Design of Simulator

2020/7/2

# System Overview:

DAG parser

Strategy Config

Raw DAG

Extended DAG 2

Device Assign

Module

Device Assign Config

Simulation Execution module

Adapter

Profiling DB

Execution Time Estimator

Parallel Strategy

Module

Execution DAG: The DAG output by Device Assign Module, contains all execution information, including node and device info. This work is done by PlanGenerator.

Execution Time Estimator module: This module will estimate the execution time of each node in the Execution DAG according to the offline profiling database, then write the execution time into Execution DAG.

Adapter module: This module parse Execution DAG file to Simulation DAG

Simulation DAG: The node in execution DAG is saved in extended Node class. This class only contains attributes used for simulator and is independent from framework.

Simulator Execution module: The simulator will read the Simulation DAG and simulate the execution. It will only use attributes belong to basic Node class. This will make it independent from specific framework.

# **Simulator Design Document**

## **Simulator Classes Document**

### [**simulator.Simulator**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\simulator.py)

Provide an instance to simulate the computation graph.

class Simulator()

#### Remarks

This class provides methods to simulate the computational time cost as well as the communication time cost of a given computation graph (DAG). The inputs of this class are typically generated by the PlanAdapter.

#### Constructor

def \_\_init\_\_(self, nodemetadata\_list, device\_info)

**Parameters**

1. nodemetadata\_list: a list of namedtuple, storing nodemetadata
2. device\_info: a list of tuple (device\_type, spec\_list) containing device info, or a list of class Device, storing all Device objects

#### Private Properties

1. self.\_\_nodes\_metadata: list of NodeMetadata. All nodes in the graph.
2. self.\_\_nodes: list. All nodes that are initialized from nodemetadata.
3. self.\_\_devices: dict. {device\_name: device\_obj}
4. self.\_\_execution\_enqueue\_time: list of tuple (node\_index, node enqueue time). Store the enqueue time of each node.
5. self.\_\_execution\_dequeue\_time: list of tuple (node\_index, node dequeue time). Store the dequeue time of each node.

#### Public Methods

1. def reset(self): Reset the Simulator, reset simulation time to 0, reset all nodes in DAG
2. def run(self): Run simulation, return total\_execution\_time: float, node\_enqueue\_time: list of (node\_index, node enqueue time), node\_dequeue\_time: list of (node\_index, node dequeue time)

**Returns**

* 1. float, the total execution time
  2. list of (node\_index, node enqueue time), the node enqueue time
  3. list of (node\_index, node dequeue time), the node dequeue time

1. get\_nodes(self): Return all nodes in the Simulator

#### Private Methods

1. def \_\_start\_all\_ready\_nodes(self): Start all nodes that are in ready status, this function will be called only once at the beginning of a simulation

**Returns**

* 1. float, the earliest complete time
  2. Device, the earliest device

1. def \_\_start\_node(self, exec\_node): Start to execute a node. Enqueue the node into device. The node will be marked as 'executing'.

**Parameters**

* 1. exec\_node: The Node obj to be executed

1. def \_\_find\_earliest\_complete\_device(self): Find the device who has a node with the earliest complete time. Return (earliest\_complete\_time, earliest\_device)
2. def \_\_next\_step(self): Wait until any executing node is done. Get the timestamp. Mark the node as 'done'. Then dequeue it from device. Update all successor nodes' dependency counter. If a successor node is ready, start it. Return current timestamp.

**Returns**

float, the earliest complete time

1. def \_\_init\_device(self, device\_info): Init self.\_\_devices via device\_info. Raise ValueError if the device\_info is invalid.

**Parameters**

* 1. device\_info: a list of tuple (device\_type, spec\_list) containing device info, or a list of class Device, storing all Device objects

### [**simulator.NodeMetadata**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\node.py)

Provide an instance to store essential data of a node

class NodeMetadata()

#### Remarks

This class contains all static essential data to initialize a class Node.

