CS21106
高级硬件设计(FPGA)

研究小课题

支持 CGRA 软件流水的调度

刘大江

计算机学院

重庆大学 CHONGQING UNIVERSITY

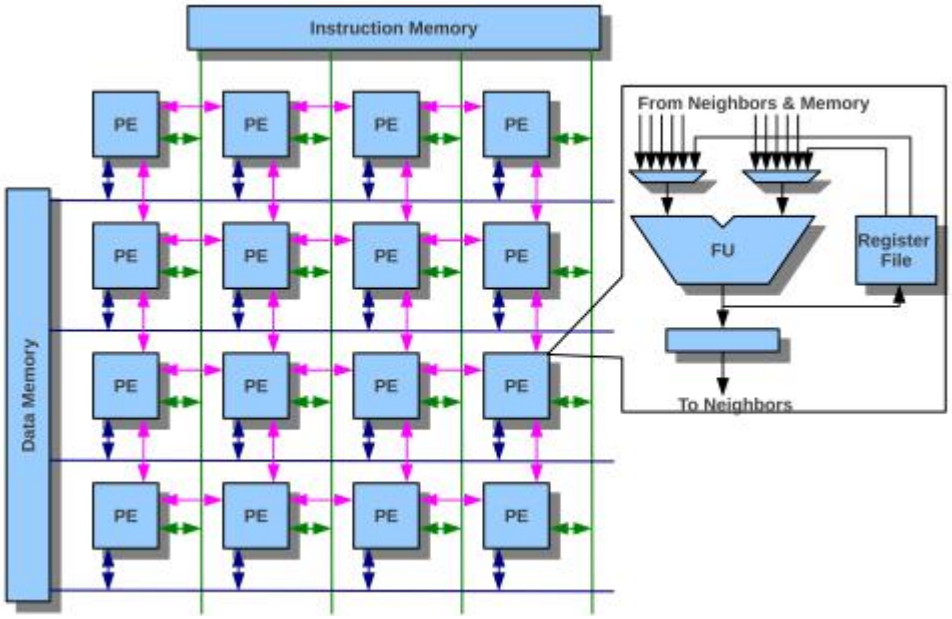
2020春季

可重构计算中的软件流水的调度

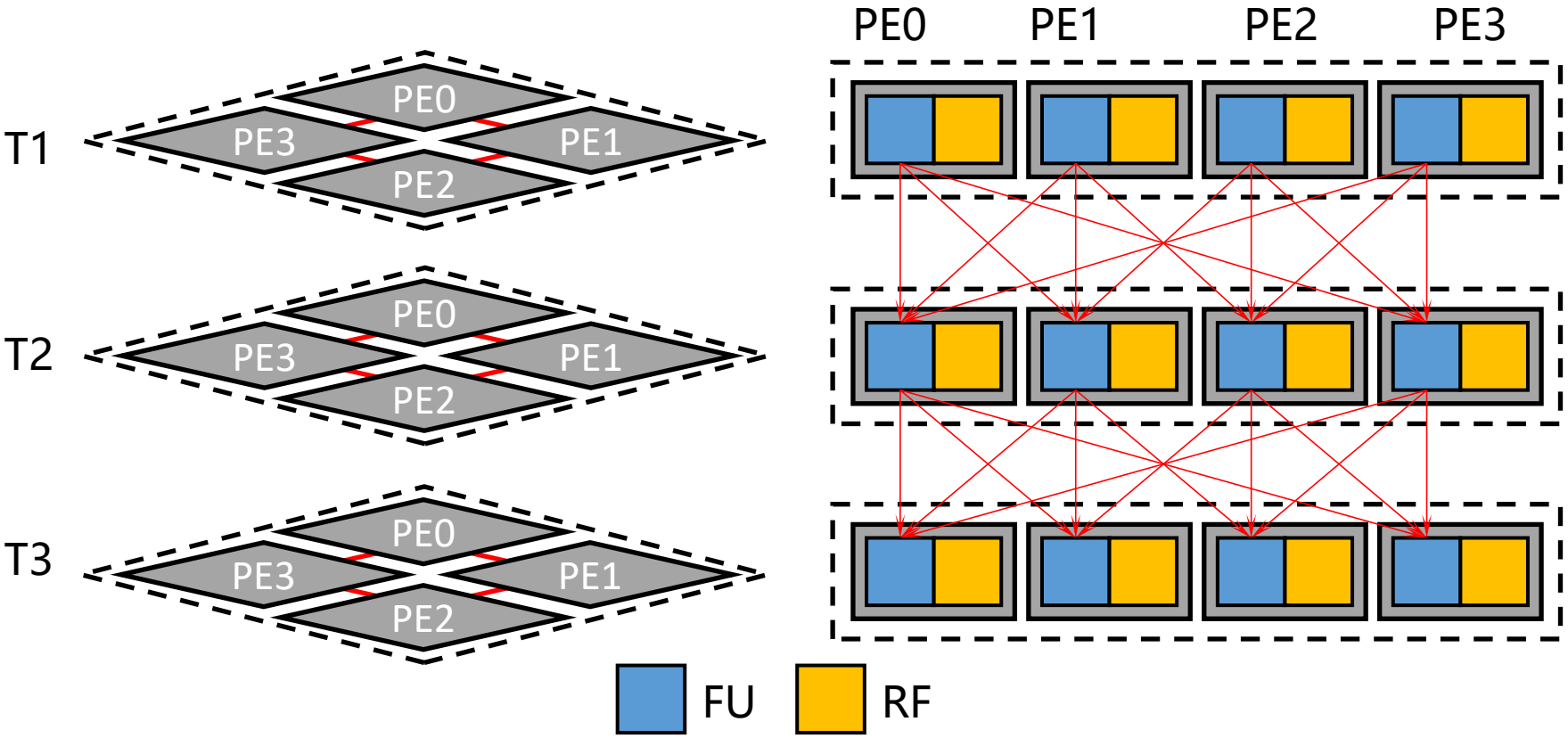
- 可重构计算：粗粒度可重构架构（Coarse-Grained Reconfigurable Architecture，CGRA）是一种兼具编程灵活性和高能量效率的一种新型计算平台，适合加速计算密集型应用。
- 软件流水：通过最小化循环启动间隔来增加循环并行性的循环优化技术。
- 问题定义：对循环体 DFG 进行调度，使调度之后的DFG 满足资源约束和长依赖约束，最小化CGRA计算阵列的处理单元使用量。

