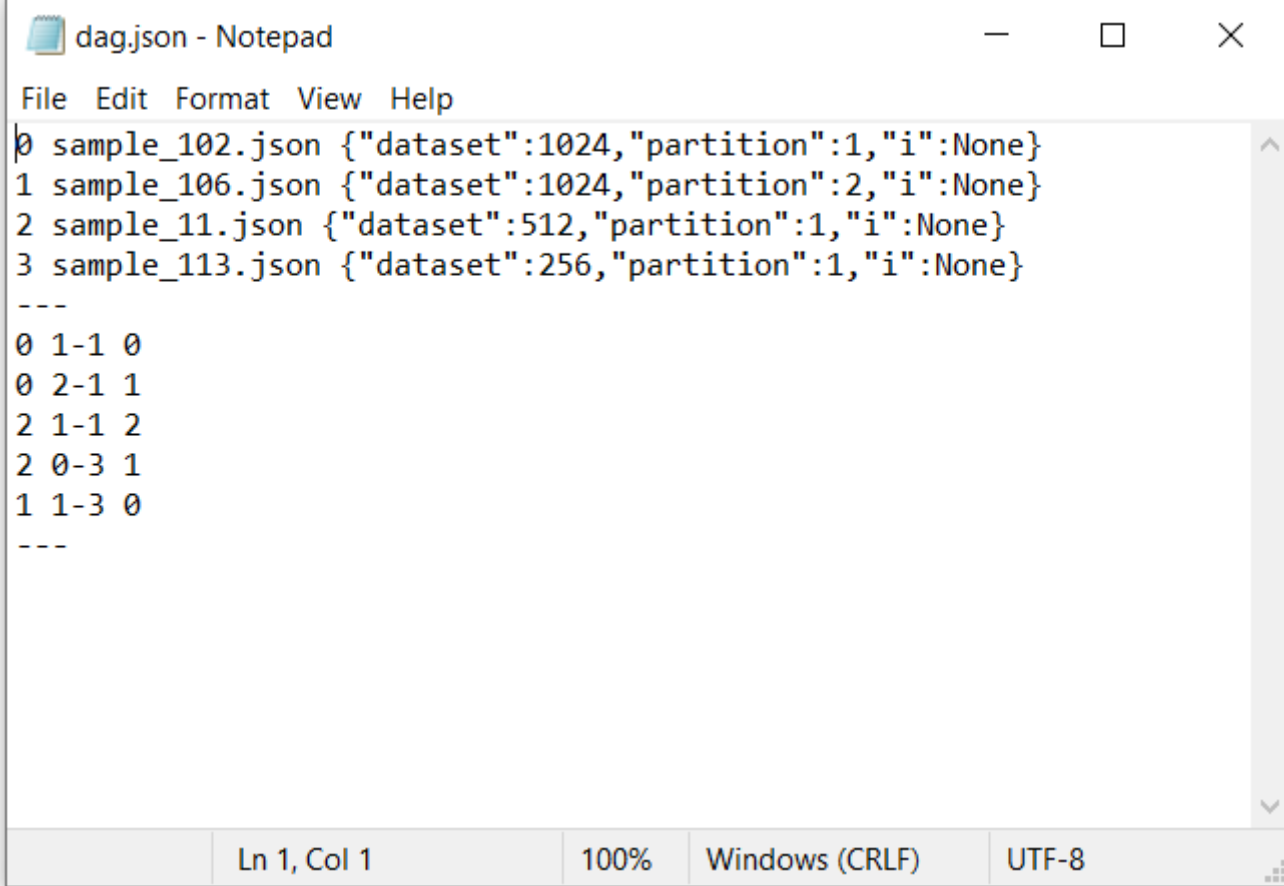


- Parameters for each kernel – edit Json text in **Parameters** window
- Symbolic variables – entry widgets at the bottom.

