Kernel调度算法及架构设计

Version History

Version	Update Log	Date	Author
v0.3	与DE/DV/Amos做了review和算法讲解，基本可用的初版完成	05/18	Yingnan Zhang
v0.31	修正/补充了关于动态图调度的处理方式	05/23	Yingnan Zhang
v0.4	修正了与SIP和Amos交互的章节的信息	07/01	Yingnan Zhang

Open Questions

问题	影响	需要谁
AOT算子和JIT算子会同时存在么?	如果会，则需要考虑如何构建合理的优先级策略？（初始化offline prio，动态更新offline prio，计算online prio的策略）	James确认
L2B地址如果是Amos分配，在SIP调度的时候，如何把分好的地址传给SIP?	1. 走MD寄存器，这个简单，Amos放在TD的里，HWS去memory上取得 2. 走sip param？是否可以让软件和Amos约定好分配地址放置的位置，通过block idx，supervisor thread主动去L3上获取其L2B的分配地址?	James & Amos
对于动态图的算子，如何保证从HDR里出来的顺序跟下发序相同?		
VC 动态分配		

答疑

Q1：

Q: 如果不能同时做存储和计算的资源分配，先做哪个无所谓

A：不是的。首先，如果所有资源的分配能在非常短的时间窗口做完，那肯定是即分即用是最好的选择。**但是**，只要不是硬化算法，不可能在短时间（100ns以内）做完，因此在更长的时间窗口下，进入us级别，就接近kernel的执行时间量级，那计算和存储资源做分配时的解空间是会剧烈变化的，无法根据一个相对的稳态计算出较优解。

因此在必须把计算资源利用率提高的大目标下，通过拆分资源分配到不同的pipeline，且先做资源分配耗时长的，且只做短期内会被执行的任务的分配，再利用硬件快速完成计算资源的分配。应该可以提高优解的概率。

Terminology

名词	全称	解释
Kernel	DAG上的一个节点，task也是一样意思	
DAG	Direct Async Graph 有向无环图 （executable - 执行图）	

Overview

SIP调度器的目标是将任务发射到符合条件的SIP上去执行。

其本身的机制是一个loop engine，不断地从上游接收已经解除依赖的新任务（后文task，kernel是可相互代替），并实时监控任务状态的更新，资源状态的更新，以最快速度尝试将任务发射到合适的SIP上。

在这个过程中关键的三个动作为：a. 计算优先级，b. 任务选择，c. 资源匹配。

调度器 计算优先级

会根据一定规则将尝试将选择的任务匹配适合的SIP，如果匹配成功，则会产生一次成功的调度。如果**所有**任务没有任何匹配成功，则为一次失败的调度尝试。

无论成功与否，调度器都会在固定的时间内完成一次尝试，并马上进行下一次调度尝试（loop engine）

Fundamental Concepts

Task 优先级分类

为了更好的做任务选择，将Task根据优先级和算法计算的信息，每次调度会分类成3种优先级的task：

Reserve-SIP Task(R-Task) > Prioritized Task(P-Task) > Opportunistic Task(O-Task)

调度器会按照上述顺序，分级对task进行资源匹配

Offline Priority vs Online Priority

Offline Priority为Task的初始优先级，是SIP调度器接到该task时候的优先级。Online Priority是调度器中算法动态生成的。每轮调度尝试都会计算最新的值

Kernel 拆分方式

每个kernel任务都会表达成：<block_num, thread_num> 形式

thread_num代表一个block内，需要**同时**启动多少个SIP，block_num代表这样需要启动多少次这样的block

详见《任务规模描述及SIP拆分和启动信息生成》

Promotion & Reserved SIP

因每个kernel的thread_num可能不同，会出现需求SIP资源越少的kernel越快占用SIP，导致其他一些kernel无法及时被调度。因此，调度器中引入了promotion机制，来确保在满足一定条件下（eg. 失败的次数超过一定阈值后），则会从P-task升级到R-task，并且预留SIP，保证当预留的SIP都free出来后，可以调度一次该R-task的block

Reserved SIP

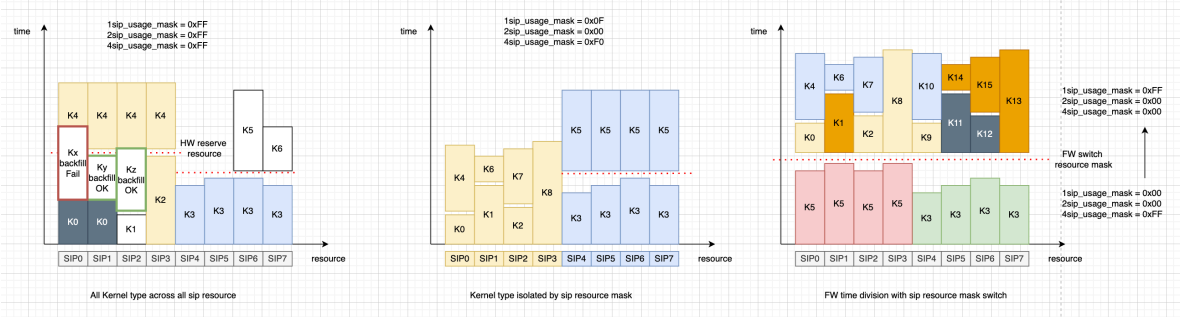
Backfilling

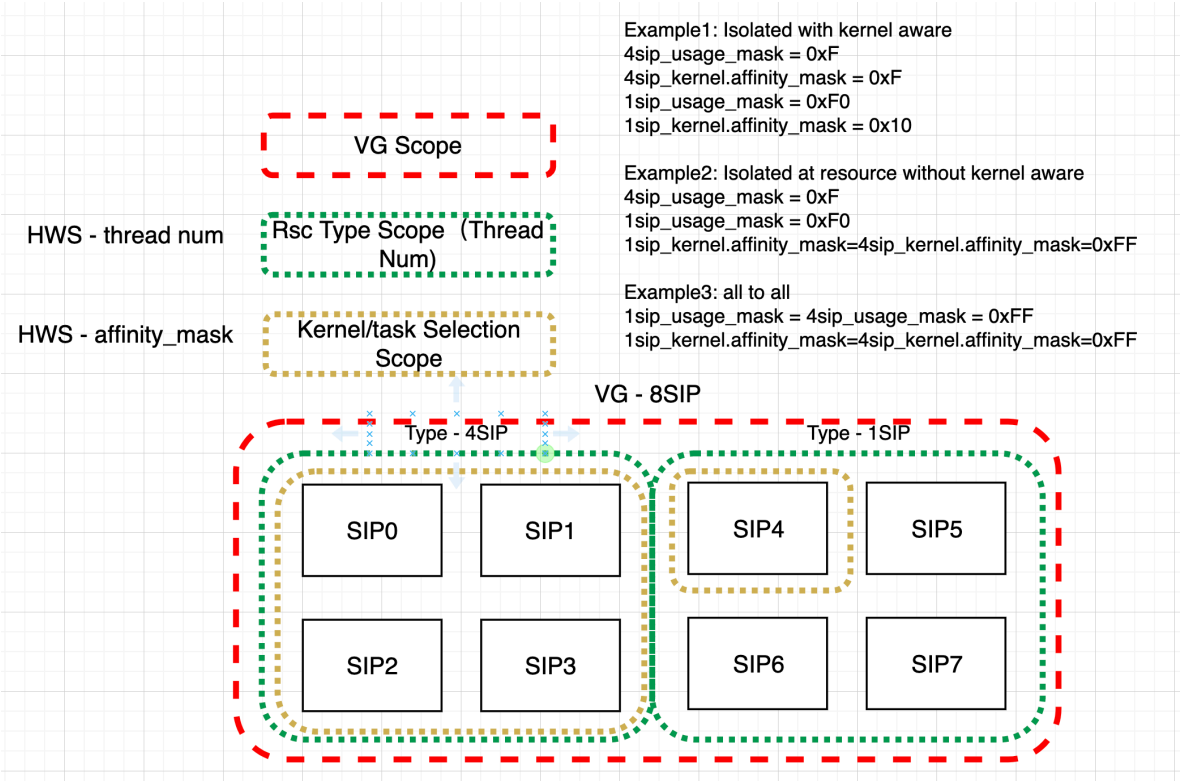
回填是一种因为Promotion引入预留SIP的机制后，因可能预留的SIP部分是free的，若完全不允许任何kernel使用，可能出现很明显的SIP idle，浪费SIP资源。因此，引入其他非R-task在满足一定条件下（eg. 通过偷用预留SIP中idle SIP的时间，当前kernel的任务的一次执行结束的时刻依然早于R-task预期能开始的时间），去偷用被预留但闲置的SIP。将非R-task投机执行在预留的SIP上的动作，称之为backfilling

HWS.sip_usage_mask vs task.sip_affinity_mask

对于SIP的资源匹配，有两个概念引入：

- SIP调度器中对于不同thread_num的kernel类型，进行SIP的分类控制，即kernel只能使用对应<thread_num>_sip_usage_mask所允许使用的SIP。eg. Kernel<N,4>只能使用4_sip_usage_mask中使能的SIP。
- 软件可以定义每个kernel希望能使用的SIP，sip_affinity_mask。
- 最终每个kernel能使用的SIP是同时满足两个维度的定义：usage_mask & sip_affinity_mask





Feature List

Baseline

- 存储最多32（之后可以调，不能低于16，决定了O-task Pool的大小）个kernel的信息
- 支持的最大DAG并行数量为24（不能低于16，决定了P-task Pool的大小）
- 支持Kernel Thread 的数量为1/2/3/4/6/8/9/12/16
  - （）中的是harvest 版本的SIP数量
- 支持根据TDP，从SM读取TD中调度相关的信息，加载进HWS。
  - 单个task trigger 读动作
  - 信息的位置可配（offset to TDP，size）
- 支持根据Kernel信息和调度决策，自动生成需要动态修改的SIP启动MD寄存器的位域（并提供位置可配置的能力）
  - thread_idx
  - Block_idx
- HWS里需要维护SIP 的实时idle（pre-idle）
- 根据HRM-SIP的状态和SIP返回的状态，更新SIP的idle和pre-idle的状态

- 从HRM-SIP查询SIP（shadow配置）的状态，并结合HWS内部维护的SIP状态，向HRM申请对应SIP的配置权
- 维护 dag图的finish kernel count
- 支持记录SIP当前执行kernel的预计完成时间，并实时进行更新，以配合backfilling功能
    - launch时初始化
    - kernel结束时清零并自动开始下一次kernel的count down（如果已经shadow launch了）
    - **uint32_t remain_exec_time[32] (cycles, or unit to be configured)**
- 算法相关：
    - 只支持一种Multi-DAG的OnlinePriority优先级计算方法
    - 支持两种任务分类根据优先级排序功能（P-task, O-task）
    - 基于kernel允许使用SIP资源与实时资源状态的匹配，做调度决策
    - 在满足一定条件下，支持非R-task的任务backfilling 被reserved SIP
    - 仅支持R-task Pool=1（或者是一个cluster一个？）
    - 支持P-task 任务promotion机制，升级到R-task，并对SIP资源进行预留
    - 做资源匹配时，是优先grant P/O-task Pool中的最高优先级任务 或 优先grant 可以将当前cluster内的剩余SIP用满的任务（eg. Cluster0中 SIP0-3是idle的，是优先launch一个thread=2 Priority = 100的P-task还是thread=4的，Priority=90的P-task）
- 更新调度后的各kernel和sip的信息
    - 包括fail times
    - sip的预计完成时间
    - sip的idle/pre-idle状态更新
    - dag图的finish kernel count

Advanced

- HWS 自动判断SIP的pre-idle的状态
    - 与SIP主动通知的模式互斥
- ~~AOT算子优先级自动更新？~~
    - ~~更新offline priorioty~~
    - ~~确认pytorch下发的行为~~
- ~~**检测除SIP资源外，**其他资源的匹配情况~~
    - ~~支持L2资源的检测~~
    - ~~Callback Amos机制（仅支持在成功调度kernel的同时进行后续block的callback）~~
- 与Amos配合的机制
    - 返回调度结果（最终结果？）（满则丢，报错）
    - ~~通过MIH异步Callback机制~~
- 算法相关：
    - 主动判断pre-idle的offset（距离EET的时间）调控（寄存器）
    - backfilling的允许的remain execution time的margin调控（寄存器）
    - ~~R-task 数量是否可调？~~

报错：

1. block内的thread使用跨cluster的sip

2.

Capability

sip phy id 0 - 31

xmc ：map(HWS用sip id -> sip phy id)

kernel.affinity_mask -> convert成HWS使用的

支持的thread 类型：

- 每个block内的thread不允许使用跨cluster的sip
- 每个block内的thread_num要小于等于HWS配置的cluster_sip_num的参数
- block仅支持以下几种thread_num: 1/2/(3or4)/(6or8)/(9or12or16)
    - 目的是在各种大小的cluster情况下，能简化复杂度
- 每个block只允许使用连续的SIP，且start sip idx对齐到 pow(2, ceil(log2(thread_num)))的sip id，eg：
    - 目的是在各种大小的cluster情况下，能使任何thread_num都不跨cluster
    - 使用连续sip来保证性能抖动小，且不容易出现碎片
    - thread_num=x=1/2/4/8/16, 则可使用的sip idx为 0~x-1 .. (xn~x(n+1)-1)
    - thread_num=3, 则可使用的sip idx为： 0~2 or 4~6 ..(4n~4n+2).. or 28~30
    - thread_num=6, 则可使用的sip idx为：0~5 or 8~13 .. (8n~8n+5).. or 24~29
    - thread_num=9, 则可使用的sip idx为：0~8 .. (16n~16n+8).. or 16 ~24
    - thread_num=12, 则可使用的sip idx为：0~11 .. (16n~16n+11).. or 16 ~27
- Thread num=1/2/3/4, 可以从cluster的高idx选sip，尽可能把低idx留给大thread