CGRA 架构

- CGRA由数据存储、配置存储和处理单元（PE）阵列组成。
- PE 阵列由10-100个PE以二维排列的方式构成，相邻的 PE 之间可以交换数据。
- 每个 PE 由一个功能单元 FU 和 一个寄存器堆 RF 组成



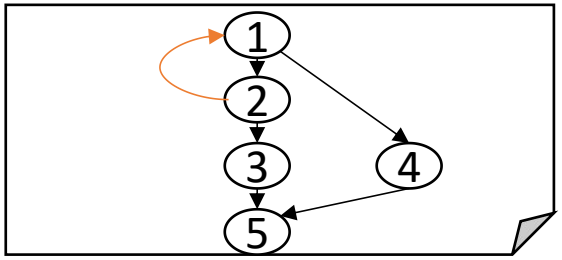
Time Extended CGRA：TEC

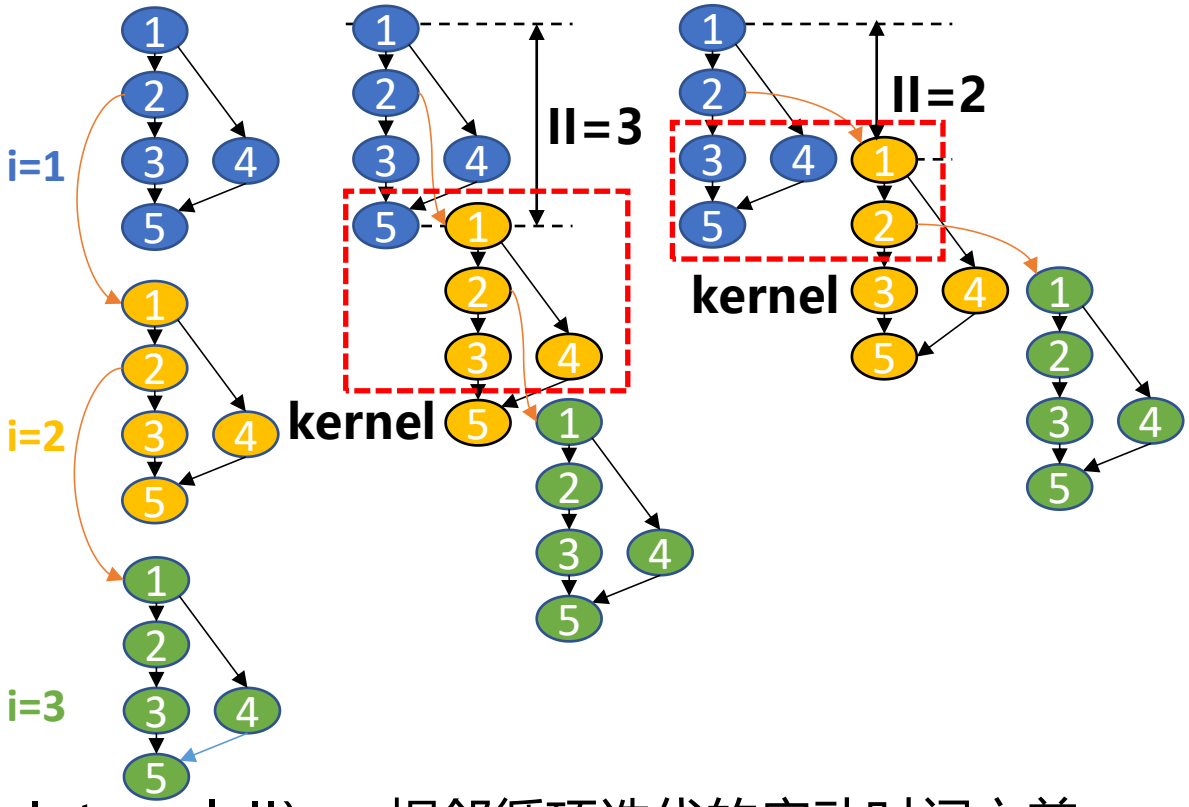


- PE 阵列可以时域扩展，分时复用执行不同的算子
