A Simulator for Intelligent Scheduling of Data-Parallel Workloads on Heterogeneous Platforms

BTP report submitted to the
Indian Institute of Technology, Kharagpur
In the fulfillment for the award of the degree

of

Dual Degree
in
Mechanical Engineering

by

Joru Saikumar

(17ME33037)

Under the supervision of

Asst. Prof. Soumyajit Dey

Department of Computer Science & Engineering

Indian Institute of Technology, Kharagpur

Autumn&Spring Semesters, 2020-21

30-4-2021

DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(d) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: 30-4-2021 (Joru Saikumar)

Place: Kharagpur (17ME33037)

Department of Computer Science & Engineering

Indian Institute of Technology, Kharagpur

Kharagpur - 721302, India

CERTIFICATE

This is to certify that the project report entitled "**A Simulator for Intelligent Scheduling of Data-Parallel Workloads on Heterogeneous Platforms**" submitted by **Joru Saikumar** (Roll No. 17ME33037) to Indian Institute of Technology, Kharagpur towards the fulfillment of requirements for the award of degree of Dual Degree in Mechanical Engineering is a record of bona fide work carried out by him under my supervision and guidance on the session 2020-2021.

Asst. Prof. Soumyajit Dey,

Department of Computer Science & Engineering,

Indian Institute of Technology, Kharagpur

Date: 30-4-2021

Acknowledgements

I am deeply grateful to my supervisor **Asst. Prof. Soumyajit Dey** who gave me the opportunity to work on this project. I am thankful for his aspiring guidance, invaluably constructive friendly advice during the course of the project. I am sincerely grateful to him for sharing his truthful and illuminating views on a number of issues related to the project. Also, I am very thankful to **Mr. Anirban Ghose** involved in this project who constantly motivated me to overcome all the challenges and helped me whenever I faced any issues.

1 Introduction

With the wide usage of computers in many applications, plenty of research is going on building High Performance Computing Systems. Such systems are needed to make the computation faster saving a lot of valuable time and resources. They find its preference especially when running larger applications which could simply take days and months to run on normal computers. Parallel Computing is a powerful concept in achieving High Performance Computing Systems. In parallel computing, the computation is shared across multiple devices and is made to run simultaneously. Of course, the code is divided such that the subcodes run independently of others i.e., only input and output values are shared across the subcodes and the intermediate variables do not affect other parts of the code. Parallel Computing can be defined in mathematical notations as solving an n-size problem by dividing its domain into $k \geq 2$ parts and solving them simultaneously on p processors [8]. Parallelism can be induced in two ways: Data Parallelism and Task Parallelism. In Data level parallelism, the entire data is divided across multiple domains to apply the same function on the subsets of the data whereas, in Task level parallelism, the same entire data is used in multiple functions and computed in parallel. All the p processors can be of the same type or different type. Heterogeneous systems are computing systems that contain different types of processors like Central Processing Units(CPUs), Graphics Processing Units(GPUs), Field Programmable Gate Arrays(FPGAs). Our work includes an efficient mapping technology of kernels(subcodes in OpenCL) to respective device types to optimize the makespan of an application. We made our study on scheduling OpenCL kernels on heterogeneous systems containing CPUs and GPUs. OpenCL is a widely used language for Parallel Computing applications. This report contains Introduction to OpenCL in Section 2 followed by the problem formulation in Section 3. Section 4 contains the design and contributions made in both semesters. Section 5 and Section 6 contain the description and results of the GUI and Kernel Mapping stages of our design.

2 Introduction to OpenCL

OpenCL has emerged as a standard choice for high-performance computing on heterogeneous architectures comprising of multi-core processors of CPU, GPU, FPGA, and other kinds of processors. The applications involving OpenCL range widely from graphics, CAD and 3D modeling, audio, video, and multimedia applications to office, games, cryptography, and scientific computing. The main idea of OpenCL is to break the program into kernels and execute them in parallel over different work-items[1].For example,the traditional loop(left) shown in figure 1 is broke into functions or kernels(right) which execute different instances(for each position i in the array) on different work-items in parallel[9].

Traditional loops

```
void
mul(const  int n,
    const  float *a,
    const  float *b,
           float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

OpenCL