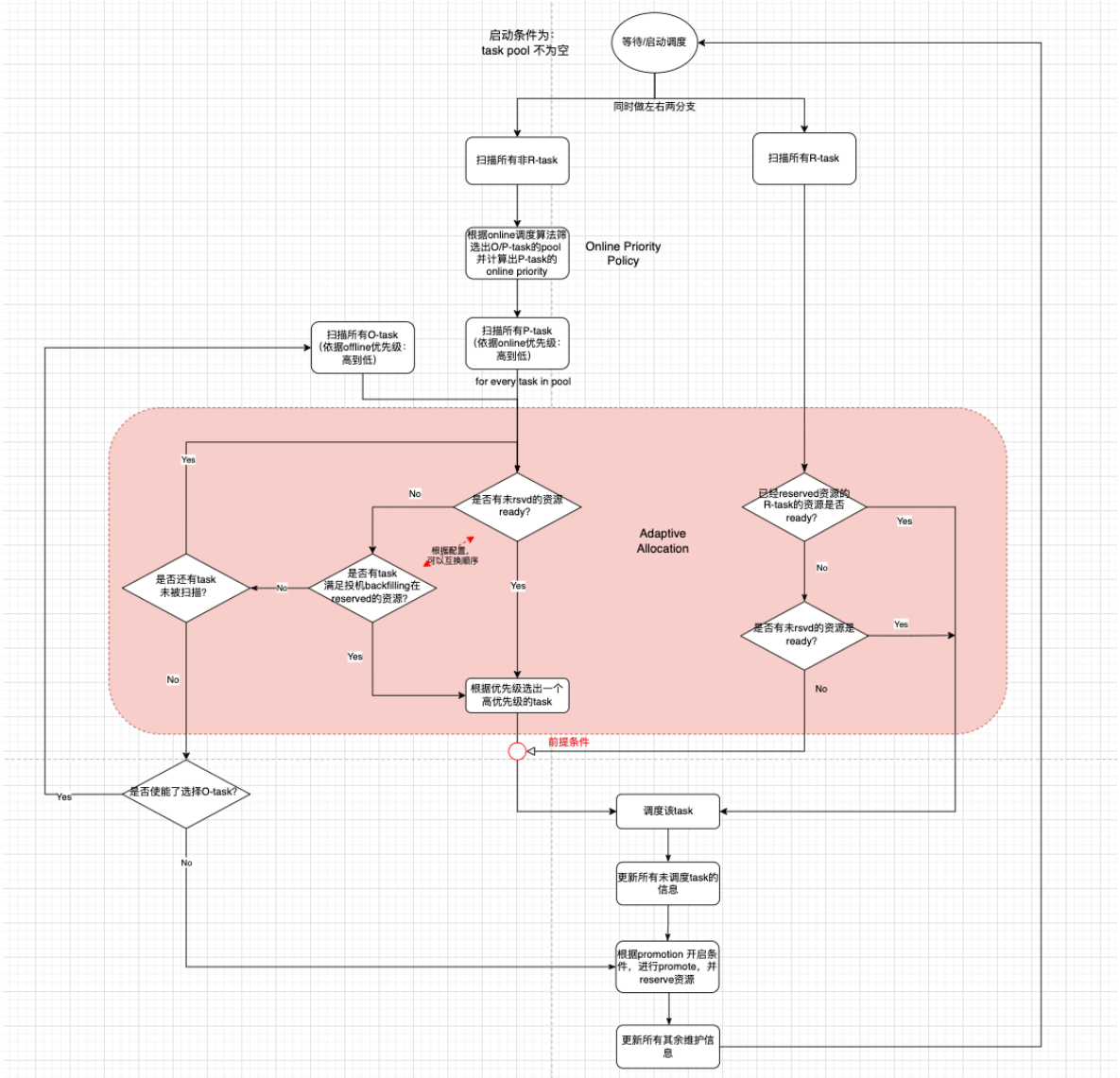
total sip num	sip per cluster	launchKernel support	launchCoopKernel(blk_num>1) support （每种组合在block不变的情况下，thread可以降低到LK支持的）
12	3	<N,1>, <N,2>, <N,3>	max= <4, 3>
	6	<N,1>, <N,2>, <N,3>,<N,4>, <N,6>	max= <2/4, 6/3>
	12	<N,1/2/3/4/6/8/9/12>	max= <2/4, 6/3>
16	4	<N,1>, <N,2>, <N,3>, <N,4>	max=<4, 4>
	8	<N,1/2/3/4/6/8>	max= <2/4, 8/4>
	16	<N,1/2/3/4/6/8/9/12/16>	max= <2/4, 8/4>
24	3	<N,1>, <N,2>, <N,3>	max= <8, 3>
	6	<N,1>, <N,2>, <N,3>,<N,4>, <N,6>	max= <4/8, 6/3>
	12	<N,1/2/3/4/6/8/9/12>	max= <2/4/8, 12/6/3>
32	4	<N,1>, <N,2>, <N,3>, <N,4>	max=<4, 4>
	8	<N,1/2/3/4/6/8>	max= <4/8, 8/4>
	16	<N,1/2/3/4/6/8/9/12/16>	max= <2/4/8, 16/8/4>

Performance Requirement

1. 每次调度小于50cycle（包含构造MDMA描述符，读取SM的时间）

调度算法设计

调度流程



算法代码原型

几点需要注意的：

1. 不推荐对于一个context(一个HWS处理的范围)，不建议混合动态图和静态图两种图混合的kernel并行调度（因为动态图的kernel的原始信息软件可能给不全，比如total_kernel_cnt，on_cp，offline_priority都不准确，因此计算出来的online_priority也不会准确），但如果真出现混合调度使用的情况下，只保证动态图的kernel能正常跑完，不会出现卡死等功能性问题，但整体的调度性能可能无法达到最优
2. 当只调度动态图时，online_priority则直接使用offline_priority。
3. FW要在initDAGprop()中标注DAG是否为动态图，以方便让硬件做判断。

主要数据结构

```
//each HWS serve one SW-context scope
struct kernel_t {
  bool out_path_mdma; //from TD，是否要直接发送给MDMA去launch SIP
  bool out_path_mih; //from TD，是否要把调度结果发送给MIH以上报Amos，与out_path_mdma可以独立配置，但是至少要有一个为1，即3中组合
  bool R_flag;
  uint16_t block_idx;
  uint16_t block_num;//gridDimX(Y=Z=1)
  uint16_t thread_num; //num of sip need to launch together for one block, BlockDimX(Y=Z=1)
  uint16_t dag_tag; //1. orignal SW dag_tag. 2. Amos Mapped 32 -> 16 bit, unique is must!
  uint32_t offline_priority;
  uint32_t online_priority; //remain_task_pert
  uint32_t sip_alloc_affinity_mask; //SW/FW给的，硬件只会读，不会修改该信息
  uint32_t estimate_exec_time; //from SW's cost model
  uint8_t fail_times;
  uint32_t age; //only a representation, use RS_age_matrix in real HW
  bool on_cp; //kernel是否在critical path上，也是SW给的
  bool coop_kernel; //if this is a CooperativeKernel，如果是，需要满足 block_num*thread_num <= total_sip_num，且thread_num <= cluster_max_sip_num
  //Design In，DV pending
  uint16_t block_trigger_cnt; //from FW，HWS read from TD every success launch
};
struct cfg_t {
  bool USE_RSVD_SIP_FIRST; //whether to select Reserved-But-Free SIP to launch first. 0: select non-reserved sip first if found, 1: select reserved-sip first if found
  bool SIP_FILLUP_TASK_FIRST; //whether prioritize a P/O-task which could use all free sips in a VG. 0: follow normal priority, 1: prioritize sip-fillup task
  uint8_t CLUSTER_SIP_NUM;

}

struct HWS_t {
  cfg_t CFG;
  kernel_t RS[32]; //32 RS entry
  uint32_t RS_age_matrix[32]; //记录每个RS的entry之间的先后顺序
  uint32_t dag_total_kernel_cnt[32]; //FW 在接到一个新的DAG时，将DAG的信息通过寄存器配置进来。静态图尽量传递SW给的信息，动态图如果拿不到，给一个默认值即可。
  uint32_t dag_finish_kernel_cnt[32]; //upto 32 DAG at same time
  //DAG_offline_priority是HWS里一个存放每个DAG所属的kernel按照offline_priority优先级从高到低,最高K个kernel，每次HDR进来新的Kernel，都需要更新此数据
  kernel_t DAG_(top_)offline_priority[32]; //upto 32 DAG，highest offline_priority per DAG
  uint32_t DAG_cp_priority[32]; //对应DAG 在Critical Path上的priority，每次DAG进新的kernel，若kernel的on_cp=1，则更新
  uint32_t sip_idle_mask; //real idle, 1: free. 0: busy
```

```
  uint32_t sip_pre_idle_mask; //1: pre_idle(close to finish), 0: busy
  uint32_t reserved_sip_bm; //所有R-task产生的Reserved的SIP的合集

  uint32_t 1sip_grp_work_info[32];//以thread_num分类的，sip执行时间的信息。 eg.
1sip_grp_work_info[0] : sip0 remain_exec_time, 2sip_grp_work_info[0]: //sip0&1:
remain_exec_time,
  uint32_t 2sip_grp_work_info[16];
  uint32_t 3_4sip_grp_work_info[8];
  uint32_t 6_8sip_grp_work_info[4];
  uint32_t 9_12_16sip_grp_work_info[2];

  uint32_t reserve_sip_kernel_thd_num[32]; //每个reserved sip是被thread_num多大的kernel
预约

  //usage_mask group:
  uint32_t usage_mask_1sip;
  uint32_t usage_mask_2sip;
  uint32_t usage_mask_3_4_sip; //3or4sip share same usage_mask，因为usage_mask本身是为了
资源类型隔离使用，3/4其实是一类大小，区分3or4也无法真正达到隔离意义
  uint32_t usage_mask_6_8_sip; //原因同上
  uint32_t usage_mask_9_12_16_sip; //原因同上
}

//Instances:
HWS_t HWS;//one of HWS instance
```

主逻辑框架

```
//HWS (a polling engine)
while(!RS.empty() && (HWS.ready_sip_bitmask != 0x0)) { //assume 1 in sip_bitmask
means corresponding SIP can accept new launch
  prepareTaskPool(input=RS, output=[RTP,PTP,OTP])

  //sort task pool
  sort(target=RTP, rule=online_priority_H2L); //RTP probably only allow 1 outstanding
  sort(target=PTP, rule=online_priority_H2L); //
  sort(target=OTP, rule=offline_priority_H2L);

  //try for RTP
  adaptive_allocation(input=RTP, output=RES);

  (task_to_launch, sip_to_launch_bm) = RES.pop_front();

  //update kernel info for selected R-task
```

```
    if (task_to_launch == null) {
      //try for P-task
      adaptive_allocation(input=PTP, output=RES);
      sip_fillup_res = null;
      if(HWS.CFG.SIP_FILLUP_TASK_FIRST) {
        sip_fillup_res = found_first_fillup_task(input=RES);
      }
      if(sip_fillup_res != null) {
        (task_to_launch, sip_to_launch_bm) = sip_fillup_res;
      } else {
        (task_to_launch, sip_to_launch_bm) = RES.pop_front();
      }
      if (task_to_launch == null) {
        //try for O-task
        adaptive_allocation(input=OTP, output=RES);
        if(HWS.CFG.SIP_FILLUP_TASK_FIRST) {
          sip_fillup_res = found_first_fillup_task(input=RES);
        }
        if(sip_fillup_res != null) {
          (task_to_launch, sip_to_launch_bm) = sip_fillup_res;
        } else {
          (task_to_launch, sip_to_launch_bm) = RES.pop_front();
        }
      }
    }
    if(task_to_launch != null) {
      (RS[task_to_launch].R_flag, RS[task_to_launch].reserved_sip_mask) = (false, 0);
      HWS.reserved_sip_bm &= ~RS[task_to_launch].reserved_sip;
      RS[task_to_launch].fillup_task = false; //clear flag incase
      //report final result:
      ConstructMIHforAmos(task_to_launch, sip_to_launch_bm);
      //launch SIP cmd:
      thread_idx=0;
      //2nd_rsc_addr = RS[task_to_launch].2nd_rsc_queue.pop_front();
      if(RS[task_to_launch].out_path_mdma) {
        for(sip_idx in sip_to_launch_bm) {
          ConstructCMDforMDMA(sip_idx, thread_idx++, RS[task_to_launch].block_idx/*,
  2nd_rsc_addr*/);
        }
      }
      if(RS[task_to_launch].out_path_mih) {
        ConstructMsgforAmos(task_to_launch, sip_to_launch_bm);
      }

      RS[task_to_launch].block_idx++;
      //RS[task_to_launch].2nd_rsc_used_block_cnt[belong_vg(sip_to_launch_bm)]++;

      //if need callback Amos for 2nd rsc allocation
      //if(RS[task_to_launch].enable_2nd_rsc_alloc &&
```

```
    //   sum(RS[task_to_launch].2nd_rsc_requested_blk_cnt+ xxx ) ==
RS[task_to_launch].block_num) {
    //  ConstructMsgforAmos(task_to_launch, sip_to_launch_bm);
    //}


    //completely finish all blocks for this kernel
    if(RS[task_to_launch].block_idx == RS[task_to_launch].block_num) {

      //update kernel fail_times
      inc_kernel_fail_cnt_upon_launch(task_to_launch);

      dag_finish_kernel_cnt[RS[task_to_launch].dag_tag]++; //注意，如果
finish_kernel_cnt已经达到最大值，就保持在最大值即可，不要overflow
      //finally, remove the task from HWS
      RS.remove(task_to_launch);
    }
  }

  //update SIP usage status and Remain-Execution-Exec-Time
  update_sip_work_info(task_to_launch, sip_to_launch_bm);

  promote_and_reserve_sip(task_to_launch);


}//while
```

任务分类

```
void prepareTaskPool(input=RS, output=[RTP,PTP,OTP]) {
  //filter kernels in RS which will input to OPA
  for(each kernel in RS) {
    //filter
    if(kernel.R_flag) { //R_flag是HWS自行维护
      RTP.push_back(kernel);
    } else {
      ok_to_try = (RS[kernel].launched_block_cnt < kernel.block_trigger_cnt);
    }
    if(ok_to_try) {
      if(check_DAG_topK_priority(kernel)) {
        kernel.online_priority = cal_online_priority(kernel);
        PTP.push_back(kernel);
      } else {
        OTP.push_back(kernel);
      }
    }
```

```
    }
  }
```

优先级策略

Offline Prioritizing Task（offline_priority)

Scorpio中使用的优先级策略是 “基于一种对静态DAG图的优先级表示方法” 之上的一种衍生算法，用于适配动态执行过程中多DAG并行的场景。

推荐使用（软件如果想尝试其他的优先级表示方法也是OK的）优先级表示方法为upward rank（倒序排列），即offline_priority的来源（**注意**：offline_priority = coeff * upward_rank，这里coeff可以为1，也可以是某种归一化的系数，只要保证一个图里每个节点通过upward rank体现出来的优先级关系与用offline_priority体现出来的优先级**完全一致即可**。）。

在计算优先级之前，需要知道每个节点的computation cost（例如2D+1D时间+DMA时间，具体cost model由软件来迭代）以及与其相关联的节点间的communication cost（eg. 跨cluster的数据传递经过L3，cluster内的数据交互可以只经过L2，会有区别）。

然后从一个图的exit node（结束节点）开始反向遍历每个节点，每个节点的优先级(RANK_UP[ni])是该节点(ni)自己的compute cost(wi) 加上 所有该节点的后续依赖节点(nj)的RANK_UP[nj]加上到该节点(ni)的communicate cost(ci,j)之和 中 最大的值：

Prioritizing tasks [edit]

In the first phase each task is given a priority. The priority of each task n_i is usually designated to be its "upward rank" which is defined recursively as follows

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j))$$

where n_j represents the j^{th} task, $\overline{w_i}$ is an average computation cost of job i among all the processor, $succ(n_i)$ is the set of all jobs that immediately depend on task n_i, and $\overline{c_{i,j}}$ is the average communication cost of the variables transferred between jobs n_i and n_j between all pairs of workers. Note that the computation of $rank_u(n_i)$ depends on the computation of the rank of all its children. The upward rank is meant to represent the expected distance of any task from the end of the computation. For averaged quantities like $\overline{w_i}$ different averages may provide different results.[2]

reference link： https://en.wikipedia.org/wiki/Heterogeneous_Earliest_Finish_Time

以上述方法，倒序遍历所有DAG中的节点即可得到每个节点的upward rank(也有些paper里叫bottom rank)。

在真实业务中的一张执行图中，每个任务（kernel）可以看做该图（一定是DAG）的一个节点。因所有不同任务之间的数据(即communication cost）交互由软件显式控制，没有统一的衡量方法，为了简化，在计算upward rank时，可以忽略communication cost为0，并在compute cost里加上适当的data movement的开销即可。

在倒序遍历完一张图后，至少会有一个entry node（开始节点） 任务的uprank是最大的，该节点就是critical path的入口。critical path决定了该图执行时间的下限值。从这些最大uprank节点正向再遍历找到产生该节点优先级的子节点(不是所有子节点都贡献了该节点的优先级，只有（ci,j+rank[nj]）最大的那个nj节点才是rank[ni]的产生者），并递归遍历直到exit node，路径上的每个节点都叫在critical path的节点。即on_cp位域的由来。一个图可以有多条cp。

同时，一张图一共有多少个节点，也是在上述优先级计算过程中可以很容易获得的，即dag_total_kernel_cnt

Example

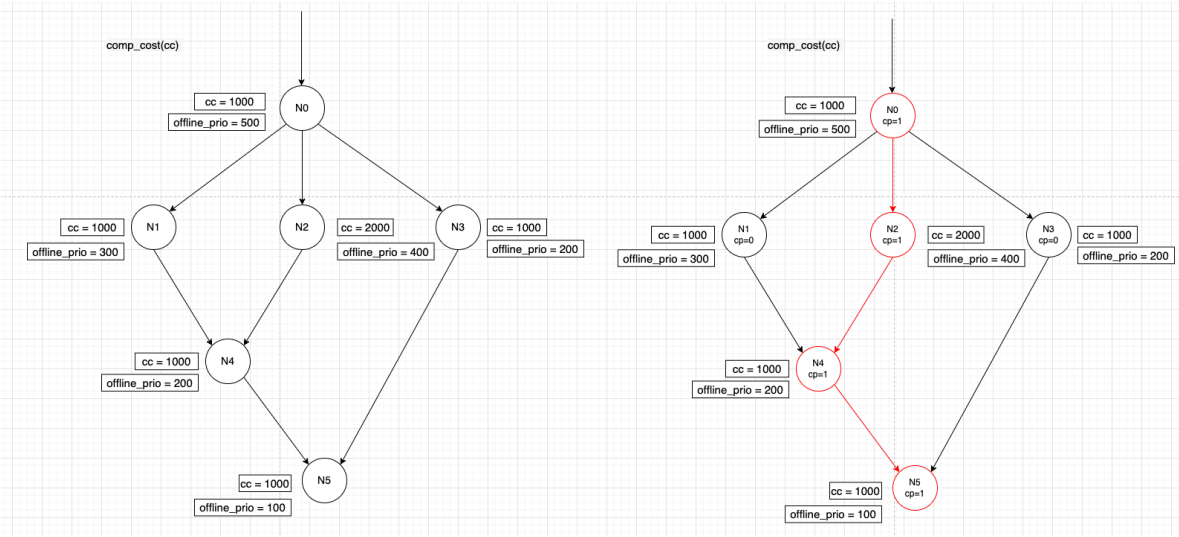
以下图的一段执行图为例，阐释上述算法的的计算方法。

N0和N5分别为该图的entry node和exit node。每个node的compute cost如图所示（communication cost均假设为0），有1000或2000两种。根据算法（这里以offline_priority = 1/10 * upward_rank为计算公式展示），倒序从exit node开始做upward rank的计算，因N5节点没有任何子节点，所以ranku(N5) = cc = 1000, offline_priority(N5)=100,

N4节点只有N5一个子节点，因此offline_priority(N4) = 2000/10 = 200.

