**Arbitrar Design:**

**Static determination of kernel instances:**

Since input image sizes are constant for Deep Learning frameworks, static computation time can be calculated for each kernel and based on the same inverse ratio, number of instances can be computed.

**Global Buffer:**

Global Buffer will have two sections: 1) Input/Output Buffer (Controlled by Host) 2) Intermediate Sub Buffers used by computational pipeline kernels.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |



**Information Exchanges and Bookkeeping:**

1. **HLS Stream Exchange Packet between Arbitrar and Computation Kernels.**

| Sequence Number | Stage Number | Stage IDs | Buffer Offset In | Buffer Offset Out |
| --- | --- | --- | --- | --- |

**Sequence Number (1 Byte)**: ID corresponding to the pipeline sequence for current operation. To make sure that the input image and computational pipeline sequence is uniquely identified as each point in time.

**Stage Number (1 Byte)**: What is the current operation happening with in the whole pipeline of multiple computational kernels. Index of operation in the pipeline kernel stage list.

**Stage IDs (Max Number of Pipeline Stages \* 1 Byte)**: Sequential list of all the computation kernel IDs in the current pipeline. Avoid storing these in GMEM (very high latency) or kernel memory (overhead of bookkeeping this information for all pipelines running in parallel).

**Sub-Buffer Offset In (1 Byte)**:Offset for sub buffer in the GMEM buffer that holds the output from previous computational kernel in the pipeline.

**Sub-Buffer Offset Out (1 Byte)**:Offset for sub buffer in the GMEM buffer that will hold the output from the current computational kernel in the pipeline.

* TODO: List out pros and cons for both designs.

Graphical user interface, application

Description automatically generated

*Figure 1 Sub-Buffer Offset in the GMEM main Buffer.*

1. **Arbitrar Bookkeeping:**

**Buffer Consumer Producer model**: Track dynamic use of sub buffers and allocate available sub buffers to new computational kernels.

* Counters for producer and consumer, holding usage of sub buffers (synchronization) and computer units (separate for separate computational kernels)
* Maintain In-Use flag array for all sub buffers. (Track whether corresponding sub-buffer is in use or not, set when arbitrar sends request to kernel; unset when kernel responds)

**Wait Queue**: Maintain pipeline state meta-data in case of non-availability of memory (sub-buffers) or compute unit (all kernel instances are busy for required operation)

* Hold entries for Packet Data in case of resource unavailability. Ordered based on time. (FIFO based)
* Separate for state starving for memory and starving for compute unit.
* Direct FIFO scheme for buffer availability. Compute unit check at time of completion of kernel (check for starving for that compute unit an accordingly allocate from wait list)

**Running Queue:** Maintains pipeline state meta-data (just number IDs of computational kernel instances executing since all other information will be received from HLS) for kernels that have been put into execution. Arbitrar needs to remember which kernels are into execution so that it can poll for results from them.

* Use non-blocking reads from HLS streams to make sure that arbitrar does not waste time in polling.
* Timing for polling logic is essential.

**Basic Algorithm Flow: (Initial Iteration, No error handling)**

**Functions:**

***poll\_running\_queue():***

*for tasks in running\_queue:*

*status = non\_blocking\_read(tasks[i].hlsstreamOut)*

*if(status):*

*if(tasks[i].nextstage == done):*

*write\_to\_out\_buffer();*

*remove from queue;*

*else if (cu\_available(tasks[i].stages[next\_stage].CU) && buffer\_available):*

*schedule tasks[i]; // next stage*

*update\_running\_queue();*

*else:*

*add\_to\_wait\_queue();*

***args (GMEM buffer start pointer, HLS stream connections)***

***arbitrar start:***

***load tasks:*** *// Load and schedule tasks until CU exhaustion*

*burst access gmem data to kernel memory;*

*while (available CUs > 0 && pipelines > 0):*

*if(pipeline[i] != SCHEDULED):*

*if(cu\_available(schedule pipeline[i].stages[0].CU)):*

*schedule pipeline[i];*

*CU\_Counter[schedule pipeline[i].stages[0].CU]++;*

*mark schedule pipeline[i] as SCHEDULED;*

*i++;*

*pipelines--;*

*else:*

*add\_to\_wait\_queue(pipeline[i]);*

*available\_CUs = calculate\_cus();*

***start scheduler:*** *// loop until wait queue and running queue both are empty*

*while (!(wait\_queue.empty() && running\_queue.empty())):*

*poll\_running\_queue ();*

*exhaust\_available\_cus:*

*// Check for more entries from host to load. (not sure??)*

*for pipelines from host (available CUs > 0 && pipelines > 0):*

*if(pipeline[i] != SCHEDULED):*

*add\_to\_wait\_queue();*

*for (tasks in wait\_queue):*

*if (cu\_available(tasks[i].stages[current\_stage].CU) && buffer\_available):*

*schedule tasks[i];*

*add\_to\_running\_queue();*

*remove\_from\_wait\_queue();*

***return:***

***Kernels Interface:***

***Input Arguments : (\*in , \*sub , HLS\_IN, HLS\_OUT)***

***\*in :*** *Pointer to start the input buffer.*

***\*sub :*** *Pointer to start of the buffer used for internal processing.*

***HLS\_IN:*** *Input HLS stream connection with arbitrar to receive the processing header with all the information mentioned before*

***HLS\_OUT:*** *Output HLS stream connection with arbitrar to send an acknowledgement to arbitrar to know that the processing is completed on the given input.*

***Header exchange needs to have a fields separately for kernel algorithm related arguments : Ex: Resizing (sizing window), Brightness (percentage) (Since kernels will be static and logic in arbitrer will have static knowledge of kernels, header population will be different for different kernels)***

***Output from Kernels: (Input Header for acknowledgement without the kernel specific inputs)’***

***General Algo for Computational Kernels:***

***kernel start:***

*read kernel header();*

*input\_sanity\_checks();*

*call\_standard\_computational\_api()*

*prepare acknowledgement header;*

*write to output stream;*

***end:***

***Since, we need to provide a common interface for different computation kernels, it is better to have APIs for different computations and use these APIs for the above implementation. These APIs can also be directly leveraged by other applications directly.***

***Common interface for APIs?? (Arguments might be different)***

***Timeline:***

***Base Branch by 4/28***

***Implementing base arbitrer logic 5/09***

***-> Computational kernel/APIs implementation***

***->Integrate with arbiter logic code.***