```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id]  = a[id] * b[id];
}
// execute over n work-items
```

FIGURE 1: OpenCL kernel

An OpenCL platform consists of one Host and one or more OpenCL devices(CPU's and GPU's). The OpenCL platform model is shown in figure 2. Each OpenCL device is composed of one or more compute units also called work-groups. Each compute unit is divided into one or more processing elements also called work items. These work items are organized as a multi-dimensional grid and a collection of work items are grouped to form a workgroup. An OpenCL application consists of a single-threaded host program and a set of data-parallel tasks of the application referred to as kernel. Each kernel instance is executed by one work item of the device and the host program is executed on the host device. During the program execution, a user-specified number of work items are launched to execute in parallel.

The OpenCL memory model shown in figure 3 describes the distribution and management of the memory during the program execution. The different types of memory in the hierarchy of memory model are distinguished as follows:

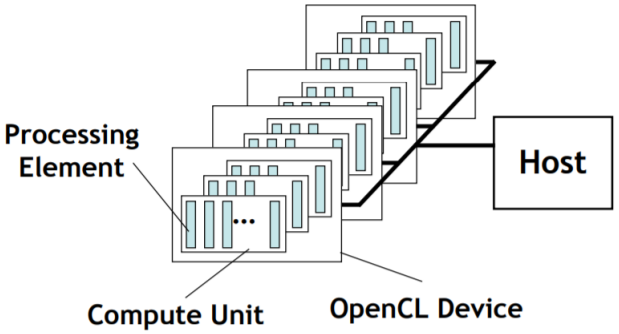


FIGURE 2: OpenCL platform model

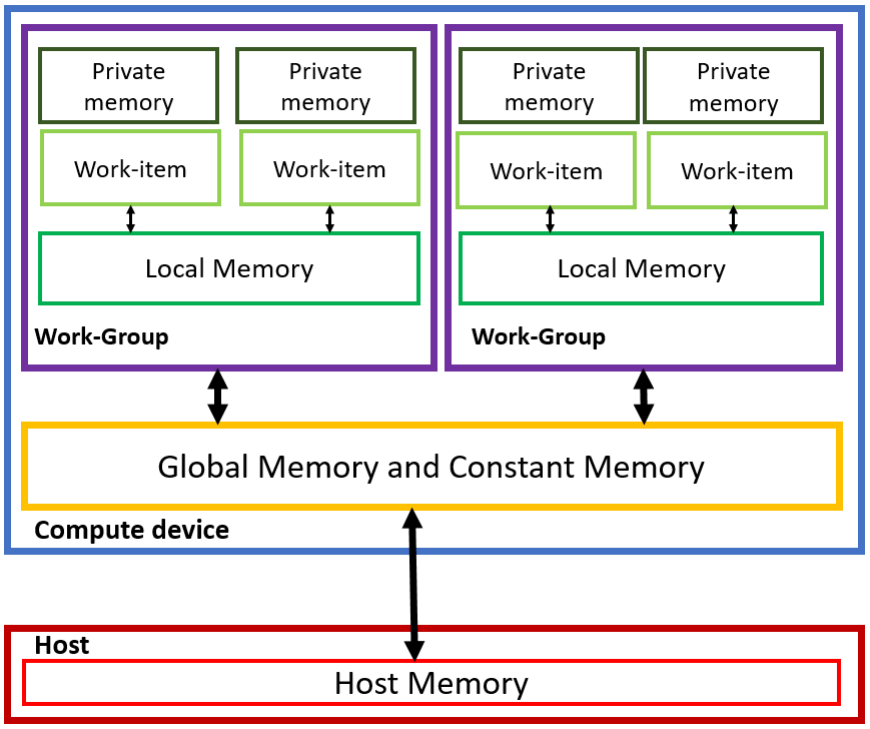


FIGURE 3: OpenCL memory model

1. Private memory: This memory is restricted to each work item where a kernel computation is done using this memory. Other work-groups or work-items of the same work-group have no access to this memory.

2. Local memory: This memory is shared within a workgroup of the device. All the work items in this workgroup have access to this memory but the other work-groups in the device don't have access to this memory. A work item can easily access private memory in comparison to local memory.

3. Global memory and constant memory: This memory is a shared memory of all the workgroups. Every workgroup and every work item has access to this memory. A workgroup can easily access its local memory compared to global memory. And a work item has the most easiness to access its private memory compared to local memory and global memory where the least and moderate easiness is in access to global and local memory respectively. Constant memory is the same as global memory in the hierarchy of the model but any work-item or kernel instance can only read it and cannot change it.

4. Host memory: A host memory is the regular memory of the host program. The OpenCL device has no access to this memory.

The host program is solely responsible for controlling the data transfer between the host and device and launching the kernels when required. It does this by maintaining the following data structures:

1. **Context:** The context is an environment within which the kernels execute and in which synchronization and memory management is defined. It includes one or more devices, device memory, and one or more command queues.

2. **Command queue:** It is a data structure used to hold all the commands involving kernel execution, synchronization, and memory operations for a device.

3. **Buffers:** Buffers are the objects that store the data to be processed by the kernels and the output data after processing by the kernels.

Hence an OpenCL host program uses context for controlling the devices and a command queue for each device for executing commands on the device in the order they are mentioned in it and buffers to share data between kernels and devices.

3 Problem Formulation

As mentioned previously the goal of our research is to find an efficient way to map the kernels of an application that is represented as DAG or may be extracted into a

DAG to the respective device types so that the makespan of the application will be optimal(least possible makespan). We have used a heterogeneous system containing 2 CPUs and 2 GPUs for scheduling the OpenCL kernels which are the nodes in the DAG. Since the processors differ in their architecture and execution, the same kernel will take a different amount of time on different processors. One kernel may take less time to run on a CPU device and termed as CPU bound kernel and some other kernel may take less time to run on a GPU device and termed as GPU bound kernel. Hence the choice of making a particular kernel to run on a particular device type may advance the performance of the system than randomly choosing the device types. Hence the problem can be formulated as follows

Given a DAG $G =< K, E >$ where K is the set of kernels $\{K_0, K_1, K_2, ...K_{n-1}\}$ and E represents the set of dependencies between kernels, output the set of the kernel to device mapping $\{D_0, D_1, D_2, ...D_{n-1}\}$ where $D_i \in \{CPU, GPU\}$.

