#### 代码文件说明

### 仿真框架

整个仿真框架是基于离散事件库Simpy库，详细介绍可参考：

<https://simpy.readthedocs.io/en/latest/contents.html>

* 仿真环境代码

**#例化仿真环境**

**env = simpy.Environment()**

**#例化硬件模型**

**wd=Wafer\_Device(env,tile\_inter\_shape=[4,4],tile\_intra\_shape=[4,4],with\_3ddram\_per\_tile=True)**

**# 任务事件0时刻的注册**

**env.process(wd.noc\_process(10,src\_id=0,des\_id=3,task\_id=1,DEBUG\_MODE=Debug))**

**# 进行10000 us时间单位的仿真**

**env.run(until=10000)**

### 主要文件

**Wafer\_device**.py定义了wafer的基本架构：由tile组成的多die系统,例如4x4个tile构成一个die,然后4x4个die构成wafer级别的系统。with\_3ddram\_per\_tile 决定是否引入tile级别的dram。此外tile内，die内均由noc link相连接，noc link资源有限，抢占式调度。默认每个x方向的边缘die会连接dram，那么上述参数定义下的dram共8个。

**wd=Wafer\_Device(env,tile\_inter\_shape=[4,4],tile\_intra\_shape=[4,4],with\_3ddram\_per\_tile=True)**

**op\_pd**.py 定义了算子参数，并行策略，存储，通信等信息

**comp\_graph**.py 定义了计算图构建和读写文件

**pipeline**.py 定义了流水线的process

**main**.py 执行一个计算图的流水线模型

**tile\_dataflow**.py 定义了tile的基本信息和process

### 其他文件

ML.py 定义了神经网络/深度学习基本的术语和策略

Util.py 定义了一些格式转换方法

monitored\_resource.py 重写部分simpy类

#### 内存分配

内存层次定义如下图所示：

假设计算阵列的计算速率基本匹配SRAM的带宽，因此不对SRAM带宽建模;

需要评估的对象是SRAM容量，3D DRAM容量与带宽与DDR容量与带宽；分析各存储层次在大模型训练中的作用。

目前认为3D DRAM与每个边缘Die的带宽是单向的，不允许同时间读写，且3D DRAM层次是否存在是可参数配置的。

**SRAM**

**3d\_DRAM**

**per\_Tile**

**Edge\_Die**

**DDR**

**Compute**

**array**

下面阐述内存分配策略：

存在的问题:

1.考虑重计算，反向计算的时候重复一遍正向计算，需要存储该stage的所有激活，理想情况仍然需要片上存储中间激活；原本流水线根据stage段数存储N份激活，现在减少到预留一部份存储空间，用于存储1份激活

#### 访存/通行phase

抽象出访存和通信的process，便于上层调度插入通信，访存事件。但抽象的具体实现方式存在很多优化空间

#### Simpy 事件注册机制

**import** simpy

**class** comm\_overlap():

**def** \_\_init\_\_(self,env) -> None:

        self.env=env

        self.cp\_worker= simpy.Resource(env, capacity=1)

        self.cm\_worker= simpy.Resource(env, capacity=1)

**def** cp\_process(self):

        '''''

        process 1

        '''

        with self.cp\_worker.request() as req:

**yield** req

**yield** self.env.timeout(20)

**print**('process 1 done @{:.3f} '.format(self.env.now))

**def** cm\_process(self):

        '''''

        process 2

        '''

        with self.cm\_worker.request() as req:

**yield** req

**yield** self.env.timeout(30)

**print**('process 2 done @{:.3f} '.format(self.env.now))

**def** overlap\_process(self):

         event\_list=[]

**while**(True):

              event\_list.append(self.env.process(self.cp\_process()))

              event\_list.append(self.env.process(self.cm\_process()))

**yield** simpy.AllOf(env,event\_list)

**print**('process overlap\_process done @{:.3f} '.format(self.env.now))

**break**

**def** order\_process(self):

**while**(True):

**yield** self.env.process(self.cp\_process())

**yield** self.env.process(self.cm\_process())

**print**('process order\_process done @{:.3f} '.format(self.env.now))

**break**

**def** short\_process(self):

**while**(True):

**yield** self.env.timeout(20)

**yield** self.env.timeout(30)

**print**('process short\_process done @{:.3f} '.format(self.env.now))

**break**

**if** \_\_name\_\_ == '\_\_main\_\_':

    env=simpy.Environment()

    test=comm\_overlap(env)

    env.process(test.overlap\_process())

    env.process(test.order\_process())

    env.process(test.short\_process())

    env.run(until=100)

**result:**

**process 1 done @20.000**

**process 2 done @30.000**

**process overlap\_process done @30.000**

**process 1 done @40.000**

**process short\_process done @50.000**

**process 2 done @70.000**

**process order\_process done @70.000**