以此类推，ranku(N0) = cc(N0) + max(ranku(N1), ranku(N2), ranku(N3)) = 1000 + ranku(N2) = 5000, 所以offline_priority(N0)=500。到此为止，所有节点的offline_priority已经全部计算完成，如下图左侧DAG标注所示

下面开始检查哪些节点会在critical path上。从upward_rank最大的node做downward的扫描，逐一找到是哪一(几)个子节点产生的当前节点的upward_rank，这些子节点就是到达该节点的最关键路径。例如N0的upward_rank来自N2，因此N2.cp=1。N2的upward_rank来自于N4，因此N4.cp=1。最终如下图右侧DAG标记所示，所有红色节点都是在critical path的节点。



Online Prioritizing Task（online_priority）

在每个节点的offline_priority和on_cp 信息以及其DAG的dag_total_kernel_cnt这些信息准备好后，我们需要在执行过程中找到如何处理多张图各自的offline_priorty，来找到合适的调度顺序。

原始想法来源Paper Reference：《Fairness resource sharing for dynamic workflow scheduling on Heterogeneous Systems》

目标有2个：

- 执行图完成的任务越多，越接近结束，越应该保证其有较高的优先级来保证其End2End latency（paper中的术语为Makespan）不会随着新DAG的到来而被delay
- 对执行图越关键的节点(比如在cp上的节点，或优先级比当前cp上的节点还高），越需要尽快执行

因此，online_priority的计算首先会挑选每个DAG里最高offline_priority的节点进行横向比较，再通过以下两个系数来横向衡量：

- kernel_critical_ratio来反映一个节点相对于其所在DAG图的最新的在cp上的节点的优先级的关系（eg. 当前时间，kernelA.on_cp = 1, kernelA.offline_priority=100, kernelB.on_cp = 0, kernelB.offline_priority = 200, 就说明cp上的kernel已经被执行的很多了，原本不在cp上的kernelB已经成为了影响该DAG图完成的最紧迫的节点）。
- remain_task_percentage来反映该图完成的比例，当不同DAG的kernel都是其所在DAG中最关键的节点且急迫性接近时，更接近完成的DAG的kernel享有更高执行权。

最终通过一个scale_ratio和curve_ratio的查找表，来给软件调控上述两个系数的占比影响。具体实现见下面的代码：

```
bool check_DAG_topK_priority(input=kernel) {
  if(kernel in HWS.DAG_offline_priority[kernel.dag_tag]) {
    return true;
  } else {
    return false;
  }
}

uint32_t cal_online_priority(input=kernel) {
  //double online_priority = kernel_critical_ratio / map(dag_remain_task_percentage, curve_scale_ratio)
  //where:
  //kernel_critical_ratio = kernel.offline_priority * 1/DAG_cp_priority[kernel.dag_tag]; 用cp_prirority作为reference，反映该kernel与之的相对关系。当ratio越小（eg. <1时），该kernel越没有那么critical，ratio越大(eg. >=1时)，代表该kernel越紧急，已经或马上就会影响DAG图执行的E2E时间。在横向比较不同DAG的kernel的online_priority时，用该部分去反映节点的紧急程度；越紧急，越可以在多DAG横向比较中，取得优先执行权。
  //map(dag_remain_task_percentage, curv_ratio): 先计算出:
  //dag_remain_task_percentage = (dag_total_kernel_cnt[kernel.dag_tag] - dag_finish_kernel_cnt[kernel.dag_tag]) / dag_total_kernel_cnt[kernel.dag_tag]，在根据5%? 档位（共20档）来查找配置表，找到对应的curv_ratio
  //curve_scale_ratio 是个分数（分子，分母可以分别配置，或者直接配置浮点数）。其有两个目的(scale_ratio and curve_raito)： 1. 拉开优先级的具体数值差，放大系数scale_ratio作用。 2. 若dag_remain_task_percentage随着任务完成，占比过快放大，影响了critical_ratio的影响，也可以通过curve_ratio配置成线性（或者任意非连续函数的点位）
  if(RS[kernel].dynamic_dag == 1) {
    //如果为动态图，则直接使用SW传进来的原始（offline）优先级。软件想要更好的效果，可以更合理的生成offline_priority
    return RS[kernel].offline_priority;
  } else {
    double kernel_critical_ratio = kernel.offline_priority * 1/DAG_cp_priority[kernel.dag_tag];
    double dag_remain_task_percentage = 1 - dag_finish_kernel_cnt[kernel.dag_tag] / dag_total_kernel_cnt[kernel.dag_tag];
    HWS.CFG.CURVE_SCALE_RATIO_RANK[32] = 100*{1.0/*100~96.875%*/,2,3 ...., 32/*0~3.125%*/}; //只是示例， 放大系数100，remain_task_percentage的倒数为1~32，最终配置为：100，200，300 ~ 3200。这是一种线性倒数，理论上应该有一定非线性度，eg. 1，1.1，1.3，1.5，2，...
    int scale_rank_index = convert dag_remain_task_percentage to  32 ranks
```

```
    return ceil(kernel_critical_ratio * CURVE_SCALE_RATIO_RANK[scale_rank_index]); //
举例：若kernel_critical_ratio = 0.85, remain_pct = 10%, 则online_priority = 100* 0.85 *
29(落在 3/32 ~ 4/32之间) = 2465
  }
}
```

动态资源匹配

```
void adaptive_allocation(input=TP, output=RES) {
  for(task in TP) {
    usable_ready_sip_bm = get_usable_ready_sip(task);
    non_rsv_ready_sip_bm = usable_ready_sip_bm & ~HWS.reserved_sip_bm/*1:rsvd,0:not
yet*/;
    rsv_ready_sip_bm = usable_ready_sip_bm & HWS.reserved_sip_bm;
    //R-task case:
    //TODO: fast pass case? or don't need fast pass for Scorpio?
    // time 0: task0.reserved_sip = 0xF(sip0-sip3), ready_sip_mask = 0x3;
    // time 1: ready_sip_mask = 0xF0;
    // time 10: reserved_sip ?= ready_sip_mask ?= 0xF
    if(task.R_flag && task.reserved_sip != 0) {
      if(task.reserved_sip & HWS.ready_sip_bitmask) == task.reserved_sip) {
        optimal_sip_bitmask = task.reserved_sip;
        RES.push_back(tuple(task, optimal_sip_bitmask));
      }
      else if(check_ready_sip_enough(task, usable_ready_sip_bm)) {
        optimal_sip_bitmask = sel_best_sip_to_launch(task, usable_ready_sip_bm);
        RES.push_back(tuple(task, optimal_sip_bitmask));
      }
    }
    //other task case:
    else {
      //check non rsv free sip:
      if(check_ready_sip_enough(task, non_rsv_ready_sip_bm)) {
        optimal_sip_bitmask = sel_best_sip_to_launch(task, non_rsv_ready_sip_bm);
        task.fillup_found = (optimal_sip_bitmask == HWS.ready_sip_bitmask); //or use
(optimal_sip_bitmask==non_rsv_ready_sip_bm) ?
        option1 = tuple(task, optimal_sip_bitmask));
      }
      //check for reserved free sip
      //HWS.CFG.backfilling_enable always 1
      if(HWS.reserved_sip_bm != 0) {
        rsv_ready_sip_ok_to_backfill_bm = filter_rsv_sip_to_backfill(task.thread_num,
rsv_ready_sip_bm);
        if(check_ready_sip_enough(task, rsv_ready_sip_ok_to_backfill_bm)) {
          optimal_sip_bitmask = sel_best_sip_to_launch(task,
rsv_ready_sip_ok_to_backfill_bm);
          task.fillup_found = (optimal_sip_bitmask == HWS.ready_sip_bitmask);
          option2 = (tuple(task, optimal_sip_bitmask);
```

```
        }
        all_ready_sip_to_backfill_bm = rsv_ready_sip_ok_to_backfill_bm |
non_rsv_ready_sip_bm;
        if(check_ready_sip_enough(task, all_ready_sip_to_backfill_bm)) {
          optimal_sip_bitmask = sel_best_sip_to_launch(task,
all_ready_sip_to_backfill_bm);
          task.fillup_found = (optimal_sip_bitmask == HWS.ready_sip_bitmask);
          option3 = (tuple(task, optimal_sip_bitmask);
        }
      }
      //pick options base on cfg
      if(option1 != null || option2 != null) {
        if(!HWS.CFG.USE_RSVD_SIP_FIRST && option1 != null)
          RES.push_back(option1);
        else if(HWS.CFG.USE_RSVD_SIP_FIRST && option2 != null)
          RES.push_back(option2);
      } else if(option3 != null) { //maybe mixed rsvd and non-rsvd
          RES.push_back(option3);
      }
    }
}

//=================辅助函数=========================
uint32_t get_allowed_sip_to_use(task) {
  allowed_sip_bm = (task.sip_alloc_affinity_mask & sip_usage_mask[task.thread_num]);
}
uint32_t get_usable_ready_sip(task) {
  allowed_sip_bm = get_allowed_sip_to_use(task);
  //2nd_rsc_enabled_sip_bm = get_2nd_rsc_enabled_sip(task);
  return (allowed_sip_bm & /*2nd_rsc_enabled_sip_bm &*/ HWS.ready_sip_bitmask);
}

//check if target_sip_bitmask has enough kernel_sip_num to launch, following certain
rules:
//1-SIP:
bool check_ready_sip_enough(task, target_sip_bitmask) {
  //if can find any valid sip
  //补充3/6/9/12的规则，应该跟之前的约束相同
  int found_block_cnt = 0;
  num_block_to_check = task.coop_kernel? task.block_num : 1; //see if need to found
multiple blocks
  for(int i=0; i<num_block_to_check; i++) {
    sip_mask_target = (0x1 << task.thread_num) - 1; //1: 0x1, 2: 0x3, 3:0x7, 4: 0xF,
6: 0x3F, 8: 0xFF, 9: 0x1FF, 12: 0xFFF, 16: 0xFFFF
    switch(task.thread_num) {
      case 1:
        for(int i=0; i < 32; i++) {
          if(((target_sip_bitmask >> i) & sip_mask_target) == sip_mask_target) {
            found_block_cnt++;
            target_sip_bitmask ^= (sip_mask_target << i);
```

```
            }
          }
          break;
        case 2: //0/2/4/6/8/..
          for(int i=0; i < 16; i++) {
            if(((target_sip_bitmask >> i*2) & sip_mask_target) == sip_mask_target) {
              found_block_cnt++;
              target_sip_bitmask ^= (sip_mask_target << i*2);
            }
          }
          break;
        case 3,4: //0/4/8/12/..
          for(int i=0; i < 8; i++) {
            if(((target_sip_bitmask >> i*4) & sip_mask_target) == sip_mask_target) {
              found_block_cnt++;
              target_sip_bitmask ^= (sip_mask_target << i*4);
            }
          }
          break;
        case 6,8: //0/8/16/24
          for(int i=0; i < 4; i++) {
            if(((target_sip_bitmask >> i*8) & sip_mask_target) == sip_mask_target) {
              found_block_cnt++;
              target_sip_bitmask ^= (sip_mask_target << i*8);
            }
          }
          break;
        case 9,12,16: //0/16/
          for(int i=0; i < 2; i++) {
            if(((target_sip_bitmask >> i*16) & sip_mask_target) == sip_mask_target) {
              found_block_cnt++;
              target_sip_bitmask ^= (sip_mask_target << i*16);
            }
          }
          break;
        default:
          return false;
      }
    }
    if(found_block_cnt == num_block_to_check) {
      return true;
    } else {
      return false;
    }
  }

  uint32_t sel_best_sip_to_launch(task, target_sip_bitmask) {
    //注意下面两个sip_mask不是互斥的:
    uint32_t target_real_free_sip_mask = target_sip_rsc_mask & HWS.sip_idle_mask;
    uint32_t target_pre_idle_sip_mask = target_sip_rsc_mask & HWS.sip_pre_idle_mask;
```

```
  //注意：这里的代码没有考虑双die亲和问题，在硬件实现时，可以优先使用local die的，也可做RR方式轮番调度双die的SIP

  int found_block_cnt = 0;
  uint32_t optimal_sip_to_pick = 0x0;
  num_block_to_search = task.coop_kernel? task.block_num : 1; //see if need to found multiple blocks
  for(int i=0; i<num_block_to_search; i++) {
    sip_mask_target = (0x1 << task.thread_num) - 1;
    switch(task.thread_num) {
      //1/2/3/4 倒序搜索，给6/9/12这种相对大的kernel但是未用满所有cluster资源的，更多的成功机会（因为他们只会用sip0选起）
      case 1:
        for(int i=31; i >=0; i++) {
          if(((target_real_free_sip_mask >> i) & sip_mask_target) == sip_mask_target) {
            optimal_sip_to_pick |= (sip_mask_target << i);
            found_block_cnt++;
            target_real_free_sip_mask &= ~(sip_mask_target << i);
            target_pre_idle_sip_mask &= ~(sip_mask_target << i); //防止下一次循环从pre_idle中选了同一个sip，因为real_free_sip_mask和pre_idle_sip_mask并非互斥
            break;
          }
        }
        if(HWS.CFG.ENABLE_EARLY_LAUNCH) {
          for(int i=31; i >=0 i++) {
            if(((target_pre_idle_sip_mask >> i) & sip_mask_target) == sip_mask_target) {
              optimal_sip_to_pick |= (sip_mask_target << i);
              found_block_cnt++;
              target_pre_idle_sip_mask &= ~(sip_mask_target << i); //此处只clear pre_idle即可，因为进入此处，说明real_free_sip_mask没有成功选出，那选择的sip一定是不存在在real_free_sip_mask
              break;
            }
          }
        }
        break;
      case 2:
        for(int i=15; i >=0; i++) {
          if(((target_real_free_sip_mask >> i*2) & sip_mask_target) == sip_mask_target) {
            optimal_sip_to_pick |= (sip_mask_target << i*2);
            found_block_cnt++;
            target_real_free_sip_mask &= ~(sip_mask_target << i*2);
            target_pre_idle_sip_mask &= ~(sip_mask_target << i*2); //防止下一次循环从pre_idle中选了同一个sip，因为real_free_sip_mask和pre_idle_sip_mask并非互斥
            break;
          }
```

```
        }
        if(HWS.CFG.ENABLE_EARLY_LAUNCH) {
          for(int i=15; i >=0; i++) {
            if(((target_pre_idle_sip_mask >> i*2) & sip_mask_target) ==
sip_mask_target) {
              optimal_sip_to_pick |= (sip_mask_target << i*2);
              found_block_cnt++;
              target_pre_idle_sip_mask &= ~(sip_mask_target << i*2); //此处只clear
pre_idle即可，因为进入此处，说明real_free_sip_mask没有成功选出，那选择的sip一定是不存在在
real_free_sip_mask
              break;
            }
          }
        }
        break;
      case 3,4: //正序倒序都行?
        for(int i=7; i >=0; i++) {
          if(((target_real_free_sip_mask >> i*4) & sip_mask_target) ==
sip_mask_target) {
            optimal_sip_to_pick |= (sip_mask_target << i*4);
            found_block_cnt++;
            target_real_free_sip_mask &= ~(sip_mask_target << i*4);
            target_pre_idle_sip_mask &= ~(sip_mask_target << i*4); //防止下一次循环从
pre_idle中选了同一个sip，因为real_free_sip_mask和pre_idle_sip_mask并非互斥
            break;
          }
        }
        if(HWS.CFG.ENABLE_EARLY_LAUNCH) {
          for(int i=7; i >=0; i++) {
            if(((target_pre_idle_sip_mask >> i*4) & sip_mask_target) ==
sip_mask_target) {
              optimal_sip_to_pick |= (sip_mask_target << i*4);
              found_block_cnt++;
              target_pre_idle_sip_mask &= ~(sip_mask_target << i*4); //此处只clear
pre_idle即可，因为进入此处，说明real_free_sip_mask没有成功选出，那选择的sip一定是不存在在
real_free_sip_mask
              break;
            }
          }
        }
        break;
      case 6,8://正序搜索
        for(int i=0; i < 4; i++) {
          if(((target_real_free_sip_mask >> i*8) & sip_mask_target) ==
sip_mask_target) {
            optimal_sip_to_pick |= (sip_mask_target << i*8);
            found_block_cnt++;
            target_real_free_sip_mask &= ~(sip_mask_target << i*8);
            target_pre_idle_sip_mask &= ~(sip_mask_target << i*8); //防止下一次循环从
pre_idle中选了同一个sip，因为real_free_sip_mask和pre_idle_sip_mask并非互斥
```

```
              break;
            }
          }
          if(HWS.CFG.ENABLE_EARLY_LAUNCH) {
            for(int i=0; i < 4; i++) {
              if(((target_pre_idle_sip_mask >> i*8) & sip_mask_target) ==
sip_mask_target) {
                optimal_sip_to_pick |= (sip_mask_target << i*8);
                found_block_cnt++;
                target_pre_idle_sip_mask &= ~(sip_mask_target << i*8); //此处只clear
pre_idle即可，因为进入此处，说明real_free_sip_mask没有成功选出，那选择的sip一定是不存在在
real_free_sip_mask
                break;
              }
            }
          }
          break;
       case 9,12,16://正序搜索
          for(int i=0; i < 2; i++) {
            if(((target_real_free_sip_mask >> i*16) & sip_mask_target) ==
sip_mask_target) {
              optimal_sip_to_pick |= (sip_mask_target << i*16);
              found_block_cnt++;
              target_real_free_sip_mask &= ~(sip_mask_target << i*16);
              target_pre_idle_sip_mask &= ~(sip_mask_target << i*16); //防止下一次循环从
pre_idle中选了同一个sip，因为real_free_sip_mask和pre_idle_sip_mask并非互斥
              break;
            }
          }
          if(HWS.CFG.ENABLE_EARLY_LAUNCH) {
            for(int i=0; i < 2; i++) {
              if(((target_pre_idle_sip_mask >> i*16) & sip_mask_target) ==
sip_mask_target) {
                optimal_sip_to_pick |= (sip_mask_target << i*16);
                found_block_cnt++;
                target_pre_idle_sip_mask &= ~(sip_mask_target << i*16); //此处只clear
pre_idle即可，因为进入此处，说明real_free_sip_mask没有成功选出，那选择的sip一定是不存在在
real_free_sip_mask
                break;
              }
            }
          }
          break;
        default:
          break;
      }
    }
    if(found_block_cnt == num_block_to_search) {
      return optimal_sip_to_pick;
    } else {
```

```
    return 0x0; //if fail to find for all blocks, then pick NONE of them
  }
}

uint32_t filter_rsv_sip_to_backfill(task_to_check, uint32_t target_rsv_sip_rsc_mask)
{
  //prerequisite: target_rsv_sip_rsc_mask must be reserved sip but free sip
  //注意：该函数不检查是否有足够的sip去launch，只是挑选出符合backfill条件的所有sip

  uint32_t steal_sip_time = RS[task_to_check].estimate_exec_time +
(HWS.CFG.BACKFILL_OFFSET[31]? HWS.CFG.BACKFILL_OFFSET[30:0] : -
HWS.CFG.BACKFILL_OFFSET[30:0]); //can be
  //for now, just find any spot to backfill, HW can implement as smallest margin to
use

  uint32_t rsv_sip_rsc_to_backfill = 0x0;
  for(int i=0; i < 32; i++) { //search all sips, if they are reserved-but-idle-sip
    if(((target_rsv_sip_rsc_mask >> i) & uint32_t(1)) == uint32_t(1)) {
      //根据对应free的sip id所属的thread size的类型（1/2/4/8），查看是否有可以插空的free-sip，
注意，当前task_to_check的尺寸不重要，只需要看当前sip被哪个尺寸的kernel所预约
      switch(HWS.reserve_sip_kernel_thd_num[i]) {
        case 2:
          if(HWS.2sip_grp_work_info[i/2]->remain_exec_time - steal_sip_time >= 0) {
            rsv_sip_rsc_to_backfill |= (0x1 << i);
          }
          break;
        case 3,4:
          if(HWS.4sip_grp_work_info[i/4]->remain_exec_time - steal_sip_time >= 0) {
            rsv_sip_rsc_to_backfill |= (0x1 << i);
          }
          break;
        case 6,8:
          if(HWS.8sip_grp_work_info[i/8]->remain_exec_time - steal_sip_time >= 0) {
            rsv_sip_rsc_to_backfill |= (0x1 << i);
          }
          break;
        case 9,12,16:
          if(HWS.16sip_grp_work_info[i/16]->remain_exec_time - steal_sip_time >= 0) {
            rsv_sip_rsc_to_backfill |= (0x1 << i);
          }
          break;
      }
    }
  }

  return rsv_sip_rsc_to_backfill;
}
```

