目录

[代码文件说明 2](#_Toc145857452)

[一. 仿真框架 2](#_Toc145857453)

[二. 文件说明 2](#_Toc145857454)

[三. 算子参数 3](#_Toc145857455)

[四. 内存分配 4](#_Toc145857456)

[五. 计算建模 6](#_Toc145857457)

[六. 通信访存模型 7](#_Toc145857458)

[七. 并行抽象 7](#_Toc145857459)

[八. 映射(创新1) 7](#_Toc145857460)

[九. 附录 8](#_Toc145857461)

# 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)**

## 文件说明

**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

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

## 算子参数

|  |  |  |  |
| --- | --- | --- | --- |
| 算子类型枚举 | 维度 | 是否分析完毕 | 检查 |
| Linear | [B,M,N,K] | Yes |  |
| Conv2 | [B,C,H,W,R,S,K] | Yes |  |
| Softmax |  |  |  |
| Embedding |  |  |  |
| LayerNorm |  |  |  |
| Transformer | [B,S,H,A] | Yes |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Oppd类 | 算子参数 | 变量定义 | 变量类型 | 影响因素 |
| 基本参数 | 类型 | type | 枚举类 |  |
| 参数维度 | param\_dim | 整数链表4-7 |  |
| 并行维度 | p\_sgy | 整数链表4-7 |  |
| ZeRO策略 | ZeRO | 枚举类 |  |
| 不绑定设备的分析参数 | 输入shape | i\_shape | 整数链表3-4 |  |
| 输出shape | o\_shape | 整数链表3-4 |  |
| 权重，优化器，梯度大小 | w\_s\_g\_size\_m | 整数链表3 |  |
| 中间激活量访存量 | intra\_act\_access\_m | 浮点数 |  |
| 中间激活量存储量 | intra\_act\_size\_m | 浮点数 | 存储到反向阶段  可被重计算优化为0 |
| 权重，优化器，梯度访存量 | w\_s\_g\_access\_m | 整数链表3 |  |
| 前向计算乘加次数 | fd\_macs\_m | 浮点数 | 反向计算量认为是2倍 |
| 并行3阶段通信量 | f\_b\_u\_comm | 浮点数链表3 | 并行维度 |
| ZeRO 通信量 | ZeRO\_comm | 浮点数链表2 | 更新阶段无通信？ |
| 设备绑定的参数 | 是否映射表示 | dpmap\_flag | Bool |  |
| 设备组 | device | 整数链表 |  |
| 并行通信算子 | f\_b\_u\_comm\_d | CompOp类 |  |
| ZeRO通信算子 | ZeRO\_comm\_d | CompOp类 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| CommOp算子参数 | 变量定义 | 变量类型 | 备注 |
| 类型 | type | 枚举类 | ALL2ALL,ALL-REDUCE |
| 通信量 | size | 浮点数 | 没有乘以字节数 |
| 通信设备组 | device\_group | 整数链表 | 设备id |
|  |  |  |  |

## 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**

根据神经网络执行机制，将计算通信访存动作表述如下，以前向计算为例，**参考simpy事件注册机制。**前向执行分为九个event如下，根据内存分配策略有选择的注册为process。目前认为这九个event原则时间上可以重叠，但具体根据任务量和dram 资源和noc link资源会有完成时间的先后顺序，以最后完成的event为准，认为是整个前向计算的整个执行时间。严格意义上，需要考虑计算和通信时间是否能overlap，且下列事件的执行更加细粒度的考虑会交织的非常紧密。

|  |  |
| --- | --- |
| Event注册顺序 | **Event** |
| 0 | **edge\_dram 权重读16bit** |
| 1 | **edge\_dram 输入激活读** |
| 2 | **tile\_dram 权重读16bit** |
| 3 | **tile\_dram 输入激活读** |
| 4 | **Zero通信** |
| 5 | **前向计算** |
| 6 | **前向通信：模型并行维度** |
| 7 | **tile\_dram 中间激活写** |
| 8 | **tile\_dram 输出激活写？** |
| 9 | **edge\_dram 中间激活写** |
| 10 | **edge\_dram 输出激活写** |

#### 前向执行event

#### 反向执行event

考虑复杂算子，重计算process不能与反向计算process overlap

|  |  |  |
| --- | --- | --- |
| Event注册顺序 | **Event** | **备注** |
| 0 | **edge\_dram 权重读16bit** | **重计算** |
| 1 | **edge\_dram 输入激活读** |
| 2 | **tile\_dram 权重读16bit** |
| 3 | **tile\_dram 输入激活读** |
| 4 | **Zero通信** |
| 5 | **正向重计算** |
| 6 | **前向通信** |
| 7 | **tile\_dram 中间激活写** |
| 8 | **edge\_dram 中间激活写** |
| 0 | **edge\_dram 权重读16bit** | **反向计算dloss** |
| 1 | **edge\_dram dloss读,(数据量与act基本同)** |
| 2 | **tile \_dram 权重读16bit** |
| 3 | **tile \_dram dloss读** |
| 4 | **Zero通信** |
| 5 | **反向计算1** |
| 6 | **反向通信：模型并行维度** |
| 7 | **tile \_dram dloss写** |
| 8 | **edge\_dram dloss写** |
| 0 | **edge\_dram 激活读** | **反向计算dW**  **优化器参数更新** |
| 1 | **edge\_dram dloss读** |
| 2 | **edge\_dram 优化器参数读** |
| 3 | **tile \_dram 激活读** |
| 4 | **tile \_dram dloss读** |
| 5 | **tile \_dram 优化器参数读** |
| 6 | **~~Zero通信~~** |
| 7 | **反向计算2** |
| 8 | **~~反向通信：模型并行维度~~** |
| 9 | **tile \_dram优化器参数写** |
| 10 | **tile \_dram权重梯度dW 32bit写** |
| 11 | **edge\_dram 优化器参数写** |
| 12 | **edge\_dram 权重梯度dW 32bit写** |
|  |  |