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.



```
dag.json - Notepad
File Edit Format View Help
0 sample_102.json {"dataset":1024,"partition":1,"i":None}
1 sample_106.json {"dataset":1024,"partition":2,"i":None}
2 sample_11.json {"dataset":512,"partition":1,"i":None}
3 sample_113.json {"dataset":256,"partition":1,"i":None}
---
0 1-1 0
0 2-1 1
2 1-1 2
2 0-3 1
1 1-3 0
---
```

Ln 1, Col 1 100% Windows (CRLF) UTF-8

Kernel mapping stage

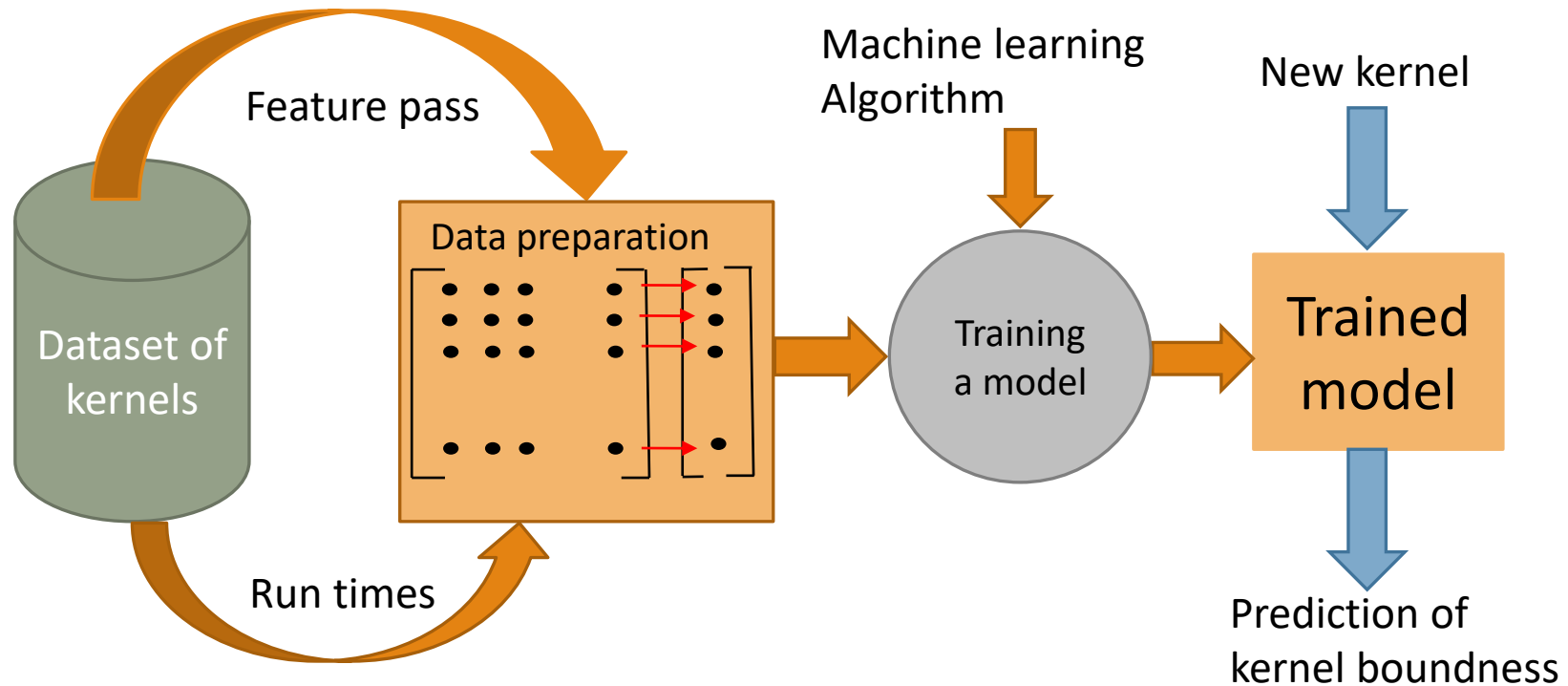
- ❑ The DAG generated through GUI-Each node mapped to run on GPU or CPU device.
- ❑ Basically, generate a command queue for each device
- ❑ Two types of mapping:
 - A)Topology oblivious mapping
 - B)Topology aware mapping

