Intel® OpenSource HD Graphics Programmer's Reference Manual (PRM)

Volume 2 Part 2: Media and General Purpose Pipeline (Ivy Bridge)

For the 2012 Intel® Core™ Processor Family

May 2012

Revision 1.0

NOTICE:
This document contains information on products in the design phase of development, and Intel reserves the right to add or remove product features at any time, with or without changes to this open source documentation.

Doc Ref #: IHD-OS-V2 Pt 2 – 05 12

Creative Commons License

You are free to Share — to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

No Derivative Works. You may not alter, transform, or build upon this work.

Contents

1. Media and General Purpose Pipeline

1.1 Introduction

This section covers the programming details for the media (general purpose) fixed function pipeline. The **media pipeline** is positioned in parallel with the 3D fixed function pipeline. It provides media functionalities and has media specific fixed function capability. However, the fixed functions are designed to have the general capability of controlling the shared functions and resources, feeding generic threads to the Execution Units to be executed, and interacting with these generic threads during run time. The media pipeline can be used for non-media applications, and therefore, can also be referred to as the **general purpose pipeline**. *For the rest of this chapter, we refer to this fixed function pipeline as the media pipeline, keeping in mind its general purpose capability.*

Concurrency of the media pipeline and the 3D pipeline is not supported. In other words, only one pipeline can be activated at a given time. Switching between the two pipelines within a single context is supported using the MI_PIPELINE_SELECT command.

Following are some media application examples that can be mapped onto the media pipeline. All these applications are functional; however, the level of performance that can be achieved depends on the hardware configuration and is beyond the scope of this document.

- MPEG-2 decode acceleration with HWMC (e.g. DXVA HWMC interface)
- MPEG-2 decode acceleration with IS/IDCT and forward (e.g. DXVA IDCT interface)
- MPEG-2 decode acceleration with VLD and forward (e.g. DXVA VLD interface)
- AVC decode acceleration with HWMC and forward including Loop Filter
- AVC decode acceleration with IT and forward including Loop Filter (with hardware IT)
- VC1 decode acceleration with HWMC and forward including Loop Filter
- VC1 decode acceleration with IT and forward including Loop Filter (with hardware IT and overlap filter)
- Advanced deinterlace filter (motion detected or motion compensated deinterlace filter)
- Video encode acceleration (with various level of hardware assistant)

1.1.1 Terminologies

AVC	Advanced Video Coding. An international video coding standard jointly developed by MPEG and ITU. It is also known as H.264 (ITU), or MPEG-4 Part 10 (MPEG).
Child Thread	A thread corresponding to a leaf-node or a branch-node in a thread generation hierarchy. All thread originated from kernels running on the execution units are child threads.
EOB	End of Block. It is a 1-bit flag in the non-zero DCT coefficient data structure indicating the end of an 8x8 block in a DCT coefficient data buffer.
IDCT	Inverse Discrete Cosine Transform. It is the stage in the video decoding pipe between IQ and MC.
ILDB	In-loop Deblocking Filter – the deblocking filter operation in the decoding loop. It is a stage after MC in the video decoding pipe.
IQ	Inverse Quantization. It is a stage in the video decoding pipe between IS and IDCT.

IS	Inverse Scan. It is a stage in the video decoding pipe between VLD and IQ. In this stage, a sequence of none-zero DCT coefficients are converted into a block (e.g. an 8x8 block) of coefficients. VFE unit has fixed functions to support IS.
IT	Inverse Integer Transform. It is the stage in AVC or VC1 video decoding pipe between IQ and MC.
MPEG	Motion Picture Expert Group. MPEG is the international standard body JTC1/SC29/WG11 under ISO/IEC that has defined audio and video compression standards such as MPEG-1, MPEG-2, and MPEG-4, etc.
MC	Motion Compensation. It is part of the video decoding pipe.
MVFS	Motion Vector Field Selection – a four-bit field selecting reference fields for the motion vectors of the current macroblock.
PRT	A persistent root thread in general stays in the system for a long period of time. It is normally a parent thread. Only one PRT is allowed in the system. Hardware is responsible of re-dispatching the incomplete PRT at context restore, and a PRT can continue operations from that previously left-over state.
Parent Thread	A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy.
Root Thread	A thread corresponding to a root-node in a thread generation hierarchy. In the general-purpose pipeline, all threads originated from VFE unit are root threads.
Synchronized Root Thread	A root thread that is dispatched by TS upon a 'dispatch root thread' message.
TS	Thread Spawner. It is the second (and the last) fixed function in the general-purpose pipeline.
Unsynchronized Root Thread	A root thread that is automatically dispatched by TS.
VFE	Video Front End. It is the first fixed function in the general-purpose pipeline.
VLD	Variable Length Decode. It is the first stage of the video decoding pipe that consists mainly of bit-wide operations. The architecture supports hardware MPEG-2 VLD acceleration in the VFE fixed function stage.

1.1.2 Hardware Feature Map in Products

The following table lists the hardware features in the media pipe.

Video Front End Features in Device Hardware

Features	
Generic Mode	**Y**
Root Threads	**Y**
Parent/Child Threads	**Y**
SRT (Synchronized Root Threads)	**Y**
PRT (Persistent Root Thread)	**Y**
Interface Descriptor Remapping	N
IS Mode (HW Inverse Scan)	N
VLD Mode (HW MPEG2 VLD)	N

Features	
AVC MC Mode	N
AVC IT Mode (HW AVC IT)	N
AVC ILDB Filter (in Data Port)	N
VC1 MC Mode	N
VC1 IT Mode (HW VC1 IT)	N
Stalling HW Scoreboard	**Y**
Non-stalling HW Scoreboard	**Y**
HW Walker	**Y**
HW Timer	**Y**
Pipelined State Flush	**Y**
HW Barrier	**Y**

1.2 Media Pipeline Overview

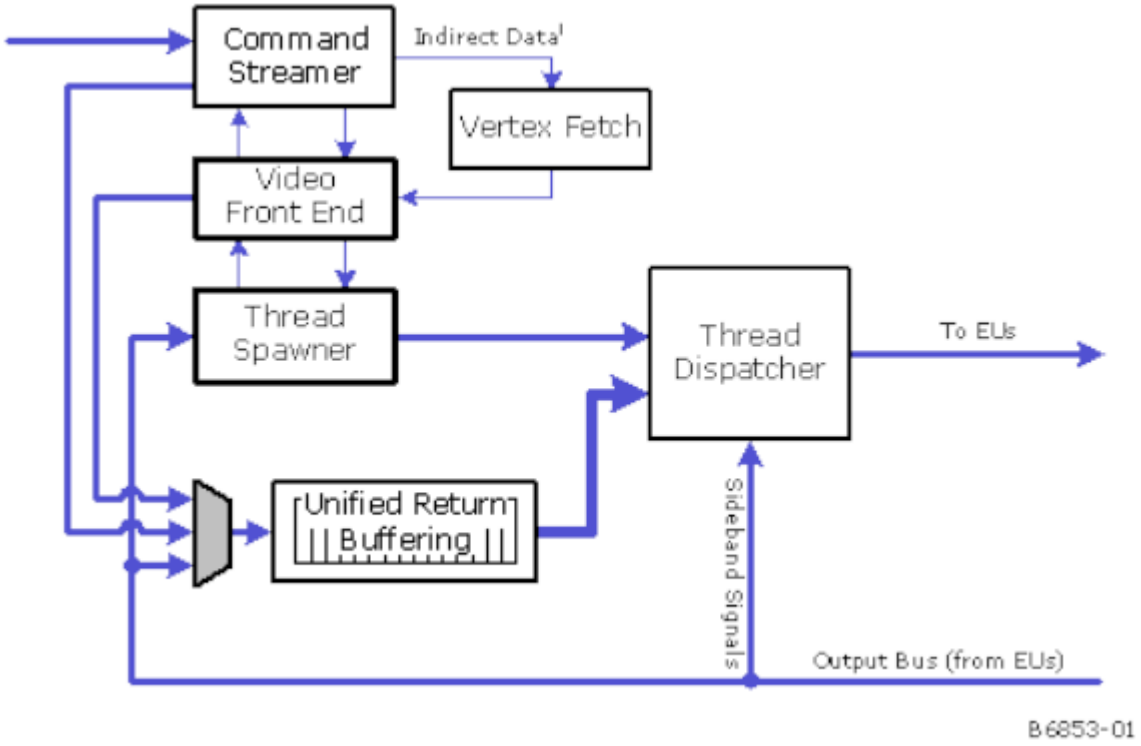
The media (general purpose) pipeline consists of two fixed function units: Video Front End (VFE) unit and Thread Spawner (TS) unit. VFE unit interfaces with the Command Streamer (CS), writes thread payload data into the Unified Return Buffer (URB), and prepares threads to be dispatched through TS unit. VFE unit also contains a hardware Variable Length Decode (VLD) engine for MPEG-2 video decode. TS unit is the only unit of the media pipeline that interfaces to the Thread Dispatcher (TD) unit for new thread generation. It is responsible for spawning root threads (short for the root-node parent threads) originated from VFE unit and for spawning child threads (can be either a leaf-node child thread or a branch-node parent thread) originated from the Execution Units (EU) by a parent thread (can be a root-node or a branch-node parent thread).

The fixed functions, VFE and TS, in the media pipeline, in most cases, share the same basic building blocks as the fixed functions in the 3D pipeline. However, there are some unique features in media fixed functions as highlighted by the followings.

- VFE manages URB and only has write access to URB; TS does not interface to URB.
- When URB Constant Buffer is enabled, VFE forwards TS the URB Handler for the URB Constant Buffer received from CS.
- TS interfaces to TD; VFE does not.
- TS can have a message directed to it like other shared functions (and thus TS has a shared function ID), and it does not snoop the Output Bus as some other fixed functions in the 3D pipeline do.
- A root thread generated by the media pipeline can only have up to one URB return handle.
- If a root thread has a URB return handle, VFE creates the URB handle for the payload to initiating the root thread and also passes it alone to the root thread as the return handle. The root thread then uses the same URB handle for child thread generation.

- If URB Constant Buffer is enabled and an interface descriptor indicates that it is also used for the kernel, TS requests TD to load constant data directly to the thread's register space. For root thread, constant data are loaded after R0 and before the data from the other URB handle. For child thread, as the R0 header is provided by the parent thread, Thread Spawner splits the URB handles from the parent thread into two and inserts the constant data after the R0 header.
- A root thread must terminate with a message to TS. A child thread should also terminate with a message to TS.
- High streaming performance of indirect media object load is achieved by utilizing the large vertex cache available in the Vertex Fetch unit (of the 3D pipeline).

Top level block diagram of the Media Pipeline



1.2.1 GPGPU Media Pipe Differences

You can access the GPGPU pipe with the GPGPU_OBJECT and GPGPU_WALKER commands. A thread group id is associated with every dispatch, which is used to allocate and track barriers and Shared Local Memory. The GPGPU pipe has access to all the shared functions. The GPGPU pipe does not use the Scoreboard and should not dispatch child threads.

You can access the Media pipe with the various MEDIA_OBJECT* commands. Barriers and Shared Local Memory are not allocated for them. All shared functions are available. The Scoreboard is available to control dispatch depending on the completion of neighboring blocks.

1.3 Programming Media Pipeline

1.3.1 Command Sequence

Media pipeline uses a simple programming model. Unlike the 3D pipeline, it does not support pipelined state changes. Any state change requires an MI_FLUSH or PIPE_CONTROL command. When programming the media pipeline, it should be cautious to not use the pipelining capability of the commands described in the Graphics Processing Engine chapter.

To emphasize the non-pipeline nature of the media pipeline programming model, the programmer should note that if any one command is issued in the “Primitive Command” step, none of the state commands described in the previous steps cannot be issued without preceding with a MI_FLUSH or PIPE_CONTROL command.

With the addition of MEDIA_STATE_FLUSH command, pipelined state changes are allowed on the media pipeline. The MEDIA_STATE_FLUSH serves as a fence for state change by flushing the VFE/TS front ends but not waiting for threads to retire.

The basic steps in programming the media pipeline are listed below. Some of the steps are optional; however, the order must be followed strictly. Some usage restrictions are highlighted for illustration purpose. For details, refer to the respective chapters for these commands.

1.3.1.1 Command Sequence

For, the media pipeline is further simplified with fixed functions like MPEG2 VLD and AVC/VC1 IT removed. The addition includes (1) CURBE command is now unique to the media pipeline and (2) the interface descriptors are delivered directly as a media state command instead of being loaded through indirect state.

The programming model is listed as the following.

- Step1: MI_FLUSH/PIPE_CONTROL
    - This step is mandatory.
    - Multiple such commands in step 1 are allowed, but not recommended for performance reason.
- Step2: State command PIPELINE_ SELECT
    - This step is optional. This command can be omitted if it is known that within the same context media pipeline was selected before Step 1.
    - Multiple such commands in step 2 are allowed, but not recommended for performance reason.
- Step3: State commands configuring pipeline states
    - STATE_BASE_ADDRESS
        - This command is mandatory for this step (i.e. at least one).
        - Multiple such commands in this step are allowed. The last one overwrites previous ones.
        - This command must precede any other state commands below.
        - Particularly, the fields **Indirect Object Base Address** and **Indirect Object Access Upper Bound** are used to control indirect Media object load in VF.

        - The fields **Dynamics Base Address** and **Dynamics Base Access Upper Bound** are used to control indirect Curbe and Interface Descriptor object load in VF.
        - *Note: This command may be inserted before (and after) any commands listed in the previous steps (Step 1 to 3). For example, this command may be placed in the ring buffer while the others are put in a batch buffer.*
    - STATE_SIP
        - This command is optional for this step. It is only required when SIP is used by the kernels.
    - MEDIA_VFE_STATE
        - This command is mandatory for this step (i.e. at least one).
        - This command cause destruction of all outstanding URB handles in the system. A new set of URB handles will be generated based on state parameters, no. of URB and URB length, programmed in VFE FF state.
        - Multiple such commands in this step are allowed. The last one overwrites previous ones.
    - MEDIA_CURBE_LOAD
        - This command is optional.
        - Multiple such commands in this step are allowed. The last one overwrites previous ones.
    - MEDIA_INTERFACE_DESCRIPTOR_LOAD
        - This command is mandatory for this step (i.e. at least one).
        - Multiple such commands in this step are allowed. The last one overwrites previous ones.
- Step4: Primitive commands
    - MEDIA_OBJECT
        - This step is optional, but it doesn't make practical sense not issuing media primitive commands after being through previous steps to set up the media pipeline.
        - Multiple such commands in step 4 can be issued to continue processing media primitives.

With the addition of MEDIA_STATE_FLUSH command, pipelined state changes are allowed on the media pipeline. In order to support context switch for barrier groups, watermark and barrier dependencies are added to the MEDIA_STATE_FLUSH command. The usage of barrier group may have strict restriction that all threads belonging to a barrier group must all be present in order to avoid deadlock during context switch. Here are the example programming sequences to allow context switch. Note that the use of MEDIA_OBJECT_PRT and MI_ARB_ON_OFF are optional.

- MEDIA_VFE_STATE
- MEDIA_INTERFACE_DESCRIPTOR_LOAD
- MEDIA_CURBE_LOAD (optional)
- MEDIA_GATEWAY_STATE (for example for barrier group 1)

- MEDIA_OBJECT_PRT (with VFE_STATE_FLUSH set and PRT NEEDED set.)
- MEDIA_STATE_FLUSH (with watermark set for group 1)
- MI_ARB_ON_OFF (OFF)// Arbitration must be turned off while sending objects for group 1
- Several MEDIA_OBJECT command (for barrier group 1)
- MI_ARB_ON_OFF (ON)// Arbitration is allowed
- MEDIA_STATE_FLUSH (optional, only if barrier dependency is needed)
- MEDIA_INTERFACE_DESCRIPTOR_LOAD (optional)
- MEDIA_CURBE_LOAD (optional)
- MEDIA_GATEWAY_STATE (for example for barrier group 2)
- MEDIA_STATE_FLUSH (with watermark set for group 1)
- MI_ARB_ON_OFF (OFF)// Arbitration must be turned off while sending objects for group 2
- Several MEDIA_OBJECT command (for barrier group 2)
- MI_ARB_ON_OFF (ON)// Arbitration is allowed
- …
- MI_FLUSH

Commands for the GPGPU pipe (GPGPU_OBJECT and GPGPU_WALKER) should be separated from commands for the Media pipe (MEDIA_OBJECT*) by an MI_FLUSH.

1.3.2 Interrupt Latency

Command Streamer is capable of context switching between primitive commands.

For all independent threads, it is not much a problem. The interrupt latency is dictated by the longest command that is likely to have the largest number of threads. For VLD mode, such a command may be corresponding to a largest slice in a high definition video frame. This is application dependent, there are not much host software can do. For Generic mode, programmer should consider to constrain the compute workload size of each thread.

In modes with child threads, a root thread may persist in the system for long period of time – staying until its child threads are all created and terminated. Therefore, the corresponding primitive command may also last for long time. The Software designer should partition the workload to restrict the duration of each root thread. For example, this may be achieved by partitioning a video frame and assigning separate primitive commands for different data partitions.

In modes with synchronized root threads, a synchronized root thread is dependent on a previous root or child thread. This means context switch is not allowed between the primitive command for the synchronized root thread and the one for the depending thread. So no command queue arbitration should be allowed between them. Software designer should also restrict the duration of such non-interruptible primitive command segments.

1.3.3 Programming the GPGPU Pipeline

1. In MEDIA_VFE_STATE choose whether to set DW2.6 Bypass Gateway Control. Usually this will be set, allowing the gateway to be used without OpenGateway/CloseGateway.
2. Set up interface descriptor with # of threads in barrier. The barrier id is not specified here because can Gen7 automatically assigns barriers to thread groups when they are free. The amount of CURBE data to deliver per thread dispatch is set in the interface descriptor.

3. Set up CURBE with thread ids and common data for all thread dispatches in the thread group.
4. Set up a GPGPU_WALKER command or a set of GPGPU_OBJECT commands with the thread group ids to dispatch the threads. The CURBE data is sent in sections for each thread dispatch in the thread group; a new thread group starts sending the CURBE data from the beginning of the buffer.

Note: Gen7 can either have the barriers and SLM automatically managed by hardware or specified by software. Mixing software managed and hardware managed in the same set of threads is allowed, but may cause stalls if there is an allocation conflict.

Note: When using GPGPU_OBJECT, finish dispatching a thread group before starting a different one.

The kernel should handle the barriers as follows:

The BarrierMsg message contains the barrier id and a way to reprogram the barrier count. The barrier count reprogram is not normally used for GPGPU workloads. When all threads in the group have reached the barrier, the gateway returns a notification bit 0.

The kernel must wait for the barrier to finish with a WAIT N0.

1.3.4 GPGPU Indirect Thread Dispatch

Indirect thread dispatch allows one thread group to control the group size of a following thread group.