These kernel-to-device mappings are used by the backend[2] of the simulator for scheduling the workloads(kernels) on respective device types. We have worked on two types of mapping technologies namely Topology oblivious mapping and Topology aware mapping. In the former case the device mapping for a kernel is made solely by considering the features of that particular kernel whereas in the latter case in addition to features of a kernel, its neighbors' features were also considered to make the device mapping. Practically, topology oblivious mapping is easy to implement but the latter makes more sense. This can be explained with an example: Suppose we have a DAG with a set of kernels and all the kernels are GPU bound. Topology oblivious mapping will generate a mapping that contains all kernels mapped to GPU devices. When scheduled by the backend all kernels are made to execute on GPU devices only and CPU devices are not used by the application for computation. This leads to underutilizing of resources and there is a chance of improvement. Topology aware mapping addresses this issue by using neighborhood information while predicting the mappings.

The simulator puts an extra burden on its users as they need a good knowledge of heterogeneous programming. They have to build the DAG by themselves which is a tedious task to type the set of kernels and dependencies in the input format. So we have designed a Graphical User Interface that is user-friendly with simple tools to select the kernels and draw the dependencies between them in a canvas.

4 Design and Contributions

The architecture of the simulator is shown in fig4. It contains three stages of processing and scheduling the DAG. The outermost stage is the GUI which allows the users to build the DAG. The second stage is a kernel mapping stage where each of the kernels is mapped to a device type. As mentioned it can be done in 2 ways: Topology oblivious mapping and Topology aware mapping. After this, the mapped DAG is input to the final stage ie., backend where the kernels are scheduled on respective devices using the OpenCL framework.

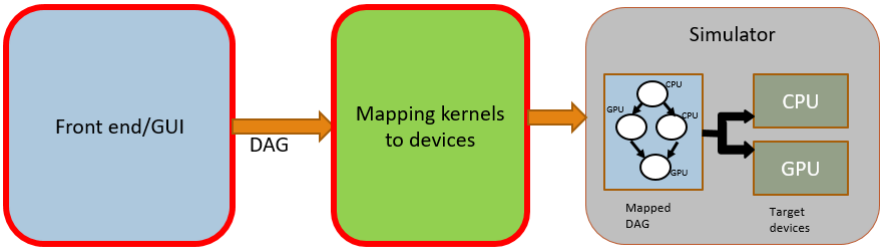


FIGURE 4: Architecture of the Simulator

The backend part of the simulator is already built and published in the name of *PyShedCL*[2]. The topology oblivious mapping was built by me in the previous semester. The GUI, topology aware mapping using Graph Convolution Neural Networks, and a Random DAG Generator for training data of the models were built in the current semester.

5 GUI

The input to the scheduling simulator is a Directed Acyclic Graph of kernels as nodes and directed edges representing the dependency between kernels and data transfer. This DAG is built using a Graphical User Interface in python using Tkinter. The first window of the GUI contains a list of kernels from a database allowing the user to select the kernels to be scheduled. The user can also select the kernels from a custom folder containing OpenCL kernels by clicking on *Choose from folder* button as shown in fig 5. Double-clicking on an item in the list shows the preview of the kernel(fig 6).

After selecting the kernels a canvas appears as shown in fig 8. It contains the selected kernels(green buttons) along with their input and output buffers(grey buttons).

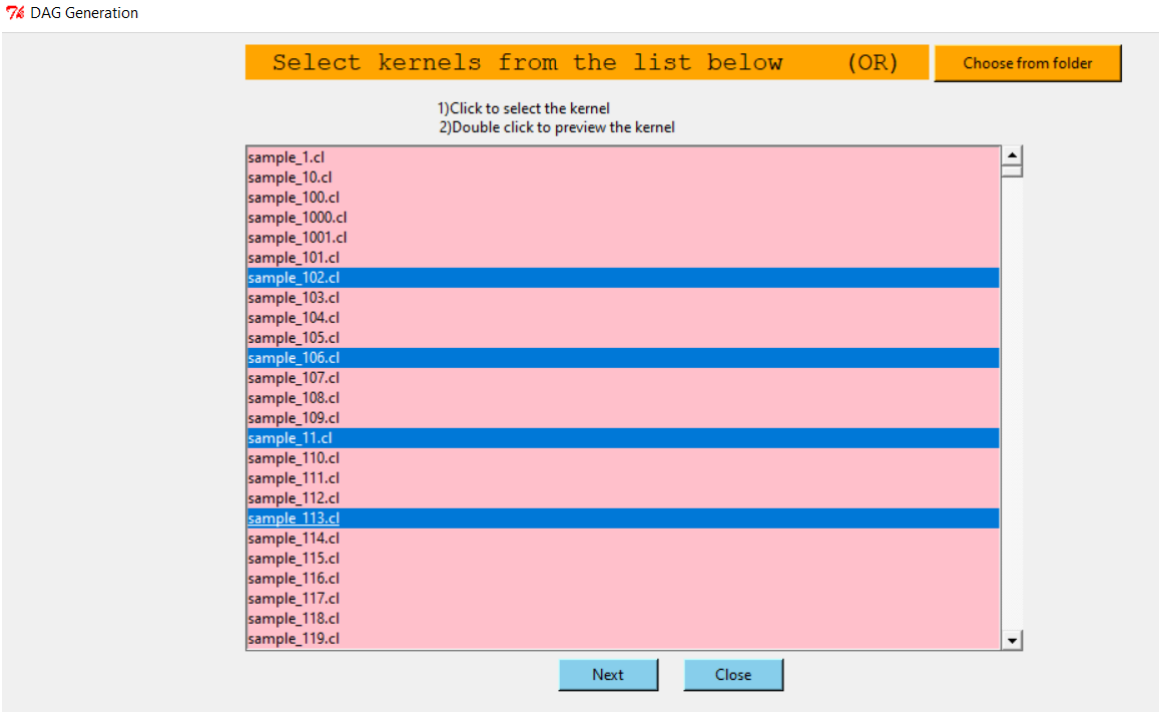


FIGURE 5: GUI First Window

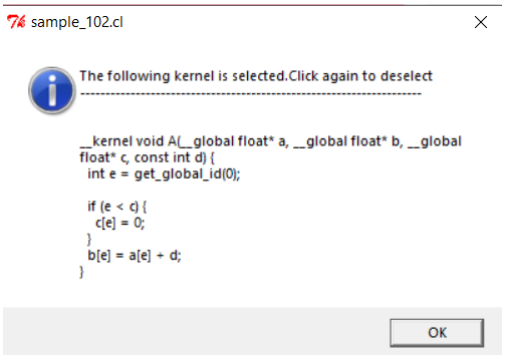


FIGURE 6: Preview of the kernel

The buffer nodes are added by parsing through the kernel's code. For example, the kernel shown in fig 7 contains three buffers a,b,c in the function call. An algorithm parses through the kernel and finds that a,b are input buffers(since they are on the right side of the expression) and c is the output buffer. Accordingly two input buffer nodes *b0,b1* corresponding to a,b and output buffer node *b2* corresponding to c, are

added in the canvas. The algorithm also generates a set of features of the kernel which is used by the mapping models discussed later. The nodes are indexed from 0 as *k0,k1,k2,...* and the corresponding kernels are shown in the right panel under *Notations* for reference. Similarly, the buffers for each kernel are also indexed from 0 as *b0,b1,b2,...* which include both input and output buffers.

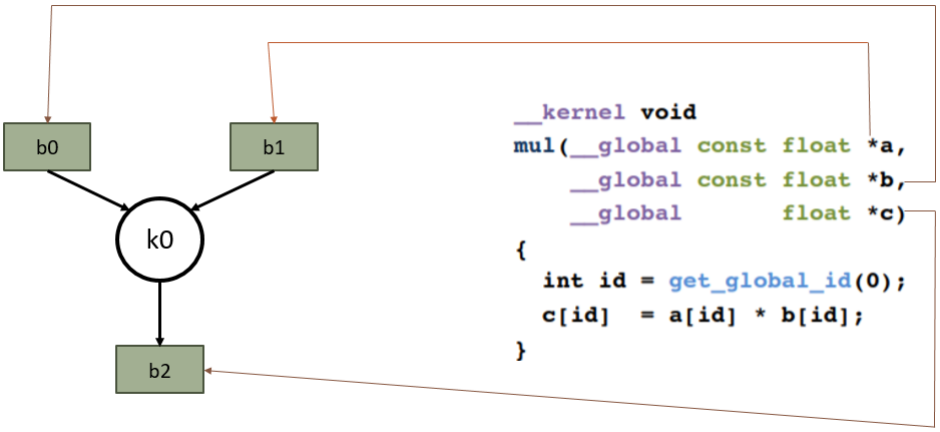


FIGURE 7: Kernel Parsing

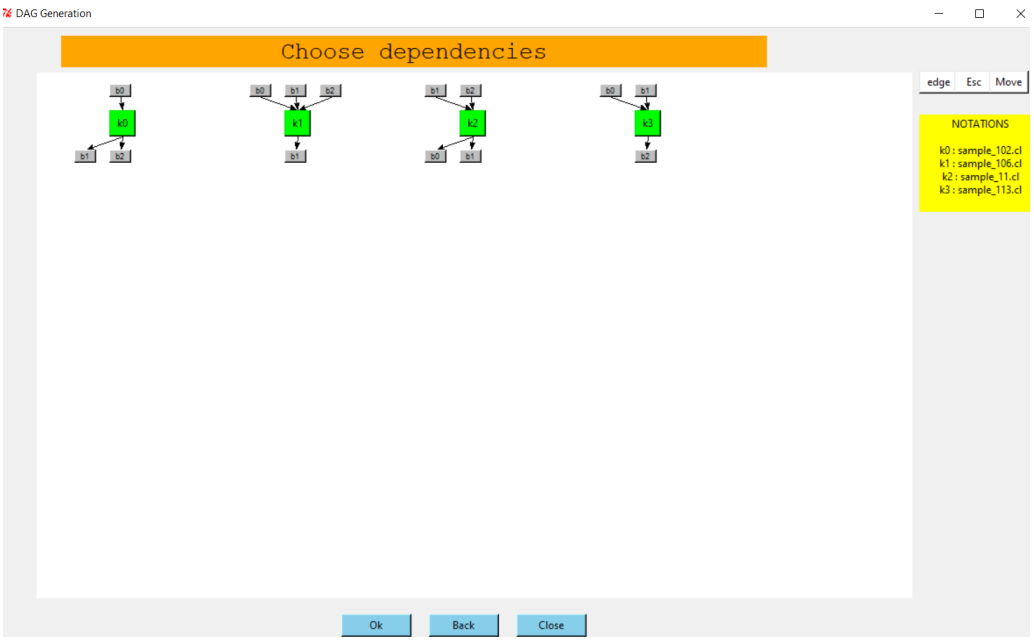


FIGURE 8: GUI Canvas

The kernel specific information like input and size of buffers, global work-size of the kernel and its work dimension can be specified for each kernel by editing the text(the

JSON info file for each kernel) that appears on the *Parameters* window when clicking on the respective kernel's button. The symbolic variables like dataset(defines input size) and partition(representing the partition of work items into workgroups) can also be specified in the entries below the text editor as shown in fig 9. Note that these parameters define the input to the application if the kernels are root nodes of the DAG and intermediate buffers if they are interior nodes. The output buffers of the leaf nodes are going to be the output of the application.