#### 权重更新event

权重更新需等待流水线排空。

|  |  |  |
| --- | --- | --- |
| Event注册顺序 | **Event** | **备注** |
| 0 | **edge\_dram 权重读32bit** | **忽略加和计算时间** |
| 1 | **edge\_dram 梯度读32bit** |
| 2 | **tile\_dram 权重读32bit** |
| 3 | **tile\_dram 梯度读32bit** |
| 4 | **同步通信:数据并行维度32bit?** |
| 5 | **tile\_dram 权重写32bit** |
| 6 | **edge\_dram 权重写32bit** |

已经考虑的影响存储，通信和计算的策略因素

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **策略** | **策略空间** | **主要影响** | **是否已经实现** |
| **1** | **单核数据流** | **WS,IS,OS** | **权重和激活，loss的存取次数** | **√** |
| **2** | **优化器** | **SGD,ADAM** | **主要影响权重相关的存储量** | **√** |
| **3** | **重计算** | **None，all** | **影响激活生存时间，等待反向** | **√** |
| **4** | **ZeRO** | **None,1,2,3** | **影响DP维度的激活存储量，通信量** | **√** |
| **5** | **流水线** | **Cerebras,Gpipe,1F1B等** | **影响不同stage段激活存储份数** | **Ing** |
| **6** | **并行方式** | **数据并行，模型并行** | **权重，激活等存储量，通信量** | **√** |
| **7** |  |  |  |  |
| **8** |  |  |  |  |
| **9** |  |  |  |  |

默认:DP通信可以与计算overlap；MP通信不能与计算overlap

#### 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，用于存储反向过程中的1份激活。认为反向的dloss不能被及时消耗，多次读写
2. 对于SRAM策略需要优化考虑，可以考虑16bit存SRAM，32比特备份以及优化器参数存dram
3. 认为dloss与激活计算得到梯度的过程中，dloss更不频繁存取的数据流为WS

## 计算建模

1. 简单的计算模型只需要用算子macs除以硬件的macs数或者算力；复杂一点的考虑，需要为计算与通信进程考虑是否可以overlap的问题；
2. 另一方面计算模型需要反应输入size的影响，形成‘’陡峭”曲线，scale-sim有对脉动阵列的简单数学建模；在本人的学习过程中，也看到直接与尺寸相关的建模方式。

    def compute\_cycles(self,param:List[int]):

        '''

        define matrix multiply [m,n,k]: (m,k)\*(k,n)=(m,n)

        SR:m SC:n T: k

        PE array num:PR\*RC

        each PE macs units: R\*C

        #reference: https://github.com/ARM-software/SCALE-Sim

        '''

        assert(len(param)==3)

        [SR,SC,T]=param

        [R,C]=self.array\_shape

        [PR,PC]=self.array\_group

        if self.cp\_model==comp\_model.SCALE\_SIM:

            sr=math.ceil(SR/PR)

            sc=math.ceil(SC/PC)

            return (2\*R+C+T-2)\*math.ceil(sr/R)\*math.ceil(sc/C)

        elif self.cp\_model==comp\_model.simple:

            return T\*math.ceil(SR/(R\*PR))\*math.ceil(SC/(C\*SC))

        elif self.cp\_model==comp\_model.abrupt\_curve:

            cost=T\*math.ceil(SR/(R\*PR))\*math.ceil(SC/(C\*SC))

            if SR%(PR\*R)==0 and SC%(PC\*C)==0:

                cost=1.0\*cost

            elif SR%(PR\*R)==0 :

                cost=1.2\*cost

            elif SR%R==0 and SC%C==0:

                cost=1.8\*cost

            elif SR%R==0 :

                cost=2.0\*cost

            else:

                cost=2.5\*cost

            return int(cost)

        else:

            raise NotImplementedError

## 通信访存模型

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

## 并行抽象

将算子对应分配好的硬件Group按照并行维度进行划分,详见util.py里的**split\_comm\_group(Group\_Id,parall\_dims)**函数。同时，每个算子的参数量与计算量会被对应的切分方式重新定义；

    Here is an example :

    suppose Group\_Id=[0,1,2,3,...,15],len=16

    1.if parall\_dims=[16,1,1,1],group=[[0:15],[],[],[]]

    2.if parall\_dims=[1,16,1,1],group=[[],[0:15],[],[]]

    3.if parall\_dims=[8,2,1,1],group=

    [[0, 2, 4, 6, 8, 10, 12, 14], [1, 3, 5, 7, 9, 11, 13, 15]]

    [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10, 11], [12, 13], [14, 15]]

    []

    []

## 映射(创新1)

1. 建立core邻接矩阵
2. 根据某种先验（算力，存储需求）划定基本core数，需要一个算法规定各个形状的位置
3. 各个形状代价可以由环路开销决定（构成环路的最小非冗余跳数）

## 附录

#### Event register 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**