This is the sequence of commands in the ring buffer:

GPGPU_OBJECT/WALKER	// Either a set of objects or a walker to dispatch a thread group which will write the next groups properties to memory
MI_FLUSH	// Make sure the thread group has finished executing
MEDIA_CURBE_LOAD	// Load the thread ids for new group
MI_LOAD_REGISTER_MEMORY	// Load the indirect MMIO GPGPU registers from the mem written by the previous group
GPGPU_WALKER (indirect)	// A walker with the indirect bit set.

The first thread group writes this data to memory:

1) The thread ids delivered in the CURBE - written where the following MEDIA_CURBE_LOAD will read them.
2) The GPGPU_WALKER parameters are written to memory where the MI_LOAD_REGISTER_MEMORY will read them.

a. GPGPU_DISPATCHDIMX - the X dimension of the number of thread groups to dispatch in dword 4.
b. GPGPU_DISPATCHDIMY - the Y dimension of the number of thread groups to dispatch in dword 6.
c. GPGPU_DISPATCHDIMZ - the Z dimension of the number of thread groups to dispatch in dword 8.

See vol1c Memory Interface and Command Stream for the MMIO register addresses and formats.

The indirect registers are not supposed to be set to 0, but sometimes the kernel computing the value wants no work done and sets them to 0. This does not work correctly, so a work-around in the command stream is needed:

GPGPU_WALKER // The thread group which writes the indirect values to memory locations

MI_CONDITIONAL_BATCH_BUFFER_END DIMX 0 // End batch buffer if X dim in memory = 0

MI_CONDITIONAL_BATCH_BUFFER_END DIMY 0 // End batch buffer if Y dim in memory = 0

MI_CONDITIONAL_BATCH_BUFFER_END DIMZ 0 // End batch buffer if Z dim in memory = 0

MI_LOAD_REGISTER_MEM GPGPU_DISPATCHDIMX DIMX // Normal load of register from memory

MI_LOAD_REGISTER_MEM GPGPU_DISPATCHDIMY DIMY

MI_LOAD_REGISTER_MEM GPGPU_DISPATCHDIMZ DIMZ

GPGPU_WALKER // The thread groups which depend on the indirect dimensions

1.3.5 GPGPU Context Switch

The GPGPU pipeline supports interruption of GPGPU workloads on thread group boundaries. This is needed for GPGPU workloads that are so large that there is a risk of the display becoming n on-responsive if the work cannot be interrupted for other jobs.

A workload is interrupted with the MI_ARB_CHECK command with the UHPTR register. The MI_ARB_CHECK command is placed throughout the command buffer. The driver updates the UHPTR register when a new context is needed; MI_ARB_CHECK checks for this and reprograms the head and tail pointers to the new batch of commands. The driver waits for the pre-emption to occur without going into RS2.

The GPGPU needs to modify this to allow a GPGPU_WALKER command to be interrupted. This is done by following each GPGPU_WALKER command with a MEDIA_STATE_FLUSH. This causes the CS to stop fetching commands until either the command completes or until the UHPTR valid bit is set.

GPGPU workloads can be dispatched with either GPGPU_OBJECT commands or GPGPU_WALKER commands. In the case of GPGPU_OJBECT, the MEDIA_STATE_FLUSH/ MI_ARB_CHECK pair must be placed in the batch buffer at thread group boundaries, since preemption cannot occur with a thread group partially dispatched. GPGPU_WALKER commands can dispatch multiple thread groups, in this case the MEDIA_STATE_FLUSH/ MI_ARB_CHECK follows each GPGPU_WALKER and the hardware takes care of noticing the UHPTR update and stopping at the next thread group boundary.

The commands in the batch buffer will look something like this:

Command Ring	Comments
MI_SET_CONTEXT	Go to GPGPU context
MI_BATCH_BUFFER_START	If new context, set address to top of batch. Otherwise, address needs to be set to the command preempted (given in the HWSP).

Command Batch	Comments
GPGPU_OBJECT	
GPGPU_OBJECT	
...	(more threads forming a complete thread group)
MEDIA_STATE_FLUSH	Check for preemption at thread group boundary. “Preemption” defined by the UHPTR valid bit set.
MI_ARB_CHECK	Move the head only if UHPTR valid bit is set.
…	
GPGPU_WALKER	
MEDIA_STATE_FLUSH	Check for preemption at thread group boundary internal to GPGPU_WALKER command. “Preemption” defined by the UHPTR valid bit set.
MI_ARB_CHECK	Move the head only if UHPTR valid bit is set.

The context saved will consist of the state commands for VFE and a modified GPGPU_WALKER command with a new starting thread group id. On context restore, the commands are executed to start the GPGPU_WALKER where it left off before continuing with the rest of the command buffer.

An example software model for starting a preemption goes like this:

1. The UHPTR is reprogrammed to point to the current tail of the ring buffer.
2. Insert new commands:
   a. LRI to UHPTR to clear valid.
   b. Store Register to mem the preempted batch offset.
   c. Store Register to mem the preempted ring offset.
   d. Pipe_control notification.
   e. An MI_SET_CONTEXT to the new context is put into the ring.
3. Insert commands for new context. i.e. batch buffers.
4. Update Tail Pointer.

Note: 2-3 items above could happen during execution of a thread group so the HW may see the tail pointer updated before preemption starts.

Note: The driver needs to turn off RC6 during items 1 and 4.

1.3.6 Media GPGPU Payload Limitations

There are 3 types of payload that the media/GPGPU instructions can have, but not all of them are allowed. The following table lists the legal combinations:

WorkLoad	Commands	Data Stored
GPGPU	GPGPU_WALKER	CURBE
	GPGPU_WALKER	INDIRECT
	GPGPU_OBJECT	CURBE
Media(Legacy)	Media_Object	CURBE
	Media_Object	INDIRECT
	Media_Object	INLINE
	Media_Object	CURBE+INLINE
	Media_Object	CURBE+INDIRECT
	Media_Object	INLINE+INDIRECT
	Media_Object	CURBE+ INLINE+INDIRECT
	Media_Object_Walker	CURBE
	Media_Object_Walker	INLINE
	Media_Object_Walker	CURBE+INLINE
Media using Barrier/SLM	Media_Object_GRPID	CURBE
	Media_Object_GRPID	INDIRECT
	Media_Object_GRPID	INLINE
	Media_Object_Walker (with group id)	CURBE
	Media_Object_Walker (with group id)	INLINE

1.3.7 Synchronization of the Media/GPGPU Pipeline

The Media/GPGPU Pipeline is synchronized in the same way as the 3D pipeline using the PIPE_CONTROL command.

See the Bspec section on 3d pipe synchronization: vol2a 3D Pipeline - Overview [SNB+] > 3D Pipeline > Synchronization of the 3D Pipeline.

1.4 Video Front End Unit

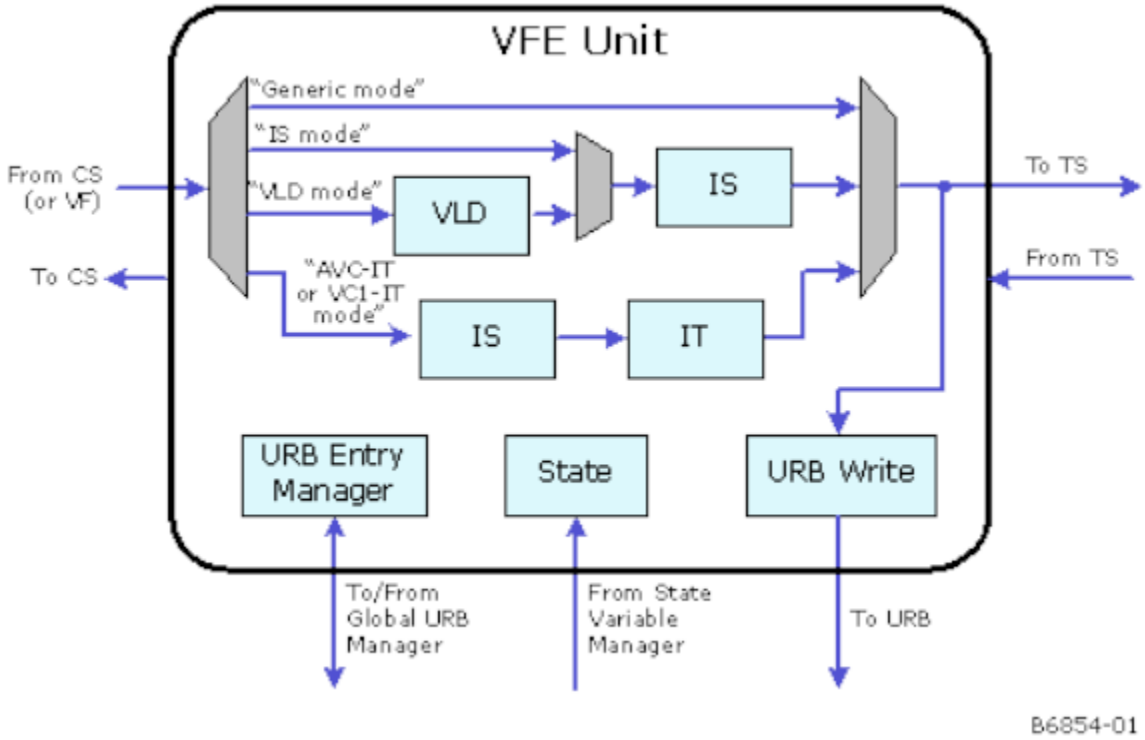
The Video Front End unit is the first fixed function unit in the media pipeline. It processes MEDIA_OBJECT commands to generate root threads by preparing the control (including interface descriptor pointers) and payload (data pushed into the GRF) for the root threads.

VFE supports three modes of operation: Generic mode, Inverse Scan mode and VLD mode.

- **Generic mode**: In the Generic mode, VFE serves as a conduit for general-purpose kernels fully configured by the host software. There is no application specific hardware enabled in this mode.
- **IS (Inverse Scan) mode**: The IS mode is a special mode for video decoding when off-host IDCT acceleration is supported by kernels running on execution units.
- **VLD mode**: It is a special mode for video decoding when MPEG-2 off-host VLD acceleration is supported by hardware.
- **AVC-IT (AVC Inverse Integer Transform) mode**: The AVC-IT mode is a special mode for AVC video decoding when off-host IDCT acceleration is supported by VFE hardware and MC and Loop Filter are supported by kernels running on execution units.
- **AVC-MC (AVC Motion Compensation) mode**: The AVC-MC mode is a special mode for AVC video decoding with host-based IDCT and MC and Loop Filter are supported by kernels running on execution units.
- **VC1-IT (VC1 Inverse Integer Transform) mode**: The VC1-IT mode is a special mode for VC1 video decoding when off-host IDCT acceleration is supported by VFE hardware and MC and Loop Filter are supported by kernels running on execution units.

The following figure illustrates the three modes of operation. The details can be found in the rest of the sections.

VFE Functional Blocks and Modes of Operations



MEDIA_STATE_POINTERS command configures VFE in one of the three modes using. Mode switching requires media pipeline state change.

1.4.1 Interfaces

VFE unit acquires its states from Sate Variable Manager, accesses URB handles from the Global URB Manager, receives state and primitive commands from CS unit, writes thread payloads to URB, and sends new thread to TS unit. It does not directly interface to Thread Dispatcher. When VFE is ready for a thread, it sends the interface descriptor pointer for the thread to TS.

1.4.1.1 Interface to Command Streamer

VFE interfaces to CS to acquire the control data, inline data and indirect data of MEDIA_OBJECT commands. The interface supports the throughput of a given mode of operation of VFE. For example, in VLD mode and IS mode, VFE consumes one dword at a time, one dword to the variable length decoder or one dword to the inverse-scan operator. In Generic mode, VFE is capable of a much higher throughput to push indirect data (as thread payload data) into URB. As throughput for indirect data is much higher than that of inline data, when large amount of user data need to be passed through VFE unit, if applicable, it is encouraged to use indirect object load.

1.4.1.2 Interface to Thread Spawner

VFE also transmits scratch memory base address received from State Variable Manager to TS, and passes on the Constant URB handle received from CS.

VFE receives URB handle dereference signal from TS.

1.4.1.3 Interface to State Variable Manager

State Variable Manager is responsible of fetching media state structure from memory. VFE only acquires its state variable upon the first primitive command. Therefore, host software is allowed to change media states before issuing primitive commands. As media pipeline does not support pipelined state change, a pipeline flush is required before any state change to make sure that there are no outstanding primitive commands in the pipeline.

1.4.1.4 Interface to Global URB Manager

VFE is responsible for managing URB handles for all root threads. Upon state change, VFE allocates URB handles through the Global URB Manager. VFE manages the URB handles in a circular buffer. URB handle referencing is in a strict order (taking from the head of the circular buffer), even though the handle dereferencing may occur out of order.

When starting a root thread, VFE reference one and only one URB handle, forwarding it to TS. TS then forwards this handle to TD for thread dispatching.

The URB handle for a root thread is used in two ways: (1) serving as buffer space for VFE to assemble thread payload, and (2) serving as the return URB buffer for the root thread to assemble child threads and their payload.

TS sends an indication to VFE when it is safe to dereference the URB handle, and VFE dereferences it. After a URB handle has been dereferenced, VFE can assign it to a new thread.

1.4.1.5 Interface to URB

VFE sends the assembled root thread payload to URB via a wide data bus. In Generic mode, the data comes from the command as inline or indirect data objects. In IS mode, the inline data is directly assembled as URB register wide payloads, and the indirect data are assembled through the Inverse Scan logic. In VLD mode, the data is decoded from the indirect object (i.e. bitstream data).

1.4.2 Mode of Operations

1.4.2.1 Generic Mode

In the Generic mode, VFE serves as a conduit for general-purpose kernels fully configured by the host software. As there is no special fixed function logic used, the Generic mode can also be viewed as a ‘pass-through’ mode. In this mode, VFE generates a new thread for each MEDIA_OBJECT command. The payload contained in the MEDIA_OBJECT command (inline and/or indirect) is streamed into URB. The interface descriptor pointer is computed by VFE based on the interface descriptor offset value and the interface descriptor base pointer stored in the VFE state. VFE then forwards the interface descriptor pointer and the URB handle to TS to generate a new root thread. Many media processing applications can be supported using the Generic mode: MPEG-2 HWMC, frame rate conversion, advanced deinterface filter, to name a few.

1.4.2.2 Interface Descriptor Selection

After populating the URB with the data, VFE notifies TS to initiate the thread. TS needs an interface descriptor pointer to fetch the information for thread initiation. A list of interface descriptors is arranged by the host software as a descriptor array in memory, as shown in the media state model in the Media State Model Figure.

VFE obtains the interface descriptor base pointer from the VFE state structure. The offset into the list of interface descriptors comes from MEDIA_OBJECT command. Each interface descriptor has a fixed size. VFE uses a multiple of the fixed size and the offset to add to the base pointer, and creates the final interface descriptor pointer to be sent to TS.

TS fetches the interface descriptor through the Instruction State Cache (ISC) using the interface descriptor pointer. TS then initializes the thread through the Thread Dispatcher. The interface descriptor pointer is given to TS by VFE for a root thread and by a thread for a child thread. The R0 header is formed by TS for a root thread and is stored in URB by the parent thread for a child thread.

1.4.2.3 Scratch Space Allocation

TS handles the allocation of scratch space. Since TS does not have a normal state interface, VFE receives the scratch space configuration with the VFE state, then forwards the configuration to TS with the interface descriptor pointer.

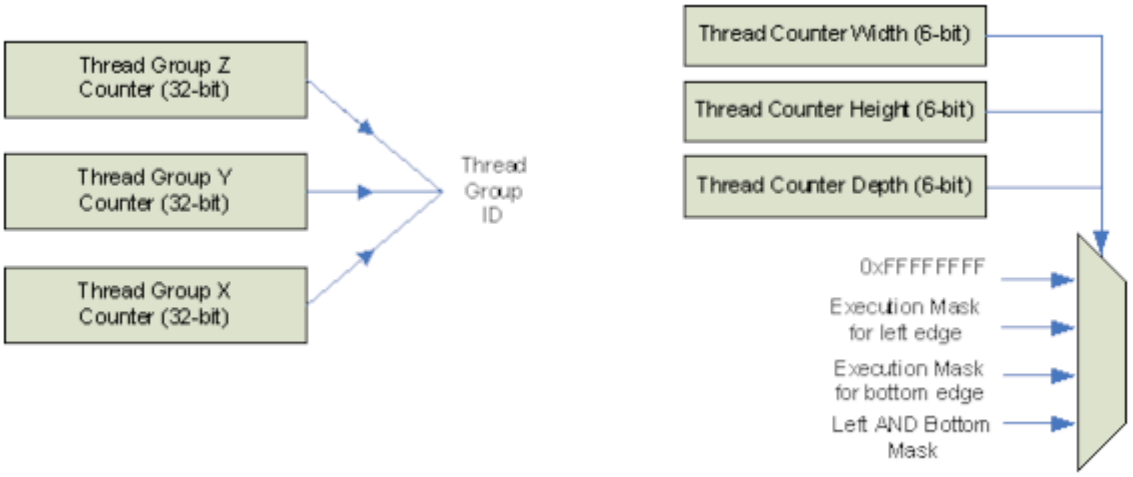
1.4.2.4 GPGPU Mode

The general purpose (GPGPU) mode allows the Gen7 architecture to be used by general purpose parallel APIs such as . This is similar to the Generic mode with additional support for automatic generation of threads, Shared Local Memory and Barriers.

1.4.2.4.1 Automatic Thread Generation

A single GPGPU job may require thousands or even millions of GPU_OBJECT commands. Rather than create them separately, it would be better to generate them algorithmically. To do this a GPGPU_WALKER command is created.

Rather than modifying the Media Walker, a simple Thread Group Walker is created instead:



The X/Y/Z counters for the thread group will have an initial and maximum value. The thread group id sent with each dispatch consists of these 3 numbers. These counters are 32-bits since the spec does not give a limit to the size of the thread id.

The 3 thread counters count the number of dispatches in a single thread group – up to 32 dispatches for SIMD32 or 64 dispatches for SIMD16/8. There are 3 of them in order to select the execution masks correctly – see section *Execution Masks* on execution masks. Each one is 6-bits in order to allow full flexibility of any dimension going to 64 while the rest do not increment.

A thread is generated each time one of the thread counters increment. When all the counters reach their maximum values, the thread group is done and the thread group counter can increment and start a new thread group. When the thread group X counter reaches its maximum it is reset to 0, and the Y counter is incremented.

The compiler determines how many SIMD channels are needed per thread group, and then decides how these will be split among EU threads. The number of threads is programmed in the thread counter, and the SIMD mode (SIMD8/SIMD16/SIMD32) is specified in the GPGPU_WALKER command.

1.4.2.4.2 Thread Payload

The payload to each thread dispatched is:

1. A thread group id which identifies the group the set of threads belong to. This is in the form of a set of 3, 32-bit X/Y/Z values.
2. The set of X/Y/Z that form the thread id for each channel. If Z is not used then only X/Y are needed.
3. The execution mask which indicates which channels are active.