- FU 可以连接到下一时刻邻居 PE 或自身PE 的 FU，但不能连到斜对角 PE
- RF 的数据只能传递到自己 PE 的 FU
- FU上可以布局算子，RF上只能传递数据

循环软件流水

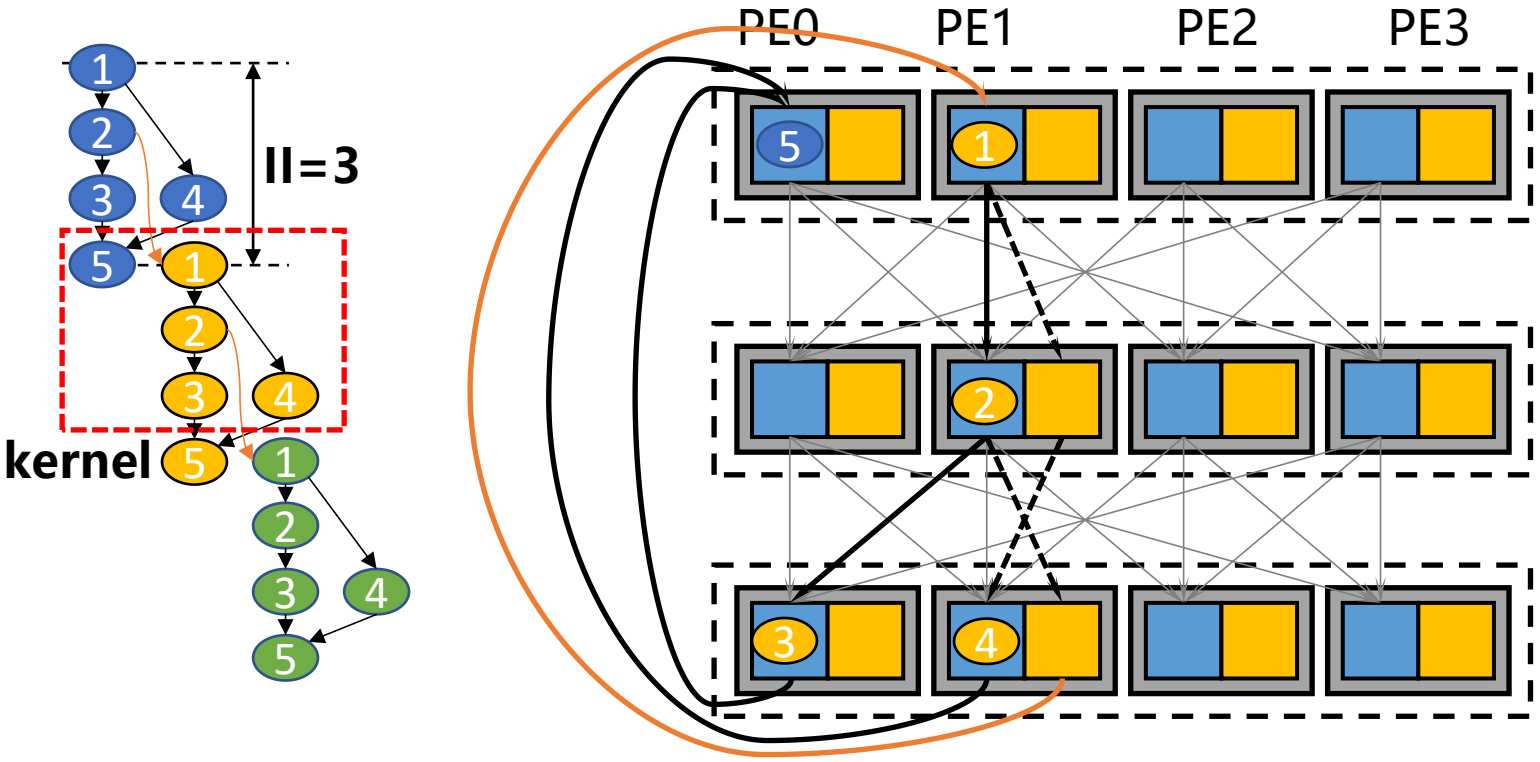
```
For i=1:N
 a[i] = b[i-1] + 2; ①
 b[i] = a[i] << 3;  ②
 c[i] = b[i] >> 2;  ③
 d[i] = a[i] - 6;   ④
 e[i] = c[i] + d[i]; ⑤
```