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

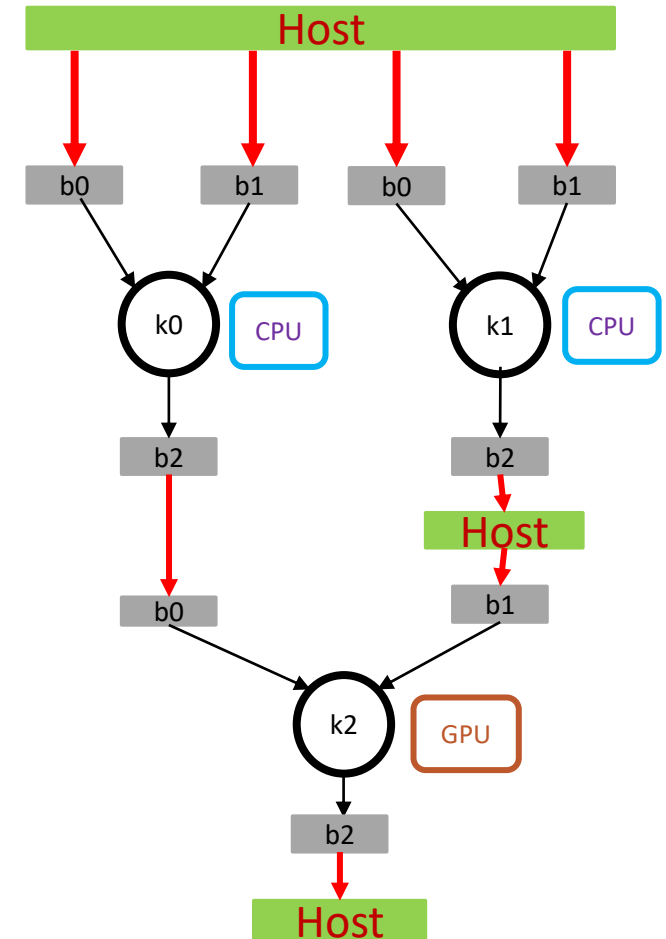
- ❑ Also trained linear regression, XGBoost regression, SVM for predicting the speedup ratios.
- ❑ Evaluated using RMSE, MAE and R^2 score.

TABLE 4: Regression models

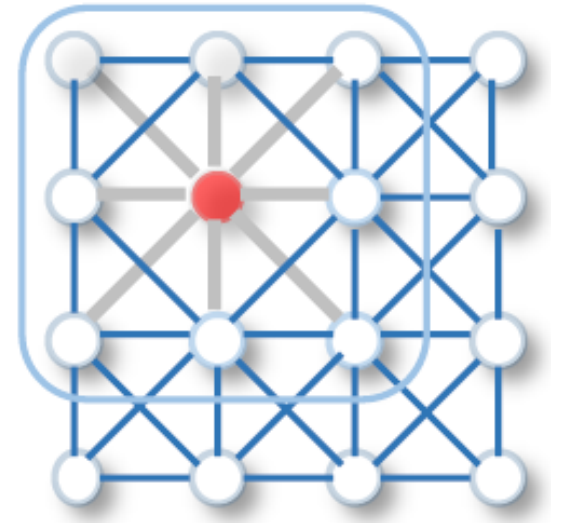
Model	RMSE	MAE	R^2 score
Linear Regression	1.2316	1.0056	0.078
XGBoost	1.1189	0.9013	0.241
Support Vector Machine(SVM)	1.1789	0.9334	0.136

B)Topology aware scheduling

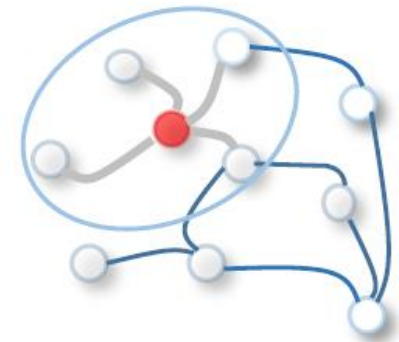
- ❑ Kernel mapping depends on state information of neighborhood kernels as well.
- ❑ Collection of kernels mapped to the same device affect synchronization and data transfer overheads.



