Appendix A

Tools and Frameworks

The appendix presents a brief overview of the different specification formats and important commands with regards to the tools and frameworks devised for this thesis. The repository containing different codebases for the works proposed is located at https://github.com/anighose25/phd. The folder structure is explained below.

1. `dag_specification`: Scripts for GUI based DAG Specification
2. `pyschedcl`: Scripts for PySchedCL execution
3. `fgfs`: Scripts for FGFS scheduling works
4. `coarsening_code_generation`: Scripts for CNN pipeline generation
5. `rlschedsim`: Scripts for RL assisted simulation
6. `ascend`: Low Level Scheduler for Periodic OpenCL applications

We also present a list of hardware configurations and software packages used for the experiments in each of the technical contributions.

A.0.1 HW/SW Configurations

The hardware configurations, compiler versions and software packages used for each of the technical contributions is listed below.
1. **Chapter 3**: The work presented in this chapter leveraged two platforms discussed below.

- Platform I: GeForce GTX 970 GPU with DRAM 6GB and NVIDIA Driver Version 384.130 + Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz
- Platform II: Tesla K40m GPU with DRAM 12GB NVIDIA Driver Version: 470.42.01 + Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz

The corresponding compilers used for this work include

- gcc 4.8.2
- Python 2.7.12
- OpenCL 1.2 profile for CPU and OpenCL 1.1 profile for GPU

2. **Chapter 4:** The experiments for the work presented in this chapter were carried out on Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz. The compilers and interpreters used are the same as above.
3. **Chapters 5 and 6:** The target embedded platform used in both the works in Chapters 5 and 6 was the ARM Odroid XU4 platform. The specifications for the same are listed below.

- Samsung Exynos5422 Cortex™-A15 2Ghz and Cortex™-A7 Octa core CPUs
- ARM Mali-T628 MP6 GPU
- 2Gbyte LPDDR3 RAM shared between CPUs and GPUs

The compilers and interpreters used for the above work is listed below.

- gcc 7.5.2 with C++14 standard
- OpenCL 1.2 Full profile

Table A.1: ***Python 2.7.12 software packages***

Package	Version
cffi	1.11.5
configparser	3.7.3
futures	3.2.0
imbalanced-learn	0.3.3
Mako	1.0.7
matplotlib	2.0.2
mpi4py	3.0.0
networkx	1.11
numpy	1.14.5
pycparser	2.18
pygraphviz	1.3.1
pyopencl	2016.2.1
pytools	2018.5.2
scikit-learn	0.19.0
scipy	1.1.0
six	1.11.0
sklearn	0.0
torch	0.3.0.post4

The complete list of python software packages with their corresponding version numbers for the python interpreter version 2.7.12 listed in each of the above technical works are presented in Table A.1. subsectionJSON format for kernel specification Let us consider the standard vector addition kernel from the CUDA SDK below.

```
__kernel void VectorAdd(__global const float* a,
__global const float* b, __global float* c, int N){
    int iGID = get_global_id(0);
    if (iGID >= N)
        return;
    c[iGID] = a[iGID] + b[iGID];
}
```

Our proposed JSON Specification file relieves the end user from designing large complex host side programs and allow focusing more on the algorithm of the application at hand. The specification file shall provide a list of guidance parameters for the framework so that it can carry out the computation of the given kernel. For the sample kernel, it is evident from the above code, that the kernel requires two input buffers, one output buffer and one variable denoting the total size of the arrays to be processed. The kernel performs an element-wise addition of numbers in both the input buffers and stores the result in an output buffer. The corresponding specification file is given below. The guidance parameters can be classified into three primary categories.

```
{
    "globalWorkSize": "[dataset]",
    "inputBuffers": [
        {

            "pos": 0,
            "size": "dataset",
            "type": "float"
        },
        {

            "pos": 1,
            "size": "dataset",
            "type": "float"
        }
    ],
    "name": "VectorAdd",
    "outputBuffers": [
        {

            "pos": 2,
            "size": "dataset",
            "type": "float"
        }
    ],
```

```
    "src": "VectorAdd.cl",
    "varArguments": [
        {
            "pos": 3,
            "type": "int",
            "value": "dataset"
        }
    ],
    "workDimension": 1
}
```