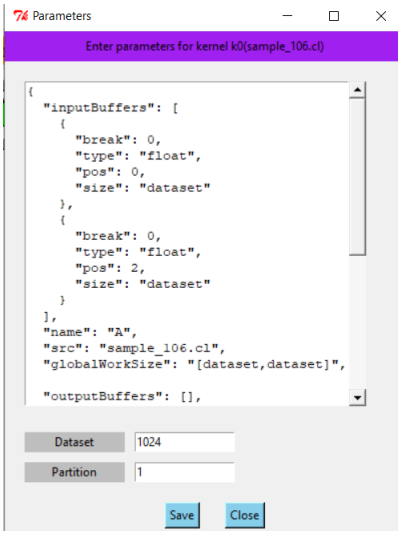


FIGURE 9: Kernel info

Now the edges between the kernels can be specified by choosing the source kernel's output buffer and dependent kernels input buffer as shown in figure 10. These edges are shown in red arrows in the figure. Also, note that the kernels can be moved within the canvas so that the edges can be built effectively without any ambiguity.

Finally, after submitting the DAG a JSON Specification file is created as shown in fig 11. The first n lines contain the kernels selected for the execution along with its input parameters and indices of the kernels. Then followed by a dotted line contains the information of the edges in the DAG in the format '*i j-k l*' describing an edge from j^{th} buffer of the i^{th} kernel to l^{th} buffer of the k^{th} kernel. This JSON Specification file is used by the backend of the simulator for creating the DAG and then scheduling the kernels effectively.

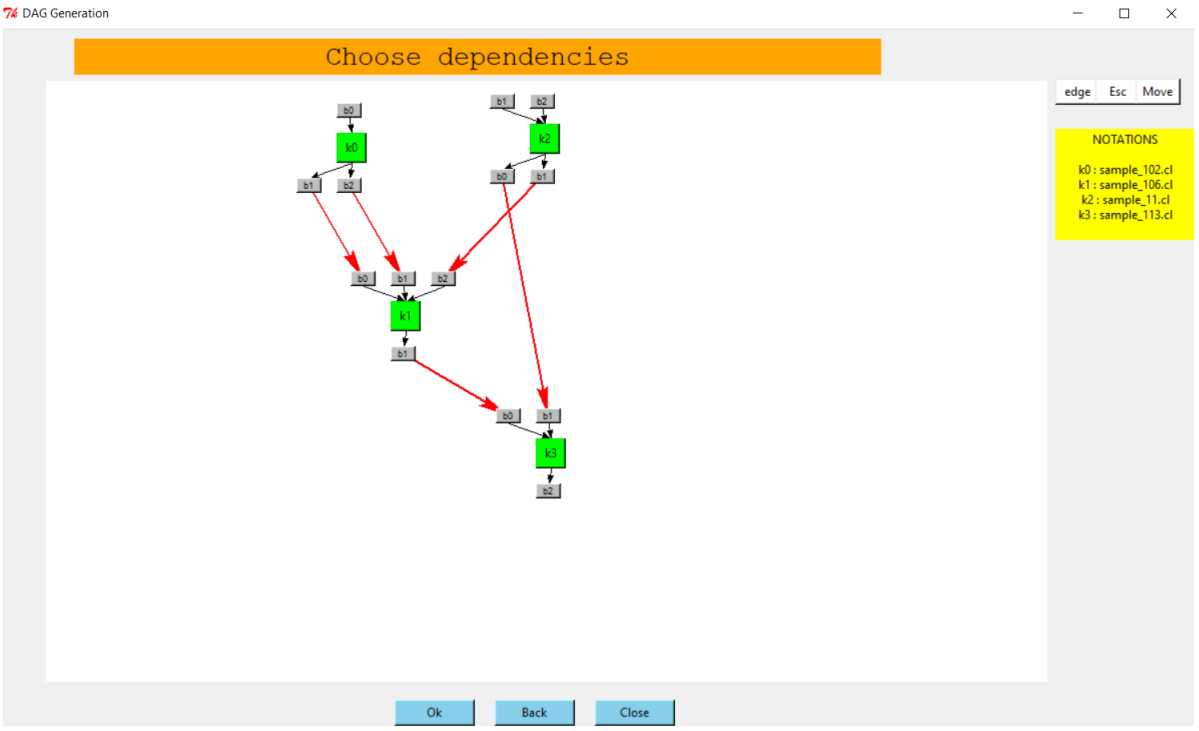


FIGURE 10: Kernel dependencies

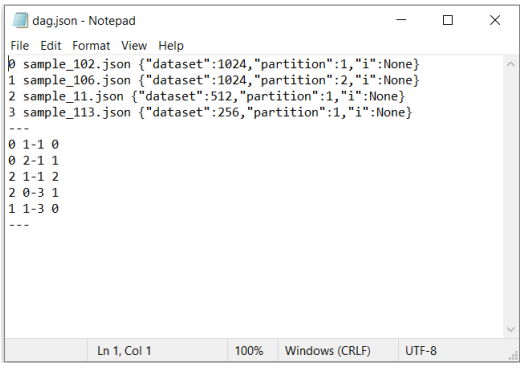


FIGURE 11: JSON Specification file

6 Kernel Mapping

6.1 Topology oblivious mapping

For each kernel(node) of the DAG application, the state information is represented as features in a feature vector. The features represent the workload of the kernel in terms of the following properties[6].

Table 1: Kernel features

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

The dataset was built using 1000 kernels. Each kernel was run with the following combinations of multiple threads on CPU and GPU devices and the run times were noted :(global work-size, local work-size)=(64,1), (128,1), (256,1), (512,1), (1028,1). The timing files were individually obtained for CPU and GPU devices. Hence in total 5463 timing files were obtained. The dataset was built by extracting a feature vector for each kernel, thread combination, and the output labels were obtained by comparing the corresponding CPU and GPU timing files. Therefore in total after removing the erroneous examples a data-set of 2685 examples is used for training the model. The size of the feature vector used is 56 which is filtered using a few statistical tools before training the models.

Various machine learning models were trained to obtain the best model for accurately predicting the kernel boundness to the devices. XGBoost(eXtreme Gradient Boosting) model[3] is found to give good results compared to other models giving an accuracy of about 93.4%. It is an implementation of gradient boosted decision trees and widely recognized as a go-to algorithm for many data scientists. Boosting is an ensemble technique where new models are added by correcting the errors made by the existing models. Models are added until no further improvement is observed. In gradient boosting new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. Gradient boosting

can be used for regression and classification. More details of evaluation metrics and results are discussed in section 6.1.1. Now, this trained model is used to predict the kernel boundness and a labeled DAG is obtained for scheduling in later stages.

6.1.1 Result

We had trained some basic to complex machine learning models for topology oblivious kernel mapping discussed in 6.1. The data extracted from timing files contains a feature vector of size 86 of which 30 features are remained the same throughout all the examples and hence are removed. The remaining 56 features were again filtered using the statistical tool *SelectKBest* of the scikit-learn package. It is implemented using correlation coefficients between each feature and the output label. The results for different K(number of features to be extracted) values are shown in Table 2 and Table 3 for logistic regression and XGBoost models respectively. The metrics used for the evaluation of models are:

1. **20% split test data**: The dataset is split such that 80% of the examples are used for training the model and the remaining 20% examples which are not used in the training are used for testing the model. The accuracy, precision, recall are measured for evaluating the model.

2. **k-fold cross-validation**: The dataset is equally split into k small sets of observations and k similar models where trained were for each model (k-1) sets are used for training the model and the remaining set is used for testing the model. The performance of the model is evaluated as the average of metrics of k models. We obtained the results by choosing k=10.