- ❑ 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.

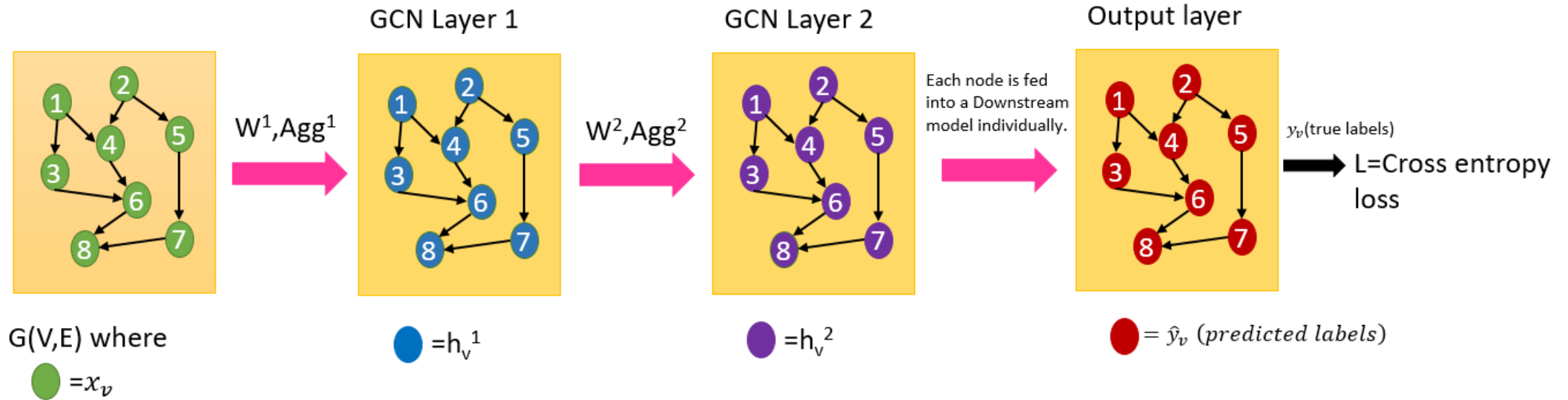


Euclidean space



non Euclidean space

GraphSAGE – A variant of GCNNs



Convolution operation on a node at each layer

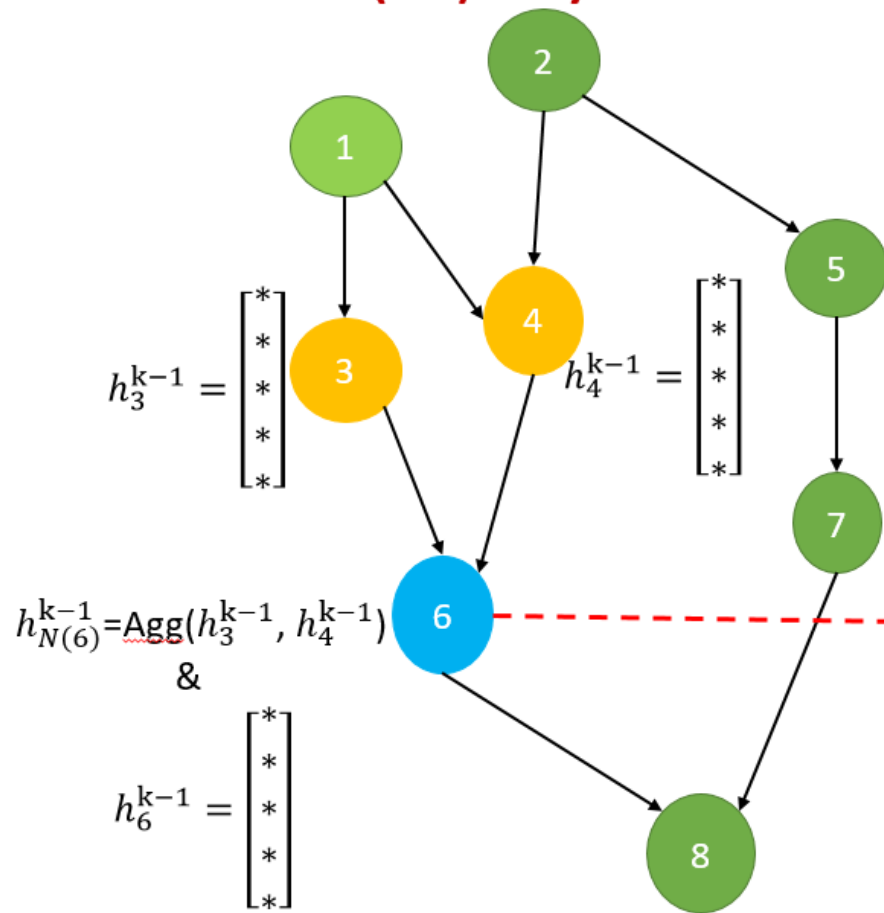
$$h_{N(v)}^k = \text{Agg}^k(\{h_u^{k-1}, \forall u \in N(v)\}) \quad // \text{Aggregation}$$