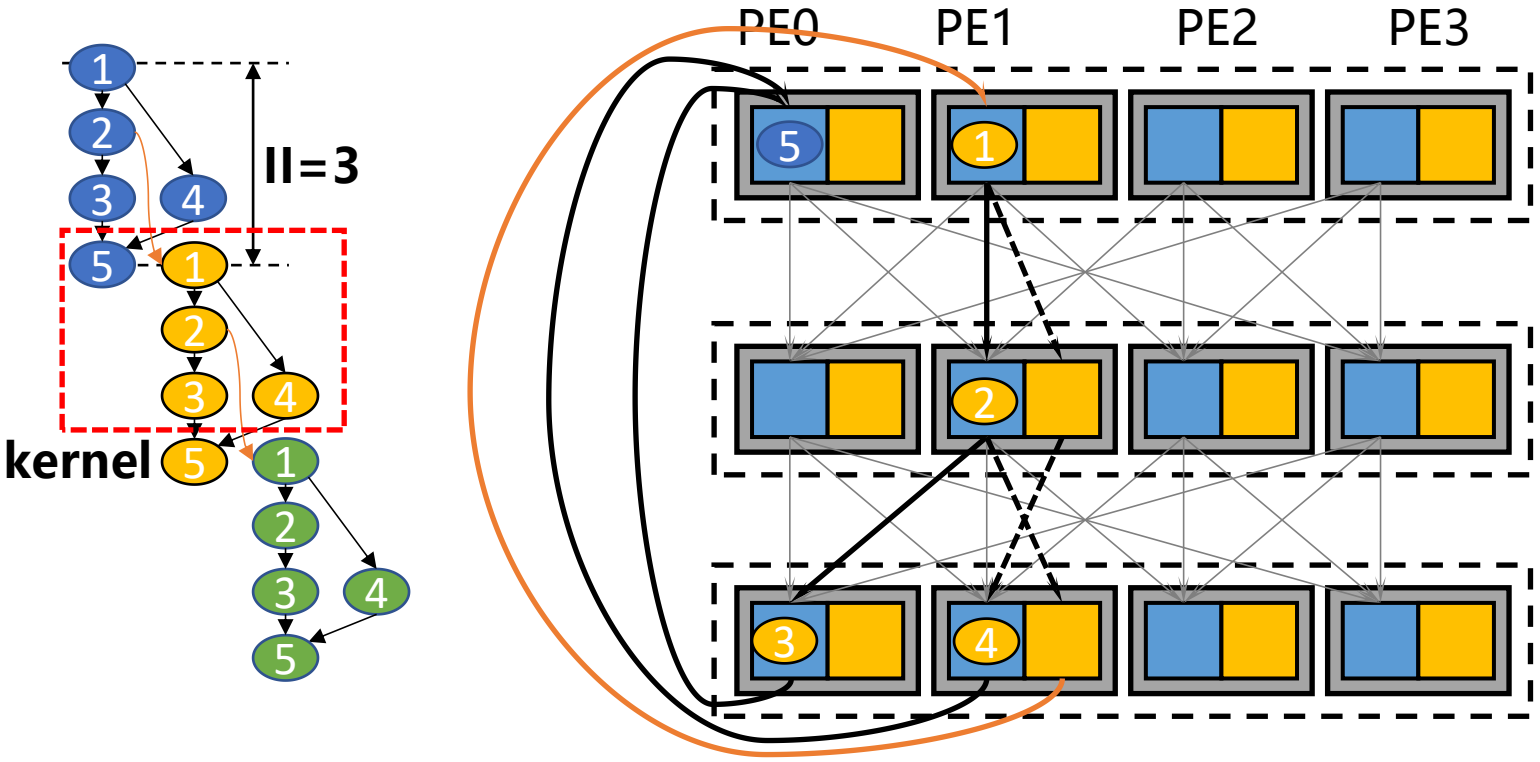
- 启动间隔（Initiation Interval, II）：相邻循环迭代的启动时间之差
- 减小 II 可以增加并行度，但是 II 也受限于循环间依赖和资源的约束，受依赖约束的最小 II 称为 RecMII，受资源约束的最小 II 称为 ResMII，实际 MII = Max(RecMII, ResMII)
- 循环流水后最小重复单元称为 Kernel，由来自不同循环迭代的算子组成。

CGRA 上的模调度（1）



- 通过计算 MII 得到初始 II
- **资源约束：**如果两个算子 v_i, v_j 的时间步对 II 取模之后相等 t_i % II = t_j % II，则他们处于 TEC 的同一层，要在一起竞争 PE 资源