3. **Leave one out cross-validation**(LOOCV): This is the same as k-fold cross validation but with k set to the size of the dataset. Hence m(size of the dataset) models were trained by using m-1 observations for training and the remained one observation for testing the model. The model performance is evaluated as the average of individual performances of m models.

Note that the dataset was highly imbalanced(number of observations labeled to CPU device is 504 and to GPU device is 2181) and we used the SMOTE technique[4] for

balancing the dataset. It stands for Synthetic Minority Oversampling Technique and works by choosing the examples that are close in feature space, drawing lines between the examples, and generating new examples on the lines.

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
LOOCV	accuracy	59.4	58.8	58.4	58.5	58	57.8
	precision	41.3	39.2	42.3	42.1	41.7	42
	recall	41.3	39.2	42.3	42.1	41.7	42

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
LOOCV	accuracy	83.6	85.7	87.2	87.37	87.2	86.9
	precision	48.3	47.4	45.9	46.15	45.7	45.3
	recall	48.3	47.4	45.9	46.15	45.7	45.3

The results(Table 2 and Table 3) show that XGBoost has a better performance than logistic regression. The 20% test accuracy is in the order of 60% and 85% respectively. The 10-fold cross-validation accuracy of logistic regression is in the range of 78-84% where for XGBoost it is 90-96%. The LOOCV accuracies are around 58% and 83-87% respectively. Similarly the precision results from 20% split(79-86% and 89-96%), 10-fold cross-validation(56% and 76-84%) and LOOCV(39-42% and

45-48%) are observed. The recall measures for 20% split(55-57% and 77-86%),10-fold cross-validation(58% and 83-88%) and LOOCV(39-42% and 45-48%) also show that XGBoost has better performance than logistic regression. Also, the results for the selection of features show that selecting only 20 best or 30 best features has good performance than selecting all the features for training the models.

The regression models for predicting the speedup ratios are evaluated using 20% split data for testing the models using the evaluation metrics(Root mean squared error(RMSE), Mean absolute error(MAE), R^2 score). The results are shown in Table 4.

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

6.2 Topology aware mapping

6.2.1 Generation of training DAGs

A random DAG Generator is built for sampling DAGs for the training of our Topology aware Graph Convolutional Neural Network models. We have used Erdos Renyi methods for sampling the kernels and edges. Given n number of kernels, the *G(n,p)* method generates the edges with a fixed probability p where each of $\binom{n}{2}$ edges exists in the DAG with an independent probability p. The algorithm is shown below and is designed such that it avoids the occurrence of cycles in the DAG. The algorithm first samples n kernels randomly from the list of database kernels and indexes from 0 to n-1. The sampled kernels are then copied into a graph file as shown in fig 12. Now for all possible edges, a random number is generated in the range $[0, 1]$ and if it is less than the probability measure p then the edge is created(lines 3-6). Now the graph has been created and the makespans for all possible enumerations of the kernel to device mappings are generated in lines 9-14. The dag is created using *DAGCreater*

class that is already implemented and is scheduled to run on 2 CPU devices and 2 GPU devices using the List Scheduling Algorithm of *SchedulingAlgorithm* class. The makespan is then noted into the timings dictionary. Finally, the dictionary is saved in JSON file format. The *graph* files and corresponding *JSON* files are used for preprocessing data before training the models.

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 $i \in 1...n$ **do**
    for $j \in i+1...n$ **do**
        if Random() <p **then**
            write the indices(i,j) seperated by white space into the graph file.
$nCPU \leftarrow 2, mGPU \leftarrow 2$
timings ←dict()
for each of 2^n enumerations **do**
    $DC \leftarrow$ DAGCreator()
    dag ← DC.create_dag_from_file(path,device_enumeration)
    $SA \leftarrow$ SchedulingAlgorithm(dag,nCPU,mGPU)
    SA.list_scheduling()
    $timings[device\_enumeration] \leftarrow$ SA.makespan()
Copy the *timings* dictionary to a $'Json'$ file

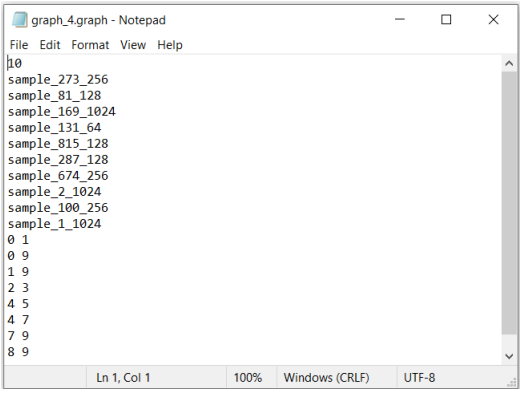


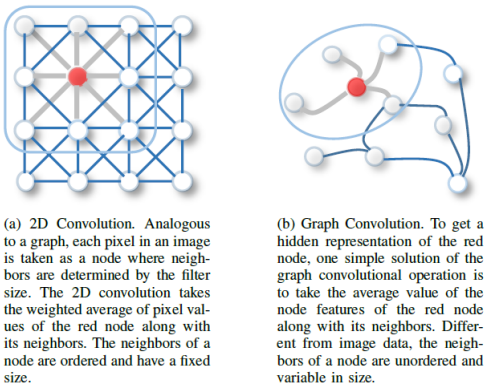
FIGURE 12: Graph file format

The probability measure p can be varied to obtain DAGs of different densities from sparse graphs to dense graphs. A higher value of p yields dense graphs and a lower

value of p yields sparse graphs. We have generated DAGs with varying probability measures for our training data.

6.2.2 Graph Convolution Neural Networks