$$h_v^k = \sigma(W^k \cdot [h_v^{k-1}, h_{N(v)}^k]) \quad // \text{Transformation}$$

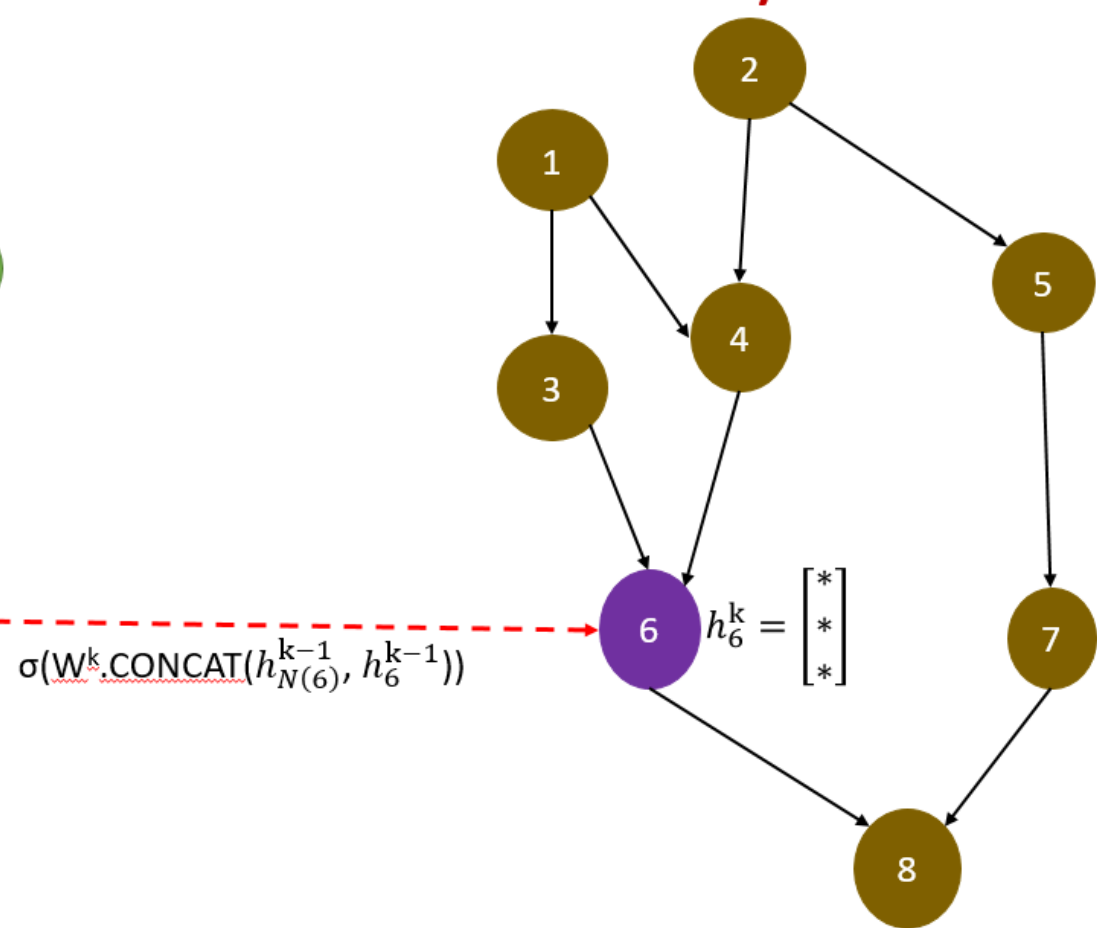
$$h_v^k = h_v^k / \|h_v^k\| \quad // \text{Normalization}$$