CGRA 上的模调度（2）

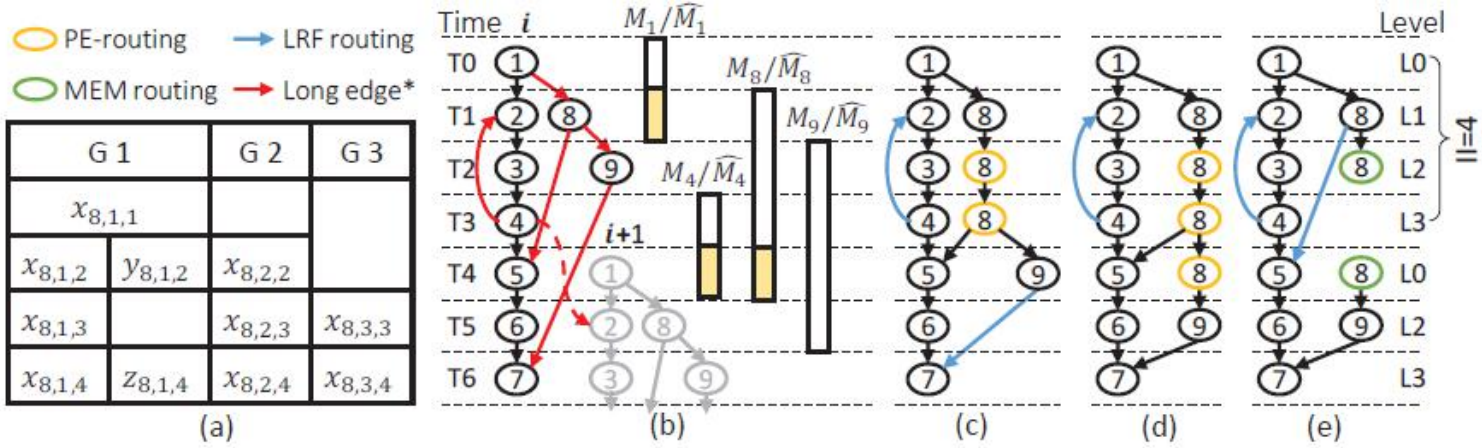


- 调度后的长依赖通过寄存器 RF 传输，而 RF 只能由自身PE的 FU 访问，因此要求长依赖的目标节点和源节点必须放在同一个 PE 上。
- **依赖长度约束：**模调度时间相同 (t_i % II = t_j % II) 的算子 v_i 和 v_j 处于 TEC 的同一层，因此调度结果不能出现依赖长度等于 II 或者 II 倍数的调度方案。

问题1：支持 CGRA 软件流水的调度

- 假设：资源类型只有一种（即PE资源），PE的数量为16，每个 PE 支持给定 DFG 中的任意算子
- 目标函数：最小化 PE 的使用数量
- 约束：
  - 依赖约束：保证依赖的算子的先后顺序
  - 资源约束：TEC 中同一层的算子数量不超过 PE 的数量
  - 依赖长度约束：没有依赖长度等于II或II倍数的依赖
- 建模方式：可以用 ILP 建模，也可以用其他启发式算法。

问题2：映射友好的CGRA调度

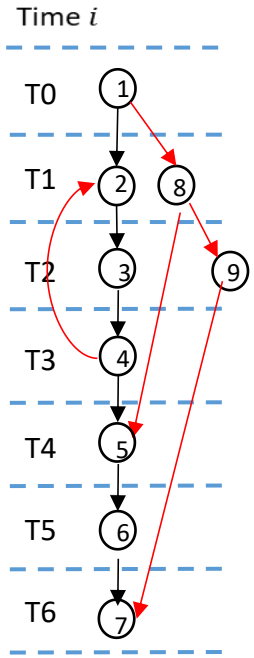


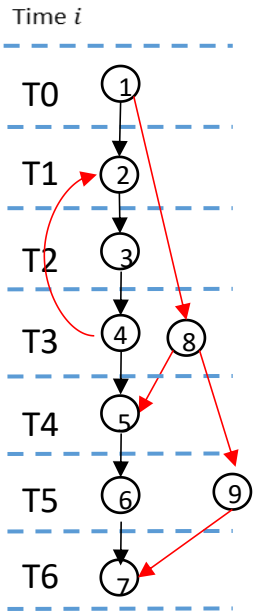
- 最大的DDG延迟(T_l):大于或等于DDG图的关键路径长度。
如图(b), T_l可等于关键路径（$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$）的长度：7

- R 可路由算子集合:如果某个算子至少有一条出边可以跨越多个时间步，那么此算子可放在R集合中。如图（b），边1 → 8，跨越T0到T3；如边4 → 2,跨越T3到T5.所以图(b)的 R集合为{1, 4, 8, 9}

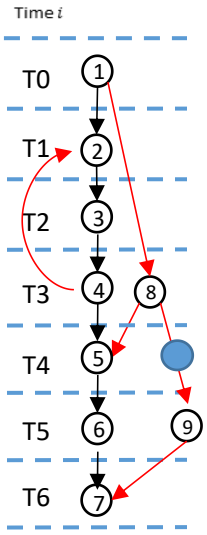
- S_n 代表算子可以被调度的最早时间步：1,4,8, 9 对应的为0,3,2,1。
- L_n 代表算子可以被调度的最晚时间步：1,4,8,9 对应的为0,3,3,5。
- $\widehat{L_n}$ 代表算子可以被插入的路由节点的最晚时间步：它等于该算子中最晚时间步的子节点的时间步减1。