Source Information: These parameters typically represent kernel source file information and associated problem size for the computation. This includes the name of the OpenCL kernel function (name), the name of the kernel source file (src) , the dimension of the kernel (workDimension) and the total number of workitems for each dimension (globalWorkSize).
Buffer Information: These parameters represent information for every input (inputBuffers) and output (outputBuffers) buffer for the program. It may be observed from the above specification file, each buffer is characterized by a tuple $< pos, size, type >$. The value of pos denotes the positional argument for that buffer in the actual kernel function. This is needed while setting up the kernel arguments for the OpenCL runtime system by the framework. The size and type variables denote the size (number of elements) and type of the buffer respectively. Here, the size of the buffer and the global work size for the kernel are the same and are represented by a symbolic expression called *dataset* which may be changed during runtime.
Argument Information: These parameters represent information for every variable argument (varArguments) for the OpenCL kernel. Each variable argument is characterized by a tuple $< pos, type, value >$. The value of pos like before denotes the positional argument of that variable in the actual kernel function. The varArguments field denotes the data type of the variable. The value field represents the corresponding value of the variable, which may be a constant or a symbolic expression. The variable kernel argument for the given function represents the total number of elements for each buffer and therefore is set to *dataset*.

A.0.2 DAG Specification

For prototyping SIMT DAGs, we rely on `.graph` file with a custom format for specifying kernel level information and dependencies between buffers. In this context, we have developed a GUI that would allow end users to select kernels for a given application and specify precedence constraints between the same. Figure A.1 presents a snapshot of our GUI framework where we can observe three kernels designated with identifiers k_0, k_1 and k_2. For each kernel, we have a corresponding JSON specification as discussed in the previous section which provides the relevant buffer and argument information. Subject to this, each kernel and its associated

buffers are drawn on the canvas as depicted in the right hand side of Figure A.1 by the GUI application. The edges (red in colour) can be drawn by the users between the buffers of the kernels for specifying the flow of data and subsequently the precedence constraints of the application. Given this visual representation of