Kernel信息维护与更新

```
void update_info_upon_new_task_arrive(new_task) {
  if(new_task.dag_tag not exist) {
    //其实可以选择2种方式，1. 每个kernel里都带total_kernel_cnt的信息，比较冗余。 2. 是FW用寄存器方式写进来的。推荐第二种。清除也是FW主动做的
    dag_total_kernel_cnt[new_task.dag_tag] = new_task.total_kernel_cnt;
    dag_finish_kernel_cnt[new_task.dag_tag] = 0; //initialization
  }
  //step1: update age relationship among all task in HWS
  for (entry_idx in RS) {
    set(RS_age_matrix[entry_idx].age older than new_task.age);
  }
  //step2: update dag's top priority if new task's priority is higher than any task in the same DAG
  if(/*new_task.dynamic_dag==0 &&  */new_task.offline_priority > HWS.DAG_offline_priority[new_task.dag_tag]) {
    HWS.DAG_offline_priority[new_task.dag_tag] = new_task;
  }
  //else if(new_task.dynamic_dag==1) {
  //  //TODO: for AOT + DynDAG
  //}
  if(new_task.on_cp == 1) {
    HWS.DAG_cp_priority[new_task.dag_tag] = new_task.offline_priority; //一个DAG可能有多个在critical path上的，但是因为在算online_priority时候，比值可以>1，代表可能有更critial的节点在critical path上。
  }
}

void inc_kernel_fail_cnt_upon_launch(input=task_to_launch, input=PTP) {
  for(each task in PTP) {
    //if task's age is older AND task_to_launch is from PTP
    //or task_to_launch is from OTP
    if ((task.age > task_to_launch.age || task_to_launch in OTP))  //task_to_launch not in RTP &&
      task.fail_times++;
  }
}
```

内部SIP状态和相关信息维护

```
//实时更新的状态：
assign HWS.ready_sip_bitmask = HWS.CFG.ENABLE_EARLY_LAUNCH? (HWS.sip_idle_mask | HWS.sip_pre_idle_mask) : HWS.sip_idle_mask;

void update_sip_work_info(input=task_to_launch, input=sip_to_launch_bm) {
  //记录或更新 sip remain_exec_time
```

```
  elapsed_time = current_time - last_schedule_time;

  for sip_idx in 0...32 and i not in {
    if (sip_idx in sip_to_launch_bm) { //sip_to_launch_bm是32bit，每bit代表sip是否被选
择，1：选择，0：未被选
      HWS.1sip_work_info[sip_idx].remain_exec_time =
RS[task_to_launch].estimate_exec_time + (HWS.sip_idle_mask[sip_idx] == 1?
HWS.CFG.LAUNCH_SIP_DELAY : 0); //注意：该EET是kernel的cycle换算成SP的频率下后的sp cycles数
      HWS.sip_work_info_count_down_en[sip_idx] = HWS.sip_idle_mask[sip_idx] == 1? 1 :
0; //如果launch新SIP的时候，SIP已经是idle，则立即开始countdown，如果SIP还在上一个kernel处理中，
则暂不开始countdown，直到接到real-complete
    } else if(HWS.sip_work_info_count_down_en[sip_idx]) { //该count down enable 需要在
接到sip real-complete的时候，置1
      HWS.1sip_work_info[sip_idx].remain_exec_time =
(HWS.1sip_work_info[sip_idx].remain_exec_time >= elapsed_time)?
(HWS.1sip_work_info[sip_idx] - elapsed_time) : 0; //具体硬件实现方式可以有多种： 1. HWS的完
成时间固定（fail/sucess），可以减掉本次调度对应的耗时cycle 2. 硬件每拍更新
    }
  }
  //基于每个单sip的EET时间，刷一下按照2/4(3)/8(6)
  //update for 2-sip group
  for(int i=0; i<16; i++) {
    //idx与sip的规则是：i for sip[i*2: i*2+1], eg. 2sip_grp_work_info[0] collect
sip[0:1] info, 2sip_grp_work_info[4] for sip[8:9]
    HWS.2sip_grp_work_info[i]->remain_exec_time =
std::max(HWS.1sip_grp_work_info[i*2]->remain_exec_time,

HWS.1sip_grp_work_info[i*2+1]->remain_exec_time);
  }
  //update for 3/4-sip group
  for(int i=0; i<8; i++) {
    HWS.4sip_grp_work_info[i]->remain_exec_time =
std::max(HWS.2sip_grp_work_info[i*2]->remain_exec_time,

HWS.2sip_grp_work_info[i*2+1]->remain_exec_time);
  }
  //update for 8-sip group
  for(int i=0; i<4; i++) {
    HWS.8sip_grp_work_info[i]->remain_exec_time =
std::max(HWS.4sip_grp_work_info[i*2]->remain_exec_time,

HWS.4sip_grp_work_info[i*2+1]->remain_exec_time);
  }
  //update for 16-sip group
  for(int i=0; i<2; i++) {
    HWS.16sip_grp_work_info[i]->remain_exec_time =
std::max(HWS.8sip_grp_work_info[i*2]->remain_exec_time,

 HWS.8sip_grp_work_info[i*2+1]->remain_exec_time);
  }
```

```
}


void promote_and_reserve_sip(task_to_launch) {
  for (task in PTP) { //assume task sorted in high->low priority, so first task is
highest priority
    if (RTP.empty() && task != task_to_launch) {
      do_promotion_thrshld = HWS.CFG.PROMOTE_RULE__THRESHOLD == 1 && task.fail_times
>= HWS.CFG.PROMOTE_THRESHOLD;
      if(!task_to_launch.R-flag && task_to_launch != PTP[0]/*if R-task or Top1-P-task
is launched for this round, it is expected behavior, no need to do promotion*/) {
        do_promotion_on_cp = HWS.CFG.PROMOTE_RULE__ON_CP == 1 && task.on_cp == 1;
        do_promotion_top1_proi = HWS.CFG.PROMOTE_RULE__TOP_1 == 1 && (task ==
PTP[0]);
      }
      if(do_promotion_thrshld || do_promotion_on_cp || do_promotion_top1_prio) {

        non_rsv_ready_sip_bm = get_allowed_sip_to_use(task) &
~HWS.reserved_sip_bm/*1:rsvd,0:not yet*/;
        new_rsv_sip_mask = sel_best_sip_rsc_to_reserve(input=task,
from=non_rsv_ready_sip_bm);
        if(new_rsv_sip_mask != 0) { //success reserved
          HWS.reserved_sip_bm |= new_rsv_sip_mask;
          task.reserved_sip_mask = new_rsv_sip_mask;
          task.R_flag = 1;
          RTP.push_back(task);
          for (sip_idx in new_rsv_sip_mask) {
            HWS.reserve_sip_kernel_thd_num[sip_idx] = task.thread_num;
          }
          break;
        }
      }
    } else {
      break;
    }
  }

}

uint32_t sel_best_sip_rsc_to_reserve(task_to_check, target_sip_rsc_mask) {
  sip_mask_target = (0x1 << RS[task_to_check].thread_num) - 1;
  switch(RS[task_to_check].thread_num) {
    case 1:
      m_1sip_grp_work_info_tmp = (HWS.1sip_grp_work_info sort by remain_exec_time);
//m_1sip_grp_work_info_tmp[grp_idx]越小, remain_exec_time越小
      for(int grp_idx=0; grp_idx < m_1sip_grp_work_info_tmp.size(); grp_idx++) {
        uint32_t new_rsv_sip_mask = sip_mask_target << m_1sip_grp_work_info_tmp[i]-
>start_sip_idx; //每个grp只记录该组的start_sip_idx
        if ((new_rsv_sip_mask & target_sip_rsc_mask) == new_rsv_sip_mask) {//检查最快结
束的sip是否在target_sip_rsc_mask内
```

```
            return new_rsv_sip_mask;
          }
        }
        break;
      case 2:
        m_2sip_grp_work_info_tmp = (HWS.2sip_grp_work_info sort by
remain_exec_time);//m_2sip_grp_work_info_tmp[grp_idx]越小，2sip组的remain_exec_time越小
        for(int grp_idx=0; grp_idx < m_2sip_grp_work_info_tmp.size(); grp_idx++) {
          uint32_t new_rsv_sip_mask = sip_mask_target <<
m_2sip_grp_work_info_tmp[grp_idx]->start_sip_idx;
          if ((new_rsv_sip_mask & target_sip_rsc_mask) == new_rsv_sip_mask) {
            return new_rsv_sip_mask;
          }
        }
        break;
      case 3,4:
        m_4sip_grp_work_info_tmp = (HWS.4sip_grp_work_info sort by
remain_exec_time);//m_4sip_grp_work_info_tmp[grp_idx]越小，3sip或4sip组的
remain_exec_time越小
        for(int grp_idx=0; grp_idx < m_4sip_grp_work_info_tmp.size(); grp_idx++) {
          uint32_t new_rsv_sip_mask = sip_mask_target <<
m_4sip_grp_work_info_tmp[grp_idx]->start_sip_idx;
          if ((new_rsv_sip_mask & target_sip_rsc_mask) == new_rsv_sip_mask) {
            return new_rsv_sip_mask;
          }
        }
        break;
      case 6,8:
        m_8sip_grp_work_info_tmp = (HWS.8sip_grp_work_info sort by
remain_exec_time);//m_8sip_grp_work_info_tmp[grp_idx]越小，6sip或8sip组的
remain_exec_time越小
        for(int grp_idx=0; grp_idx < m_8sip_grp_work_info_tmp.size(); grp_idx++) {
          uint32_t new_rsv_sip_mask = sip_mask_target <<
m_8sip_grp_work_info_tmp[grp_idx]->start_sip_idx;
          if ((new_rsv_sip_mask & target_sip_rsc_mask) == new_rsv_sip_mask) {
            return new_rsv_sip_mask;
          }
        }
        break;
     case 9,12,16:
        m_16sip_grp_work_info_tmp = (HWS.16sip_grp_work_info sort by
remain_exec_time);//m_16sip_grp_work_info_tmp[grp_idx]越小，9sip或12sip组或16sip组的
remain_exec_time越小
        for(int grp_idx=0; grp_idx < m_16sip_grp_work_info_tmp.size(); grp_idx++) {
          uint32_t new_rsv_sip_mask = sip_mask_target <<
m_16sip_grp_work_info_tmp[grp_idx]->start_sip_idx;
          if ((new_rsv_sip_mask & target_sip_rsc_mask) == new_rsv_sip_mask) {
            return new_rsv_sip_mask;
          }
        }
```