Thread ids form a 2D or 3D surface which has to be mapped into SIMD32, SIMD16 or SIMD8 dispatches. Rather than have the hardware force a particular mapping of thread ids to channels, the mapping will be supplied by the compiler. The VFE will receive a simple count of the number of threads per thread group which will be used to count the number of dispatches. The thread ids for all threads in a thread group are put in a constant buffer with the MEDIA_CURBE_LOAD command. A single set of thread ids can be used repeatedly for all thread groups, since the thread ids are the same for each thread group id output by the GPGPU_WALKER.

The data required is up to the compiler, but here is an example set of payloads for a 2 Z x 2Y x 12 X and a SIMD16 dispatch. This thread group requires 3 dispatches:

3 2 1 0 11 10 9 8 7 6 5 4 3 2 1 0	Thead id X for dispatch 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0	Thead id Y for dispatch 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Thead id Z for dispatch 0
7 6 5 4 3 2 1 0 11 10 9 8 7 6 5 4	Thead id X for dispatch 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	Thead id Y for dispatch 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0	Thead id Z for dispatch 1
11 10 9 8 7 6 5 4 3 2 1 0 11 10 9 8	Thead id X for dispatch 2
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0	Thead id Y for dispatch 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	Thead id Z for dispatch 2

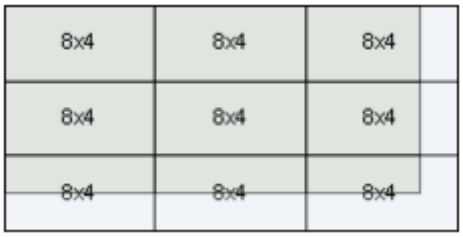
In this case the thread counter width would be programmed with a maximum value of 3 (since all the execution masks are all F, it doesn't matter how the thread counters are programmed as long as they count to 3 before finishing the thread group).

The first dispatch would tell the TS (who would tell the TD) that the payload starts at the beginning of the constant buffer and has a length of 3. The next dispatch would have a payload starting at constant_buffer_start + 3. The final dispatch payload starts at constant_buffer_start + 6. If there are more thread groups in the command they would get exactly the same payload – the only difference is the thread group id (as well as a different barrier and shared local memory space).

1.4.2.4.3 Execution Masks

The number of channels required by the GPGPU job may not evenly fit into the number of SIMD channels. That can leave some channels idle. The execution mask is used to tell the hardware which channels are to be used.

A thread group is modeled as a 3D solid with each channel acting as one X/Y/Z point in the solid. This can take the form of a line with 1024 channels with X from 0 to 1023 and constant Y/Z, a square with X=0 to 32 and Y=0 to 32, or a cube with X=0 to 9, Y=0 to 9, Z=0 to 9. Software needs to determine how these shapes are mapped onto the 32 SIMD32 channels per dispatch (or 16 SIM16, etc). The mapping per thread is assumed to be a 2D square of channels such as 8x4, 16x2, 32x1. Below is a diagram of a 22x6 thread group that is mapped onto a set of 8x4 SIMD32 channels:

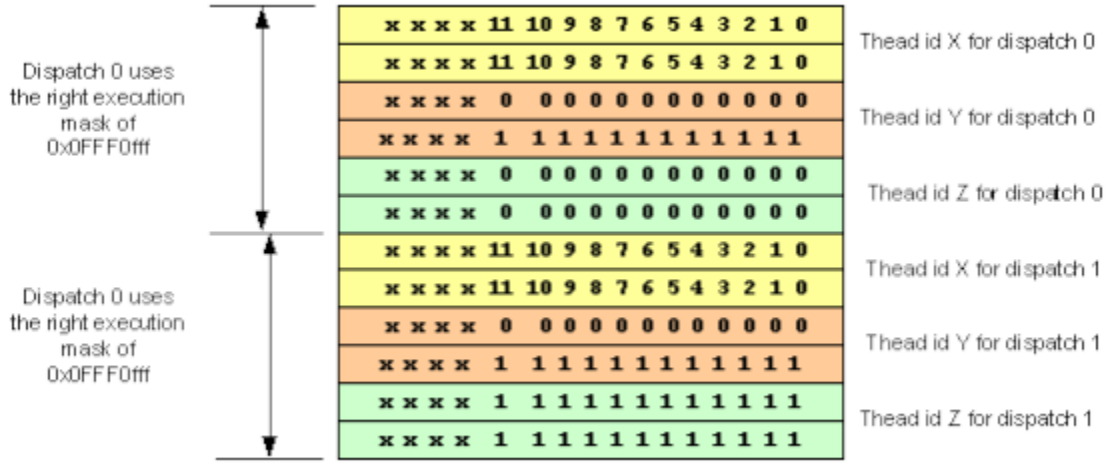


Note that the dispatches to the top and left have execution masks of all-F, while all the right edge dispatches have the same execution mask; likewise all the bottom edge dispatches have the same execution mask. The bottom right is the logical-AND of the right and bottom edge dispatches.

A 32-bit right and bottom mask is sent with the GPGPU_WALKER command, and the thread width, height and depth counters are used to determine when they are used (width, height and depth are used instead of X/Y/Z, since it is not required that width = X – width and height are the two variables that are changing in a single SIMD dispatch even if they are Y and Z).

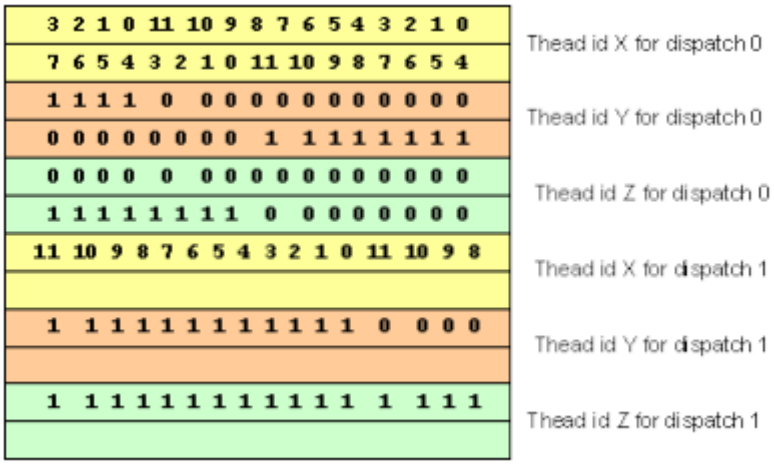
For each dispatch the width counter is incremented until it reaches the maximum – the dispatch with width=max will use the right execution mask. The height counter is then incremented and process repeated. If at any time the height counter = max then the execution mask is the bottom execution mask. When the height and width counters are both max then the dispatch will be the AND of the right and bottom and the depth counter will increment.

The same 2Z x 2Y x 12X thread group described above dispatched as SIMD32 with each dispatch delivering a 16X x 2Y shape would require 2 dispatches with empty bits in the right execution mask and all F in the bottom.



The width and height counter would have a maximum of 1, and the depth counter would have a maximum of 2. The two dispatches would use the AND of the two masks, but since the bottom mask is F it would be the same as just the right mask.

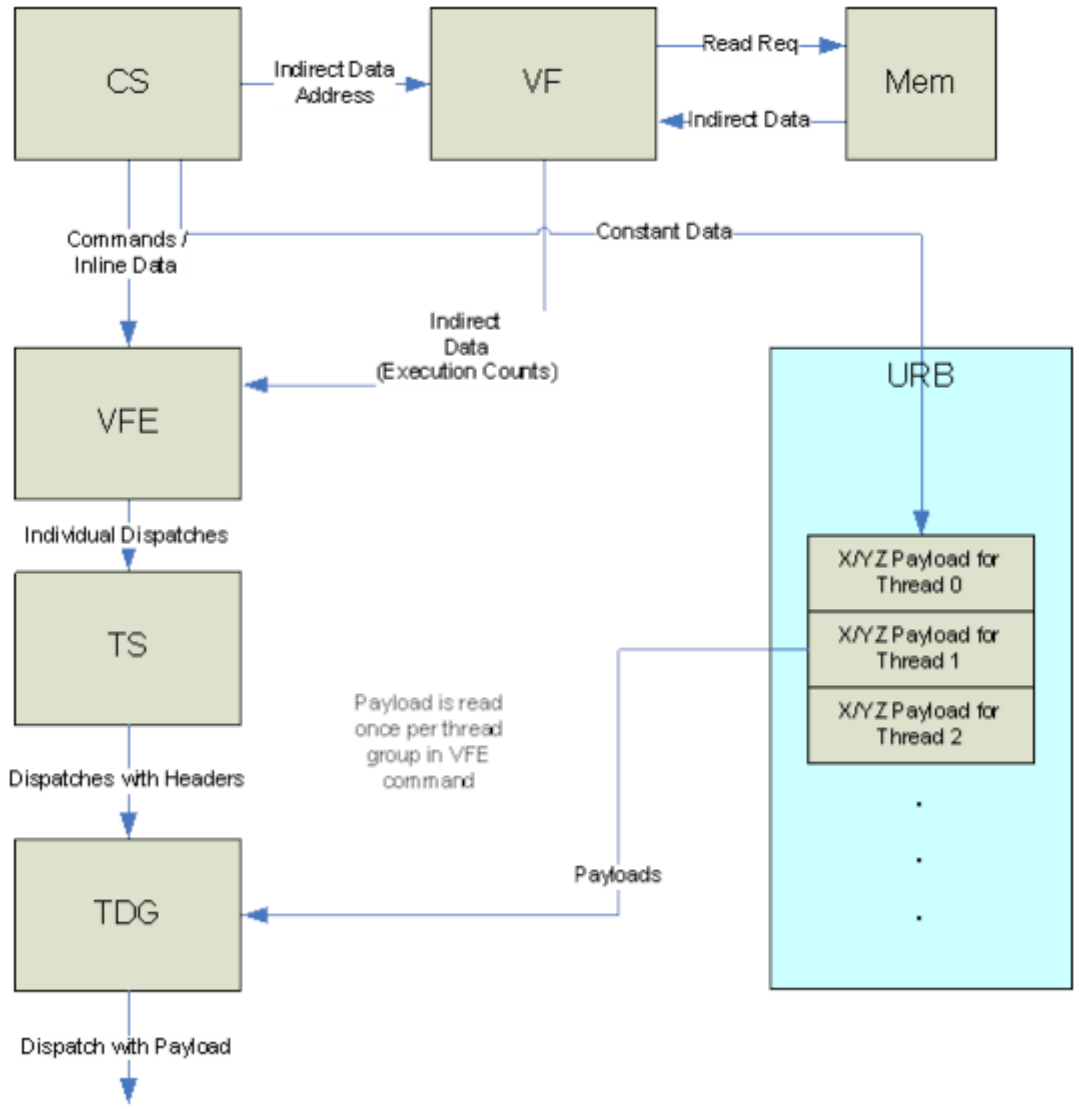
The execution masks can also be used when the software wants to pack the channels rather than lay them out in a regular pattern:



In this case the width counter can have a maximum of 2, and the height and depth counters with a maximum of 1. The first dispatch will use the bottom mask only (all-F) and the second will use the right AND bottom mask to remove the channels that are not used.

1.4.2.4.4 Payload Storage

The MEDIA_CURBE_LOAD constant data is stored in the URB by CS and read out by TDL when the dispatch occurs. The inline payload with the execution counts is sent to VFE from CS. The execution counts are stored internal to VFE.



Only 32 threads are allowed for SIMD32 to match the 1024 thread limit, requiring 32 execution count bytes, or 8 DW payload. SIMD16 and SIMD8 allow the full 64 thread per half-slice, and so require as much as 16 DWords.

The X/Y/Z payload size per dispatch is specified in the command, but a maximum size is 3 16-bit numbers per 1024 SIMD channels, or 6 kbytes.

1.4.2.4.5 URB Management

The VFE manages the URB in GPGPU and generic/media modes. The first 32 URB entries are reserved for the interface descriptor, and CURBE data is placed after the IDs. URB handles are needed for indirect data and parent/child communication; when the VFE starts up it creates up to 64 handles by partitioning the remaining URB space into evenly spaced addresses and saving the resulting handles in a FIFO. The handles can then be treated just like ones created by the URBM – send to TD on dispatch and recovered on the handle return bus.

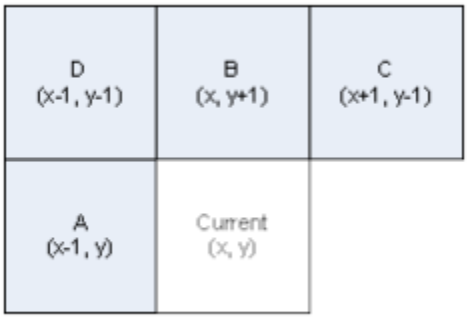
MEDIA_VFE_STATE specifies the amount of CURBE space, the URB handle size and the number of URB handles. The driver must ensure that ((URB_handle_size * URB_num_handle) – CURBE – 32) <= URB_allocation_in_L3.

1.4.2.4.6 AVC-Style Dependency Example

For AVD decoding, dependencies for a given macroblock may be set based on the availability of neighbor macroblocks, namely A, B, C, D and left-bottom neighbors (left-bottom only if MbAff = 1), as well as the current macroblock's address, MbAff flag and FieldMbFlag. For a macroblock in a progressive frame picture or a field picture, one macroblock may depend on up to four neighbors, A, B, C and D as shown in *AVC Style Dependency Example*. For a macroblock in a MbAff pair, it may depend on up to three, five or eight neighbors as shown in *AVC Style Dependency Example* and *AVC Style Dependency Example*, based on the current macroblock's address and FieldMbFlag.

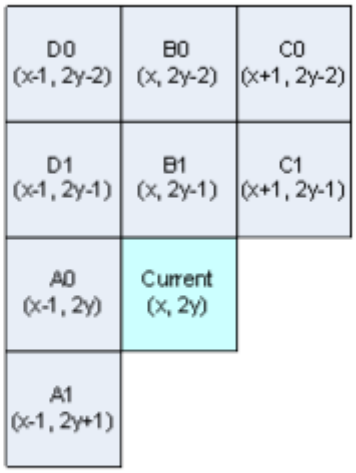
The neighbor's availability depends on the corresponding **IntraPredAvailFlagA|B|C|D|E** flags for the macroblock (or the macroblock pair). Hardware assumes that the flags are set correctly in the MEDIA_OBJECT_EX command as shown in Macroblock indices for field picture destination. For simplicity, the left neighbor pair (A0 and A1) availability for a MbAff macroblock can be determined as a group by **IntraPredAvailFlagA | IntraPredAvailFlagE**. For the second macroblock in a 'frame' MbAff pair, it depends on the first macroblock in the pair and it is always available.

Neighbor addresses of a macroblock in a progressive frame picture (MbAff = 0) or a field picture with up to 4 dependencies



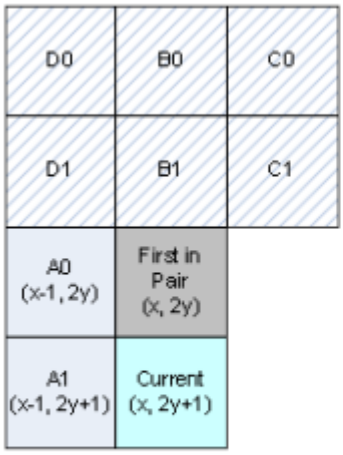
Neighbor addresses of the first macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies

(a) Neighbors for the first macroblock in a 'frame' MbAff pair

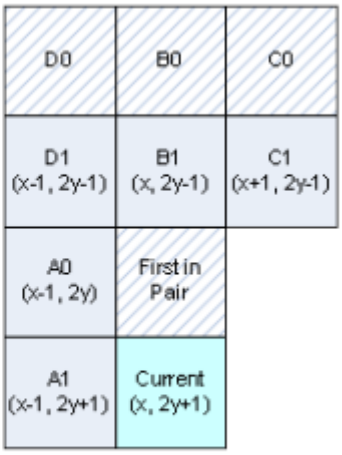


(b) Neighbors for the first macroblock in a 'field' MbAff pair

Neighbor addresses of the second macroblock in a MbAff frame picture (MbAff = 1) with up to 8 dependencies



(b) Neighbors for the second macroblock in a 'frame' MbAff pair



(b) Neighbors for the second macroblock in a 'field' MbAff pair

Neighbor Availability

MbAff	FieldMbFlag	VertOrigin[0]	A	B	C	D	LB	Comments
0	0/1	0/1	√	√	√	√		Progressive or Field picture
1	0	0	√	√	√	√	√	1st Frame MbAff macroblock

MbAff	FieldMbFlag	VertOrigin[0]	A	B	C	D	LB	Comments
1	0	1	√	na	0	na	√	2nd Frame MbAff macroblock
1	1	0	√	√	√	√	√	1st Field MbAff macroblock
1	1	1	√	√	√	√	√	2nd Field MbAff macroblock

1.4.2.5 VC1-Style Dependency Example

For VC1, only one dependency may be set depending on the availability of the upper neighbor macroblock.

Macroblock sequence order in a VC-1 picture with WidthInMblk = 5 and HeightInMblk = 6

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24
5	25	26	27	28	29

1.4.3 Parameterized Media Walker

The Parameterized Media Walker is a hardware thread generation mechanism that creates threads associated with units in a generalized 2-dimensional space, for example, blocks in a 2D image. With a small number of unit step vectors, the walker can implement a large number of walking patterns as described hereafter. This command may provide functions that are normally handled by the host software, thus, may be used to simplify the host software and GPU interface.

The walker described herein is doubly nested, where essentially a “local” walker can perform a variety of 2-dimensional walking patterns and a “global” walker can perform similar 2-dimensional walking patterns upon many local walkers. The local walker has 3 levels (outer, middle, and inner) while the global walker has 2 levels (outer and inner). Thus, the algorithm has 5-nested loops that modify local state based on user-defined unit step vectors.

The Walker’s programmability is derived from:

- The walker traverses a unit-normalized surface. Some example unit sizes:
    - 1x1: Walking pixels
    - 4x4: Walking sub-blocks
    - 16x16: Walking macro-blocks
    - 32x16: Walking macro-block-pairs
- The use of unit step vectors to describe the motion at each of level of nesting

- Starting locations for the local and global walkers
- Block sizes of the local and global walker
- And a small number of special mode controls for the inner-most loop which are aimed at efficiently dividing an image into two balanced workloads for dual-slice designs.