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

(k-1)th Layer



kth Layer



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), $\forall k \in \{1, \dots, K\}$; 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.

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

□ $W^k \Rightarrow$ Weight matrices for transformation.

□ $Agg^k \Rightarrow$ 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

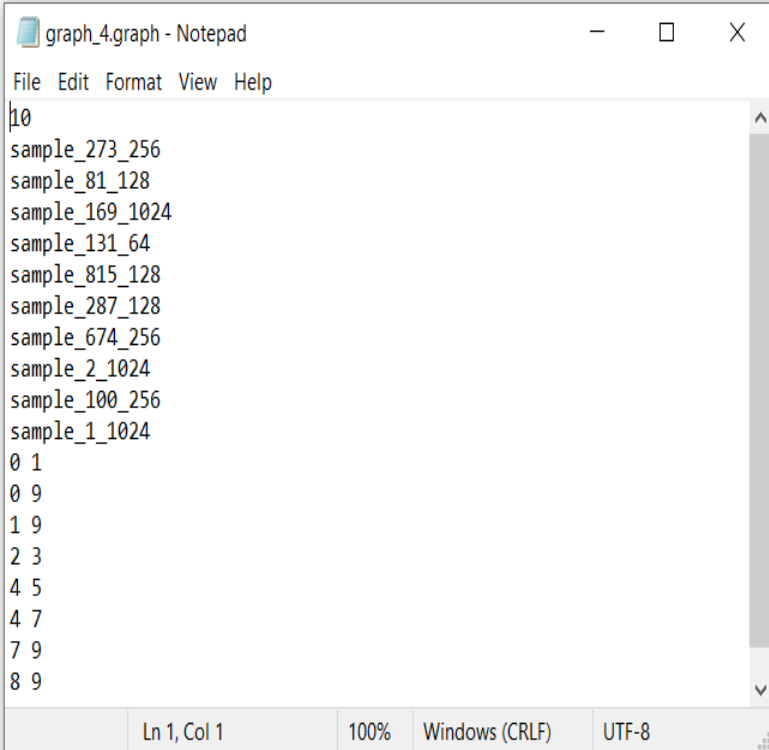
3)Pooling aggregator-element wise max pooling after transforming each neighbor vector with a fully connected layer(trainable parameters)