#### Constructor

def \_\_init\_\_(self, index=0,

op='',

name='',

device\_name='',

execution\_time=0.0,

output\_tensors=[],

input\_ids=[],

dependency\_ids=[],

successor\_ids=[]

)

**Parameters**

| **Name** | **DataType** | **Description** |
| --- | --- | --- |
| index | int | ID of node |
| op | string | Operation |
| name | string | node name |
| device\_name | string | Name of device to run this node |
| execution\_time | float | Estimated execution time, in microsecond |
| output\_tensors | List(Tensor) | Store the output Tensors’ information |
| input\_ids | List(int) | ID of all input data nodes |
| dependency\_ids | List(int) | ID of all nodes that only have control dependency |
| successor\_ids | List(int) | ID of all nodes depends on this node |

#### Public Properties

1. self.index
2. self.op
3. self.name
4. self.device\_name
5. self.execution\_time
6. self.output\_tensors
7. self.input\_ids
8. self.dependency\_ids
9. self.successor\_ids

#### Public Methods

1. def to\_dict(self): return the dict representation of obj self.

**Returns**

a dict, representing self obj

1. def assign\_from\_dict(self, input\_dict): assign self.key to input\_dict[key]

### [**simulator.NodeStatus**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\node.py)

An Enum class, providing the status of node. Inherit from Enum

1. waiting = 0. Not started.
2. executing = 1. Started, not finished. Only for asynchronized node.
3. done = 2. Finished.

### [**simulator.NodeException**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\node.py)

An Exception exclusive for class Node. Inherit from Exception.

#### Constructor

def \_\_init\_\_(self, error\_info)

**Parameters**

1. error\_info: string, the description string of this exception

#### Public Properties

1. self.error\_info: string, the description string of this exception

#### Public Methods

1. def print\_error\_info(self): print self.error\_info

### [**simulator.Node**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\node.py)

Provide the run-time Node in simulator.

class Node()

#### Remarks

This class contains all static and dynamic information of a Node. It will store the node's execution status, the remain\_dependency\_cnt and etc.

#### Constructor

def \_\_init\_\_(self, metadata, device)

**Parameters**

1. metadata: class Nodemetadata.
2. device: class Device, the device obj on which this node will be executed.

#### Private Properties

1. self.\_\_metadata: class Nodemetadata, the metadata of the node.
2. self.\_\_status: the Enum of NodeStatus, The status of a node.
3. self.\_\_remain\_dependency\_cnt: int, the number of remaining dependencies
4. self.\_\_device, class Device, the device runtime object that this node is running on
5. self.\_\_successor\_nodes, list of Node, Node of successor nodes depends on this node

#### Public Methods

1. def reset(self): reset self.\_\_remain\_dependency\_cnt and self.\_\_status based on self.\_\_metadata
2. def is\_ready(self): return whether this Node is ready to be executed

**Returns**

bool, whether this Node is ready to be executed

1. def get\_index(self): return node index

**Returns**

int, the index number

1. def get\_op(self): return op string

**Returns**

string, name of 'op'

1. def get\_name(self): return node's name

**Returns**

string, node's name

1. def get\_device\_name(self): return the device name of this node

**Returns**

string, the device name

1. def get\_execution\_time(self): return node's execution time

**Returns**

float, execution time

1. def set\_execution\_time(self, execution\_time): set the self.\_\_metadata.execution\_time of current node.

**Parameters**

* 1. execution\_time: float, the execution time of the Node

1. def get\_tensors(self): return node's tensor list

**Returns**

list of Tensor, the node's tensor list

1. def get\_status(self): return Node status

**Returns**

NodeStatus, the node's current status

1. def get\_remain\_dependency\_cnt(self): return the number of remaining dependency Nodes

**Returns**

int, the number of remaining dependency

1. def decrease\_remain\_dependency\_cnt(self, cnt): decrease the number of remaining dependency Nodes by cnt.

**Parameters**

* 1. cnt: int

1. def renew\_successor\_nodes(self, node\_list): use successor index, find and add successor Nodes from node\_list to current Node's self.\_\_successor\_nodes

**Parameters**

* 1. node\_list: dict. {index: Node\_obj}

1. def get\_successor\_nodes(self): return the successor nodes list

**Returns**

list of Node, the successor nodes list

1. def execute(self, time\_now): execute the Node if ready. Enqueue this Node to the device with time\_now. Change the NodeStatus to executing.