1.4.3.1 Walker Parameter Description

The global and local loops are both described by the same four parameters:

- Resolution,
- Starting location,
- Outer unit vector,
- Inner unit vector

The local inner loop has some special modes that will be described later. A table of the user inputs and some example values are given below:

GLOBAL LOOP PARAMETERS							
Global Resolution		Global Start		Outer Loop Unit Vector		Inner Loop Unit Vector	
X	Y	X	Y	X	Y	X	Y
120	68	0	0	32	0	0	32

LOCAL LOOP PARAMETERS							
Block Resolution		Local Start		Outer Loop Unit Vector		Inner Loop Unit Vector	
X	Y	X	Y	X	Y	X	Y
32	32	0	0	1	0	-2	2

LOCAL INNER LOOP SPECIAL MODE SELECTS							
Dual Mode	Repel	Attract			ExtraSteps	X	Y
TRUE	FALSE	FALSE			1	0	1

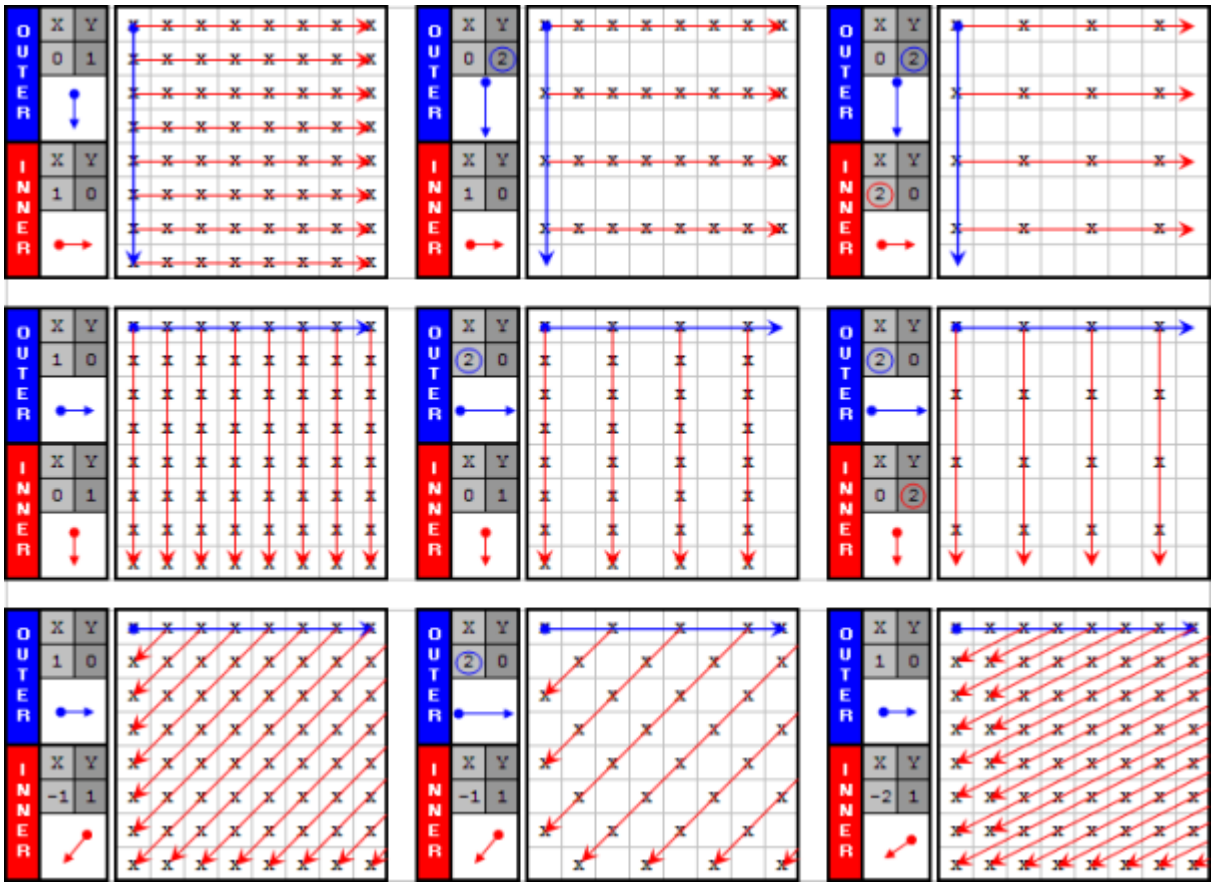
It should be emphasized that the value of what a “unit” represents is implicitly defined by the user. In other words, the walker traverses a “unit normalized space” that is not inherently bound to pixel walking. If the smallest unit of work the user wants to walk is a 4x3 block of pixels, you can program the inner loop to step (4,3) or (1,1):

- In the first case (4,3) the user is walking in units of pixels
- In the second case (1,1) the user is walking in units of 4x3 blocks of pixels.

It should be noted that hardware doesn’t contain enough bits for pixel walking for pixel resolution like 1920x1088. The intended usage of the walker is for block walking whereas the block size is not relevant to the walker parameters.

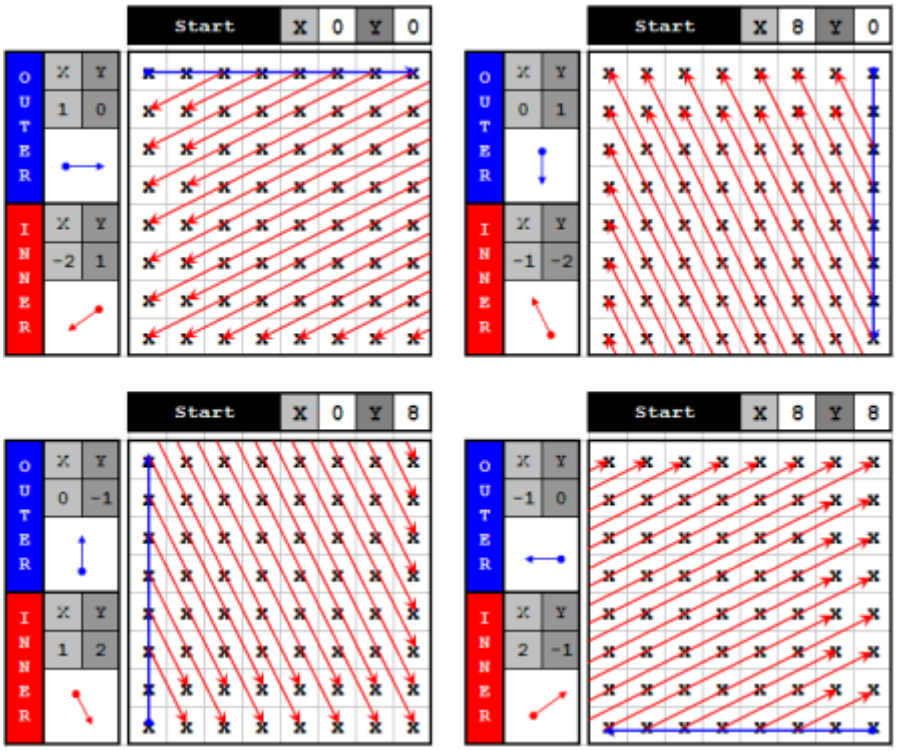
1.4.3.2 Basic Parameters for the Local Loop

The local inner and outer loop xy-pair parameters alone can describe a large variety of primitive walking patterns. Below are 9 primitive walking patterns generated by varying only the inner and outer unit step vectors of the local loop:



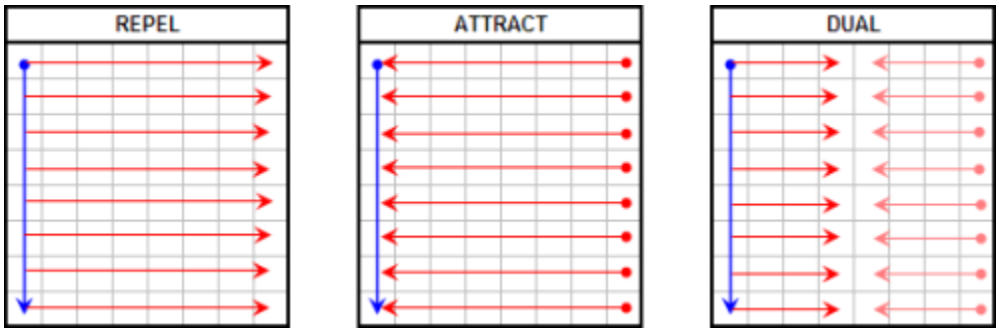
- The top row shows the outer unit vector pointing down (+Y) and the inner unit vector pointing right (+X). Rows and columns can easily be skipped by increasing the unit step vectors above one.
- The middle row the outer unit vector pointing right (+X) and the inner unit vector pointing down (+Y). Again, rows and columns are skipped by increasing the unit step vectors beyond one.
- The last row shows the capability to walk angles not perpendicular to the edge. The 1st shows a 45º walking pattern by setting the inner unit vector to (-1,1). The 2nd shows a checkerboard pattern by skipping every other outer loop and retaining the inner unit vector of (-1,1). The 3rd shows a 26.5º walking pattern by setting the inner unit vector to (-2,1).

The block resolution, shown as [8,8], and the starting location, currently [0,0], can be varied and the above patterns can be stretched and rotated many ways. The diagram below shows an example of where the start position and unit step vectors can be set to achieve a full rotation of the same pattern:

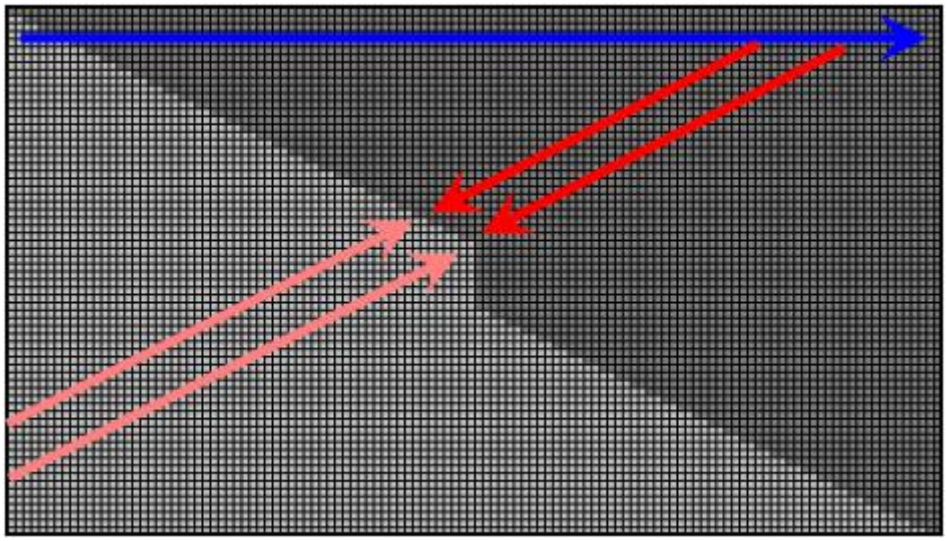


1.4.3.3 Dual Mode of Local Loop

The local Inner Loop Special mode selects are included to aid in the distribution of work, specifically with two slices in mind. Essentially, the local inner loop can be bisected and each half-walk can be directed inward towards the center of the image (dual). The local inner loop need not be bisected, and can either move away from the outer loop (repel) or move towards it (attract) when an even split is not desired:

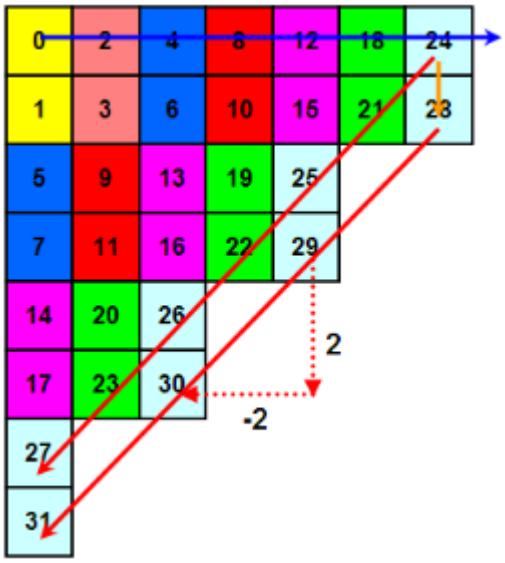


In Dual mode, the sequence will alternate between two half-walks such that every-other output would go to the same slice. This effect will produce a more balanced workload to two slices as shown in the example below where the color of the block represents which slice it was dispatched to. This is the walker's approach to fine-grained parallelism.



1.4.3.4 MbAff-Like Special Case in Local Loop

The local loop has an additional middle loop that is used to achieve some specific walking patterns, with MBAFF mode especially in mind. A pattern to handle MBAFF AVC content is to walk the top macroblocks of all macroblock pairs (MB-pairs) on a wavefront followed by the respective bottom macroblocks. The pattern is shown below.

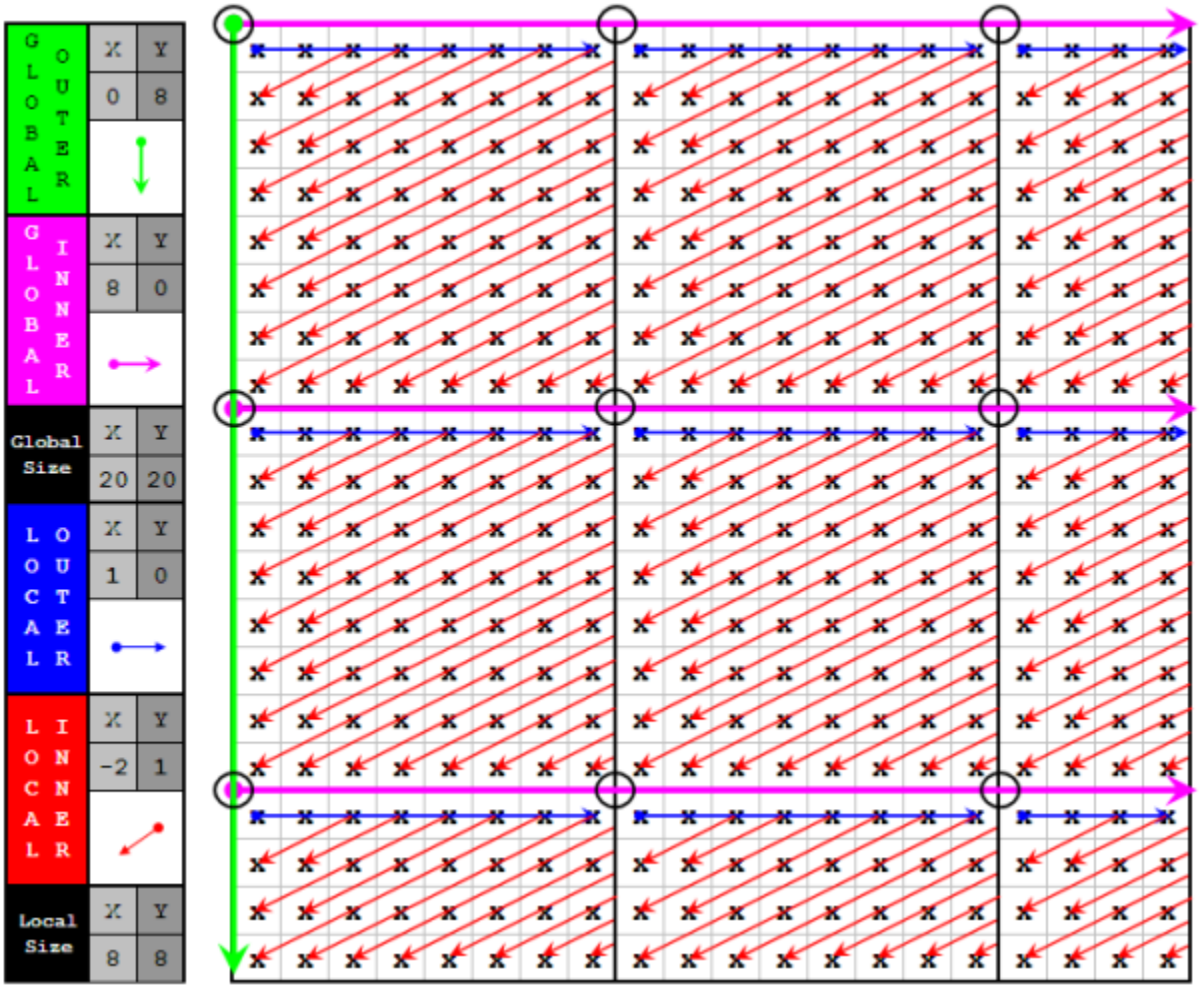


The outer loop unit step vector would be [1, 0] and the inner loop unit step vector would be [-2, 2]. A third loop is necessary to repeat the inner loop, only shifted down a unit before restarting. Thus, a middle loop with a unit step vector of [0,1] would achieve this MBAFF pattern. Additionally, the number of “extra steps” taken by the middle loop would be 1 in this case.

The addition of a middle loop also creates more overall flexibility, which seems necessary due to the integer-based unit step vector solution proposed (Manhattan distance issues etc.).

1.4.3.5 Global Loop

The same set of general parameters is used to describe the global loop as well. Thus, a global loop that is walking a raster-scan pattern can be combined with a local loop that is walking a 26.5º pattern (or vice-versa). As shown in the example below, if the local block size [8,8] is not an even multiple of the global resolution [20,20], the slack is still processed by dynamically changing the local block resolution.



The global loop will always resolve to be the upper-left corner of the local loop, shown above black circles. Note that local loop can still start in any corner of the local block, but the local (0,0) will always be the location where global loop begins the local loop, hence the upper-left corner.

The user can specify where the staring location of the global loop as with the local loop. If the user were to set the global starting location to (16,16) in the previous example, after inverting the global outer and global inner unit step vectors the same pattern would be achieved in the reverse order. Note that the slack would still be handled along the right and bottom edge of the global image in that case. The user could have also started at (12,12) in which case the slack would be handled on the left and top faces.

1.4.3.6 Walker Algorithm Description

The walker algorithm has been tested and optimized in software. A high-level pseudo-code description is given below:

```
Walker(){ //C-Style Pseudo-Code of Walker Algorithm
```

```
Load_Inputs_And_Initialize();
While (Global_Outer_Loop_In_Bounds()){
  Global_Inner_Loop_Intialization();
  While (Global_Inner_Loop_In_Bounds()){
    Local_Block_Boundary_Adjustment();
    Local_Outer_Loop_Initialization();
    While (Local_Outer_Loop_In_Bounds()){
      Local_Middle_Loop_Initialization();
      While (Local_Middle_Steps_Remaining()){
        Local_Inner_Loop_Initialization();
        While (Local_Inner_Loop_Is_Shrinking()){
          Execute();
          Calculate_Next_Local_Inner_X_Y();
        } //End Local Inner Loop
        Calculate_Next_Local_Middle_X_Y();
      } //End Local Middle Loop
      Calculate_Next_Local_Outer_X_Y();
      Calculate_Next_Local_Inverse_Outer_X_Y();
    } //End Local Outer Loop
    Calculate_Next_Global_Inner_X_Y();
  } //End Global Inner Loop
  Calculate_Next_Global_Outer_X_Y();
} //End Global Outer Loop
} //End Walker
```

The pseudo-code has the following characteristics:

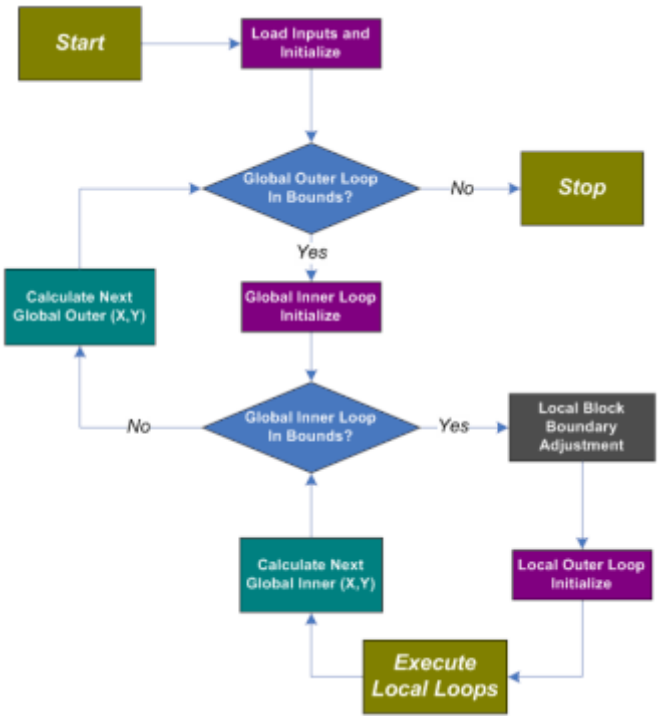
- There are 5 levels of iteration
- The highest 2 levels are called “global” and the lowest 3 levels are called “local”
    - The global loop is split into an outer and an inner loop.
    - The local loop is split into an outer, a middle, and an inner loop.
    - A bounding box for the global and local resolution is defined by the user.
    - The starting location within each bounding box is also specified by the user.
- Each of the 5 loops has its own persistent
    - Current position (x,y)
    - Unit step vector (x,y)
- The final output (x,y) is a summation of the global x,y and the local x,y.