Training data – Random DAG Generator

Algorithm 1 DAG Generating Method

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

- 1: $dagkernels \leftarrow$ Sample n kernels randomly from database kernels
 - 2: $path \leftarrow$ Create a *'graph'* file and write list of $dagkernels$ into it.
 - 3: **for** $i \in 1 \dots n$ **do**
 - 4: **for** $j \in i + 1 \dots n$ **do**
 - 5: **if** $\text{Random}() < p$ **then**
 - 6: write the indices(i, j) separated by white space into the graph file.
 - 7: $nCPU \leftarrow 2, mGPU \leftarrow 2$
 - 8: $timings \leftarrow \text{dict}()$
 - 9: **for** each of 2^n enumerations **do**
 - 10: $DC \leftarrow \text{DAGCreator}()$
 - 11: $dag \leftarrow DC.\text{create_dag_from_file}(path, device_enumeration)$
 - 12: $SA \leftarrow \text{SchedulingAlgorithm}(dag, nCPU, mGPU)$
 - 13: $SA.\text{list_scheduling}()$
 - 14: $timings[device_enumeration] \leftarrow SA.\text{makespan}()$
 - 15: Copy the $timings$ dictionary to a *'Json'* file
-



```
graph_4.graph - Notepad
File Edit Format View Help
10
sample_273_256
sample_81_128
sample_169_1024
sample_131_64
sample_815_128
sample_287_128
sample_674_256
sample_2_1024
sample_100_256