In Topology aware mapping, the topology of the DAG is considered for making mapping decisions of its kernels. For a DAG of n kernels, we can have 2^n enumerations of kernel-device mappings out of which only one has an optimal makespan. As one approach to building a model that predicts the kernel-device mapping, we can generate a flat feature vector combining the feature vectors of all nodes or create a multi-dimensional grid and apply convolution on it. But there are disadvantages with these approaches. First, they don't apply on arbitrary graphs with a varying number of nodes because they are designed only to be applying to a fixed size of input data. They don't consider the topology in the way we would like to ie., the edges data is nowhere taken as input in these models. Also for large size DAG, the feature vector will become large which needs a complex network that is difficult to train. So we have come up with a new model known as Graph Convolution Neural Networks(GCNN) which overcomes the above disadvantages. GCNNs stand apart from traditional CNN/DNN pipelines in the sense that the input data under consideration are generated from non-Euclidean domains and are typically represented as graphs with dependency relationships. This is in contrast different from traditional real-valued feature representations which inherently are represented in some Euclidean space(fig 13). For example, CNN processes an image, by sliding a filter of weights and performing neighborhood operations such as convolution and pooling which operate on a subset of pixels of an image arranged in a 2D space. The CNN processes the image through a series of convolution and pooling operations with the objective of transforming the input image to a feature representation that would be subsequently used for classification using a feed-forward network. In a similar vein, GNNs, specifically GCNNs operate on input data by passing a filter over the graph, looking for essential vertices and edges that can help classify nodes within the graph[10].This number of vertices is arbitrary and entirely depends on the structure of DAG. But the depth of vertices is fixed and predetermined(hyperparameter).



(a) 2D Convolution. Analogous to a graph, each pixel in an image is taken as a node where neighbors are determined by the filter size. The 2D convolution takes the weighted average of pixel values of the red node along with its neighbors. The neighbors of a node are ordered and have a fixed size.

(b) Graph Convolution. To get a hidden representation of the red node, one simple solution of the graph convolutional operation is to take the average value of the node features of the red node along with its neighbors. Different from image data, the neighbors of a node are unordered and variable in size.

FIGURE 13: Comparision between 2D Conv networks and Graph Conv Networks

We have generated a dataset containing DAGs of size 10 and different densities using Algorithm 1. The GraphSAGE model which is a variant of Graph Convolution Neural Networks with the setting of inductive learning is shown in fig 14[5]. Algorithm 2 shows the forward propagation of the model. The input to the model is a DAG along with the information of its kernels in form of their feature vectors($x_v, \forall v \in V$). Number of layers(K), Nonlinear activation function σ, Aggregator functions for each of the GCN layer, and downstream model are the hyperparameters of the model. The parameters associated with aggregator function, weight matrices(W^k), and downstream model are the parameters of the model which are to be learned while training. The input feature vectors are transformed into embedding vectors through a number of GCN layers and the embedding vectors are fed into a downstream model which makes a binary classification of each kernel to a particular device type. Algorithm 2 shows how these parameters are used to generate an embedding vector transformation for nodes in the graph. Each iteration in the outer loop of the algorithm corresponds to a GCN layer.

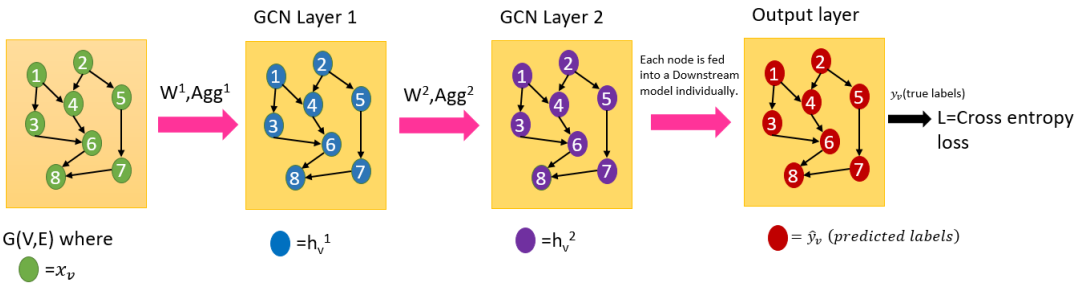


FIGURE 14: GCNN

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**
    $z_v \leftarrow D(h_v^K)$
$\hat{y_v} \leftarrow z_v$

For each layer k, h_v^k denotes the node's representation at that layer. Initially, h_v^0 is set equal to the input feature vector for each node. Now at each layer from 1 to K, the following steps are made for each node v: First the state representation of node's immediate neighbors are aggregated into a vector $h_{N(v)}^k$ using the aggregator function. Various types of aggregator functions implemented are described in section 6.2.3. This aggregation step is made on state representations of the previous layer(h_u^{k-1}) which in turn is dependent on its previous layers. GraphSAGE then concatenates the neighborhood state representation vector $h_{N(v)}^k$ and state representation of a current node in the previous layer h_v^{k-1} into