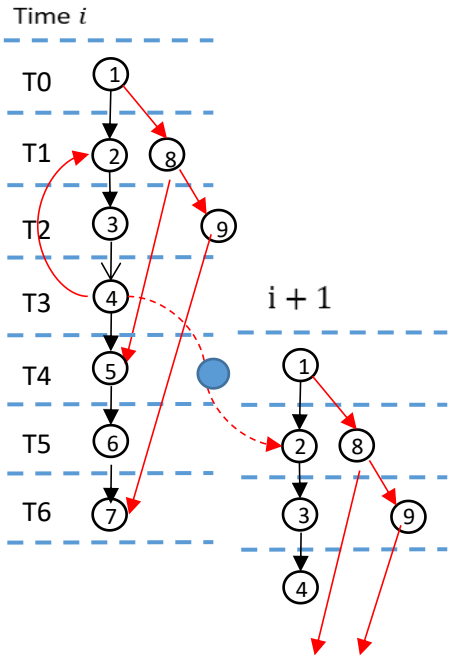
- 算子n可放置的最早时间步（S_n）：如图，算子1,4,8,9最早时间步分别为0,3,1,2
- 算子调度可放置的最晚时间步（L_n）：如图，算子1,4,8,9的最晚时间步分别为0,3,3,5





- 最晚路由时间步($\widehat{L_n}$)：算子n的路由算子插入的最晚时间步为$\max(L_{n'}-1)$，n'为算子的子算子；
如算子8有两个子算子5和9, L_5=4, L_9=5，那么$\widehat{L_8}$=L_9-1=4
如果是跨越周期的依赖，如算子4，最晚时间步为$\max(L_{n'}+II-1)$，
$L2$ = 1 ,II=4, $\widehat{L_4}$=1+4-1=4





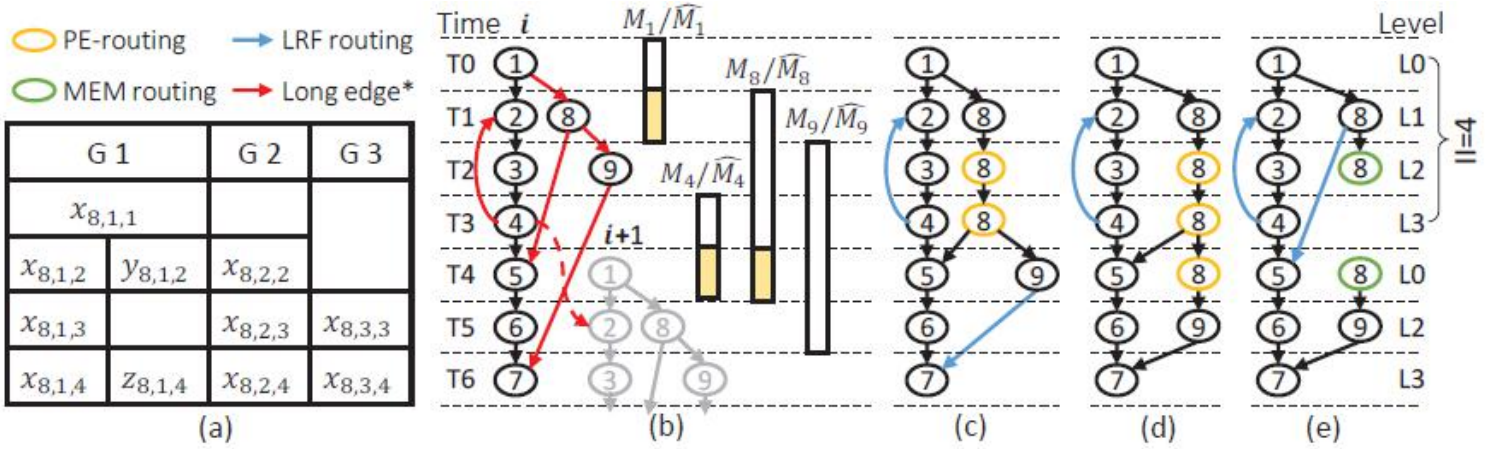
- 算子n的机动性(M_n),它的取值范围为[最早时间步,最晚时间步]，如图
算子1,算子4,算子8 和算子9 的机动性分别为 [0,0], [3,3], [1,3] ,[2,5]