sample_1_1024
0 1
0 9
1 9
2 3
4 5
4 7
7 9
8 9
Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

□ Loss function:

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

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

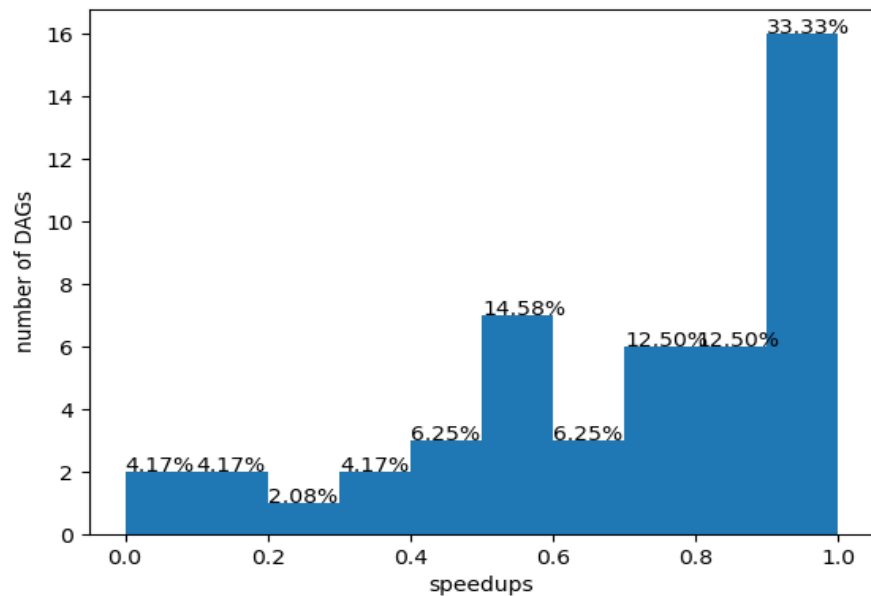
□ Trained model with 3 GCN layers – dimension of state vectors at each layer = 16, downstream model = Logistic Regression.

□ Highly unbalanced model –
Predicting single class always.

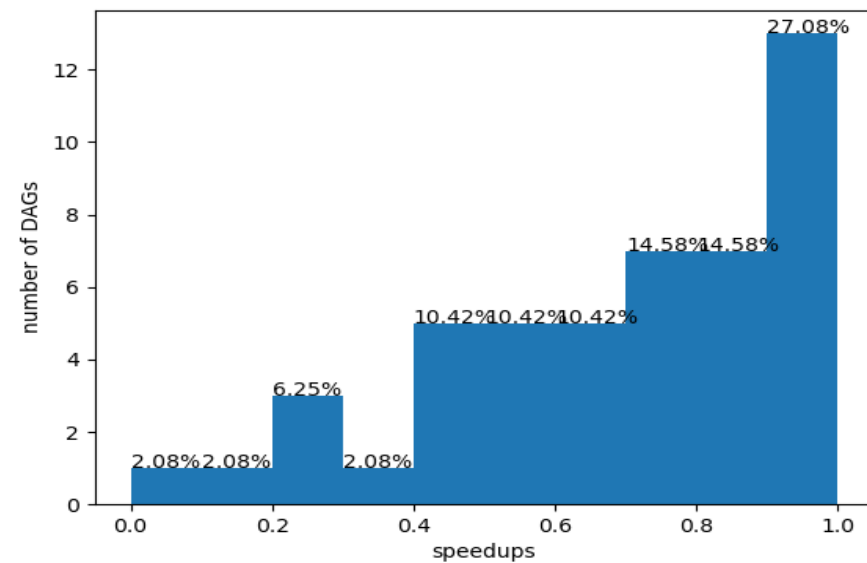
Aggregator function	Test loss	Accuracy	Average Speedup
Mean Aggregator	0.6083	77.33%	0.83913
MeanPool Aggregator	0.5385	77.33%	0.83913
MaxPool Aggregator	0.6575	77.33%	0.83913
LSTM Aggregator	0.5935	77.33%	0.83913

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

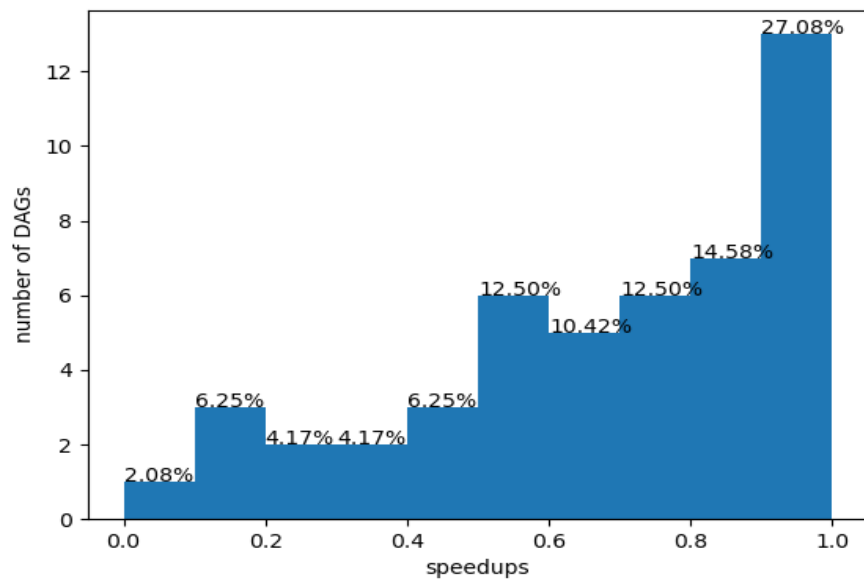
Aggregator function	Test loss	Accuracy	Average Speedup
Mean Aggregator	0.69275	67.34%	0.6955
MeanPool Aggregator	0.69532	61.25%	0.6957
MaxPool Aggregator	0.7349	59.16%	0.6874
LSTM Aggregator	0.7304	56.875%	0.6779



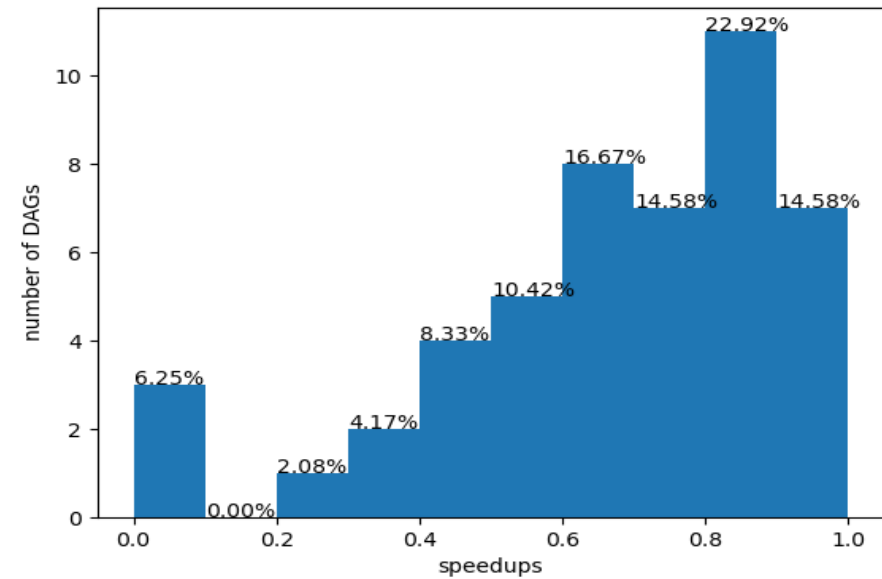
a) Mean aggregator



b) Mean Pool aggregator



a) Max Pool aggregator



a) LSI M aggregator

Thank you