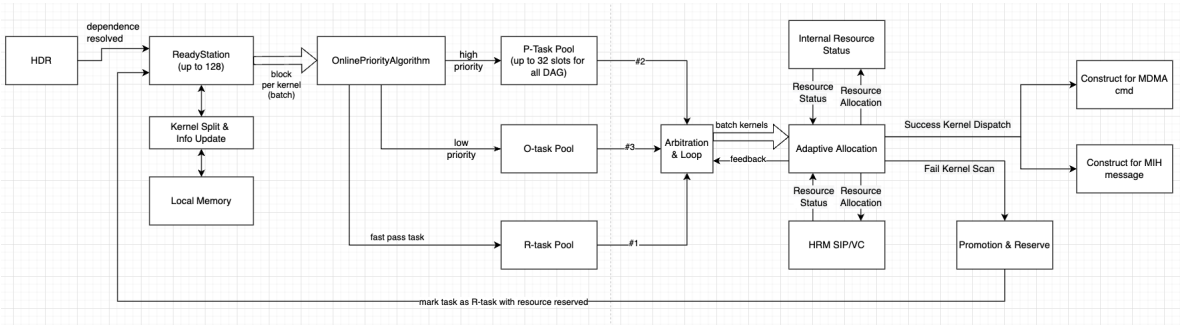
```
            break;
          default:
            break;
        }
        return 0x0;

    }
```

调度算法模块 架构

架构设计上HWS的参数

可调硬件参数	默认值	解释
MAX_WORKING_DAG_COUNT	24 or 32	同时并行的DAG图的数量
READY_STATION_SIZE	32	RS最大表项数量

主要结构图 及 子模块介绍



上图为SIP硬件调度架构的整体功能框架图，主要功能分成以下几部分：

ReadyStation（RS）

主要功能：

- 此模块接收HDR内已经解除依赖的task，将task信息移动到此模块中
- 存储从Share Memory（SM）中读取的该task的task desc中必要的信息
- 部分位域需要Kernel Split&Info Update模块来更新，详见KSIU模块描述
- 为后续模块存储所有kernel的状态信息和调度所需要的信息。

RS表项信息：

信息	位宽	来源	语义
sip_alloc_affinity_mask	32	TD@SM	SIP调度时，此kernel允许使用的sip的id
kernel_offline_priority	16	TD@SM	基于offline算法计算得出的DAG图内节点的优先级 可能需要归一化? 对于静态图，软件需要在图编译时进行节点优先级计算 对于动态图，可以在runtime是？遗留问题
kernel_dag_tag	16	TD@SM	所属图+用户的唯一标识，保证有限生命周期内的唯一性
kernel_on_cp	1	TD@SM	是否在DAG计算图的critical path上，可以有多条critical path
kernel_est_exec_time	32	TD@SM	该kernel的每个block的estimate execution time预计执行时间（注意：是多sip并行执行的绝对时间）
start_block_idx	16	TD@SM	该kernel的block_idx从start_block_idx开始往上递增(包含start_block_idx)
blockDimX(Y=1/Z=1)	16	TD@SM	Block的三个维度上thread的个数，以thread为单位 Scropio中Y=Z=1, X代表thread_num, 对应sip的个数 threadIdx(z,y,x)，在block内是local的thread idx
gridDimX(Y=1/Z=1)	16	TD@SM	Grid的三个维度上block的个数，以block为单位 Scropio中Y=Z=1, X代表block_num blockIdx(z,y,x)，在一次LaunchKernel中逐block递增
cooperative_kernel	1	TD@SM	不确定需要加，可能可以让FW launch cooperativeKernel的时候
dynamic_dag	1	寄存器	Amos在新DAG出现时，配置给HWS的信息，标记该kernel_dag_tag对应的DAG是否为动态图， 1：动态图，0：静态图。默认不配置则为0
dag_total_kernel_cnt	32	寄存器	Amos在新DAG出现时，配置给HWS的信息
---------- 下面是由硬件维护的	----------	----------	需要实时更新的逻辑 ----------
kernel_finished_block_cnt	16	HWS	记录该kernel已经调度完成的block的数量
schedule_fail_times	16	HWS	kernel被其他更低优先级的任务超车调度的次数（通常是因为该任务的资源不ready，其他任务资源ready导致）
kernel_reserve_flag	1	HWS	被promoted的kernel的标记，即R-task（在代码实现中写成了R_flag）
reserved_sip_mask	32	HWS	被promoted的kernel已经reserved_sip_mask
dag_finish_kernel_cnt	32	HWS	HWS完成一个kernel的后更新+1

Kernel Split & Info Update(KSIU)

- 每次调度完成，更新被调度的kernel的信息，详见RS中硬件维护的信息
- 负责从Share Memory中读取Task Desc中必要的信息，并更新到RS的entry表项中
- 维护DAG图相关的历史信息（见下表）
- HDR新push进来的kernel，若为动态图的算子kernel，需要更新对应DAG的所有现存Kernel的offline priority

额外需要维护的RS/KSIU中的信息	位宽	语义
dag_finished_kernel_cnt[32]	16 per DAG(32 max)	每个dag已经finished的kernel，由HWS更新
dag_total_kernel_cnt	16	所属DAG所有node的数量，可以让软件在每个DAG开始前（对于动态图，可以在stream的开始认为是DAG的开始）用API设置进来
~~dag_is_dynamic~~	~~1~~	~~节点所属的图是动态图还是静态图~~ ~~软件如果有能力在下发前分析DAG并通过优先级标记了执行顺序，则该bit置0.~~ ~~若为动态图，无法提前分析，则用API软件可以置1，硬件会根据软件的下发顺序，更新offline_priority~~

OnlinePriorityAlgorithm（OPA）

根据RS中kernel信息，使用对应的multi-DAG scheduling的算法：（具体见算法中的onlinePriorityAlgorithm()函数实现）

- 优先从每个DAG的kernel中 pick 一个offline priority最高的kernel进入P-task pool
- 若总数超过P-task pool上限，则停止pick（理论上如果配置的P-task pool size = 最大并行的DAG数 * P-task Top-K value，则不应该出现此问题）
- 所有DAG扫描结束完成P-task pool的准备后，计算出所有P-task的online priority
- 所有没被pick的任务，自动进入O-task pool
- ~~若有软件指定的fast pass task，则直接进入R-task pool~~

P-task Pool(PTP)/O-task Pool(OTP)/R-task Pool(RTP)

根据RS中每个kernel entry里的信息，通过OPA计算结果分类得到三种任务组：

Task Pool

1. R-task Pool

- 至多8个，每个cluster（VG）上至多出现一个R-task（但未必把所有SIP都占满了）
- RTP内不再对多kernel区分优先级，因为资源不重合

2. P-task Pool

- 每个kernel会有一个新的online priority优先级，根据OPA算法得到

- PTP的上限与支持的最大并行DAG图的数量关联，暂定32个
- 需要按优先级排序，排序方式见算法的中的：sortPTPkernel()

3. O-task Pool

- 为RS中除去RTP和PTP中的kernel的集合
- 按优先级排序，排序方式见算法的中的：sortOTPkernel()
- [实现简化] 若减少开销，可只存优先级高的8~16个 Kernel

优先级排序方式

其实kernel任务的优先级排序本身属于OPA算法的一部分，但具体实现是放在OPA中还是TP中，无所谓。

Arbitration & Loop

- 仲裁顺序：RTP > PTP > OTP
- 若一次无法将Pool中kernel全部输出给AA，则需要按优先级分成几组输出给AA模块
    - 每个batch 做成8 ~ 32个kernel的info bus输出到AA模块

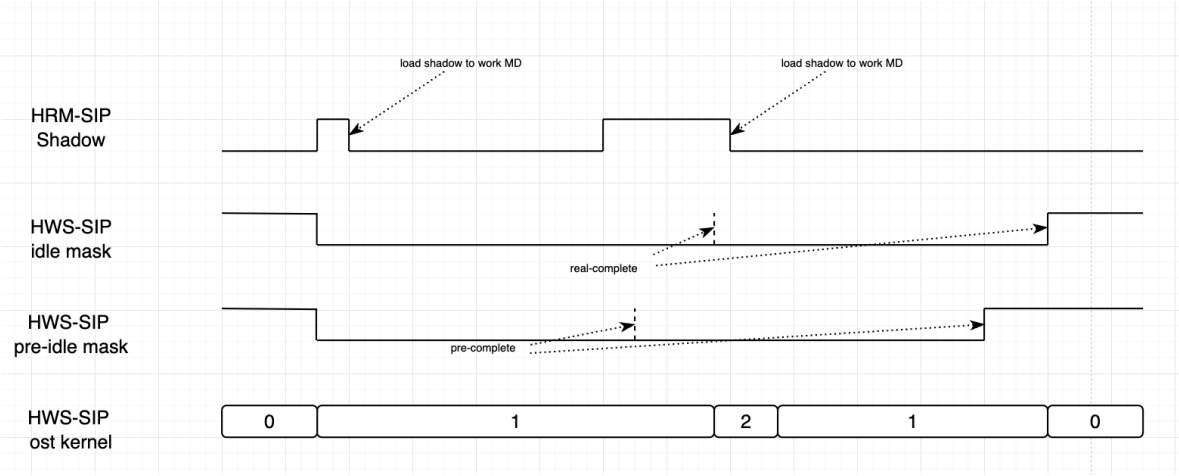
Internal Resource Status（IRS）

该模块是HWS中非常重要的模块，需要记录：

1. 除HRM资源之外的资源使用情况
2. 每个SIP的Estimate Remain Time

详细信息如下：

位域	位宽	语义	更新条件
sip_idle_status	32	sip是否idle，idle的定义：不仅sip未在working，也没有pending shadow config 1：idle， 0：not idle 注：依赖于SIP的行为是先发real-complete给SP，再发shadow-config load finish（见下面的波形图）	见波形图（图中取反即可）
sip_pre_idle_status	32	sip是否进入pre-idle，pre-idle的定义：sip接近当前执行的结尾，但还是busy，且没有pending shadow config 1：pre-idle， 0：not pre-idle 注：若pre-complete是SIP主动产生的，则依赖于SIP的行为是先发pre-complete给SP，再发shadow-config load finish（见下面的波形图） 若pre-complete是SP自己预判的，则需要保证当sip_idle_status 0->1时，sip_pre_idle_status也需要同时0->1	见波形图（图中取反即可）
sip_ost_kernel_cnt (待定是否要做)	3		
sip_remain_exec_time[32]	32bit per sip	记录32个SIP每个还有大概多久会结束当前kernel。时间单位可以配置。1us？如果touch max，就max记录即可	每次launch新的kernel，会将占用的sip的值更新成kernel的EET+某个固定的offset。 启动倒计时的条件：若launch时sip是已经idle的，则可立即启动（offset此时可以多加些） 若launch时以pre-idle的判断的SIP，则等real-complete从SIP返回给HWS时，才启动
reserved_sip_mask	32	记录32个SIP哪个是被reserve给某个task了	每次promote一个task，reserve sip后，更新。 此位域是所有R-task的kernel的reserved_sip_mask的并集



注意：即使在HWS没有被使用的时候，HWS中logic也需要通过根据HRM-SIP，以及SIP回报的real-complete和pre-complete来维护sip的idle 和pre-idle mask。对于没有通过HWS调度的SIP，pre-idle mask只能在 “使用SIP回报”方式下有意义。

Adaptive Allocation （AA）

该模块结合IRS和HRM的资源和状态信息，根据上游task pool送来的kernel，与资源进行匹配，完成选择最终调度任务的功能。具体算法见 adaptive_allocation() 函数的实现。几个特性：

- 根据配置，可以选优先进行backfilling（即匹配reserved idle sip）方式，还是未reserved idle sip
- 根据配置，是否可以使用pre-idle的sip
- 对于R-task，若reserved的sip没有ready，但其他该task可以适用的sip ready了，则会直接使用这些未被（或部分未被）ready的sip，且释放之前未被使用的reserved 的sip
- 根据kernel和HWS的配置，可以进行三种资源的组合匹配：
    - 只看sip是否ready
    - ~~看SIP和对应亲和的L2是否有分配好且未被使用的存储（per block）~~
    - ~~SIP+L2B+VC的全部资源都满足~~

Promote & Reserve（PR）

该模块负责监控每次调度过程中，任务是否符合promotion的条件，若符合，则进行任务优先级的promotion（标记成R-task），并完成资源预约，以保证当资源可用时，第一时间调度该任务。

具体可以看算法实现： promote_task_and_reserve_sip()

promotion有几种情况，其中一种promotion的方式是always会发生，另外几种种机制是为了更好的执行重要的任务。（都有开关，静态配置）

Options	触发条件	说明
默认ON,也可以关	每当一个kernel所有block都dispatch完成，则会对PTP中所有比该刚调度的kernel老的kernel.schedule_fail_times变量递增，当fail次数达到阈值后，则promote (optional) 若有多个kernel同时超过阈值，则promote fail次数最多的。 该条件触发的kernel promotion优先级高于其余两种方式。会优先promote	为防止一个kernel进入PTP中，被新进入的kernel反复反超，设计的强制promotion机制本质上只有一个kernel优先级达到了进入PTP的条件后，就会记录失败次数，如果所有后进的kernel都执行过一遍（或者个数够多了），则说明饥饿过久
可打开	kernel的优先级排序为PTP的Top1 且 block需要VG内所有sip资源	该kernel在与其他只使用部分sip的kernel竞争时，很难在某一个时刻拿到所有资源（SIP天然的launch就有斜率，结束也是一个一个结束的）
可打开	若调度成功，但选择的kernel的并非PTP中最高优先级，则promote PTP中最高优先级的	
可打开	若调度成功，但选择的kernel的并非在CP上，且PTP中有其他在CP上的kernel，则promote PTP中在CP上的最高优先级的	

必要的寄存器

寄存器	方向	语义/值	HWS的行为
------ 控制寄存器-------			
HWS_WORK_MODE	FW->HWS	0: HW mode; 1: FW mode	0：硬件自动启动，并直接启动MDMA 启动 1：完成调度后，通过MIH把结果回给Amos
CLUSTER_SIP_NUM	FW->HWS	指定一个cluster(VG)内SIP的数量，以保证在选取SIP的时候不会出现跨cluster的边界问题，如果出现任何kernel的thread_num > CLUSTER_SIP_NUM，则需要报错。理论上FW就不应该下发。	
SIP_REPORT_COMPLETE	SIP->SP	将LaunchMD中的sip_rpt_complete位域的值写到此寄存器	将收到的SIP ID对应内部的sip_idle_mask对