- $\widehat{M_n}$：由算子的机动性范围 M_n扩展而来，代表算子的路由节点可放置的范围,即为：$[S_n, \widehat{L_n}]$

- 边集合 (E):原始DDG图的依赖边集合

- 内存访问延迟(T_m)：，它是内存路由的最小延迟，包括数据存，数据持有，数据取；
假设数据存取执行时间为一个时间步，那么T_m=3

- N_p: PEA 阵列中PE的数量



1：变量的定义

- $x_{n,i,j}$:算子n在i时间步被调度，在j时间步插入它的PE路由结点，$j \geq i$
  $x_{8,1,2}$表示算子8在时间步1被调度，在时间步2插入PE路由节点

$$i \in [S_n, L_n], j \in [i, \widehat{L_n}]$$

- $y_{n,i,j}$:算子n在i时间步存在内存中，在j时间步插入一个store算子，
  $y_{8,1,2}$表示算子8在时间步1存，在时间步4插入$store$算子

$$i \in [S_n, L_n], j \in [i+1, \widehat{L_n} - T_m + 1]$$

- $z_{n,i,j}$:算子n在i时间步存在内存中,在j时间步插入一个load算子;
  $z_{8,1,4}$表示算子8在时间步1存，在时间步4插入$load$算子

$$i \in [S_n, L_n], j \in [i + T_m, \widehat{L_n}]$$

- n_{pe}:使用的PE最多数量，取模后每一时间步的op算子个数要小于或等于n_{pe}

路由方式：

- PE 路由：数据通过PE进行路由，占用PE资源
- 寄存器路由：数据通过PE的RF进行路由，即在长依赖中，算子在不同时间步只能存放在同一个PE上
- Memory routing:将长依赖数据，先存起来，快要使用时再取出来的方法进行路由
- Path Sharing: ILP需要支持该优化方式。如节点5和节点9可以共享相同节点8的路由路径。

2：约束

- 唯一性：算子的调度时间步是唯一的

$$\sum_{i \in [S_n, L_n]} x_{n,i,i} = 1, \forall n \in R$$

- 排他性：一旦算子的调度时间步确定，该算子其他不同开始调度时间步的情形都是不可行的

$$x_{n,i_1,i_1} + x_{n,i_2,j} \leq 1, \forall i_2 \neq i_1$$
$$x_{n,i_1,i_1} + y_{n,i_2,j} \leq 1, \forall i_2 \neq i_1$$
$$x_{n,i_1,i_1} + z_{n,i_2,j} \leq 1, \forall i_2 \neq i_1$$

- 插入load算子和插入store算子要同时存在

$$y_{n,i,j_1} - z_{n,i,j_2} = 0, \forall j_1, j_2$$

- PE路由和memory路由是互斥的,即一旦使用了pe进行路由就不能使用memory进行路由。
  针对的是某一条长依赖不能同时被两种方式路由。

$$y_{n,i,j_1} + x_{n,i,j_2} \leq 1, \forall j_1, j_2$$

- 依赖约束：算子节点的调度时间必须满足早于其子节点，同时算子节点的路由时间步会有所不同，以节点8来举例。

$$i_1 \times x_{n_1,i_1,j_1} < \sum_{i_2 \in [S_{n_2}, L_{n_2}]} (i_2 \times x_{n_2,i_2,i_2}), \forall (n_1, n_2) \in E$$

$$j_1 \times x_{n_1,i_1,j_1} < i_2 \times x_{n_2,i_2,i_2}, n_2 = \underset{\{n|(n_1,n)\in E\}}{\operatorname{argmax}} S_n$$

- PE资源约束：

$$\sum x_{n,i,j} + \sum y_{n,i,j} + \sum z_{n,i,j} \le n_{pe},$$
$$\forall (j \mod II) = t, \forall t \in [0, II-1]$$

- 寻找次优解：

$$\sum_{v \in B} v - \sum_{v \in A} v \le k - 1$$

A = {v|v = 0, ∀v ∈ S* }，B = {v|v = 1, ∀v ∈ S* }。

- 绑定约束优化：

$$O > O_{lb}$$

O为当前求得的最优值，O_{lb} 是上一个目标值。

3：映射灵活度评估

$$n_{ins} = \sum_{n \in R, i \in [S_n, L_n], j \in [i, L_n]} (x_{n,i,j} + \alpha(y_{n,i,j} + z_{n,i,j}))$$

目标函数：

$$\min_{x_{n,i,j}, y_{n,i,j}, z_{n,i,j}, n_{pe}} O = \beta \cdot n_{pe} - n_{ins}$$

β权重系数，Npe表示最大资源使用量（即调度后DFG最大的宽度），Nins表示插入的路由算子的数量。

等式右边的第一项表示最小化每一个时间步上使用的PE数，使得布局布线的灵活度更高；等式右边的第二项表示在满足资源约束的情况下插入的最多节点数。

ILP 编程环境 PuLP

- PuLP is an LP modeler written in python. PuLP can generate MPS or LP files and call GLPK, COIN CLP/CBC, CPLEX, and GUROBI to solve linear problems.
  - x = LpVariable("x", lowBound=None, upBound=None, cat='Integer')
  - y = LpVariable("y", lowBound=None, upBound=None, cat='Integer')
  - prob = LpProblem("myProblem", LpMinimize)
  - prob += x + y <= 2
  - prob += -4*x + y
  - status = prob.solve()

  注意：cat 用来指定变量是离散(Integer,Binary)还是连续(Continuous).