a single vector and feeds into a fully connected layer with nonlinear activation function(σ) which transforms into a new embedding vector of dimensions of the current layer. Figure 15 shows the depiction these operations for a node in a layer. The dimensions of the transformation matrix W^k determine the dimensions of the state vectors at k^{th} layer. These dimensions of the state vectors at each layer are to be set as hyperparameter of the model. After transforming through all the layers, the final state representation of each node at layer K is fed individually through a downstream model which makes a binary class prediction of the node or kernel to CPU device or GPU device. The final output is a probability in $[0, 1]$, and based on the true labels of the kernels a

weighted loss function based on Binary Cross Entropy Loss is evaluated.

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

where

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

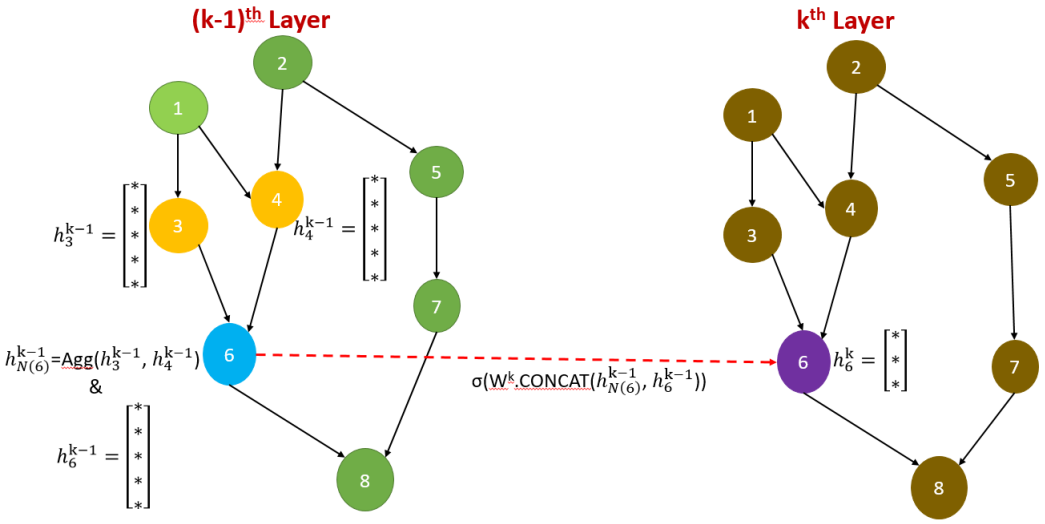


FIGURE 15: Aggregation and Transformation operations in a GCNN layer.

In the above expression, M denotes the number of DAGs in our training dataset. For each DAG the average binary cross entropy of all its nodes is evaluated. The overall loss function for the model is taken as a weighted average of losses calculated for each of the DAG present in the training set where the weights w_m are nothing but the ratio of makespan of the application with predicted kernel-to-device mapping and the optimal makespan.

$$w_m = \frac{predictedmakespan}{optimalmakespan}$$

This way the loss function allows the model to give more weightage to the training examples which are far away from its optimal makespan and less weightage to the training examples which are near to their optimal makespan and needs no further improvement. The gradients of the loss function with respect to model parameters are

calculated through backpropagation and the model is updated at every optimizer's step.

6.2.3 Aggregators

Unlike Convolution Neural Networks on images, the neighborhood nodes in a graph are not ordered and hence the aggregator function should apply on an unordered set of vectors and should be invariant of the order. Some of the aggregators used in our work are Mean Aggregator, Mean Pool Aggregator, Max Pool Aggregator, and LSTM Aggregator.

Mean Aggregator: This aggregator function is an element-wise mean operator and the simplest aggregator function. All the state vectors of immediate neighbors of a node are taken as input and the element-wise average of them is evaluated.

$$h^k_{N(v)} = MEAN(\{h^{k-1}_u, \forall u \in N(v)\})$$

Since we are performing only the average of the state vectors, this aggregator function has no parameters to be learned by the model. It is a symmetric aggregator ie., it doesn't depend on the order of neighborhood state vectors. The size of the aggregate is the same as that of the state vectors.

LSTM Aggregator: It is a more complex aggregator based on LSTM architecture. Compared to the Mean aggregator, the LSTM aggregator has the more expressive capability. It is also trainable. But since LSTM architecture aggregates the input in a sequential manner this aggregator is not symmetric. The dimension of the aggregate vector is twice that of the state vectors of the previous layer.

Pooling Aggregator: This aggregator is both symmetric and trainable. State representation of each of the neighbors is fed individually into a fully connected neural network layer. This operation is known as a pooling operation. After performing pooling operation on all the neighbor vectors aggregation is done on the outputs to generate a single aggregate vector. An element-wise mean operation can be applied and this aggregator is known as Mean Pooling Aggregator. An element-wise max operation can also be applied and that aggregator is known as Max Pooling

Aggregator.

$$h^k_{N(v)} = MAX(\{\sigma(W_{pool} h^{k-1}_u + b), \forall u \in N(v)\})$$

The dimensions of the transformation matrix W_{pool} decides the dimension of the aggregate vector.

6.2.4 Result

We have trained a GCNN architecture described above with 3 GCN layers having 16,16,16 dimensions of state vectors at each layer. Logistic Regression is used as the downstream model. The performance results of the models using different aggregator functions are shown in table 5. Also, the histograms of speedup ratios for the examples in the test set are shown in fig 16.

TABLE 5: Performance of GCNN models

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

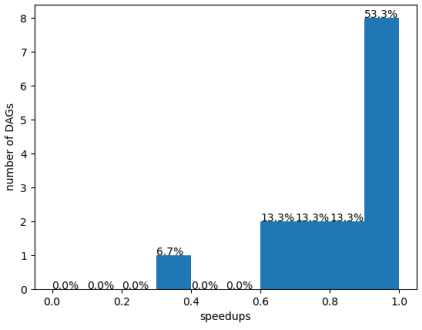


FIGURE 16: Histogram of Speedup ratios in the test set

Even though the models show good accuracy of 77.33% and an average speedup of 0.839 they are observed to be predicting only one class(GPU class) all the time. This might be because of the presence of unbalance in our database kernels. As