**Parameters**

* 1. time\_now: float. timestamp of simulation

1. def finish(self): dequeue this Node from device, set status to done

### [**simulator.TensorException**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\tensor.py)

An Exception exclusive for class Tensor. Inherit from Exception.

#### Constructor

def \_\_init\_\_(self, error\_info)

**Parameters**

1. error\_info: string, the description string of this exception

#### Public Properties

1. self.error\_info: string, the description string of this exception

#### Public Methods

1. def print\_error\_info(self): print self.error\_info

### [**simulator.Tensor**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\tensor.py)

Provide the essential data for Node

class Tensor()

#### Remarks

This class will be used to initialize NodeMetadata.

#### Constructor

def \_\_init\_\_(self,

tensor\_type='int8',

tensor\_size=0):

**Parameters**

1. tensor\_type: string, the type of current tensor.
2. tensor\_size: int, element number.

#### Public Properties

1. Tensor.valid\_type: dict, supported tensor typeAIA
2. self.type: string, must be a valid type in Tensor.valid\_type
3. self.size: int, total number of elements. The total\_bytes = size \* sizeof(type)

#### Public Methods

1. def get\_bytes\_size(self): get total bytes of current Tensor

**Returns**

int, tensor size in byte

### [**simulator.Device**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\device.py)

Provide the interface for all devices that will be used in Simulator.

class Device()

#### Remarks

This class acts as an interface. All devices that are used in Simulator should be inherited from this class, add they should implement is\_idle(self), get\_next\_node(self), enqueue\_node(self, node, time\_now) and dequeue\_node(self) methods

#### Constructor

def \_\_init\_\_(self, name)

**Parameters**

1. name: string, the name of current Device

#### Private Properties

1. self.\_\_name: string, name of device, every child class should have a name.
2. self.\_\_next\_finish\_time: float, the finish time of current node. The child classes could ignore this property.

#### Public Methods

1. def name(self): get the device name

**Returns**

string, device' name

1. def is\_idle(self): check whether the device is idle, every child class should override this method.

**Returns**

bool, whether or not the device is idle

1. def get\_next\_node(self): get the first completed node and its complete time, every child class should override this method.

**Returns**

* 1. Node, the first completed Node
  2. float, the complete time of node

1. def enqueue\_node(self, node, time\_now): enqueue a new node into this device on time\_now, every child class should override this method.

**Parameters**

* 1. time\_now: float, the enqueue simulation time

1. def dequeue\_node(self): dequeue the first completed node from the device, every child class should override this method. Note that **DO NOT** modify the attribute of the node, just modify info of device.

### [**simulator.FIFODevice**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\fifo_device.py)

Provide a device in which nodes are scheduled using FIFO. Inherited from Device

class FIFODevice(Device)

#### Remarks

This FIFO device will execute the first enqueued node.

#### Constructor

def \_\_init\_\_(self, name)

**Parameters**

1. name: string, the name of current Device

#### Private Properties

1. self.\_\_node\_queue: list, the reference of enqueued nodes.
2. self.\_\_queue\_head: int, the head pointer of node queue

#### Public Methods

1. def is\_idle(self): check whether there is node being executed in the device

**Returns**

bool, whether or not the device is idle

1. def get\_next\_node(self): get the first completed node and its finish time

**Returns**

* 1. Node, the first completed Node
  2. float, the complete time of node

1. def enqueue\_node(self, node, time\_now): enqueue a new node into this device on time\_now.

**Parameters**

* 1. time\_now: float, the enqueue simulation time

1. def dequeue\_node(self): dequeue the first completed node from the device. If still has node in queue, reset head end time

### [**simulator.ComputationDevice**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\computation_device.py)

Provide the base class for all computational devices. Inherited from FIFODevice.

class ComputationDevice(FIFODevice)

#### Remarks

This class is the base class of CPU, GPU and other computational devices. This class will have a performance property to measure its ability of computation.