			应bit 置1
SIP_REPORT_PRE_COMPLETE	SIP->SP	将LaunchMD中的sip_rpt_pre_complete位域的值写到此寄存器	将收到的SIP ID对应内部的sip_pre_idle_mask对应bit 置1
------ HWS特性的配置寄存器-------			
~~ENABLE_SECOND_RSC_SCH~~	~~FW->HWS~~	~~是否需要HWS检查除SIP之外的资源（如L2）~~ ~~0：只看SIP资源，1：检查二类资源~~	
ENABLE_EARLY_LAUNCH_SIP	FW->HWS	是否需要HWS使能使用pre-idle作为提早启动SIP（early-launch）的判断条件。 0：只允许启动真正idle的SIP，1：允许使用进入pre-idle（pre-complete）状态的SIP	
EALRY_LAUNCH_MODE	FW->HWS	EARLY_LAUNCH的模式，0：由SIP主动上报，1：由HWS自行计算判定	
RSVD_SIP_BACKFILL_MARGIN_OFFSET	FW->HWS	Backfill到reserved sip时，允许的正负值	
SCALE_RATIO_TABLE_RANK_0~19	FW->HWS	每5%一档	是否分子分母都需要?
SIP_FILLUP_TASK_FIRST	FW->HWS		
USE_RSVD_SIP_FIRST	FW->HWS		
1/2/3/4/6/8/9/12/16_SIP_USAGE_MASK	FW->HWS	对应kernel thread_num大小的SIP_USAGE_MASK	

Interface

HWS与外围的接口分为两类，一类为HWS需要接收和输出的与调度任务相关的业务接口；另一类为HWS的控制者（eg. FW）设置并监控HWS的交互接口，主要以寄存器为主。

上层软件 Kernel任务信息

下表为调度需要的信息，有些为per kernel的，会基于Task Desc（TD），有些是per DAG的，可以考虑使用API方式

位域	位宽	语义	传递给硬件的方式	影响
Kernel_id（task_id?)	32	此kernel的唯一标识，为软件辨识任务的id，全局应该具有唯一性	TD	FW可以将该id直接传给HW，也可以做1对1映射，方便FW管理
sip_alloc_affinity_mask	32	SIP调度时，此kernel允许使用的sip的id	TD	
kernel_est_exec_time	32	cost model给出的预计执行时间，是per block	TD	
offline_priority	16/32	基于offline算法计算得出的DAG图内节点的优先级 可能需要归一化?	TD	对于静态图，软件需要在图编译时进行节点优先级计算 对于动态图，可以在runtime是
dag_tag	16	所属图+用户的唯一标识，保证有限生命周期内的唯一性	TD	
on_cp	1	是否在DAG计算图的critical path上，可以有多条critical path	TD	对于静态图，软件有机会对图的拓扑做计算
block_trigger_cnt	16	FW允许HWS启动的block的数量，可以动态的修改 该位域对coop-kernel无效，因为coop只发射一次。	TD	HWS只读不改。FW可读可写。 最大值<=gridDimX 当已启动的block数量==trigger_cnt是，该任务认为暂停发射权
~~total_node_cnt~~	~~16~~	~~所属DAG所有node的数量~~	~~API~~	~~对于静态图，软件有机会对图的拓扑做计算~~
blockDimX/Y/Z	16	Block的三个维度上thread的个数，以thread为单位	TD	Scropio中Y=Z=1, X代表thread_num, 对应sip的个数 threadIdx(z,y,x)，在block内是local的thread idx
gridDimX/Y/Z	16	Grid的三个维度上block的个数，以block为单位	TD	Scropio中Y=Z=1, X代表block_num blockIdx(z,y,x)，在一次LaunchKernel中逐block递增
out_path_mdma	1	是否要直接把调度结果发送给MDMA去launch SIP	TD	与out_path_mih可以独立配置，但是至少要有一个为1，即3中组合
out_path_mih	1	是否要把调度结果发送给MIH以上报Amos	TD	与out_path_mdma可以独立配置，但是至少要有一个为1，即3中组合

API 完备性：

TODO: 每个API对应的使用的HWS的寄存器方法和顺序

Type	API	推荐实现方式	作用	使用场景
配置	setPriorityLookupTable(int rank_id, int value);	CF寄存器/静态配置	onlinePriority的算法的中的参数，每个rank_id都需要配置有效值。 详见onlinePriority算法的实现	
	configEarlyLaunch(bool enable_early_launch，int pre_idle_source)	CF寄存器/静态配置	FW配置HWS的Early-Launch相关特性。 包括： 1. 是否使能EL 2. pre_idle的来源， 0：来自HWS自行判断 1：SIP主动回报	
	configPromotionPolicy(uint8_t promotion_rule, uint16_t promotion_threshold)	CF寄存器/静态配置	Promiton_rule是按bit来开关几种promotion触发条件，可以都关或开启任意几种（见PR模块的详细解释）。当选择为threshold方式时，则需要配合threshold，单位是次数。推荐值是个位数。	
	configBackfill(int backfill_time_offset)	CF寄存器/静态配置	backfill成功的条件： sip_remain_exec_time >= kernel.estimate_exec_time - backfill_time_offset 若backfill_time_offset为正数时，代表backfill需要预留一些margin来保证backfill到reserved sip的kernel不应延后R-task的预计启动时间。当backfill_time_offset为负值时，则相反，允许在规定的offset内延后R-task启动时间。offset的单位应该与EET的单位相一致。	
	configRscAllocation(bool use_reserved_sip_first, bool sip_fillup_task_first)	CF寄存器/静态配置	use_reserved_sip_first： 在找idle sip的时候优先使用reserved sip做backfill还是用未被reserved sip sip_fillup_task_first：1：如果检测到有可以用掉现在所有idle sip的task（即使不是高优先级），则优先发射。0：按照优先级顺序	
	~~setHWSmode(int out_path, int in_path)~~	~~CF寄存器/静态配置~~	~~FW配置HWS的模式，分别用任务来源和任务输出路径两端做组合：~~ ~~out_path控制HWS的输出路径，0: 发给MDMA，1：发给MIH~~ ~~in_path控制HWS的输入路径，0：来自HDR，1：来自寄存器配置~~ ~~(in, out)=(0, 0)：默认配置，任务来自HDR，HWS调度完成直接传给MDMA cmd;~~ ~~(in, out)=(0, 1)：任务来自HDR，HWS通过MIH只将结果回传给Amos，但是不发给MDMA~~ ~~(in, out)=(1, 0)：任务来自FW，HWS只负责成功调度一次（无论block有多少）。调度成功完成前，不返回结果。返回结果走MDMA~~ ~~(in, out)=(1, 1)：同2，返回结果走MIH~~	
	~~enableHWSreportAmos(bool enable)~~	~~CF寄存器/静态配置~~	~~在HWSmode的out_path=0时，如果打开report，会在启动MDMA的同时，也给MIH传一份信息~~	
				1. 在每个executable结束后或stream边界 2. 在软件上层每次创

FW使用	resetDAG(uint32_t dag_tag)	CF寄存器/动态	清除HWS内对应dag_tag相关记录的历史信息，例如dag_finished_node（不要overflow即可）	建一个新DAG时，防止dag tag重复，在command packet中调用API 使Amos call一下该寄存器接口
	initDAGProperty（uint32_t dag_tag, uint32_t total_kernel_cnt，bool dynamic_dag)	CF寄存器/动态	在DAG开始之前，初始化该DAG信息。若为动态图，也需要标记是否为动态图（注：动态图时，total_kernel_cnt则不再使用） 该信息是为了更好的调度。	
	getSIPStatus(uint32_t sip_idle_mask, uint32_t sip_pre_idle_mask)	CF寄存器/动态	FW可以借用HWS的逻辑，实时获得SIP的两种状态，sip real idle, 和sip pre-idle的状态	FW在预配的时候可以在更准确的时间点进行预配置。
	sipUsageMaskConfig(uint8_t kernelThdSize, uint32_t usage_bitmask)	CF寄存器/动态配置	配置per context 的每种sip kernel size的usage mask 硬件只支持1/2/4(3)/8(6)/9/12/16 可以把3/4合并，6/8合并，9/12/16合并 也支持随时修改，用于实时控制HWS可以调度的资源范围，方便FW做混合调度	可能需要修改，比如FW调整HWS可调度的SIP的范围。
SIP使用	sipPreComplete(uint32_t sip_id)	CF寄存器/动态	该context id内对应的sip_id可以标记为即将结束。可以用于early-launch的候选sip	Kernel代码主动提前向SP-HWS回报其即将完成当前kernel per（SW） context id
	sipComplete(uint32_t sip_id)	CF寄存器/动态		
	-------- 进哥根据具体设计来设计，原则上此文档中的counter和status都暴露出来最好			
Debug				

DEBUG

- COUNTDOWN_CNT
-

流程：

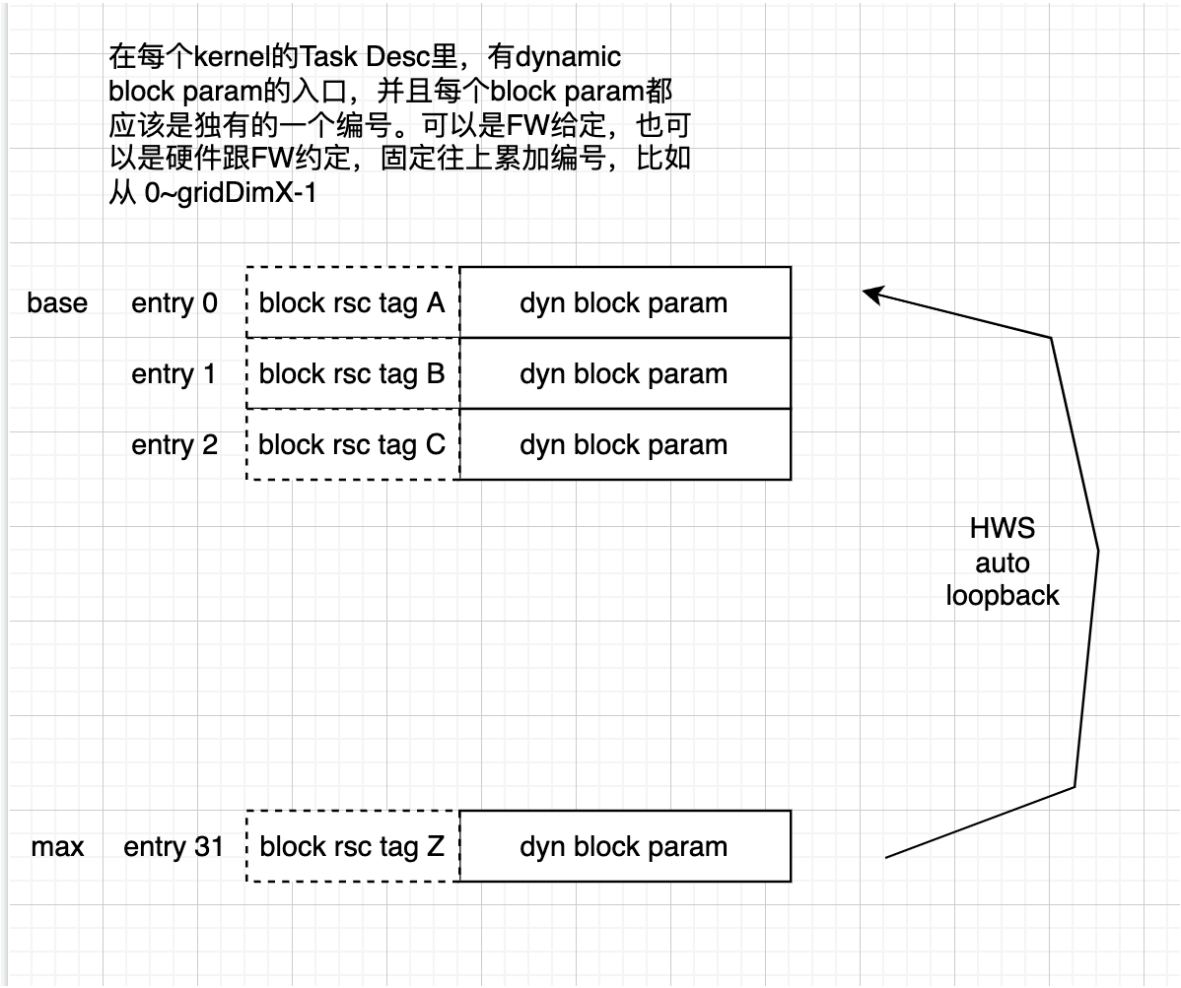
完备性：

1. FW借用HWS调度+后处理
2. FW同时调度，同时调度HWS也在调度的SIP资源
3.

数据流交互

Amos & SIP Kernel 交互

1. 硬件per Kernel支持dynamic block param的queue，在TD中提供该queue在SM上的base addr，queue size，以及param entry size。硬件自动提供回滚机制每次读取的block_param addr = base_addr + (block_id%queue_size) * param_entry_size
2. 每个dynamic block param需要有一个独有的资源编号，方便FW在收到乱序完成的block时，可以匹配找到对应可回收的资源。该资源编号可以暂时使用block_id这个等价信息标识。



在HWS与SIP交互的信息中，需要再SIP返回给HWS的signal data中增加信息：

每次HWS在launch每个block时，会在launch_info表中记录kernel，block和sip的使用信息，并把signal data内除sip_id外，再额外填入对应的launch_id和block_id。当SIP完成后，返回给HWS时，HWS可以监控对应block的完成状态。当所有SIP都完成后，HWS会向FW发送信息（见下一段文字）

每次启动launch SIP，HWS都会分配一个独有的launch_id，并记录该id对应的kernel_id和block_id

也就是说，每次SIP signal给HWS的signal data = {launch_id[5:0], block_id[15:0], sip_id[4:0]} （之前的版本只有sip_id[4:0]）

launch_id	kernel_id or hdr_entry_id	block_id	used_sip_mask
0	100	3	0xF
1	500	5	0xF00
...	...	...	...
10	500	15	0xF0

在HWS与FW的交互中修改为以下两种信息：

HWS只在两个时间点给FW发送信息：
1. 每次调度决策后，给FW传递Schedule Info, FW通过首次获得该信息，得知该kernel已经在HWS中被开始调度了，并知道所在的HWS_Entry的位置，FW可以开始向对应的寄存器，写入更新的kernel的block_trigger_cnt

同时可以通过获得的blockRscTag来知道SM上哪个dynamic block param的存放slot可以被覆盖填入新的内容。

2. 每次有block任务的所有SIP都返回结束信息后，会通知FW一个kernel的某个block的完成，FW可以通过获得的blockRscTag来知道哪组dynamic block param的资源可以被回收

Message Type= Schedule Info	HDR_Entry_Id	HWS_Entry_Id	BlockRscTag/Block_id

Message Type= Block/Kernel Finish Info	HDR_Entry_Id	BlockRscTag/Block_id

与MDMA

最终调度SIP的cmd需要使用MDMA，launch一个kernel的block，需要构造出thread_num个数的SIP的cmd，每个cmd的信息包含：

位域	位宽	值	谁产生	语义
thread_idx	16	0~blockDimX-1	AA	每个SIP的thread id不同
block_idx	16		KSIU	当前launch的block_idx，所有SIP的cmd都一样
sip_rpt_complete_val	8	对应CTX的该SIP的逻辑ID		SIP在finish该kernel时回报
sip_rpt_complete_addr	48?	HWS的寄存器地址，可能是双die上任意一个SP的任意一个HWS		写回sip_rpt_complete的目标CF地址
sip_rpt_pre_complete	8	对应CTX的该SIP的逻辑ID		SIP在即将finish该kernel时回报，时间点由sw决定（eg.最后一次DMA搬运前）
Sip_rpt_pre_complete_addr	48?	HWS的寄存器地址，可能是双die上任意一个SP的任意一个HWS		写回sip_rpt_pre_complete的目标CF地址
~~-----下面两组二选一—~~	~~—~~	~~---根据模式不同~~	~~—~~	~~------~~
~~cdte_vc_base~~	~~8 or 16~~			
~~cdte_vc_num~~	~~8 or 16~~			
~~cdte_vc_alloc_1~~	~~32~~			~~[31:0] = {vc3, vc2, vc1, vc0}~~ ~~每个8bit，[7] = valid, [6:0] = 0~127~~
~~cdte_vc_alloc_2~~	~~32~~			~~[31:0] = {vc7, vc6, vc5, vc4}~~ ~~每个8bit，[7] = valid, [6:0] = 0~127~~