- The next (x,y) for given level can be calculated while the next lower level is still executing. Additionally, the result can be used to check to see if the current level will execute again once control is returned.

The flow of the global outer and inner loops is:

1. Check a bound condition
2. Initialize the next level loop
3. Execute the next level loop
4. When the next level loop fails its condition, calculate the next position for the current loop level and repeat.

Walker algorithm flowchart for the Global Loop

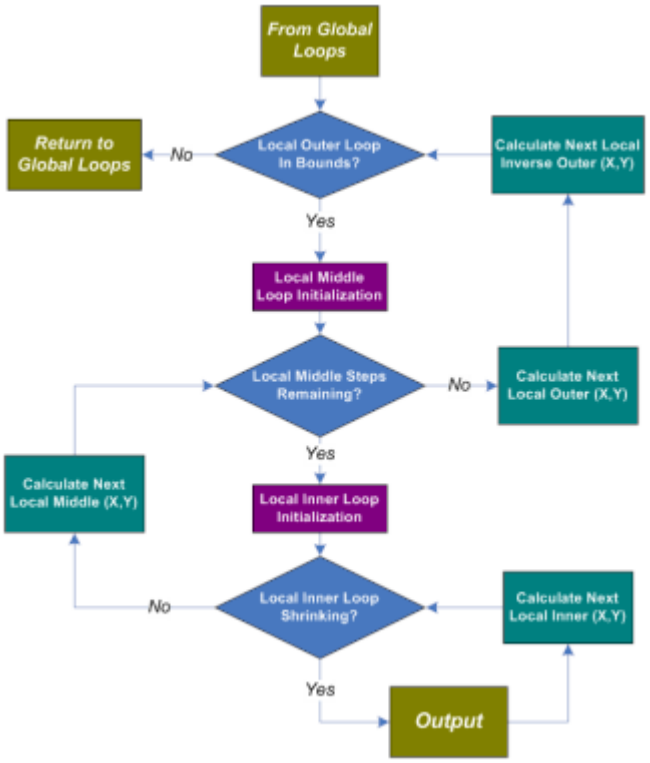


Take note of the grey box “Local Block Boundary Adjustment”. This logic is necessary to adjust the local block size when the distance between the current global position to the edge of the image is less than the local resolution. Additionally, the local starting positions might be modified here as well if the defined starting position is larger than the new local block size.

The flow of the 3 local loops does not vary much from the 2 global loops. The differences are:

- In addition to a boundary check, the local middle loop also ensures the number of middle steps is less than or equal to the user defined “number of extra steps”.
- The local inner loop only checks to see if the prior distance between the x,y starting and ending points are greater than their current distance. If this is true, it implies that the two inner loops are converging towards each other.
- When the middle loop check fails, both the starting points (local outer) and ending points (local inner) are updated.

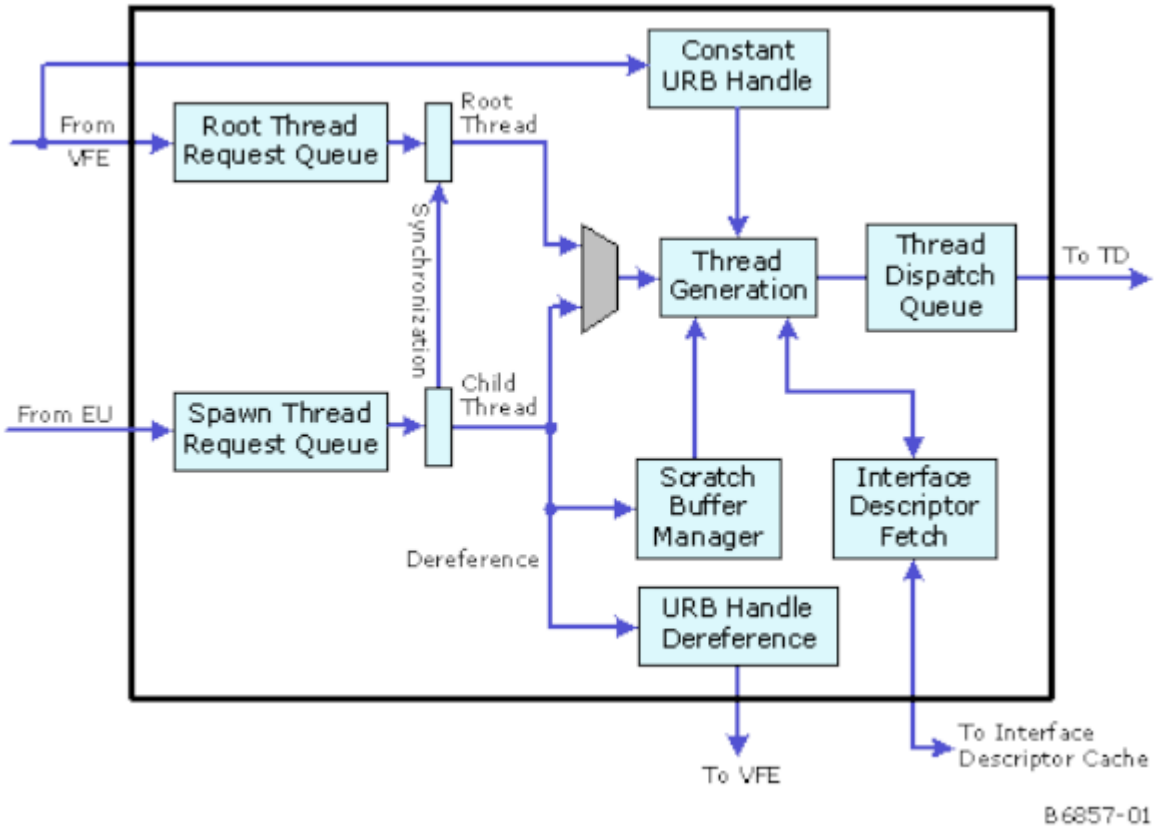
Walker algorithm flowchart for the Local Loop



1.5 Thread Spawner Unit

The Thread Spawner (TS) unit is responsible for making thread requests (root and child) to the Thread Dispatcher, managing scratch memory, maintaining outstanding root thread counts, and monitoring the termination of threads.

Thread Spawner block diagram



1.5.1 Basic Functions

1.5.1.1 Root Threads Lifecycle

Thread requests sourced from VFE are called **root threads**, since these threads may be creating subsequent (child) threads. A root thread may a macroblock thread created by VFE as in VLD mode, or may be a general-purpose thread assembled by VFE according to full description provided by host software in Generic mode.

Thread requests are stored in the Root Thread Queue. TS keeps everything needed to get the root threads ready for dispatch and then tracks dispatched threads until their retirement.

TS arbitrates between root thread and child thread. The root thread request queue is in the arbitration only if the number of outstanding threads does not exceed the maximum root thread state variable. Otherwise, the root thread request queue is stalled until some other root threads retire/terminate.

Once a root thread is selected to be dispatched, its lifecycle can be described by the following steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.

    - Once TS receives the interface descriptor, it checks whether maximum concurrent root thread number has reached to determine whether to make a thread dispatch request or to

stall the request until some other root threads retire. If the thread requests the use of scratch memory, it also generates a pointer into the scratch space.

2. TS then builds the transparent header and the R0 header.
3. Finally, TS makes a thread request to the Thread Dispatcher.
4. TS keeps track of dispatched thread, and monitors messages from the thread (resource dereference and/or thread termination). When it receives a root thread termination message, it can recover the scratch space and thread slot allocated to it. The URB handle may also be dereferenced for a terminated root thread for future reuse. It should be noted that URB handle dereference may occur before a root thread terminates. See detailed description in the Media Message section.

    - It is the root thread's responsibility (software) to guarantee that all its children have retired before the root thread can retire.

1.5.1.2 URB Handles

VFE is in charge of allocating URB handles for root threads. One URB handle is assigned to each root thread. The handle is used for the payload into the root thread.

If Children Present is not set (root-without-child case), TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched.

If Children Present is set (root-with-child case), the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread.

Children Present is a command variable in the _OBJECT command.

1.5.1.3 Root to Child Responsibilities

Any thread created by another thread running in an EU is called a **child thread**. Child threads can create additional threads, all under the tree of a root which was requested via the VFE path.

A root thread is responsible of managing pre-allocated resources such as URB space and scratch space for its direct and indirect child threads. For example, a root thread may split its URB space into sections. It can use one section for delivering payload to one child thread as well as forwarding the section to the child thread to be used as return URB space. The child thread may further subdivide the URB section into subsections and use these subsections for its own child threads. Such process may be iterated. Similarly, a root thread may split its scratch memory space into sections and give one scratch section for one child thread.

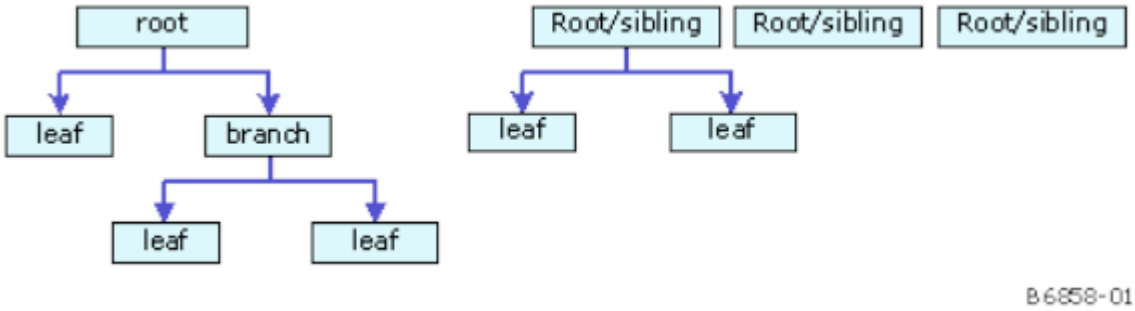
TS unit only enforces limitation on number of outstanding root threads. It is the root threads' responsibility to limit the number of child threads in their respected trees to balance performance and avoid deadlock.

1.5.1.4 Multiple Simultaneous Roots

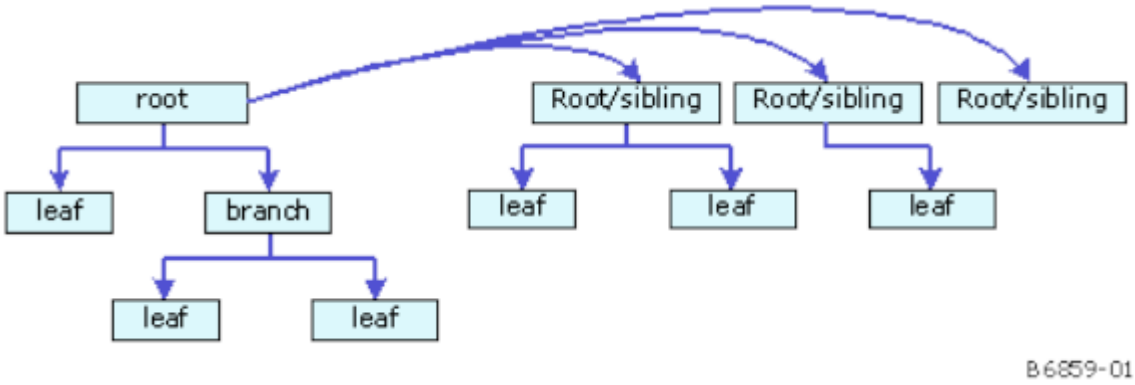
Multiple root threads are allowed concurrently running in execution units. As there is only one scratch space state variable shared for all root threads, all concurrent root thread requiring scratch space share the same scratch memory size. *Multiple Simultaneous Roots* depicts two examples of thread-thread relationship. The left graph shows one single tree structure. This tree starts with a single root thread that generates many child threads. Some child threads may create subsequent child threads. The right graph shows a case with multiple disconnected trees. It has multiple root threads, showing sibling roots of disconnected trees. Some roots may have child threads (branches and leafs) and some may not.

There is another case (as shown in *Multiple Simultaneous Roots*) where multiple trees may be connected. If a root is a synchronized root thread, it may be dependent on a preceding sibling root thread or on a child thread.

Examples of thread relationship



A example of thread relationship with root sibling dependency



1.5.1.5 Synchronized Root Threads

A synchronized root thread (SRT) originates from a MEDIA_OBJECT command with Thread Synchronization field set. Synchronized root threads share the same root thread request queue with the non-synchronized roots. A SRT is not automatically dispatched. Instead, it stays in the root thread request queue until a spawn-root message is at the head of the child thread request queue. Conversely, a spawn-root message in the child thread request queue will block the child thread request queue until the head of root thread request queue is a SRT. When they are both at the head of queues, they are taken out from the queue at the same time.

A spawn-root message may be issued by a root thread or a child thread. There is no restriction. However, the number of spawn-root messages and the number of SRT must be identical between state changes. Otherwise, there can be a deadlock. Furthermore, as both requests are blocking, synchronized root threads must be used carefully to avoid deadlock.

When Scoreboard Control is enabled, the dispatch of a SRT originated from a MEDIA_OBJECT_EX command is still managed by the same way in addition to the hardware scoreboard control.

1.5.1.6 Deadlock Prevention

Root threads must control deadlock within their own child set. Each root is given a set of preallocated URB space; to prevent deadlock it must make sure that all the URB space is not allocated to intermediate children who must create more children before they can exit.

There are limits to the number of concurrent threads. The upper bound is determined by the number of execution units and the number of threads per EU. The actual upper bound on number of concurrent threads may be smaller if the GRF requirement is large. Deadlock may occur if a root or intermediate parent cannot exit until it has started its children but there is no space (for example, available thread slot in execution units) for its children to start.

To prevent deadlock, the maximum number of root threads is provided in VFE state. The Thread Spawner keeps track of how many roots have been spawned and prevents new roots if the maximum has been reached. When child threads are present, it is software's responsibility to constrain child thread generation, particularly the generation of child threads that may also spawn more child threads.

Child thread dispatch queue in TS is another resource that needs to be considered in preventing deadlock. The child thread dispatch queue in TS is used for (1) message to spawn a child thread, (2) message to spawn a synchronized root thread, and (3) thread termination message. If this queue is full, it will prevent any thread to terminate, causing deadlock.

For example, if an application only has one root thread (max # of root threads is programmed to be one). This root thread spawns child threads. In order to avoid deadlock, the maximum number of outstanding child thread that this root thread can spawn is the sum of the maximum available thread slots plus the depth of the child thread dispatch queue minus one.

Max_Outstanding_Child_Threads = (Thread Slot Number – 1) + (TS Child Queue Depth – 1)

Adding other root threads (synchronized and/or non-synchronized) to the above example, the situation is more complicated. A conservative measure may have to use to prevent deadlock. For example, the root thread spawning child threads may have to exclude the max number of root threads as in the following equation to compute the maximum number of outstanding child threads to be dispatched.

Max_Outstanding_Child_Threads = (Thread Slot Number – 1) + (TS Child Queue Depth – 1) – (Max Root Threads-1)

1.5.1.7 Child Thread Lifecycle

When a (parent) thread creates a child thread, the parent thread behaves like a fixed function. It provides all necessary information to start the child thread, by assembling the payload in URB (including R0 header) and then sending a spawn thread message to TS with following data:

- An interface descriptor pointer for the child thread.
- A pointer for URB data

The interface descriptor for a child may be different from the parent – how the parent determines the child interface descriptor is up to the parent, but it must be one from the interface descriptor array on the same interface descriptor base address.

The URB pointer is not the same as a URB handle. It does not have an URB handle number and does not appear in any handle table. This is acceptable because the URB space is never reclaimed by TS after a child is dispatched, but rather when the parent releases its original handles and/or retires.

The child request is stored in the child thread queue. The depth of the queue is limited to 8, overrun is prevented by the message bus arbiter which controls the message bus. The arbiter knows the depth of the queue and will only allow 8 requests to be outstanding until the TS signals an entry has been removed.

As mentioned previously, child threads have higher priority over root threads. Once TS selects a child thread to dispatch, it follows these steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.
2. TS then builds the transparent header but not the R0 header.
3. Finally, TS makes a thread request to the Thread Dispatcher.
4. Once the dispatch is done, TS can forget the child – unlike roots, no bookkeeping is done that has to be updated when the child retires.

If more data needs to be transferred between a parent thread and its child thread than that can fit in a single URB payload, extra data must be communicated via shared memory through data port.

1.5.1.8 Arbitration between Root and Child Threads

When both root thread queue and child thread queue are both non-empty, TS serves the child thread queue. In other words, child threads have higher priority over root threads. The only condition that the child thread queue is stalled by the root thread queue is that the head of child thread queue is a root-synchronization message and the head of root thread queue is not a synchronized root thread.

1.5.1.9 Persistent Root Thread

A persistent root thread in general stays in the system for a long period of time. It is normally a parent thread, and only one PRT is allowed in the system at a time. The PRT can be restarted later, *even if it had completed normally the last time it was executed*. Therefore, the PRT must always save enough context (via data port messages to a predefined surface) to allow it to restart from where it left off (including determining that it has nothing left to do). However, since only one PRT can execute at a time, once the next PRT starts, the previous one will never be restarted, thus the context save surface can be reused from one PRT to the next.

A PRT may check the Thread Restart Enable bit in the R0 header to find out whether it is a fresh start or resumed from a previous interrupt and then can continue operations from that previously saved context.

PRT can be interleaved with other root (such as parent root thread, or synchronized root thread) and child threads. A parent root thread is not necessarily a PRT, and doesn't have to be as long as it can be finished in deterministic time that is shorter than required for fine-grain context switch interrupt.