#### Constructor

def \_\_init\_\_(self, name, performance='0bps')

**Parameters**

1. name: string, the name of current Device
2. performance: string, computational performance. e.g. '16Gibps'

#### Private Properties

1. self.\_\_performance: float, the performance in bps

#### Public Methods

1. def get\_performance(self): Return the performance in bps

**Returns**

* 1. float, the performance in bps

### [**simulator.CPU**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\computation_device.py)

Provide a class representing a CPU device. Inherited from ComputationDevice.

class CPU(ComputationDevice)

#### Constructor

def \_\_init\_\_(self, name, performance='0bps')

**Parameters**

1. name: string, the name of current CPU
2. performance: string, CPU's computational performance. e.g. '16Gibps'

### [**simulator.GPU**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\computation_device.py)

Provide a class representing a GPU device. Inherited from ComputationDevice.

class GPU(ComputationDevice)

#### Constructor

def \_\_init\_\_(self, name, performance='0bps')

**Parameters**

1. name: string, the name of current GPU
2. performance: string, GPU's computational performance. e.g. '16Gibps'

### [**simulator.DeviceFactory**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\device_factory.py)

Provide a class that can generate Device objects.

class DeviceFactory()

#### Remarks

This class is used as a object factory to generate other Device objects

#### Constructor

def \_\_init\_\_(self)

#### Public Property

1. DeviceFactory.valid\_device\_type: dict, static variable, storing valid device\_type string

#### Public Methods

1. def generate\_device(self, device\_type, \*spec\_list): Return a (device\_type) class obj based on spec\_list

**Parameters**

* 1. device\_type: string, denoting the type of this device.
  2. spec\_list: list, the parameters to initialize the class.

**Returns**

Device, the device object generated

### [**simulator.network\_simulator.Flow**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\network_simulator\flow.py)

Provide the Flow representation of nodes.

class Flow()

#### Remarks

Communication Nodes (Send/Recv) will be converted to a Flow in this flow based NetworkSimulator. Each Flow maintain its capacity (aka. bandwidth), and their transfer status, and it may occupy several Links.

#### Constructor

def \_\_init\_\_(self, node, time\_now)

**Parameters**

1. node: class Node, a communication node
2. time\_now: float, current simulation time

#### Public Properties

1. self.node: class Node, the Node that this Flow represents

#### Private Properties

1. self.\_\_total\_data\_len: int, the total total bytes of output tensors
2. self.\_\_remain\_len: float, the remain length of data in bits
3. self.\_\_estimated\_finish\_time: float, the estimated flow's finish time
4. self.\_\_available\_bandwidth: float, the current bandwidth of this Flow
5. self.\_\_last\_start\_time: float, the time of latest status change

#### Public Methods

1. def get\_estimated\_finish\_time(self): Return the estimated finish time of current flow
2. def get\_available\_bandwidth(self): Return the current available bandwidth
3. def set\_available\_bandwidth(self, available\_bandwidth, time\_now):Set self.\_\_current\_available\_bandwidth, then change estimated\_finish\_time and other flow status. Calculate total executed bytes during [last\_start\_time, time\_now], then update \_\_remain\_len and calculate new \_\_estimated\_finish\_time

**Parameters**

* 1. current\_available\_bandwidth: float, representing tansfer rate(bps)
  2. time\_now: float, current time of simulation

#### Private Methods

1. def \_\_lt\_\_(self, other): override '<' operator, Flow is ordered by estimated\_finish\_time

**Parameters**

* 1. other: Flow

### [**simulator.network\_simulator.Link**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\network_simulator\link.py)

Provide the unidirectional links connecting every devices.

class Link()

#### Remarks

This class is a unidirectional Link from src\_name to dst\_name with specific link properties. Each link can hold several flows.

#### Constructor

def \_\_init\_\_(self, link\_id, source\_name,

dest\_name, capacity='0bps', latency='0s')

**Parameters**

1. link\_id: int, the identity of current Link
2. source\_name: string, name of source device
3. dest\_name: string, name of destation device
4. capacity: string, the capacity of this link
5. latency: string, the propagation\_latency of the link, reserved for future usage

#### Public Properties

1. link\_id: int, the unique link identity
2. source\_name: string, the source device name
3. dest\_name: string, the destination device name
4. capacity: float, the capacity stored in bps
5. latency: string, the propagation\_latency of the link, reserved for future usage
6. flows: list of Flow, the flows that are occupying current Link