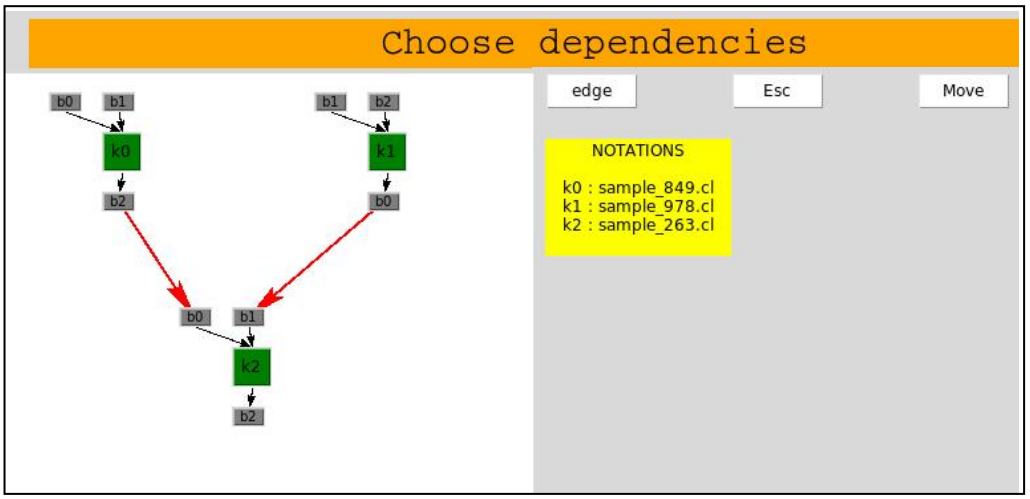

Figure A.1: ***GUI for DAG specification***

the DAG, our framework produces the following `.graph` file which is consumed by our runtime frameworks for scheduling. The first three lines of the DAG specification file presents a mapping of kernel ids with names of their corresponding JSON specification files. For each kernel, we have a list of name-value parameters denoting symbolic variables used in the kernel along with their designated values. The edge level information in the `.graph` file is captured as per the DAG illustrated in Figure A.1. Each edge is of the form $src\ b_{out} - dst\ b_{in}$ where src and dst represents the source and destination kernels, b_{out} represents the output buffer of src, and b_{in} represents the input buffer of dst. The `.graph` file is subsequently consumed by our runtime system presented in *PySchedCL* for scheduling the same.

```
# Kernel Level Information
0 sample_849.json {"dataset":1024}
1 sample_978.json {"dataset":1024}
2 sample_263.json {"dataset":1024}
# Edge Level Information
---
0 2-2 0
1 2-2 1
---
```

A.0.3 PySchedCL Execution Scripts

The base repository for PySchedCL is located in the pyschedcl folder inside the repository maintained at `https://github.com/anighose25/phd`. The folder struc-

ture for the same is discussed below.

- **scheduling:** Folder containing scripts for scheduling
- **utils:** Folder containing additional utility scripts
- **database:** Folder containing kernel source files and kernel level json files used by framework
- **dag_info:** Folder containing DAG json files used by framework
- **logs:** Folder containing timing logs and dumps for various data parallel applications
- **profiling:** Folder containing python notebooks for evaluating performance of different applications

Application Creation Scripts: For the applications presented in Chapter 3, we have scripts inside the `scheduling` folder which may be used for generating the required JSON specification files and graph files for the same. These include i) `create_transformer.py`, ii) `create_siamese_network.py` and iii) `create_resnext.py`. Each application DAG as discussed in Chapter 3 is denoted by a collection of subDAGs. The scripts are used to specify the total number of subDAGs in the application, the number of subDAGs to be mapped to the CPU device and application specific parameters such as batch size (siamese and resnext) and GEMM dimension (transformer). The corresponding commands for generating the same are discussed below.

1. **Siamese**: The command for generating siamese networks is:
   `python create_siamese_network.py batch #subdags_on_CPU #subdags`
2. **Resnext**: The command for generating ResNext networks is:
   `python create_resnext.py batch #subdags_on_CPU #subdags`
3. **Transformer**: The command for generating Transformer networks is:
   `python create_transformer.py gemm_dim #subdags_on_CPU #subdags`

Each of the above commands produce the necessary `.graph files` for the corresponding application inside `dag_info` folder.

Application Profiling Scripts: We have a set of profiling scripts present in `profiling` folder of the repository for obtaining execution times of constituent kernels of application on both CPU and GPU devices of the target platform. These are required for obtaining the ranking metrics for each application at the time of scheduling. The relevant scripts in this regard include i) `dump_siamese_profiles.py`, ii) `dump_resnext_profiling.py` and iii) `dump_execution_map.py`. The first two represent profiling scripts for obtaining execution profiles of Siamese Networks and Resnext Networks, while the last represents the script for that of Transformer

Networks. We note that the scripts in their current form obtain execution time statistics as per the application specific parameters (batch size and GEMM dimension) used in our experiments and dumps the same in the `logs` folder. This profiling information is used by our proposed scheduling schemes discussed below.

Application Scheduling Scripts The scheduling scripts of interest are kept in the `scheduling` folder of the repository. These include the coarse-grained scheduling schemes contained `eager_scheduler.py`, `dheft_scheduler.py` and the fine-grained scheduling scheme `setup_cq_and_deploy.py` The command line parameters for each of the schemes take the form illustrated below.

```
python scheduling/algorithm.py
    \ -f ./dag_info/dag_folder/
    \ -ng 1 -nc 1 -thd
    \ -ef logs/profile_stats.json
    \ -fdp logs/schedule_profile.json
```

The description of the parameters used are enumerated below.