Use of PRT must follow the following rules:

- There can only be one PRT in the media pipeline at a given time. That means, there shall not be any other media primitive commands (MEDIA_OBJECT or MEDIA_OBJECT_EX) between it and the previous MI_FLUSH command. In other words, when multiple such PRTs are used in a sequence of media primitive commands, MI_FLUSH must be inserted.

1.5.1.10 GPGPU Functions

In GPGPU mode, the Thread Spawner allocate and track the barriers and shared local memory per thread group.

1.5.1.10.1 Thread Group Tracking

The TSG needs to keep track of the threads outstanding in a group to know when the thread group barrier and Shared Local Memory can be reclaimed. This can be done by keeping a counter per active thread group (up to 16 per half-slice) which increments when a new thread is sent out and decremented

when the thread retires. The assigned barrier ID (with half-slice bit) is unique per thread group and much smaller than the thread group id and so will be used to keep track of the thread group instead.

Since TSL sends the thread retirement via the Message Channel rather thant the thread retirement bus, the barrier ID used to identify the thread group can be sent at the same time. A CAM will then match the ID with the counter to decrement.

There is a potential corner case of a thread group without barriers being partly dispatched, then retiring before the rest of the thread group is sent. This should be OK, since the lack of barriers means that there is no dependencies between threads.

1.5.1.10.2 Shared Local Memory Allocation

The Shared Local Memory is a 64k block per half-slice in the L3 that must be shared between all thread groups on that half-slice. A new memory manager simular to the Scratch Space memory manager is used to allocate this space.

We are only dispatching threads from a single Interface Descriptor at a time. If a new Interface Descriptor is requested the pipe is drained and all shared memory recovered before starting to allocate new shared memory. This means that only a single size of shared memory needs to be supported at once.

For simplicity, only power-of-2 sizes from 4k to 64k are allowed. The thread request will specify how much is needed. The first thread of a Thread Group is marked as requiring a new shared local memory – if not the old Shared Local Memory offset is sent with the dispatch.

A simple set of 16-bits is used to allocate 4k shared memory, with fewer bits used for larger sizes. A priority encoder finds the first unused bit and the offset remembered as being associated with a particular barrier id. The barrier id is then used to track the thread group.

When the Thread Group Tracking indicates that a thread group is completely retired, that section of shared local memory can be reclaimed.

1.5.1.10.3 Software Managed Shared Local Memory

Software can optionally manage shared local memory. In this case, each thread command or thread group command will have the shared memory offset included – each command in a thread group must have the same offset, of couse. If the offset requested is still being used then the command is stalled until the thread group using that offset is done.

Hardware will track the usage of this section of shared memory as before, recording the offset as being used and recording it as being available after the thread group is done.

1.5.1.10.4 Automatic Barrier Management

Since we have an automatic shared memory allocation it makes sense to make barrier management automatic too.

Instead of the barrier id in the Interface Descriptor, there is now a thread count per thread group. If a new thread group id comes in without a barrier allocated (checked with a CAM match across> 16 barriers), the TSG picks an unused barrier and sends this count in a message to GWunit. It then needs to wait for an accept message back from GW before sending the dispatch to ensure that a barrier message doesn't arrive at the GW before the barrier is programmed. The barrier id picked is sent with every dispatch from this thread group.

When the thread group tracker determines that a thread group has finished, the barrier becomes available to new thread groups.

1.5.1.10.5 Local Memory / Scratch Space

The Local Memory (not to be confused with Shared Local Memory, which is shared by all thread in a thread group) is allocated per thread dispatched to the EU.

The existing Scratch Space manager is used to provide between 1k and 12k bytes memory per thread. A small change to the kernel can be used to provide more scratch space – the pointer that TSG provides is simply:

Scratch Space Pointer = Scratch Space Base Pointer + FFTID * Per Thread Scratch Space

To increase the amount of scratch space per thread, each kernel needs to do this operation on its **Scratch Space Pointer:**

New Scratch Space Pointer = Old Scratch Space Pointer + FFTID * (New Per Thread Scratch Space – Old Per Thread Scratch Space)

The old Scratch Space Pointer and FFTID are available in the R0 header. The driver needs to allocate enough memory for the total number of threads in the system * the new per thread scratch space.

1.5.1.10.6 Dispatch Payload

The payload for a general purpose thread will have to include the execution mask with a bit per 32-channel. SIMD16 and SIMD8 use the LSB bits of the execution mask. The 5-bit number transferred from VFE will be expanded to produce the 32-bit mask. This will use the Dmask currently used by the pixel shader dispatch in the transparent header.

1.5.2 Interfaces

1.5.2.1 Interface to VFE

TS receives an interface descriptor pointer and a URB handle from VFE. It uses the interface descriptor pointer to fetch the interface descriptor. TS uses the information in the interface descriptor along with the URB handle to fill out the transparent header in the message to TD for all threads. For root thread, TS also generate the R0 header.

TS transmits URB handle dereference signal to VFE. As described previously, the derefernce signal may be at dispatch time or at later time depending on Children Present. No matter which case, there is one and only one URB handle dereference for a thread.

1.5.2.2 Interface to Thread Dispatcher

TS creates the transparent header, assembles the URB handles and calls TD to dispatch a new thread. For an unsynchronized root thread, there is one URB handle managed by VFE and optionally one Constant URB handle managed by CS. For a synchronized root thread, there is one URB handle managed by VFE, a URB handle created by the synchronizing thread (the one that sends the ‘spawn root thread’ message, and optionally one Constant URB handle managed by CS. For a child thread, there is one URB handle managed by the parent thread plus an optional Constant URB handle.

1.6 Media State Model

The media state model is based on in-line state load mechanism. VFE state, URB configuration and Interface Descriptors are loaded to VFE hardware through state commands.

All Interface Descriptors have the same size and are organized as a contiguous array in memory. They can be selected by Interface Descriptor Index for a given kernel. This allows different kinds of kernels to coexist in the system.

Pipeline	Opcode	Sub Opcode	Command
(Media) Bits [28:27]	Bits [26:24]	Bits [23:16]	
2h	0h	00h	MEDIA_VFE_STATE
2h	0h	01h	MEDIA_CURBE_LOAD
2h	0h	02h	MEDIA_INTERFACE_DESCRIPTOR_LOAD

1.7 Media Messages

All message formats are given in terms of dwords (32 bits) using the following conventions.

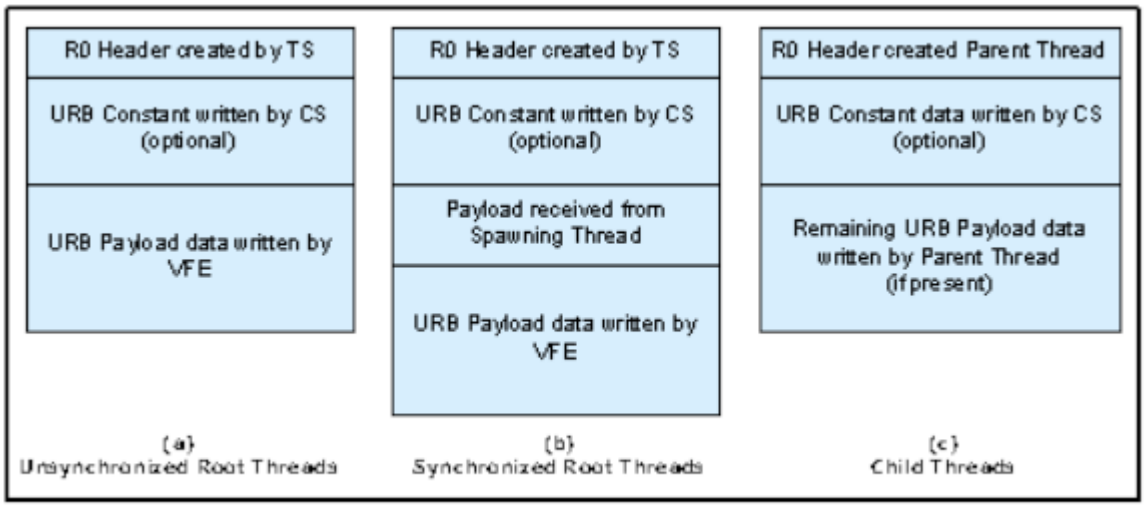
Dispatch Messages: **R**p.d

SEND Instruction Messages: **M**p.d

1.7.1 Thread Payload Messages

The root thread's register contents differ from that of child threads, as shown in *Thread Payload Messages*. The register contents for a synchronized root thread (also referred to as 'spawned root thread') and an unsynchronized one are also different. Whether the URB Constant data field is present or not is determined by the interface descriptor of a given thread. This applies to both root and child threads. When URB Constant data field is present for a synchronized root thread, URB constant data field is before the data field received from the spawning thread, which is also before the URB payload data.

Thread payload message formats for root and child threads



1.7.1.1 Generic Mode Root Thread

The following table shows the R0 register contents for a Generic mode root thread, which is generated by TS. The remaining payloads are application dependent.

R0 Header of a Generic Mode Root Thread

DWord	Bit	Description
R0.7	31:0	
R0.6	31:0	**Reserved**
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]
	9:8	Reserved: MBZ.
	9:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion. Format = U8. Bits 9:8 are Reserved, MBZ.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved : MBZ
R0.3	31:5	**Sampler State Pointer.** Specifies the 32-byte aligned pointer to the sampler state table. Format = GeneralStateOffset[31:5]
	4	Reserved : MBZ
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space, in 16-byte quantities, allowed to be used by this thread. The value specifies the power that two will be raised to, to determine the amount of scratch space. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2	31:28	Reserved : MBZ
	27:24	**BarrierID**. This field indicates which one from the 16 Barriers this kernel is associated. Format: U4
	23:16	**Barrier.Offset.** This is the offset for the Barrier to indicate the offset from the requester's RegBase (which may be 0 if Bypass Gateway Control is set to 1) for the broadcast barrier message. Barrier.Offset + RegBase must be in the valid GRF range. Otherwise, hardware behavior is undefined. It is in unit of 256-bit GRF register.

DWord	Bit	Description
		The most significant bit of this field must be zero. Format = U8 Range = [0,127]
	15:9	Reserved : MBZ
8:4	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors. Format = U5	
	3:0	**Scoreboard Color** (only with MEDIA_OBJECT_EX)**:** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
R0.2	31:10	Reserved: MBZ
	9:4	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors. Format = U5
	3:0	**Scoreboard Color** (only with MEDIA_OBJECT_EX)**:** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
R0.1	31:28	**Scoreboard Mask 4-7** (only with MEDIA_OBJECT_EX)**:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the VFE_STATE_EX. **Bit n (for n = 4…7):** Scoreboard n is dependent, where bit 28 maps to n = 4. Format = TRUE/FALSE Reserved : MBZ
	27:26	Reserved : MBZ
	25	**Reserved.** MBZ
	24:16	**Scoreboard Y** This field provides the Y term of the scoreboard value of the current thread. Format = U9 only with MEDIA_OBJECT_EX.

DWord	Bit	Description
		Reserved : MBZ
	15:12	Reserved : MBZ
	11:9	**Reserved.** MBZ
	8:0	**Scoreboard X** This field provides the X term of the scoreboard value of the current thread. Format = U9 only with MEDIA_OBJECT_EX. Reserved : MBZ
R0.0	31:24	**Scoreboard Mask:** Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the MEDIA_VFE_STATE. **Bit n (for n = 0…7):** Scoreboard n is dependent, where bit 24 maps to n = 0. Format = TRUE/FALSE
	23:16	Reserved : MBZ
	15:0	**URB Handle.** This is the URB handle where indicating the URB space for use by the root thread and its children.

1.7.1.2 Root Thread from MEDIA_OBJECT_PRT

The root thread payload message for an MEDIA_OBJECT_PRT command has a fixed format independent of the VFE mode (e.g. Generic mode or AVC-IT mode). One example GRF register location is given for the condition that CURBE is disabled.

Root thread payload layout for a MEDIA_OBJECT_PRT command

GRF Register	Example	Description
R0	R0	**R0 header**
R1 – R(m)	n/a	**Constants from CURBE when CURBE is enabled** m is a non-negative value
R(m+1)	R1	**In-line Data block**.

The R0 header field is as the following, which is the same as in other modes except the Thread Restart Enable bit (bit 0 of R0.2).

R0 header of the thread payload of a MEDIA_OBJECT_PRT command

DWord	Bit	Description
R0.7	31	
	27:24	
	23:0	

DWord	Bit	Description
R0.6	31:24	
	23:0	
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1k-byte aligned pointer to the scratch space. This field is only valid when Scratch Space is enabled. Format = GeneralStateOffset[31:10]
	9:8	Reserved : MBZ
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by the thread upon thread completion.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved : MBZ
R0.3	31:5	**Sampler State Pointer.** Specifies the 32-byte aligned pointer to the sampler state table. Format = GeneralStateOffset[31:5]
	4	Reserved : MBZ
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space, in 16-byte quantities, allowed to be used by this thread. The value specifies the power that two will be raised to, to determine the amount of scratch space. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2	31:4	**Interface Descriptor Pointer.** Specifies the 16-byte aligned pointer to *this thread's* interface descriptor. Can be used as a base from which to offset child thread's interface descriptor pointers from. Format = GeneralStateOffset[31:4]
	3:1	Reserved : MBZ
	0	**Thread Restart Enable.** If set, indicates that the persistent root thread (PRT) is being restarted, and context should be restored from the context save area before executing. Format = Enable
R0.1	31:0	Reserved : MBZ
R0.0	31:16	Reserved : MBZ
	15:0	**URB Handle.** This is the URB handle where indicating the URB space for use by the root thread and its children.

The inline data block field is the same as in the MEDIA_OBJECT_EX command with zero-filled partial GRF.

1.7.1.3 Root Thread from MEDIA_OBJECT_WALKER

The root thread payload message for an MEDIA_OBJECT_WALKER command, which must be in Generic mode, has the same format as that of the generic mode root thread format.

Root thread payload layout for a MEDIA_OBJECT_WALKER command

GRF Register	Example	Description
R0	R0	**R0 header**
R1 – R(m)	n/a	**Constants from CURBE when CURBE is enabled** m is a non-negative value
R(m+1)	R1	**In-line Data block**.

The R0 header field is identical to that of Generic Mode Root Thread.

The inline data block field is the same as in the MEDIA_OBJECT command with zero-filled partial GRF.

There is no indirect data block field.

1.7.1.4 Child Thread

The tread initiation for the child thread is determined by the data stored in the URB by the parent that spawns it. No hardware-defined header is generated. However, software should follow the header field definition similar to that for a root thread, when the same fields are used, to be consistent and to reduce message header assemble overhead.

The Parent Thread Count field should be the Thread Count field of the parent thread itself (e.g. copying R0.6[23:0] to R0.7[23:0]. The Thread Count field should have a unique value for each child thread and the unique value should not be dependent on the execution order. This is mostly important for the cases when the child thread generation order may vary depending on the thread completion order. For example, when generating child threads for macroblock-based processing, the Thread Count field for a child thread should be deterministic for a macroblock position.

The following table shows the R0 register contents for a child thread, which is generated by its parent thread. The remaining payloads are application dependent.

DWord	Bit	Description
R0.7	31	
	27:24	
	23:0	
R0.6	31:24	
	23:0	
R0.5-R0.0	31:0	**Software defined**

1.7.1.5 GPGPU Thread

The RO header of the Thread Dispatch Payload for the GPGPU thread:

DWord	Bit	Description
R0.7	31:0	**Thread Group ID Z:** This field identifies the Z component of the thread group. That this thread belongs to.
R0.6	31:0	**Thread Group ID Y**: This field identifies the Y component of the thread group that this thread belongs to.
R0.5	31:10	**Scratch Space Pointer.** Specifies the 1k-byte aligned pointer to the scratch space (used for the

DWord	Bit	Description
		GPGPU local memory space). Format = GeneralStateOffset[31:10]
	9:8	Reserved: MBZ.
	9:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent threads (of any thread group). It is used to free up resources used by the thread upon thread completion. Format = U8. Bits 9:8 are Reserved, MBZ.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4	Reserved
	3:0	Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.3	31:5	**Sampler State Pointer.** Specifies the 32-byte aligned pointer to the sampler state table. Format = GeneralStateOffset[31:5]
	4	Reserved : MBZ
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space, in 16-byte quantities, allowed to be used by this thread. The value specifies the power that two will be raised to, to determine the amount of scratch space. Format = U4 Range = [0,11] indicating [1k bytes, 12k bytes]
R0.2	31	**Reserved: MBZ**
	30	Reserved: MBZ
	29	**Barrier Enable:** This field indicates that a barrier has been allocated for this kernel Reserved: MBZ
	28	**SLM Enable:** This field indicates that Shared Local Memory has been allocated for this kernel Reserved: MBZ
	27:24	**BarrierID:** This field indicates the barrier that this kernel is associated with. Format: U4
	23:15	
	23:16	This key is a free running count of the number of dispatches.
	14:10	Reserved: MBZ
	9:4	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors.

DWord	Bit	Description
		Format = U5
	3:0	**Reserved.** MBZ
R0.1	31:0	**Thread Group ID X**: This field identifies the X component of the thread group that this thread belongs to.
	30:28	**Reserved:** MBZ
	27:24	**Shared Local Memory Index:** Indicates the starting index for the shared local memory for the thread group. Each index points to the start of a 4k memory block, 16 possibilities cover the entire 64k shared memory per half-slice. Format = U4
	23:16	**Reserved: MBZ**
	15:0	**URB Handle.** This is the URB handle where indicating the URB space for use by the thread.

Cross-thread CURBE if present is in R1 and above, followed by the X/Y/Z thread id values for each channel in the thread.

1.7.2 Thread Spawn Message

The thread spawn message is issued to the TS unit by a thread running on an EU. This message contains only one 8-DW register. The thread spawn message may be used to

- Spawn a child thread
- Spawn a root thread (start dispatching a synchronized root thread)
- Dereference URB handle
- Indicate a thread termination, dereference other TS managed resource and may or may not dereference URB handle
- Release a PRT_Fence ()

In order to end a root thread, the end of thread message must be targeted at the thread spawner. In this case, the root thread sends a message with a “dereference resource” in the Opcode field. The thread spawner does *not* snoop the messages sideband to determine when a root thread has ended. Thread Spawner does not track when a child thread terminates, to be consistent a child thread should also terminate with a “dereference resource” message to the Thread Spawner. Software must set the Requester Type (root or child thread) field correctly.

As TS dispatches one synchronized root thread upon receiving a ‘spawn root thread’ message (from a synchronization thread). The synchronizing thread must send the number of ‘spawn root thread’ message exactly the same as the subsequent ‘synchronized root thread’. No more, no less. Otherwise, hardware behavior is undefined.

URB Handle Offset field in this message (in M0.4) has 10 bits, allowing addressing of a large URB space. However, when a parent thread writes into the URB, it subjects to the maximum URB offset limitation of the URB write message, which is only 6 bits (see Unified Return Buffer Chapter for details). In this case, the parent thread may have to modify the URB Return Handle 0 field of the URB write message in order to subdivide the large URB space that the thread manages.