#### Private Properties

1. self.\_\_link\_id: int, in one NetworkSimulator, every Link should have a unique link\_id
2. self.\_\_source\_name: string, the source device name
3. self.\_\_dest\_name: string, the destination device name
4. self.\_\_capacity: float, the capacity stored in bps
5. self.\_\_latency: string, reserved for future usage
6. self.\_\_flows: list of Flow, the flows that are occupying this Link

#### Public Methods

1. def add\_flow(self, flow): Add a new flow to this link
2. def delete\_flow(self, flow: Flow): Delete specific flow from this link

### [**simulator.network\_simulator.LinkManager**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\network_simulator\link_manager.py)

LinkManager is mainly used by NetworkSimulator to maintain the Links.

class LinkManager()

#### Remarks

This class provides methods to maintain the Link objects and the routing information. NetworkSimulator will initialize this class and use it to get Links' information.

#### Constructor

def \_\_init\_\_(self, links\_spec, routing\_info\_dict)

**Parameters**

1. links\_spec: list of dict, [{'link\_id': int, 'source\_name': str, 'dest\_name': str, 'capacity': str, 'latency': str}]

src/dest\_name format: /server/hostname/DeviceType/DeviceIndex/ /switch/switch\_name/

1. routing\_info\_dict: dict, {(src\_name, dst\_name, route\_index):[id0, id1..]}

#### Public Properties

1. LinkManager.link\_essential\_data: dict, contains all essential attributes that a link\_spec should have.
2. LinkManager.link\_extra\_data: dict, contains optional attributes that a link\_spec may have.

#### Private Properties

1. self.\_\_links\_dict: dict, {link\_id: Link obj} containing all links
2. self.\_\_routing\_path: dict, storing routing path: {(src\_name, dst\_name, route\_index):[Link0, Link1..]}

#### Public Methods

1. def get\_routing\_path(self, src\_name, dst\_name, route\_index): Return a list containing the sequence of Link in this path, return None if not found.

**Parameters**

* 1. src\_name: string, the source name of the device
  2. dst\_name: string, the destination name of the device
  3. route\_index: int, the routing index

1. def get\_routing(self, node\_name): Parse the node\_name string and return routing path.

**Parameters**

* 1. node\_name: string, format: ":send:src\_name:dst\_name:route\_index:" **Returns** list of Link, [Link0, Link1..]

1. def get\_link(self, link\_id): Return a Link with specific link\_id
2. def get\_links\_dict(self): Return a dict {link\_id: Link obj} containing all links

#### Private Methods

1. def \_\_init\_links(self, links\_spec): Check validity of links\_spec, and init self.\_\_links using links\_spec. This method will only be called during initialization.
2. def \_\_init\_routing\_path(self, routing\_info\_dict): Init self.\_\_routing\_path via routing\_info\_dict. This method will only be called during initialization.

### [**simulator.network\_simulator.NetworkSimulator**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\network_simulator\network_simulator.py)

NetworkSimulator is a Device that will calculate communication Nodes' time cost.

class NetworkSimulator(Device)

#### Remarks

This class is the main module to calculate the time cost of communication Nodes. It is inherited from class Device.

#### Constructor

def \_\_init\_\_(self, name, links\_spec, routing\_info)

**Parameters**

1. links\_spec: list of dict, [{'link\_id': int, 'source\_name': str, 'dest\_name': str, 'capacity': str, 'latency': str}]

src/dest\_name format: /server/hostname/DeviceType/DeviceIndex/ /switch/switch\_name/

1. routing\_info\_dict: dict, {(src\_name, dst\_name, route\_index):[id0, id1..]}

#### Private Properties

1. self.\_\_flows: list of Flow, containing all Flows that being executed.
2. self.\_\_link\_manager: class LinkManager, controlling all Links in the NetworkSimulator

#### Public Methods

1. def is\_idle(self): Check whether the network simulator is idle
2. def get\_next\_node(self): Return (next finish node, estimated finish time)
3. def enqueue\_node(self, node, time\_now): Enqueue a node and update all the flows. The node will first be turned into a Flow, then add to the list, and update all Flow's capacities
4. def dequeue\_node(self): Dequeue the flow with the smallest estimated\_finish\_time

#### Private Methods

1. def \_\_update\_all\_flows\_capacities(self, time\_now): Calculate all flow's capacities, and update flows' status during [last\_start\_time, time\_now). There are two events that will change the flows' capacities: enqueue and dequeue.