- -f : name of input DAG folder
- -ng: number of target GPU devices
- -nc: number of target CPU devices
- -thd: spawn separate threads for setting up independent DAG instances
- -ef: name of file containing kernel level execution timings
- -fdp: output file containing timing statistics of schedule

Experiment Scripts: The key scripts used for our experiments is in the folder `scripts`. For obtaining the results of the fine-grained and coarse-grained schemes of each candidate DAG, the key scripts are as follows.

- run_all_resnext_coarse.sh
- run_all_siamese_coarse.sh
- run_all_transformer_coarse.sh
- run_all_resnext_simulate.sh
- run_all_siamese_simulate.sh
- run_all_transformer_simulate.sh

For obtaining the Tabular results and plots in the Experimental Results section of Chapter 3, refer to the python notebook `TC_revised_visualization.ipynb` in the same folder.

A.0.4 Feature Guided Frontier Scheduling

The base repository for FGFS and related experiments are located in fgfs folder of the codebase. The key components for our simulation framework are present in `simulate.py`. These include the following.

1. **DAG Creation**: We have created a class called `DAGCreator` which supports creating DAGs using text based specifications that capture node and edge information similar to that used in `PySchedCL`. The class also has support for dumping graph visualizations for DAGs.

2. **Core Classes**: The core classes include `SimTask`, `SimTaskComponent` and `SimTaskDAG` which are internally maintained while setting up application DAGs. While `SimTask` and `SimTaskDAG` maintains static objects for tasks and DAGs respectively, `SimTaskComponent` is used to maintain clusters of tasks mapped to a particular device at the time of scheduling.

3. **ML Training Module**: The `CLTrainer` class supports training of classifier models for our proposed Feature Guided Frontier Scheduling algorithms. This includes interfaces for training, SMOTE for feature imbalance and cross-validation based testing.

4. **Scheduling Engine:** The `ScheduleEngine` class maintains interfaces for resource level queues, task level frontiers and scheduling algorithms. The class also supports experimentation for online scheduling using arrival patterns based on probability distributions.

For running our proposed linear clustering algorithm on an application DAG, one can use the scripts `execute_ml_experiment.py` and `run_lc_algorithm.py` as illustrated in the code snippet below.

```
python execute_ml_experiment.py global_map.txt <graph>
python run_lc_algorithm.py global_map.txt <graph>
```

The `global_map.txt` file contains information pertaining to kernels used for our experiments. The graphs used are located in the `GraphsNew` folder of the codebase.

Experiment Scripts: A set of scripts used for generating the results of Chapter 4 are included in the `scripts` subfolder inside the `fgfs` folder of the codebase. These include the following.

- *Offline Experiments*: `distributed_execute_experiments_list_new_ml.py` for list and `distributed_execute_experiments_new_ml.py` for cluster based scheduling of all DAGs.

- *Online Experiments*: `distributed_online_list_experiments_mm_correct.py` for list and `distributed_online_lc_experiments_mm_correct.py` for cluster based scheduling for all mixture sets of DAGs considered.

- *Plots*: `plot_offline_statistics.py to get bar plots categorized by various DAG parameters and process_mixture_model_experiments.py to get line plots for online experiments.`

A.0.5 RL Training Setup

The primary repositories of interest involving setups for our proposed RL assisted scheduling works are contained in `coarsening_fusion_code_generation)`, `ascend` and `rlschedsim` folders of our main repository. For setting up the training process, we require collecting profiling data, setting up configuration parameters for periodicity, neural network architectures and specifying training hyperparameters. We elaborate on this below.

Code Generation: We have implemented an OpenCL based code generator that synthesizes optimized kernel variants for a set of Convolution Neural Network architectures. The codebase for the same is located in the folder `coarsening_code_generation`. Our first step involves generating the kernel source and json files required for profiling. This is done by the following code snippet.

```
python generate_coarsened_code.py <cnn configuration> \
<height> <width>
```

The arguments to the above function require name of the `cnn configuration`, `height` and `width` of the images to be processed by the CNN pipeline. The user can specify the `cnn configuration` to be used in the `generate_configuration_for_network` function inside `cnn.py` source file located in the `code` subdirectory. The script produces the required json files and .cl files inside the `CNN` folder.