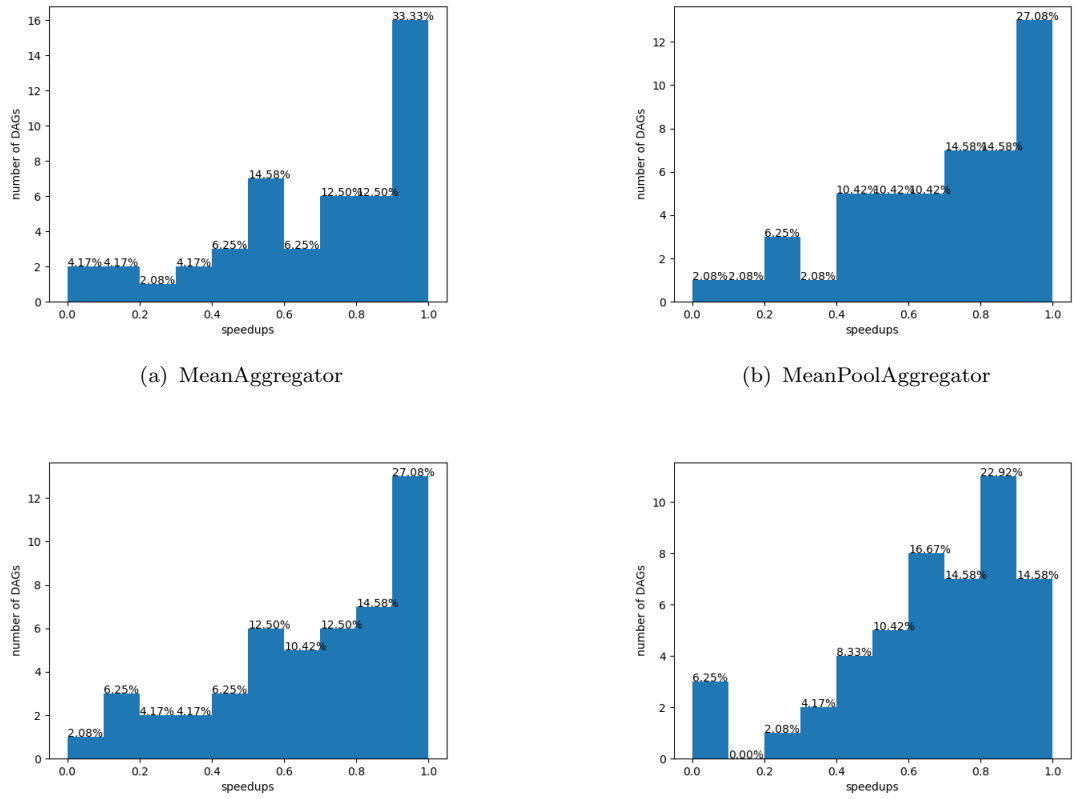
discussed in section 6.1.1 the database contains 2181 GPU bound kernels and 504 CPU bound kernels. Hence, there is a high chance that the DAG Generator selects more GPU-bound kernels than CPU-bound kernels while generating DAGs. So each of the DAG in our training set and test set contains more than 60% of GPU bound kernels and our model simply learns to predict GPU class always. To overcome this issue of unbalance in the dataset we induced a new parameter(b) representing the diversity in the algorithm of DAG Generator. b lies in [0,1] and ensures b fraction of the sampled kernels in line 1 of Algorithm 1 contains CPU bound kernels and the remaining fraction of sampled kernels contains GPU bound kernels.

We have generated a new balanced dataset of training DAGs with 10 kernels each and with varying density(p=[0.2,0.4,0.6,0.8]) and diversity(b=[0.2,0.4,0.6,0.8]). We have trained and tested GCNN models on the new training data and the results are shown in Table 6.

Table 6: Performance of GCNN models on the 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

The speedup ratios of all the examples in the test set are plotted as histograms in fig 17. We can observe most of the examples having a speedup ratio greater than 0.7. Hence we have obtained satisfactory results with GCNN based models for Topology aware mapping. Table 6 shows that MeanPool Aggregator and Mean Aggregator has the best performance in predicting the kernel-to-device mapping with 61.25% and 61.042% of accuracies respectively and average speedup ratios of 0.6957 and 0.6955 respectively. Compared to these aggregators, MaxPool Aggregator and LSTM Aggregator have low performance with less than 60% accuracy and an average speedup ratio of 0.6874 and 0.6779. Figure 17 also shows that LSTM Aggregator has poor performance compared to others.



(a) MeanAggregator

(b) MeanPoolAggregator

(c) MaxPoolAggregator

(d) LSTMAggregator

FIGURE 17: Histograms of speedup ratios of GCNN models trained on balanced data

7 Conclusion

We proposed a frontend for the simulator in our research work. We have worked on an efficient way to map kernels of a DAG to a device type so that when scheduled according to the mapped devices the makespan of the application will be optimal. We built a Graphical User Interface on top of all the stages of the simulator which allows users to easily build their own graphs by selecting the kernels from the database and drawing the dependencies between kernels in the DAG. This Graphical User Interface allows the users to avoid the tedious task of writing the input format for building the DAG. We have worked on two types of kernel-to-device mapping policies: Topology oblivious mapping and Topology aware mapping. Topology oblivious mapping is the simplest mapping technique where only state information of the current kernel is used to map the kernel to device type. In Topology aware mapping the neighbors' state

information is also considered for making mapping decisions of the current kernel. Various machine learning models were trained for the former case and the results are shown in 6.1.1. XGBoost classification model is observed to be the best choice for topology oblivious mapping. Similarly, Graph Convolution Neural Network models were trained for Topology aware mapping and the results are discussed in the section 6.2.4. With these models, we were able to achieve only up to 62% of the accuracy and average speedup ratio of 0.6957. Other variants of Graph Neural Networks can be tried and implemented for better performance as to future work to our current research. Reinforcement Learning models like Decima [7] can also be a good option to look on.

Bibliography

[1] A Gentle Introduction to OpenCL. https://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854.

[2] Vivek Kulaharia Lokesh Dokara Srijeeta Maity Anirban Ghose, Siddharth Singh and Soumyajit Dey. Pyschedcl: Leveraging concurrency inheterogeneous data-parallel systems.

[3] Jason Browniee. A Gentle Introduction to XGBoost for Applied Machine Learning. https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/.

[4] Jason Browniee. SMOTE for Imbalanced Classification with Python. https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/.

[5] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[6] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 455–466, 2014.

[7] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[8] Cristobal Navarro, Nancy Hitschfeld, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15:285–329, 09 2013.

[9] Smith Simon McIntosh. *Introduction to OpenCL*. University of Bristol, 2019.

[10] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.