**Parameters**

* 1. time\_now: float, the current simulation time

1. def \_\_schedule\_link(self, link, flow\_current\_capacity, unfinished\_schedule\_flow): Schedule flows that in link.flows and in unfinished\_schedule\_flow, allocate remain capacities equally to these flows. return a dict {flow\_obj: flow\_schedule\_capacity}, which contains all flow in link.flows, denoting capacities.

**Parameters**

* 1. link: class Link
  2. flow\_current\_capacity: dict, {flow\_obj: capacity}, denoting the current capacity for each flow
  3. unfinished\_schedule\_flow: a set containing all flows who require more capacity

## **Adapters Document**

### [**adapter.GraphAdapter**](file:///D:\Projects\SuperScaler\ai_simulator\adapter\adapter.py)

The base class for framework adapters. A virtual class.

class GraphAdapter(object)

### [**adapter.PlanAdapter**](file:///D:\Projects\SuperScaler\ai_simulator\adapter\plan_adapter.py)

Provide an interface for converting plans generated by PlanGen to nodes in the AISimulator.

class PlanAdapter()

#### Constructor

def \_\_init\_\_(self)

#### Public Methods

1. def set\_plan(self, plan): Set self.\_\_plan from a node\_list, return False if input is invalid

**Parameters**

* 1. plan: list

1. def get\_plan(self): Return the adapted plan

## **Utility Functions Document**

### [**simulator.utility.transfer\_rate\_to\_bps(rate\_str)**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\utility.py)

Get transfer rate based on the rate\_str

def transfer\_rate\_to\_bps(rate\_str)

**Parameters**

1. rate\_str: string, representing transfer rate, e.g. '1Kibps' == '1024bps', '1Kbps' == '1000bps'

**Returns**

float, transfer rate in bps

**Support suffix**:

"bps", "bit/s", "[kK]bps", "[kK]bit/s", "[kK]ibps", "[kK]ibit/s",

"[mM]bps", "[mM]bit/s", "[mM]ibps", "[mM]ibit/s",

"[gG]bps", "[gG]bit/s", "[gG]ibps", "[gG]ibit/s",

"[tT]bps", "[tT]bit/s", "[tT]ibps", "[tT]ibit/s"

### [**simulator.utility.data\_size\_to\_bit(size\_str)**](file:///D:\Projects\SuperScaler\ai_simulator\simulator\utility.py)

Get datasize based on the size\_str

def data\_size\_to\_bit(size\_str)

**Parameters**

1. size\_str: string, representing data size suffix, e.g. '1Kib' == '1024b'

**Returns**

float, data size in bps

**Support suffix**:

'Bit', 'Byte', 'KiB', 'MiB', 'GiB',

'TiB', 'PiB', 'EiB',

'kB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB', 'Kib',

'Mib', 'Gib', 'Tib', 'Pib', 'Eib', 'kb', 'Mb', 'Gb', 'Tb',

'Pb', 'Eb', 'Zb', 'Yb'

# Simulation Execution Module:

Defined in simulator.py, simulator\_execution(). Run the simulation.

The result of simulation is stored in Graph object.

## Algorithm overview:

Regard all computation devices (CPU, GPU and NetworkSimulator) as **asynchronized** devices. Each device maintains its job queue, and the finish time of its current job.

Init Simulator

Sync a pending node

Start ready nodes

Return

The simulator keeps a ready list. All nodes whose dependency are fulfilled will be added to the ready list.

Simulator run in two phases:

Phase 1: still has nodes not started. In this phase, the simulator will run a loop, includes 2 steps:

1. Wait until one enqueued op is finished. Dequeue it from device and update all successor nodes’ dependency counter. If the counter is zero, add the successor node into ready list.

This step will handle only one node at a time.

1. Start all nodes in ready list. Enqueue them into device and mark status as ‘pending’

Phase 2: wait until all nodes finished. Record the time of last finished node.

# Flow chart:

### Flow chart for the algorithm





### Flow char for the experiment