Kernel Level Profiling: For profiling the kernels, we leverage the low level scheduler optimized for the ARM odroid platform. The corresponding codebase is located in the `ascend` folder of the main repository. The user needs to copy the json and source files created in the previous step to the `info` and `kernels` folders inside `ascend`. The command for obtaining the profiled times is illustrated below.

```
./build/scheduling/execute_kernel/execute_kernel \
./database/info/<json_file> \
device timing/<exprofile.timing> <mode>
```

In the above snippet, the first argument is the json file for the kernel to be profiled, `device` can be `cpu` or `gpu`. The third argument represents the timing file where the execution times are dumped. The fourth argument `mode` can take values `normal` or `interference`, where in `normal` mode, only the kernel is executed on the device. In `interference` mode, a microkernel benchmark is executed on the other device in addition to the kernel to increase resource contention during the profiling process. The user requires to run this script for all the json files created for the CNN configuration in the previous step.

The timing files need to be copied into the `profile` subdirectory inside `coarsening_code_generation` folder. The following code snippet is used to generate a relevant dag.graph file for the low-level scheduling framework.

```
python parameterized_fusion_profiling.py \
<cnn configuration> <height> <width>
```

The above snippet produces the required dag.graph file inside the `dags` subdirectory inside `code` folder.

Task Component Profiling: In addition to kernel level execution profiles, we require timing statistics of different task components for each CNN pipeline. For obtaining this, we first require task component configurations comprising node ids of each task component. This is obtained using the following code snippet available in the `code` subdirectory of `coarsened_code_generation` folder.

```
python generate_fused_configurations.py <cnn>
```

This produces a text file containing all the task component configurations of the given network `cnn`.

In the `database/dags` subfolder inside `ascend`, we need to create a folder with the name of the CNN network. The `dag.graph` file produced in the previous step needs to be copied into this new folder. Additionally, a folder called `output` needs to be created inside this folder which would contain all the required json files for the kernels in the CNN pipeline. Once this is setup, the following code snipped needs to be executed to obtain the task component level profiling information.

```
./build/tests/profile_task_components/profile_tc \
<cnn> <device> \
./scripts/fused_configurations.txt \
./profiling/<exprofile.timing> <mode>
```

The first argument `cnn` is the name of the CNN pipeline and the second argument `device` is the name of the device (`cpu` or `gpu`). The third argument is the name of the fused configurations file produced in the previous step and the fourth argument is the name of profiling dump file which will contain all the required execution time statistics of task components in the CNN pipeline.

Once the kernel level and task component profiles have been obtained, they need to be copied inside the `profile` subdirectory inside the `coarsening_code_gen--eration` folder. The following script will produce the necessary `.graph` file required for our simulation framework.

```
python fusion_overhead_profiling.py \
<cnn configuration> <height> <width>
```

Simulation Setup: The simulation framework for our RL assisted scheduling works is located in `rlschedsim` inside our codebase. The key folders are.

1) ADAS_Graphs: The `.graph` files produced in the previous steps are kept inside this folder.

2) `Period_Configurations` This folder contains the required period configurations used in our experiments. A sample file is illustrated below.

```
yololite_32.graph=[60,90,120]
edlenet_32.graph=[100,150,250]
yololite_32.graph=[60,90,120]
edlenet_32.graph=[100,150,250]
```

Each line in the above file represents the period configurations chosen for a particular DAG. The name of the `.graph` files correspond to those files kept in the `ADAS_Graphs` folder. The periods chosen for each DAG are represented in a comma separated list.
3) Configurations : This folder contains the configuration files for our neural network used at the time of training the RL agent. A sample file is illustrated below. The `layer` argument represents the architecture configuration for the neural network and contains a list of hidden dimensions for each layer. The `num_states` and `num_actions` represent the number of states and actions respectively used at the time of training.

```
layer=[('L', 18), ('L', 24)]
num_states=10
num_actions=24
replay_size=30000
BATCH_SIZE=16
GAMMA=1
EPS_START=0.9
EPS_END=0.05
EPS_DECAY=100
```