与HDR

这个进哥根据设计的微架构定义一下即可

与Share Memory（SM）

主要由KSIU逻辑部分去访问Share Memory，接口有设计自行定义。

有两个时间点需要访问

- 在kernel任务从HDR首次加载到RS中时，需要从SM中读取kernel对应的TD的信息，并进行必要的处理工作，存储到RS entry中
- 要求每次调度完成后（AA决策后锁住资源后即可开始，不需要所有事情都做完），可能需要从SM读取信息更新被调度的kernel的最新存储信息。若不enable L2 allocation，则不需要去SM更新信息。

与MIH(与Amos交互的接口)

该接口是HWS与FW交互的主要业务接口，返回调度的信息结果回FW。

注意，因为MIH只支持64bit的message，因此，若超过64bit，需要保证HWS连续（原子性的）写入N个message

返回调度Message 格式

DW（4byte）	位域	位宽	语义
DW0	type	2	现在只有一种，0：result
	src	2	从HDR还是寄存器接口直接配置进来的
	hdr_id	4	HDR id[3:0] + HDR entry id[7:0], 软件使用组合的12bit来反向找到对应的task_id是什么
	hdr_entry_id	8	
DW1	sip_sel_mask	32	HWS 最终调度的结果，每个bit代表一个sip，对应的sip_idx是软件理解的sip_id. 1：该sip被选择，0：该sip未被选择

使用约束

kernel的允许使用多cluster场景的约束：

当一个kernel允许调度上多个cluster时（即affinity_mask 内使能的SIP是所属在多个cluster的），在4C32S，2C32S时候若kernel的thread num远小于cluster_sip_num时，若只允许该kernel独占该cluster会明显有浪费SIP的问题。因此，需要FW对于这种kernel，指定一个cluster，直接修改SW原始的affinity mask保证不跨cluster，保证调度系统只handle以下两种情况：

1. thread num < cluster sip num 且 affinity mask 不跨cluster，允许则会用多个kernel共享一个cluster。硬件回使用kernel（唯一一个）的block_trigger_cnt，做好调度流控，保证发送block的数量不超过FW允许的值，以使能FW动态分配回收SM和cluster内的共享(L2,VC)等资源的功能。
2. thread num == cluster sip num 且affinity mask跨cluster，因为kernel会独享一个cluster，因此dynamic block param的内容应该与cluster的亲和性无关。当然此时trigger_cnt还可以用于流控，但目的主要是防止dynamic_block在SM中占据过多容量。

Validation Use-Case Requirement

架构目标：

1. 支持FW静态预配置调度和动态调度（FW+HW）混合的架构目标：
    1. 支持对同一个SIP资源，无缝地在预配置的静态调度和动态调度间切换
    2. 对不同的SIP资源，支持FW预分配的静态调度和HW动态调度同时在一个Context内存在，且动态可以调整动态调度的资源范围
2. 能够在动态调度的流程上，通过FW和HW配合来加速调度性能，并能够offload FW的负载，释放更多cpu cycle给其他功能

####

使用方法

1. 每个HWS服务一个VF
2. 每个HWS有4套各context允许使用SIP的配置寄存器，context.sip_usage_mask。当对应bit的usage mask为0时，代表该context暂时无法使用对应SIP。HWS不会再发出
3. 支持动态修改VF内context的usage mask，并在固定时间内生效（可以通过write usage mask -> read usage mask -> 固定时间方式）
4.

硬件特性要求：

1. HWS做调度决策计算必须在固定的cycle数内，完成一次判断，无论成功与否。
2. Usage_mask修改之后，HWS在完成当下这次调度决策后，对应的context的SIP使用权需更新到最新usage mask。即如果某个SIP的usage mask bit变成0，则此时不会有任何该context的任何kernel使用该SIP

使用流程：

Usecase1: 全部资源HWS调度，且资源静态分配

Usecase2: 全部资源FWS调度

Usecase3: 不同SIP资源，由HW和FW分别（独立）调度，资源静态分配

Usecase4：相同SIP资源，有HW和FW同时（竞争）调度，资源静态分配

Usecase5: HWS参与调度（包含与FW独立和竞争调度），且资源动态调整（包括与FW之间，也有多context之间）

usecase6:

validation point：

1. FW修改， 硬件特性要求的所有点
2.

Verification Guide

1. 除基础功能外，只验典型与Amos的组合流程（Amos再补一些）
2. 可以接受调度算法上有bug，但是必须要保证的是资源的分配不出错（overlap或leak），状态的监控不出错（也是为了SIP不overlap），保证不违背被启动engine的接口行为（比如VC还未idle就启动了下一次）

软件使用（ToBeReWrite, 暂时先不看）

使用方式

Host软件层

- 对于静态执行图的业务，建议软件在host侧拿到图之后，根据每个执行节点的cost model，填写好estimate_execution_time（EET），并按照HEFT算法给每个节点提供
  - offline_priority
  - on_cp
  - remain_path_len

  再根据调度所要map到的VG范围，填好affinity_mask。

  上述内容都在Interface章节Kernel信息中有详细写，只是来源和产生方式不同
- 对于动态图，软件每次只能看到少数几次LaunchKernel(), 除了能给出EET，需要使用HEFT算法 完成对**单节点**的信息计算（on_cp 可以置0）

Device/Amos Runtime

场景，要求和建议

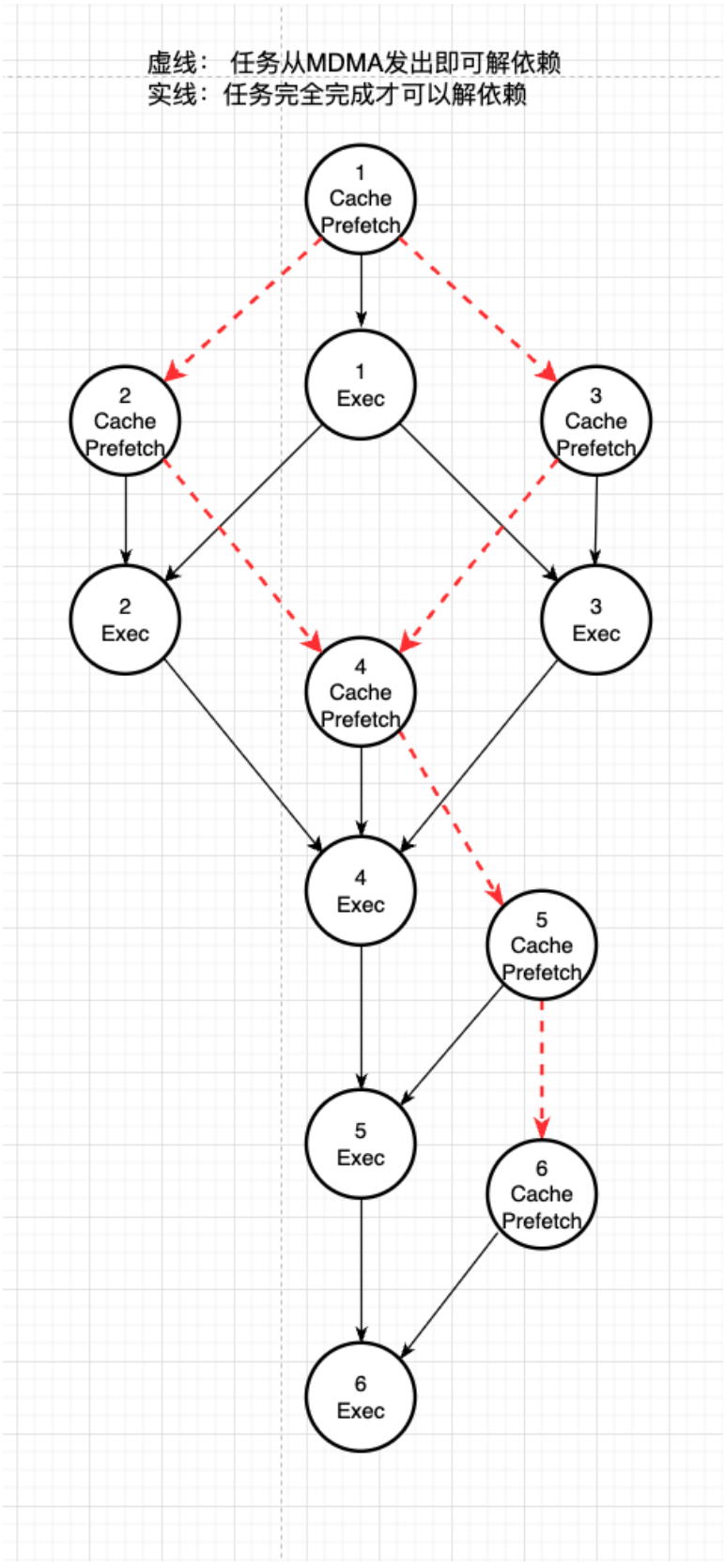
Note：

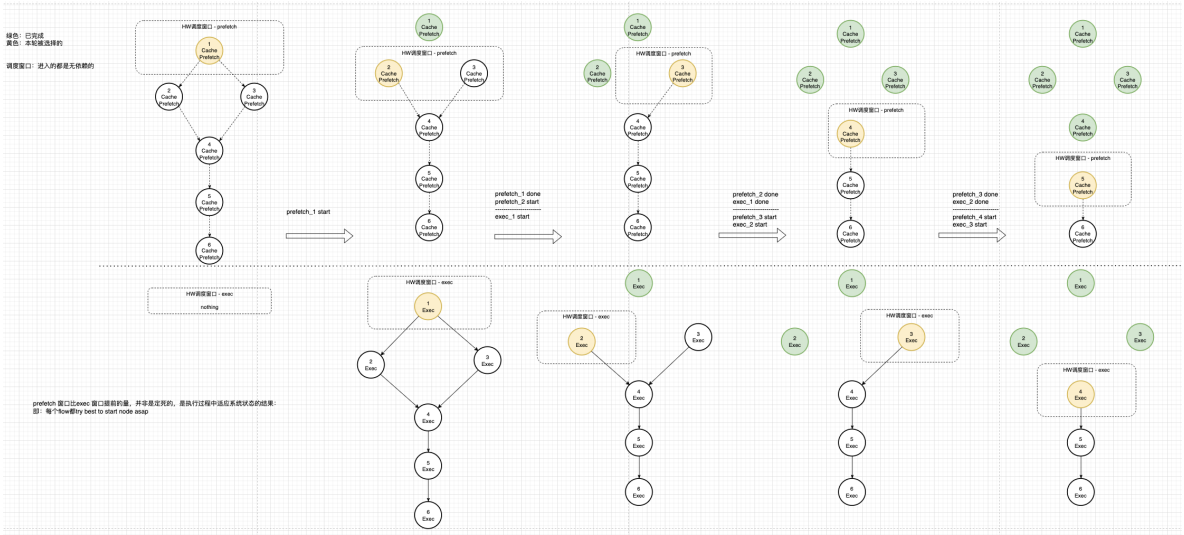
讲一下：

- 静态图时需要软件作什么
  - offline作什么，online怎么配置
  - 如何使用配置
- 动态图时怎么做
- 软件如何用动态调度，同时完成计算资源，存储，VC的分配

Case1: Prefetch L2B with Scheduling

1. Host Runtime
   - 按照原图的计算节点，额外建立两类依赖：
     - 本计算节点依赖于新增的prefetch节点
     - 新增的prefetch节点依赖于其计算节点依赖的计算节点的prefetch节点
2. Device/Amos Runtime
   - FW需要标记Prefetch节点只enable_l2_buff_alloc
   - FW需要将prefetch 和对应的exec节点使用同样的task desc

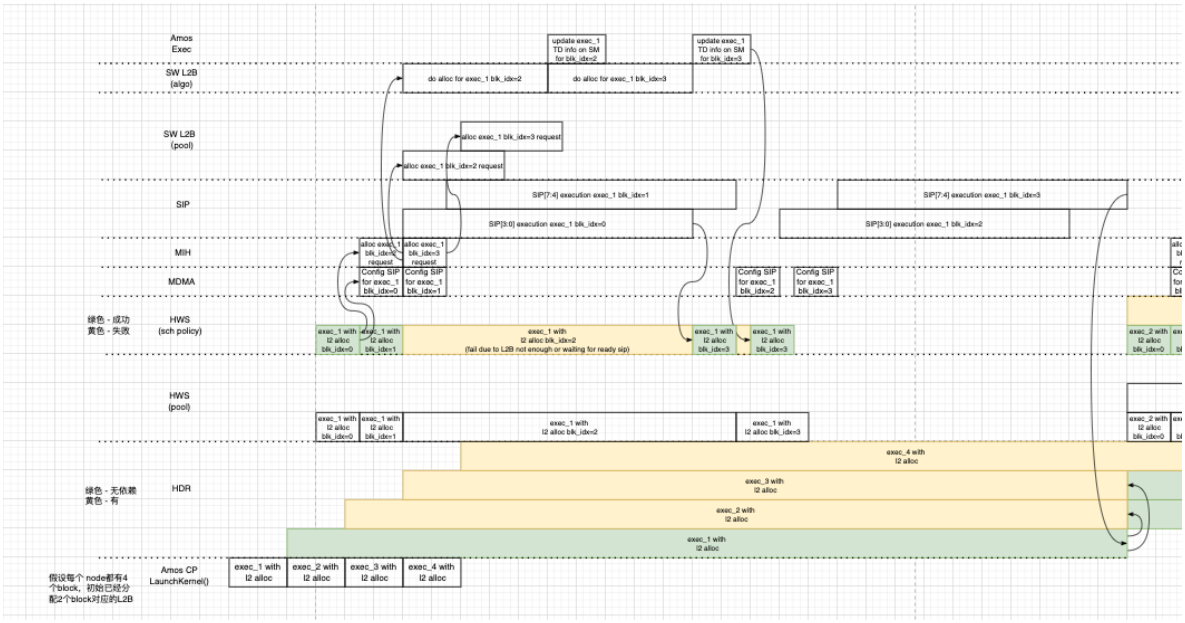

虚线： 任务从MDMA发出即可解依赖
实线：任务完全完成才可以解依赖
1 Cache Prefetch
1 Exec
2 Cache Prefetch
3 Cache Prefetch
2 Exec
3 Exec
4 Cache Prefetch
4 Exec
5 Cache Prefetch
5 Exec
6 Cache Prefetch
6 Exec