Only a persistent root thread can use this message to dispatch a root thread if preemption exceptions are possible. The root thread requested by this message is not guaranteed to dispatch, and the persistent

root thread must handle the case where it does not dispatch. When a context switch interrupt is recognized by the persistent root thread, all other root threads that had been dispatched have completed and no more will be dispatched. Child threads requested by this message are guaranteed to dispatch in all cases, so long as the persistent root thread does not also dispatch synchronized root threads. A child thread will not dispatch if it is behind a synchronized root thread that is not dispatched due to preemption exception.

In a system running child threads, a parent thread is monitoring the status of the child threads by communications through Message Gateway. When a child thread is about to terminate, it sends a message to the parent through Message Gateway and then sends a second message of EOT (end of thread) to TS.

There is a latency between sending a message to parent thread and the EOT to TS due to message bus arbitration. The parent thread may acknowledge the GW message and issue a new child dispatch before the EOT was processed; basically threads are issued faster than retired.

Because the messages for new child dispatch and EOT go to the same queue in TS, if the queue gets full, EOTs will get blocked. In the case when all the EUs/Threads are full, this will create a system deadlock: no EOTs can be acknowledged by TS (to free up EU resource) and no child threads can be dispatched (to free up TS queue to receive EOT message).

1.7.2.1 Message Descriptor

The following table shows the lower 16 bits of the message descriptor (lower 20 bits for) within the SEND instruction for a thread spawn message.

Bit	Description
19	**Header Present** This bit must be set to **zero** for all Thread Spawner messages. **:**this bit is not part of the shared function specific message descriptor**.**
18:5	Reserved: MBZ Bits 18:16 are not part of the shared function specific message descriptor.
4	**Resource Select.** This field specifies the resource associated with the action taken by the Opcode. If Opcode is “Spawn thread”, this field selects whether it is a child thread or a root thread. 0: spawn a *child* thread 1: spawn a *root* thread or (only) release a PRT_Fence If Opcode == “Dereference Resource”, this field indicates whether the URB handle is to be dereferenced. The URB handle can only be dereferenced once. 0: The URB handle is dereferenced 1: The URB handle is NOT dereferenced
3:2	**Reserved:** MBZ
1	**Requester Type.** This field indicates whether the requesting thread is a root thread or a child thread. If it is a root thread, when Opcode is 0, FF managed resources will be dereferenced. If it is a child thread and Opcode is 0, no resource will be dereferenced – basically no action is required by the TS. 0: Root thread

Bit	Description
	1: Child thread
0	**Opcode.** Indicates the operation performed by the message. A root thread must terminate with a message to TS (Opcode == 0 and EOT == 1). A child thread *should* also terminate with such a message. A thread cannot terminate with an Opcode of “spawn thread”. 0: dereference resource (also used for end of thread) 1: spawn thread

1.7.2.2 Message Payload

DWord	Bit	Description
M0.7	31:0	
M0.6	31:0	
M0.5	31:8	Ignored
	7:0	**FFTID.** This ID is assigned by TS and is a unique identifier for the thread in comparison to other concurrent root threads. It is used to free up resources used by a root thread upon thread completion. This field is valid only if the **Opcode** is “dereference resource”, and is ignored by hardware otherwise.
M0.4	31:16	Ignored
	15:10	Dispatch URB Length. Indicates the number of 8-DW URB entries contained in the Dispatch URB Handle that will be dispatched. When spawning a child thread, the URB handle contains most of the child thread's payload including R0 header. When spawning a root thread, the URB handle contains the message passed from the requesting thread to the spawned "peer" root thread. The number of GRF registers that will be initialized at the start of the spawned child thread is the addition of this field and the number of URB constants if present. The number of GRF registers that will be initialized at the start of a spawned root thread is the addition of this field, the number of URB constants if present, and the URB handle received from VFE. This field is ignored if the Opcode is "dereference resource". Length of 0 can be used while spawning child threads to indicate that there is no payload beyond the required R0 header. Length of 0 while spawning a root thread indicates that there is no payload at all from the parent thread. A spawned root has R0 supplied by the Media_Object command indirect/inline data. Format = U6 Range = [0,63] for child threads
	9:0	**URB Handle Offset.** Specifies the 8-DW URB entry offset into the URB handle that determines where the associated dispatch payload will be retrieved from when the spawned child or root thread is dispatched. This field is ignored if the **Opcode** is “dereference resource”. Format = U10 Range = [0,1023]
M0.3	31:0	Ignored
M0.2	31:28	Ignored
	27:24	**BarrierID**. This field indicates which one from the 16 Barriers this kernel is associated. Format: U4
	23:16	**Ignored**

DWord	Bit	Description
	15:10	Ignored
	9:4	**Interface Descriptor Offset.** This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors. Format = U5
	3:0	**Scoreboard Color** (only with MEDIA_OBJECT_EX)**:** This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control. Format = U4
M0.1	31:0	Ignored
M0.0	31:28	Ignored
	27:24	**Shared Local Memory Index:** Indicates the starting index for the shared local memory for the thread group. Each index points to the start of a 4k memory block, 16 possibilities cover the entire 64k shared memory per half-slice. Format = U4
	23:16	Reserved : MBZ
	15:0	**Dispatch URB Handle** If **Opcode** (and **Requester Type**) is “spawn a child thread”: Specifies the URB handle for the child thread. If **Opcode** (and **Requester Type**) is “spawn a root thread”: Specifies the URB handle containing message (e.g. requester’s gateway information) from the requesting thread to the spawned root thread. If **Opcode** is “dereference resource”: This field is required on end of thread messages if the **Children Present** bit is set, as the handle must be dereferenced, otherwise this field is ignored.

1.8 Media State and Primitive Commands

1.8.1 MEDIA_VFE_STATE Command

MEDIA_VFE_STATE		
Length Bias: 2		
An MI_FLUSH is required before MEDIA_VFE_STATE unless the only bits that are changed are scoreboard related: Scoreboard Enable, Scoreboard Type, Scoreboard Mask, Scoreboard * Delta. For these scoreboard related states, a MEDIA_STATE_FLUSH is sufficient. MEDIA_STATE_FLUSH (optional, only if barrier dependency is needed) MEDIA_INTERFACE_DESCRIPTOR_LOAD (optional)		
DWord	**Bit**	**Description**
0	31:29	**Command Type**

MEDIA_VFE_STATE

		Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 0h Format: OpCode
	23:16	**SubOpcode** Default Value: 0h MEDIA_VFE_STATE Format: OpCode
	15:0	**DWord Length** Format: =n Total Length - 2 **Value** / **Name** / **Description** 06h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1)
1	31:10	**Scratch Space Base Pointer** Format: GeneralStateOffset[31:10] Specifies the 1k-byte aligned address offset to scratch space for use by the kernel. This pointer is relative to the **General State Base Address.**
	9:8	**Reserved** Format: MBZ
	7:4	**Reserved** Format: MBZ
	3:0	**Per Thread Scratch Space** Format: U4 Specifies the amount of scratch space allowed to be used by each thread. The driver must allocate enough contiguous scratch space, pointed to by the Scratch Space Pointer, to ensure that the maximum threads in the device each get Per Thread Scratch Space size without exceeding the driver-allocated scratch space.. **Value** / **Name** / **Description** / **Project** [0,11] / / Indicating [1k bytes, 12k bytes] : 0->1k, 1->2k, 2->3k … 11->12k /
2	31:16	**Maximum Number of Threads** Format: U16-1 representing thread count Specifies the maximum number of simultaneous root threads allowed to be active. Used to avoid potential deadlock. Note that MSB will be zero due to the range limit below. Range: [0, n-1] where n = (# EUs) * (# threads/EU). See *Graphics Processing Engine* for listing of #EUs and #threads in each device.
	15:8	**Number of URB Entries**

MEDIA_VFE_STATE

		Format: U8 Specifies the number of URB entries that are used by the unit. **Value** / **Name** / **Description** / **Project** [0,64] / / [0,64] Entries /
	7	**Reset Gateway Timer** This field controls the reset of the timestamp counter maintained in Message Gateway. **Value** / **Name** 0h / Maintaining the existing timestamp state 1h / Resetting relative timer and latching the global timestamp
	6	**Bypass Gateway Control** This field configures Gateway to use a simple message protocol. **Value** / **Name** 0h / Maintaining OpenGateway/ForwardMsg/CloseGateway protocol (legacy mode) 1h / Bypassing OpenGateway/CloseGateway protocol
	5	**Reserved**
	4:3	**Gateway MMIO Access Control** The Gateway allows messages from EUs to read and write MMIO registers. This field limits this feature for security reasons **Value** / **Name** 0h / No MMIO read/write allowed 2h / MMIO read/write to any address
3	7:2	**Reserved** Format: MBZ
	1:0	**Reserved** Format: MBZ
4	31:16	**URB Entry Allocation Size** Format: U16 **Description** / **Project** Specifies the length of each URB entry used by the unit, in 256-bit register increments - 1. ROB address for URB starts after CURBE Allocated region. (URB Entry Allocation Size * Number of URB Entries) + CURBE Allocation Size + Interface Descriptor Entries) must be less than or equal to the number of entries in the URB as described in *vol5c.5 Shared Functions Unified Return Buffer,* under the section “URB Size”. / Interface Descriptor Entires is 32 / **Programming Notes** When Inline data is used with MEDIA_OBJECT or MEDIA_OBJECT_WALKER, then the URB entry allocation size must match the Inline data size. If Indirect data is being used with MEDIA_OBJECT then the allocation size does not matter, but the total Allocation Size * Number of URB Entries should be sufficient for the Indirect data. If both Inline and Indirect are being used, then the allocation size must match the Inline and the total space must be enough for both the Indirect and Inline.
	15:0	**CURBE Allocation Size**

MEDIA_VFE_STATE

		Format: U16

Description	Project
Specifies the total length allocated for CURBE, in 256-bit register increments. ROB address for CURBE starts at address 32 . (URB Entry Allocation Size * Number of URB Entries) + CURBE Allocation Size + Interface Descriptor Entries) must be less than or equal to the number of entries in the URB as described in *vol5c.5 Shared Functions Unified Return Buffer,* under the section “URB Size”.	
Interface Descriptor Entires is 32	
There are additional restrictions on CURBE Allocation:	

Value	Name	Project
[0,480]		
[0,2016]		

5	31	**Scoreboard Enable** This field enables and disables the hardware scoreboard in the Media Pipeline. If this field is cleared, hardware ignores the following scoreboard state fields.

Value	Name
0h	Scoreboard disabled
1h	Scoreboard enabled

	30	**Scoreboard Type** This field selects the type of scoreboard in use.

Value	Name	Project
0h	Stalling Scoreboard	
1h	Non-Stalling Scoreboard	

	29:8	**Reserved** Format: MBZ
	7:0	**Scoreboard Mask** Format: Enable[8] Each bit indicates the corresponding dependency scoreboard is enabled. The scoreboard is based on the relative (X, Y) distance from the current threads’ (X, Y) position. **Bit n (for n = 0…7):** Score n is enabled.
6	31:28	**Scoreboard 3 Delta Y** Format: S3 Relative vertical distance of the dependent instance assigned to scoreboard 3, in the form of 2’s compliment.
	27:24	**Scoreboard 3 Delta X** Format: S3 Relative horizontal distance of the dependent instance assigned to scoreboard 3, in the form of 2’s compliment.
	23:20	**Scoreboard 2 Delta Y** Format: S3 Relative vertical distance of the dependent instance assigned to scoreboard 2, in the form of 2’s

		compliment.
	19:16	**Scoreboard 2 Delta X** Format: S3 Relative horizontal distance of the dependent instance assigned to scoreboard 2, in the form of 2's compliment.
	15:12	**Scoreboard 1 Delta Y** Format: S3 Relative vertical distance of the dependent instance assigned to scoreboard 1, in the form of 2's compliment.
	11:8	**Scoreboard 1 Delta X** Format: S3 Relative horizontal distance of the dependent instance assigned to scoreboard 1, in the form of 2's compliment.
	7:4	**Scoreboard 0 Delta Y** Format: S3 Relative vertical distance of the dependent instance assigned to scoreboard 0, in the form of 2's compliment.
	3:0	**Scoreboard 0 Delta X** Format: S3 Relative horizontal distance of the dependent instance assigned to scoreboard 0, in the form of 2's compliment.
7	31:28	**Scoreboard 7 Delta Y** Format: S3 Relative vertical distance of the dependent instance assigned to scoreboard 7, in the form of 2's compliment.
	27:24	**Scoreboard 7 Delta X** Format: S3 Relative horizontal distance of the dependent instance assigned to scoreboard 7, in the form of 2's compliment.
	23:20	**Scoreboard 6 Delta Y** Format: S3 Relative vertical distance of the dependent instance assigned to scoreboard 6, in the form of 2's compliment.
	19:16	**Scoreboard 6 Delta X** Format: S3 Relative horizontal distance of the dependent instance assigned to scoreboard 6, in the form of 2's compliment.
	15:12	**Scoreboard 5 Delta Y** Format: S3

MEDIA_VFE_STATE		
		Relative vertical distance of the dependent instance assigned to scoreboard 5, in the form of 2's compliment.
	11:8	**Scoreboard 5 Delta X** Format: S3 Relative horizontal distance of the dependent instance assigned to scoreboard 5, in the form of 2's compliment.
	7:4	**Scoreboard 4 Delta Y** Format: S3 Relative vertical distance of the dependent instance assigned to scoreboard 4, in the form of 2's compliment.
	3:0	**Scoreboard 4 Delta X** Format: S3 Relative horizontal distance of the dependent instance assigned to scoreboard 4, in the form of 2's compliment.

1.8.2 MEDIA_CURBE_LOAD

MEDIA_CURBE_LOAD		
Length Bias:		2
DWord	**Bit**	**Description**
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 0h Format: OpCode
	23:16	**SubOpcode** Default Value: 1h Media_CURBE_LOAD Format: OpCode
	15:0	**DWord Length** Project: All Format: =n Total Length - 2 **Value** / **Name** / **Description** 2h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1)

MEDIA_CURBE_LOAD

1	31:0	**Reserved** Project: All Format: MBZ
2	31:17	**Reserved** Project: All Format: MBZ
	16:0	**CURBE Total Data Length** Project: All Format: U17 In Bytes **Description** (Project) This field provides the length in bytes of the CURBE data. This field must have the same alignment as the Curbe Object Data Start Address.As the CURBE data are sent directly to ROB, range is limited to CURBE Allocation Size. This field must be DWord (32-byte) aligned.
3	31:0	**CURBE Data Start Address** Project: All Format: DynamicStateOffset[31:0] CURBE **Description** (Project) Specifies the 32-byte (DWord) aligned address of the CURBE data. This pointer is relative to the **Dynamics Base Address**. **Value** / **Name** [0,FFFFFFFFh] **Programming Notes** (Project) Driver must invalidate the vertex fetch cache thru the **VF(address based) Cache Invalidation Enable** thru a PIPE_CONTROL command prior to reusing the same graphics memory space. P

1.8.3 MEDIA_INTERFACE_DESCRIPTOR_LOAD Command

MEDIA_INTERFACE_DESCRIPTOR_LOAD

Length Bias: 2

A Media_State_Flush should be used before this command to ensure that the temporary Interface Descriptor storage is cleared.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 0h Format: OpCode
	23:16	**SubOpcode** Default Value: 2h MEDIA_INTERFACE_DESCRIPTOR_LOAD Format: OpCode
	15:0	**DWord Length** Project: All Format: =n Total Length - 2 Value / Name / Description 2h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1)
1	31:5	**Reserved** Project: All Format: MBZ
	4:0	**Reserved** Project: All Format: MBZ
2	31:17	**Reserved** Project: All Format: MBZ
	16:0	**Interface Descriptor Total Length** Project: All Format: U17 In bytes This field provides the length in bytes of the Interface Descriptor data. This field must have the same alignment as the Interface Descriptor Data Start Address.It must be DQWord (32-byte) aligned. As the Interface Descriptor data are sent directly to ROB, range is limited to CURBE Allocation Size. Value / Name / Project [32,1024] / [1,32] interface descriptor entries /
3	31:0	**Interface Descriptor Data Start Address**

MEDIA_INTERFACE_DESCRIPTOR_LOAD

Project:	All
Format:	DynamicStateOffset[31:0]INTERFACE_DESCRIPTOR_DATA*32

Description	Project
This bit specifies the 32-byte aligned address of the Interface Descriptor data. This pointer is relative to the Dynamics Base Address.	

Value	Name
[0,FFFFFFFFh]	

Programming Notes	Project
Driver must invalidate the vertex fetch cache thru the **VF(address based) Cache Invalidation Enable** thru a PIPE_CONTROL command prior to reusing the same graphics memory space.	

1.8.4 INTERFACE_DESCRIPTOR_DATA

Interface Descriptor Data payload as pointed to by the Interface Descriptor Data Start Address:

INTERFACE_DESCRIPTOR_DATA

Default Value: 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000

DWord	Bit	Description
0	31:6	**Kernel Start Pointer** Project: All Format: InstructionBaseOffset[31:6]Kernel Specifies the 64-byte aligned address offset of the first instruction in the kernel. This pointer is relative to the **Instruction Base Address.**
	5:0	**Reserved** Project: All Format: MBZ
1	31:19	**Reserved** Project: All Format: MBZ
	18	**Single Program Flow (SPF)** Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). Value: 0h — Name: Multiple Program Flow Value: 1h — Name: Single Program Flow
	17	**Thread Priority** Specifies the priority of the thread for dispatch.