The `replay_size` parameter represents the size of the replay buffer. The `BATCH_SIZE` represents the batch size containing state action transition tuples used for training the network. The value of `GAMMA` represents the discounting factor used in RL. The last three parameters represents how the probability of randomly selecting actions changes over time as a result of using the ϵ-greedy policy. In the above snippet, the value of ϵ starts with 0.9, decaying with a factor of 100 after every episode until reaching a value of 0.05.
Training and Testing Scripts: The training script provides mechanisms for learning DQNs using Experience Replay and is contained in `train_dqn_agent_all.py` file and can be used as follows.

```
python train_dqn_agent_all.py
  -p PERIOD -c CONFIG -m
  -r REWARDS -s STATS -nr NUMRUNS
  -ne NUMEPOCHS -mo MODE -i INIT_TYPE
  -a ACTIVATION -l LOSS -lr RATE
  -bn BATCHNORM
```

The description of the arguments used are discussed below.

- PERIOD: Name of period configuration file containing period values of constituent DAGs in input task set.
- CONFIG: Name of configuration file containing parameters of neural network as discussed above
- MODEL: Model file name for dumping neural network weights
- REWARDS: Name of file where rewards obtained for each episode over time are dumped.
- STATS: Name of filename containing scheduling level statistics in terms of average lateness, tardiness, deadlines missed per episode.
- NUMRUNS: Number of runs to be repeated for each period configuration
- NUMEPOCHS: Number of epochs for entire RL training
- MODE: Type of RL Agent which can assume a total of 8 values. Descriptions for the same are discussed below.

  1 - Prioritized Replay = No Double Q Learning = No Dueling = No

  2 - Prioritized Replay = Yes Double Q Learning = No Dueling = No

  3 - Prioritized Replay = No Double Q Learning = Yes Dueling = No

  4 - Prioritized Replay = Yes Double Q Learning = Yes Dueling = No

  5 - Prioritized Replay = No Double Q Learning = No Dueling = Yes

  6 - Prioritized Replay = Yes Double Q Learning = No Dueling = Yes

  7 - Prioritized Replay = No Double Q Learning = Yes Dueling = Yes

  8 - Prioritized Replay = Yes Double Q Learning = Yes Dueling = Yes
- INIT_TYPE: Type of Initialisation for neural network weights, assuming the following values.

  1 - Xavier Initialisation

  2 - Kaiming Initialisation
- ACTIVATION: Type of Activation Function

  1 - ReLU

  2 - Sigmoid
- LOSS: Type of loss function

  1 - Huber

  2 - Cross Entropy
- RATE: Learning Rate for training

- BATCH_NORM: Use Batch Normalisation when set to 1

The training script dumps the model file, rewards file and statistics file in the `Results` subdirectory inside `rlschedsim` folder. The following testing script can be used to understand the efficacy of training results.

```
python test_dqn_agent_all.py -p PERIOD -c CONFIG
-m MODEL -t TRACE_F -s STATS
```

The arguments supplied include.

- PERIOD: Name of period configuration file. (Can be different from training)
- CONFIG: Name of model configuration file used at the time of training.
- MODEL: Name of Model file where neural network weights were dumped at the time of training.
- TRACE_F: Name of Trace Folder where task device mapping decisions for each period configuration in PERIOD are dumped.
- STATS: Name of statistics file containing schedule level statistics (deadlines missed, tardiness etc.) for each period configuration in PERIOD.

Experiment Scripts: The scripts used to obtain the reward plots (`plot_static_rewards_pdf.py` and percentage of deadlines missed plots `plot_statistics_mod.py` are present in the `scripts` subfolder inside the `rlschedsim` folder of the codebase. For the experiments carried out with regards to Chapter 6, a list of notebooks is available in the `notebooks` subfolder.