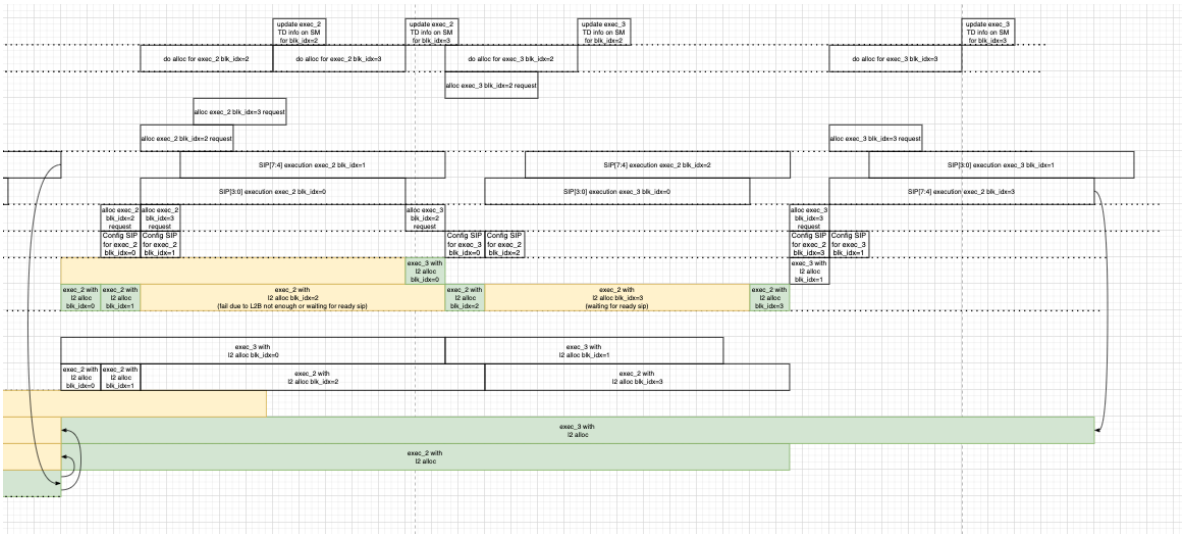


核心思路：

1. 调度器按照节点被调度执行的相同顺序排序prefetch节点，并按此顺序提前提交给Amos

Case2: Dynamic L2B Allocation with Scheduling

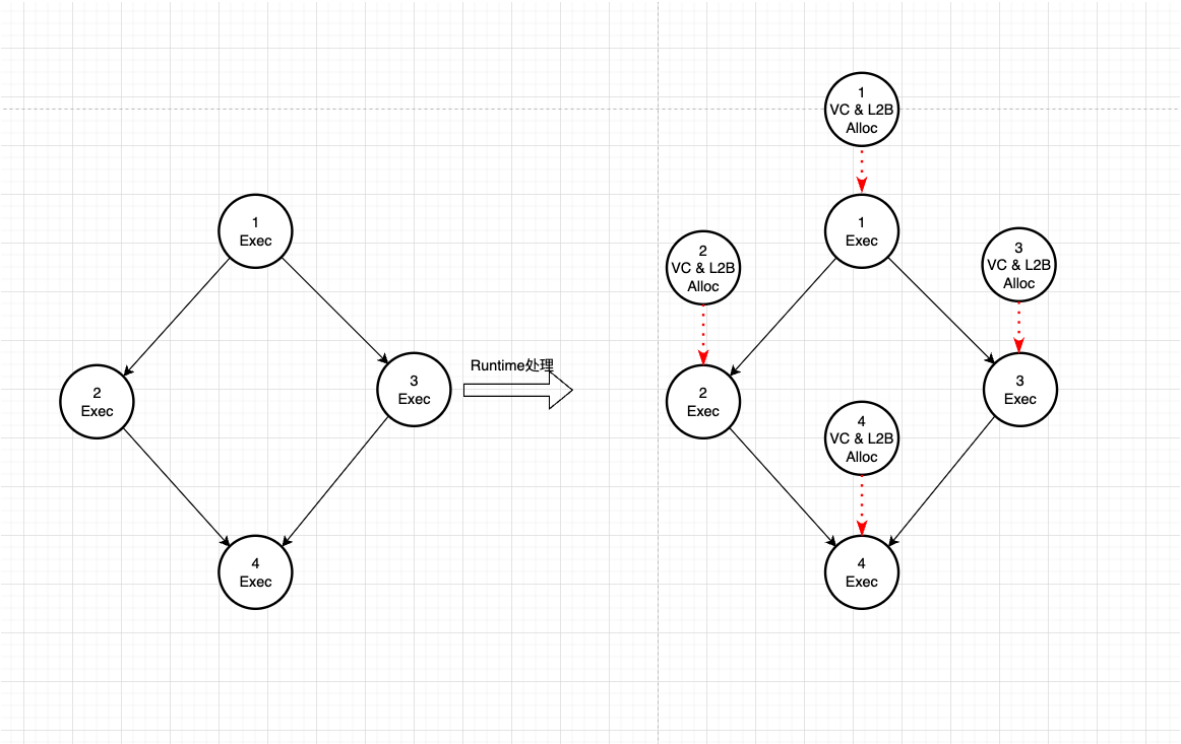




软件可以利用kernel调度算法应对的场景：

场景	Host Runtime	async launch前	async launch后	性能收益
SW CTX与VG一一对应	无	1. 除了SIP外，所有资源都在async launch前分配 2. 按照原始拓扑序建立SIP kernel间的依赖关系	HWS-SIP负责调度kernel以及SIP的分配	Pros： 最简单的能启用硬件动态调度的机制的方法 Cons： 1. launch前占用的资源多，Amos launch的时间点可能远早于kernel启动时间。会因为资源耗尽，限制住outstanding的kernel的数量。 2. 且async launch的kernels如果有依赖关系，在占用资源的同时又无法执行，导致SIP、算力利用率进一步降低
		1. 除了L2B和SIP外，所有资源提前分配好 2. 挂载L2B分配+预取节点，并与SIP kernel建立依赖关系	1. HWS-SIP负责调度L2B分配和SIP启动任务。 2. 根据Task Desc的信息，判断	

```
struct KernelFuncAttrs {
    // Kernel版本号表示软件相关的兼容性
    uint16_t kernel_major_version;      // kernel大版本，大版本不同不兼容
    uint16_t kernel_minor_version;      // kernel小版本，小版本不同必须兼容
    // Machine版本号表示硬件相关的兼容性
    uint16_t machine_major_version;     // ISA大版本，大版本不同不兼容
    uint16_t machine_major_version;     // ISA大版本，大版本不同不兼容
    uint32_t calling_convention;        // Calling Convention的类型

    uint32_t explicit_param_size;       // 显式参数的总大小
    uint32_t implicit_param_size;       // 隐式参数的总大小
    uint32_t boot_param_size;           // boot code参数的总大小
    uint32_t param_aligned_size;        // 128 Bytes对齐的参数大小（包含上述三种参数）

    uint32_t stack_size;                // 需要的stack mem大小，每个thread一份，不需要分配，供兼容性检测（在冯闯提议的全动态分配逻辑下，需要Supervisor分配）
    uint32_t local_mem_size;            // 需要的local mem大小，每个thread一份，不需要分配，供兼容性检测（在冯闯提议的全动态分配逻辑下，需要Supervisor分配）
    uint32_t shared_mem_size;           // 需要的shared mem大小，每个block一份，需要SP分配
    uint32_t local_vc_count;            // 需要的SDMA VC的数量，每个thread一份，不需要分配，供兼容性检测（在冯闯提议的全动态分配逻辑下，需要Supervisor分配）
    uint32_t local_mbx_count;           // 需要的Mailbox的数量，每个thread一份，不需要分配，供兼容性检测（在冯闯提议的全动态分配逻辑下，需要Supervisor分配）
    uint32_t shared_vc_count;           // 需要的CDMA VC的数量，每个block一份，需要SP分配
    uint32_t shared_gsync_counter_count;// 需要的GSYNC计数器的数量，每个block一份，需要SP分配
    uint32_t shared_gsync_entry_count;  // 需要的GSYNC entry的数量，每个block一份，需要SP分配
    uint32_t global_gsync_counter_count;// 需要的GSYNC计数器的数量，总共一份，需要SP分配
    uint32_t global_gsync_entry_count;  // 需要的GSYNC entry的数量，总共一份，需要SP分配

    // 以上属性是Pavo和Dorado需要的，以下是Scorpio需要新增的
    uint32_t local_mutex_count;         // 需要的本地mutex数量，每个thread一份，不需要分配，供兼容性检测（在冯闯提议的全动态分配逻辑下，需要Supervisor分配）
    uint32_t local_barrier_count;       // 需要的本地barrier counter数量，每个thread一份，不需要分配，供兼容性检测（在冯闯提议的全动态分配逻辑下，需要Supervisor分配）
    uint32_t local_latch_count;         // 需要的本地latch counter数量，每个thread一份，不需要分配，供兼容性检测（在冯闯提议的全动态分配逻辑下，需要Supervisor分配）
    uint32_t local_queue_count;         // 需要的本地queue数量，每个thread一份，不需要分配，供兼容性检测（在冯闯提议的全动态分配逻辑下，需要Supervisor分配）

    uint32_t shared_mutex_count;        // 需要的共享mutex数量，每个block一份，需要SP分配
    uint32_t shared_barrier_count;      // 需要的共享barrier counter数量，每个block一份，需要SP分配
    uint32_t shared_latch_count;        // 需要的共享latch counter数量，每个block一份，需要SP分配
    uint32_t shared_queue_count;        // 需要的共享queue数量，每个block一份，需要SP分配


    uint32_t global_mutex_count;        // 需要的共享mutex数量，总共一份，需要SP分配
    uint32_t global_barrier_count;      // 需要的共享barrier counter数量，总共一份，需要SP分配
    uint32_t global_latch_count;        // 需要的共享latch counter数量，总共一份，需要SP分配
    uint32_t global_queue_count;        // 需要的共享queue数量，总共一份，需要SP分配
}
```



Problem to Solve

引入HWS的根本原因在于软件当想走动态执行时，在device runtime时候动态的获取信息完成调度的时间点可能无法及时追踪资源变化状态。并且在高频率的调度需求下，可能来不及做决策。因此引入了HWS去更好的适应实时变化的资源和更快的决策。

但也因为HWS的引入，使得原本都由FW做的所有资源分配策略出现了割裂，导致SIP的调度，L2资源分配，VC资源分配的决策者就不是同一个master，也不在同一个时间点。就会出现资源分配的配合问题，最终影响调度效果。

在三个场景下收益：

1. prefetch 在更接近执行的位置再回调
2. 在非prefetch，单cluster内，资源动态分配时total footprint 太大，现用现分
3. 225W时候，多cluster，共享复用cluster的SIP，L2B的现用现分

目标：

软件能控制硬件与Amos协同完成资源，调度的协同工作

拆分：

1. Runtime通过API及附加指引（eg. Kernel_Action），告诉Amos如何配置每个任务，并且设置好任务之间的关联关系
2. Amos根据cmd中的指引，结合定义的Task Desc（Amos与HW的编程接口），完成从cmd到TD的配置以及必要转换
3. 硬件支持资源是否ready的检测及pause功能，根据kernel的TD的配置，进行必要的callback Amos，完成资源分配的复杂算法。

```
struct Kernel_Action { //Produce by Runtime, consume by Amos
    //L2 buffer allocation action
    enable_l2_buffer_alloc;
    l2_buffer_alloc_size_per_block;
    num_block_to_alloc_l2_buffer;
    outstanding_block_num; //how much blocks' l2 can be allocated ahead of current block launched for this kernel. eg. block_idx=0 launched to sip, if ost=8, then HWS can call back Amos at most 8 more times for block's l2 allocation.

    //VC
    enable_cdte_vc_alloc;
    enable_hw_alloc; //amos or hws, hw only support continous VC allocation
    cdte_vc_alloc_num;

    //SIP
    enable_sip_alloc;
```

```
    //more
}
//produce by Amos, consume by HW:
//info comes from Kernel_Action and KernelFuncAttrs
struct task_desc {
    //L2 buffer allocation action
    enable_l2_buffer_alloc;
    l2_buffer_alloc_size_per_block;
    num_block_to_alloc_l2_buffer;
    num_block_l2_alloc_done[8]; //indicate number of block's l2 address allocated
already on each VG; Amos write, HW read-only

    //VC
    enable_cdte_vc_alloc;
    enable_hw_alloc; //amos or hws, hw only support continous VC allocation
    cdte_vc_alloc_num;
    num_block_cdte_vc_alloc_done[2]; //indicate number of block's cdte_vc allocated
already on each Die; Amos write, HW read-only

    //SIP
    enable_sip_alloc;
    //more attrs
    sip_alloc_affinity_mask;
    offline_priority;
    //more
}
//=========================================================================
// Host Runtime Setup
//=========================================================================
//setup for allocation node
/*alloc_node.kernel_action() {
    enable_l2_buffer_alloc = 1;//enable
    l2_buffer_alloc_size_per_block = 256KB;
    num_block_to_alloc_l2_buffer = 1; //assume block_num=16, only allocate 1 block's
L2B before start to execute
    outstanding_block_num = 1;

    enable_cdte_vc_alloc = 1;
    enable_hw_alloc = 0; //callback amos for VC allocation
    cdte_vc_alloc_num = 4; //4 VC

    enable_sip_alloc = 0; //allocation node don't need sip scheduling
}*/

//setup for execution node
exec_node.kernel_action() {
    enable_l2_buffer_alloc = 1;//enable
    l2_buffer_alloc_size_per_block = 256KB;
    num_block_to_alloc_l2_buffer = 16; //assume block_num=16, only allocate 1 block's
L2B before start to execute
```

```
    outstanding_block_num = 4;

    enable_cdte_vc_alloc = 1;
    enable_hw_alloc = 0; //callback amos for VC allocation
    cdte_vc_alloc_num = 4; //4 VC

    enable_sip_alloc = 1;
}

//=====================================================================
// Amos Preparation
//=====================================================================
exec_node.task_desc() {
    enable_l2_buffer_alloc = 1;//enable
    l2_buffer_alloc_size_per_block = 256KB;
    num_block_to_alloc_l2_buffer = 16; //assume block_num=16, only allocate 1 block's
L2B before start to execute
    outstanding_block_num = 4;
    num_block_l2_alloc_done[3:0] = {0,0,1,1}; //

  //block 0 laucnh sip
  //read share TDP[16]

    enable_cdte_vc_alloc = 1;
    enable_hw_alloc = 0; //callback amos for VC allocation
    cdte_vc_alloc_num = 4; //4 VC
    num_block_cdte_vc_alloc_done[1:0] = {8, 4}; //example

    enable_sip_alloc = 1;

}

//=====================================================================
// HW Logic
//=====================================================================
bool schedule_kernel() {
  while() {
    for(each task in Pool) {
      pick_sip_schedule_tasks(input=task, output=SIPTaskPool);
      pick_l2_schedule_tasks(input=task, output=L2TaskPool);
      pick_vc_schedule_tasks(input=task, output=VCTaskPool);
    }

    thread1.run(schedule_sip(SIPTaskPool));
    thread2.run(alloc_l2(L2TaskPool));
    thread3.run(alloc_vc(VCTaskPool));
  }
}
//8 kernels(8 block per kern) pending
//4 vc per block, 8 vc total
```

```
//kernel 0, vc0-3 assign , 4 vc left
//kernel



void pick_l2_schedule_tasks(input=task, output=L2TaskPool) {
  if(task.enable_l2_buffer_alloc == 1 &&
     (sum(num_block_l2_alloc_done[7:0]) - task.block_idx) < outstanding_block_num) {

    L2TaskPool.push_back(task);//add the task as l2 allocation candidate
  }
void pick_sip_schedule_tasks(input=task, output=SIPTaskPool) {
  if(task.enable_sip_alloc == 1 && (sum(num_block_l2_alloc_done[7:0]) >
task.block_idx) {

    SIPTaskPool.push_back(task);//add the task as sip scheduling candidate
  }

}

}
```

----------分割线----------

部署策略

1. 介绍软件如何配合硬件的这个设计来达到预期效果：
    1. 静态图/graph这种，需要软件提前计算哪些信息，（每个节点的HEFT 计算方式求得的优先级，每个节点到exit node的critical path 长度，dag的编号）
    2. 动态下发，create多个sub-dag或者section的概念，当软件检测到动态图产生分叉，则create section，合并时则可以重用/merge section。软件需要不断的累计并更新section的优先级，随着新的任务实时的传给调度系统。然后调度算法是基于每个dag中的section的优先级，来决策每个section内节点的优先级。

       同时，还需要引入Normalization来应对不同DAG的绝对计算cost不相同的问题，软件需要统筹所有动态图的优先级标准，并把section优先级的更新考虑上normalization

性能注意点

1. HWS内部的ReadyStation在的entry进来的时候先去Share Memory进行Task Descriptor的读取，保证此时HWS访问Share Memory的带宽是均化的，而不会出现HDR解完依赖后，突发需要读多个

动态优先级调度

Addon：

1. 可以考虑HWS里增加一个寄存器，来接收early-finish的状态，到时候让supervior或compute thread来主动回call
2. HWS里一定会有接收SIP kernel结束状态的寄存器

Reference

1.
2. 《Queue Waiting Time Aware Dynamic Workflow Scheduling in Multicluster Environments》可以参考一些queue wait time的影响，以及评估节点cost的方式