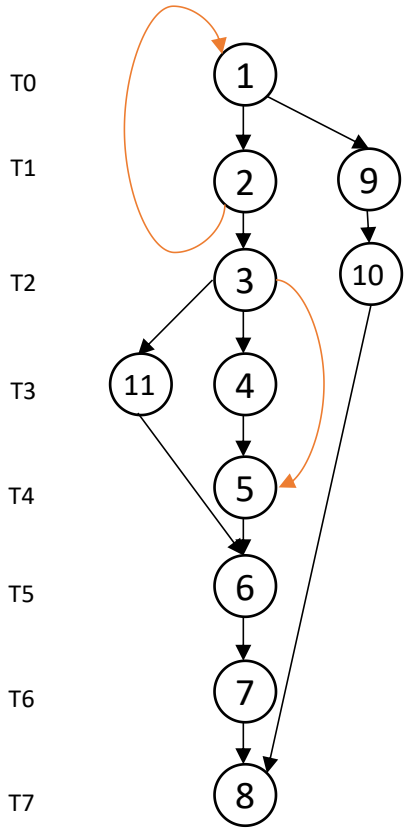
研究项目评分

- 研究报告 20 分
  - 背景描述
  - 问题定义
    - 目标+约束
  - 问题求解
  - 结果及讨论
    - 至少给10个DFG（算子数量10-100），分析调度结果和编译时间
  - 小组分工
- 代码实现 20 分
  - 书写规范
  - 功能验证
    - 助教检查时任意给定算子数量小于100的 DFG，看能否产生正确调度
  - 性能验证
    - 助教检查时给定算子数量为100的DFG，看编译时间多长

输入格式

边的类型：0 迭代内依赖；1 迭代间依赖

节点类型：0 表示无机动性的节点；1 表示具有机动性的节点

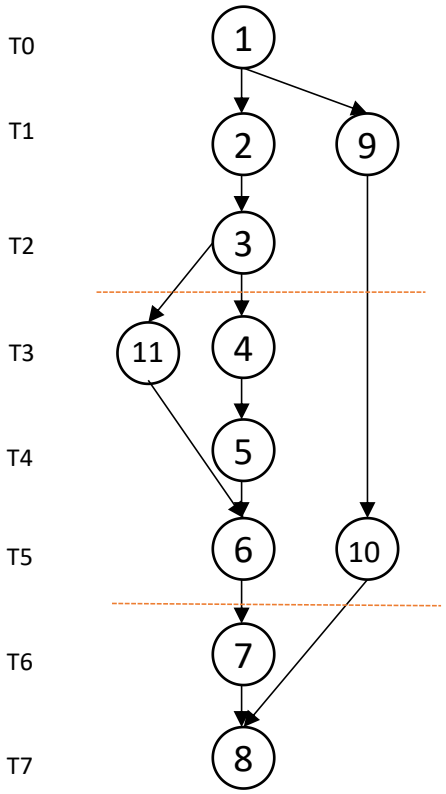


节点编号	子节点1	边类型	子节点2	边类型	子节点3	边类型	子节点4	边类型	最早时间步	最晚时间步	节点类型	有无父节点
1	2	0	9	0	0	0	0	0	0	0	0	0
2	3	0	1	1	0	0	0	0	1	1	0	1
3	4	0	11	0	5	1	0	0	2	2	0	1
4	5	0	0	0	0	0	0	0	3	3	0	1
5	6	0	0	0	0	0	0	0	4	4	0	1
6	7	0	0	0	0	0	0	0	5	5	0	1
7	8	0	0	0	0	0	0	0	6	6	0	1
8	0	0	0	0	0	0	0	0	7	7	0	1
9	10	0	0	0	0	0	0	0	1	5	1	1
10	8	0	0	0	0	0	0	0	2	6	1	1
11	6	0	0	0	0	0	0	0	3	4	1	1

- 不同节点以**换行符**分割
- 节点内部不同字段以**逗号**分割

输出格式

节点类型：0-原始的算子节点；1-插入的路由节点
原节点编号：路由节点传递的原始节点的编号



One scheduled result with resource constraint is 4 at each time step (II = 3)

节点编号	时间步	子节点1	子节点2	子节点3	子节点4	节点类型	原节点编号
1	0	2	9	0	0	0	1
2	1	3	0	0	0	0	2
3	2	4	11	0	0	0	3
4	3	5	0	0	0	0	4
5	4	6	0	0	0	0	5
6	5	7	0	0	0	0	6
7	6	8	0	0	0	0	7
8	7	0	0	0	0	0	8
9	2	10	0	0	0	0	9
10	5	8	0	0	0	0	10
11	4	6	0	0	0	0	11

参考资料

- PuLP 2.0: https://pypi.org/project/PuLP/

谢谢