INTERFACE_DESCRIPTOR_DATA

		Value / **Name** 0h — Normal Priority 1h — High Priority
	16	**Floating Point Mode** Specifies the floating point mode used by the dispatched thread. **Value** / **Name** 0h — Use IEEE-754 Rules 1h — Use alternate rules
	15:14	**Reserved** Format: MBZ
	13	**Illegal Opcode Exception Enable** Project: All Format: Enable This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions and ISA Execution Environment*.
	12	**Reserved** Format: MBZ
	11	**MaskStack Exception Enable** Project: Format: Enable This bit gets loaded into EU CR0.1[11]. See *Exceptions and ISA Execution Environment*.
	10:8	**Reserved** Format: MBZ
	7	**Software Exception Enable** Format: Enable This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions and ISA Execution Environment*.
	6:0	**Reserved** Format: MBZ
2	31:5	**Sampler State Pointer** Format: DynamicStateOffset[31:5]SAMPLER_STATE Specifies the 32-byte aligned address offset of the sampler state table. This pointer is relative to the **Dynamic State Base Address.** *This field is ignored for child threads.*
	4:2	**Sampler Count** Format: U3 Specifies how many samplers (in multiples of 4) the kernel uses. Used only for prefetching the associated sampler state entries. *This field is ignored for child threads.* *If this field is not zero, sampler state is prefetched for the first instance of a root thread upon the startup of the media pipeline.* **Value** / **Name** [0,4] 0h — No samplers used

INTERFACE_DESCRIPTOR_DATA

		1h	Between 1 and 4 samplers used
		2h	Between 5 and 8 samplers used
		3h	Between 9 and 12 samplers used
		4h	Between 13 and 16 samplers used
	1:0	**Reserved**	
		Format:	MBZ
3	31:16	**Reserved**	
		Project:	
		Format:	MBZ
	15:5	**Binding Table Pointer**	
		Project:	
		Format:	SurfaceStateOffset[15:5]BINDING_TABLE_STATE*256
		Specifies the 32-byte aligned address of the binding table. This pointer is relative to the **Surface State Base Address.** *This field is ignored for child threads.*	
	4:0	**Binding Table Entry Count**	
		Format:	U5
		Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** The maximum number of prefetched binding table entries is limited to 31. For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache. *This field is ignored for child threads.* *If this field is not zero, binding table and surface state are prefetched for the first instance of a root thread upon the startup of the media pipeline.*	
		Value	**Name**
		[0,31]	
4	31:16	**Constant URB Entry Read Length**	
		Format:	U16
		Specifies the amount of URB data read and passed in the thread payload for the Constant or Indirect URB entry, in 8-DW register increments. A value 0 means that no Constant or Indirect URB Entry will be loaded. The Constant URB Entry Read Offset field will then be ignored.	
		Value	**Name**
		[0,63]	
	15:0	**Constant URB Entry Read Offset**	
		Format:	U16
		Specifies the offset (in 8-DW units) at which Constant URB data is to be read from the URB before being included in the thread payload.	

Value	Name	Description	Project
[0,2015]		indicating [0,2015] 256-bit register increments. ROB has 64KB of storage; 2048 entries. However, lowest 32 entries are reserved for VFE/TS to store interface descriptor data. Hence, URB Entry Read Offset plus Read Length shall not exceed 2016.	
[0, 479]			

5	31:24	**Reserved**	
		Project:	
		Format:	MBZ

INTERFACE_DESCRIPTOR_DATA

	31:24	**Barrier Return GRF Offset** Project: | Format: | U8 This field specifies the offset into the GRF that the barrier return byte will be written to. **Value** | **Name** 0,127 |
	23:22	**Rounding Mode** Project: | Format: | U2 **Value** | **Name** | **Description** 00b | RTNE **[Default]** | Round to Nearest Even 01b | RU | Round toward +Infinity 10b | RD | Round toward –Infinity 11b | RTZ | Round toward Zero
	21	**Barrier Enable** | Format: | Enable This field specifies whether the thread group requires a barrier. If not, it can be dispatched without allocating one.
	20:16	**Shared Local Memory Size** | Format: | U5 This field indicates how much shared local memory the thread group requires. The amount is specified in 4k blocks, but only powers of 2 are allowed: 0, 4k, 8k, 16k, 32k and 64k per half-slice. **Value** | **Name** | **Description** [0,16] | | Encodes 0k to 64k in powers of 2
	15:8	**Reserved** Project: | Format: | MBZ
	15:8	**Barrier Return Byte** Project: | Format: | U8 This field specifies the byte that will be returned by the gateway when the barrier is reached.
6	31:0	**Reserved** Project: | Format: | MBZ
7	31:0	**Reserved** Format: | MBZ

1.8.5 MEDIA_STATE_FLUSH command

The MEDIA_STATE_FLUSH command is updated to specify all the resources required for the next thread group via an interface descriptor – if the resources are not available the group cannot start.

Two MEDIA_STATE_FLUSH commands need to be used to ensure that the flush is complete.

MEDIA_STATE_FLUSH

Length Bias: 2

This command updates the Message Gateway state. In particular, it updates the state for a selected Interface Descriptor.

This command can be considered same as a MI_Flush except that only media parser will get flushed instead of the entire 3D/media render pipeline. The command should be programmed prior to new Media state, curbe and/or interface descriptor commands when switching to a new context or programming new state for the same context.

With this command, pipelined state change is allowed for the media pipe.

It should be cautious when using this command when child_present flag in the media state is enabled. This is because that CURBE state as well as Interface Descriptor state are shared between root threads and child threads. Changing these states while child threads are generated on the fly may cause unexpected behavior.

Combining with MI_ARB_ON/OFF command, it is possible to support interruptability with the following command sequence where interrupt may be allowed only when MI_ARB_ON_OFF is ON:

MEDIA_STATE_FLUSH
VFE_STATE // VFE will hold CS if watermark isn't met
MI_ARB_OFF // There must be at least one VFE command before this one
MEDIA_OBJECT …. MI_ARB_ON

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 0h Format: OpCode
	23:16	**SubOpcode A** Default Value: 4h MEDIA_STATE_FLUSH Format: OpCode
	15:0	**DWord Length** Project: All Format: =n Total Length - 2 Value / Name / Description 0h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1)
1	31:9	**Reserved** Project: All Format: MBZ
	8	**Disable Pre-emption** Project: Format: Enable This bit causes the video front-end to ignore pre-emption requests if set. If this bit is set then ARB_CHECK commands should not be used with it. A subsequent MEDIA_STATE_FLUSH command with this bit cleared will honor previous pre-emption

requests.

7	**Reserved**
	Format: MBZ
6	**Watermark Required**
	Project: All
	This is a single bit specifying if the MEDIA_STATE_FLUSH should stall further commands until there is enough room in a half-slice for the following thread group. The characteristics of the thread group are specified in the Interface Descriptor Offset. If set, the MEDIA_STATE_FLUSH stalls CS until there are enough threads in a half-slice, and enough SLM available in the same half-slice, and a free barrier if one is required. An Interface Descriptors can be updated after a Watermarked MEDIA_STATE_FLUSH only if it has not been used in the current context. Reusing an interface desciptor requires that this bit is clear to ensure the ID cache is reloaded. If clear, the MEDIA_STATE_FLUSH stalls CS until the TDL has dispatched the last thread, allowing the CURBE and Interface Descriptors to be updated by following commands.
	Programming Notes / **Project**
5:0	**Interface Descriptor Offset**
	Format: U5
	This field specifies the offset from the interface descriptor base pointer to the interface descriptor which describes what resources are required to meet the watermark.

1.8.6 MEDIA_OBJECT Command

The MEDIA_OBJECT command is the basic media primitive command for the media pipeline. It supports loading of inline data as well as indirect data. At least one form of payload (either inline, indirect or CURBE) must be sent with the MEDIA_OBJECT.

MEDIA_OBJECT

Length Bias: 2

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Media Command Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 1h MEDIA_OBJECT Format: OpCode
	23:16	**Media Command Sub-Opcode** Default Value: 0h MEDIA_OBJECT Format: OpCode
	15:0	**DWord Length** Default Value: 4h DWORD_COUNT_n Format: =n Total Length - 2 Excludes DWords 0,1 **Generic Mode:** DWord Length = N+4, where N is in the range of [0,504]. The maximum is 504 DW (equivalent to 63 8-DW registers). When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than 112 (with both inline data length N and indirect data length rounded up to 8-DW aligned individually). The minimal inline data length is 0.
1	7:6	**Reserved** Format: MBZ
	5:0	**Interface Descriptor Offset** Format: U5 This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors.
2	31	**Children Present** Format: Enable Indicates that the root thread may send spawn messages to spawn child threads and/or synchronized root threads. If Children Present is not set, TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched. If Children Present is set, the URB handle is forwarded to the root thread and serves as the return

URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread.
In order avoid deadlock, such dereference must be issued once and only once for each URB handle.

30:25 **Reserved**

Format:	MBZ

24 **Thread Synchronization**

This field when set indicates that the dispatch of the thread originated from this command is based on the “spawn root thread” message.

Value	Name
0	No thread synchronization
1	Thread dispatch is synchronized by the “spawn root thread” message

23 **Reserved**

Format:	MBZ

22 **Reserved**

Format:	MBZ

21 **Use Scoreboard**

This field specifies whether the thread associated with this command uses hardware scoreboard. Only when this field is set, the scoreboard control fields in the VFE Dword are valid. If this field is cleared, the thread associated with this command bypasses hardware scoreboard.

Value	Name
0	Not using scoreboard
1	Using scoreboard

20 **Reserved**

Format:	MBZ

19 **Reserved**

Format:	MBZ

18:17 **Half-Slice Destination Select**

This field selects the half slice that this thread must be sent to.

Value	Name	Description
10b	Half-Slice 1	Cannot be used in products without a Half-Slice 1.
01b	Half-Slice 0	
00b	Either half-slice	Hardware will choose the slice based on load.

Programming Notes
If "Either half-slice" is selected then the Slice Destination Select must also specify "Either slice".

16:0 **Indirect Data Length**

Format:	U17 In bytes

This field provides the length in bytes of the indirect data. A value zero indicates that indirect data fetching is disabled – subsequently, the Indirect Data Start Address field is ignored.
This field must have the same alignment as the Indirect Object Data Start Address.
It must be DQWord (32-byte) aligned. As the indirect data are sent directly to URB, range is limited to

		496 DW. When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than 112 (with both inline data length and indirect data length rounded up to 8-DW aligned).
3	31:0	**Indirect Data Start Address** **Description** / **Project** This field specifies the Graphics Memory starting address of the data to be loaded into the kernel for processing. This pointer is relative to the **Indirect Object Base Address**. Hardware ignores this field if indirect data is not present. Alignment of this address depends on the mode of operation. This field specifies the DWord aligned address of the indirect data. **Value** / **Name** [0,512MB] **Programming Notes** / **Project** Driver must invalidate the vertex fetch cache through the VF(address based) Cache Invalidation Enable through a PIPE_CONTROL command prior to reusing the same graphics memory space. Bits 31:29 MBZ
4	31:25	**Reserved** Format: MBZ
	24:16	**Scoredboard Y** Format: U9 This field provides the Y term of the scoreboard value of the current thread.
	15:9	**Reserved** Format: MBZ
	8:0	**Scoreboard X** Format: U9 This field provides the X term of the scoreboard value of the current thread.
5	31:20	**Reserved** Format: MBZ
	19:16	**Scoreboard Color** Format: U4 This field specifies which dependency color the current thread belongs to. It affects the dependency scoreboard control.
	15:8	**Reserved** Format: MBZ
	7:0	**Scoreboard Mask** Format: Boolean Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with

MEDIA_OBJECT		
		the corresponding Scoreboard Mask field in the MEDIA_VFE_STATE command. **Bit n (for n = 0…7):** Scoreboard n is dependent, where bit 0 maps to n = 0.
6..n	31:0	**Inline Data** Generic Mode: The format of this data is specified by software. Hardware does not interpret this data; it merely passes it to the kernel for processing. The total size for the inline data and indirect data must not exceed 112 registers.

1.8.7 MEDIA_OBJECT_PRT Command

MEDIA_OBJECT_PRT		
Length Bias:		2
command is for generating Persistent Root Thread for the media pipeline. It only supports loading of inline data but not indirect data. This command should be used for a root thread that might have to be present in the system for a period longer than the certain minimal context-switch interrupt latency. It has to honor the context interrupt signal to terminate upon request. It should also handle replay from theinterrupted point upon context restore (the same thread being dispatched more than once). In contrary, if a thread is not a Persistent Root Thread, if dispatched, it must run to completion. The command can be used in all VFE modes, except VLD mode.		
For simplification, _PRT command has a fixed size of 16 DWORD		
DWord	**Bit**	**Description**
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 1h Format: OpCode
	23:16	**SubOpcode** Default Value: 2h MEDIA_OBJECT_PRT Format: OpCode
	15:0	**DWord Length** Project: All Format: =n Total Length - 2 Note: Regardless of the mode, inline data must be present in this command. The command size must fit within 16 dwords. **Value** / **Name** / **Description** 14h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1)
1	31:6	**Reserved**

MEDIA_OBJECT_PRT

		Project: All Format: MBZ
	5:0	**Interface Descriptor Offset** Project: All Format: U5 This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors.
2	31	**Children Present** Project: All Format: Enable Indicates that the root thread may send spawn messages to spawn child threads and/or synchronized root threads. If Children Present is not set, TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched. If Children Present is set, the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread. In order avoid deadlock, such de-reference must be issued once and only once for each URB handle.
	30:24	**Reserved** Project: All Format: MBZ
	23	**PRT_Fence Needed** Project: All Format: Enable This field specifies that a PRT_Fence is generated after dispatching the thread associated with this MEDIA_OBJECT_PRT. The PRT_Fence prevents additional threads following this persistent root thread until a thread spawn message is sent. The PRT_Fence is generated on first dispatch of the persistent root, as well as on re-dispatches of the persistent root after context restore.
	22	**PRT_FenceType** Project: All This field specifies the type of fence the PRT thread uses. If this field is set to 0, the fence is set at the end of the root thread queue. It will block the dispatch of the next root thread, but allowed these root threads to be populated through VFE to the root thread queue in TS. If this field is set to 1, the fence is set at the entry of VFE, similar to the fence set by the MEDIA_STATE_FLUSH command. No more command can go into the media pipe until a thread spawn message is sent (by the PRT). This field is only valid when PRT_Fence Needed is set to 1. Otherwise, it is ignored by hardware.

Value	Name	Description	Project
0h	Root thread queue	Root thread queue fence	All
1h	VFE state flush	VFE state flush fence	All

	21:0	**Reserved** Project: All Format: MBZ
3	31:0	**Reserved** Format: MBZ
4..15	31:0	**Inline Data** Project: All Format: U32

1.8.8 MEDIA_OBJECT_WALKER Command

The MEDIA_OBJECT_WALKER command uses the hardware walker in VFE for generating threads associated with a rectangular shaped object. It only supports loading of inline data or CURBE but not indirect data. At least one form of payload must be sent. Control of scoreboards (up to 8) is implicit based on the (X, Y) address of the generated thread and the scoreboard control state.

The command can be used only in Generic modes.

When **Use Scoreboard** field is set, the (X, Y) address and the Color field of the generated thread are used in the hardware scoreboard and the thread dependencies are set by states from the MEDIA_VFE_STATE command.

One or more threads may be generated by this command. This command doesn't support indirect object load. When inline data is present, it is repeated for all threads it generates. Unlike CURBE, which requires pipeline flush for change, continued change of this kind of 'global' (in the sense of shared by multiple threads from this command) data is supported when MEDIA_OBJECT_WALKER commands are issued without a pipeline flush in between.

MEDIA_OBJECT_WALKER		
Length Bias:		2
DWord	**Bit**	**Description**
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 1h Format: OpCode
	23:16	**SubOpcode** Default Value: 03h MEDIA_OBJECT_WALKER Format: OpCode
	15:0	**DWord Length** Default Value: Eh DWORD_COUNT_n Project: All Format: =n Total Length - 2 **Note:** If this field is greater than 15, it indicates that inline data is present. If present, inline data is common for all threads generated from this command, If this field is 15, it indicates that inline data is not present. It should be noted that unlike other media object command, inline data is optional for this command.
1	7:6	**Reserved** Format: Reserved
	5:0	**Interface Descriptor Offset**

		Format: U5 This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors.
2	31	**Children Present** Format: Enable Indicates that the root thread may send spawn messages to spawn child threads and/or synchronized root threads. If Children Present is not set, TS signals VFE to dereference the URB handle immediately after it receives acknowledgement from TD that the thread is dispatched. If Children Present is set, the URB handle is forwarded to the root thread and serves as the return URB handle for the root thread. TS does not signal deference at the time of dispatch. TS signals URB handle deference only when it receives a resource dereference message from the thread. *In order avoid deadlock, such dereference must be issued once and only once for each URB handle.*
	30:25	**Reserved** Format: MBZ
	24	**Thread Synchronization** This field when set indicates that the dispatch of the thread originated from this command is based on the “spawn root thread” message. Value: 0 — Name: No thread synchronization Value: 1 — Name: Thread dispatch is synchronized by the “spawn root thread” message
	23:22	**Reserved** Format: MBZ
	21	**Use Scoreboard** This field specifies whether the thread associated with this command uses hardware scoreboard. Only when this field is set, the scoreboard control fields in the VFE Dword are valid. If this field is cleared, the thread associated with this command bypasses hardware scoreboard. Value: 0 — Name: Not using scoreboard Value: 1 — Name: Using scoreboard
	20:17	**Reserved** Format: MBZ
	16:0	**Indirect Data Length** Format: U17 in bytes This field provides the length in bytes of the indirect data. A value zero indicates that indirect data fetching is disabled – subsequently, the Indirect Data Start Address field is ignored. This field must have the same alignment as the Indirect Object Data Start Address. It must be DQWord (32-byte) aligned. As the indirect data are sent directly to URB, range is limited to 496 DW. When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than or equal to 63 (with both inline data length and indirect data length rounded up to 8-DW aligned).
3	31:0	**Indirect Data Start Address** Description: This field specifies the Graphics Memory starting address of the data to be loaded into the — Project:

		kernel for processing. This pointer is relative to the **Indirect Object Base Address.** Hardware ignores this field if indirect data is not present. Alignment of this address depends on the mode of operation. It is the DWord aligned address of the indirect data.

Value	Name	Description
[0 - 512MB]		(Bits 31:29 MBZ)

4	31:0	**Reserved** Format: MBZ
5	7:0	**Scoreboard Mask** Format: Boolean Each bit indicates the corresponding dependency scoreboard is dependent on. This field is AND'd with the corresponding Scoreboard Mask field in the MEDIA_VFE_STATE. All threads generated by this walker command share the same dynamic mask. **Bit n (for n = 0…7):** Scoreboard n is dependent, where bit 0 maps to n = 0.
6	31	**Dual Mode** Project: Format: Boolean

Programming Notes	Project
Dual mode should be used in products that have 2 half-slices.	

	30	**Repel** Format: Boolean

Programming Notes
Repel should not be combined with either Dual Mode or Quad Mode.

	29	**Reserved** Format: MBZ
	28	**Reserved** Format: MBZ
	27:24	**Color Count Minus One** Format: U4 This field specifies the number of repeat of the inner most loop of the walker. Each repeated walk position is assigned with an incremental Color number. The Color number together with the X and Y position of the thread is used for dependency scoreboard control. **Usage Example:** This allows multiple sets of dependency threads to be dispatched.
	23:21	**Reserved** Format: MBZ
	20:16	**Middle Loop Extra Steps** Format: U5
	15:14	**Reserved** Format: MBZ

MEDIA_OBJECT_WALKER

	13:12	**Local Mid-Loop Unit Y** Format: S1
	11:10	**Reserved** Format: MBZ
	9:8	**Mid-Loop Unit X** Format: S1
	7:0	**Reserved** Format: MBZ
7	31:26	**Reserved** Format: MBZ
	25:16	**Global Loop Exec Count** Format: U10
	15:10	**Reserved** Format: MBZ
	9:0	**Local Loop Exec Count** Format: U10
8	31:25	**Reserved** Format: MBZ
	24:16	**Block Resolution Y** Format: U9 Vertical resolution of the local loop.
	15:9	**Reserved** Format: MBZ
	8:0	**Block Resolution X** Format: U9 Horizontal resolution of the local loop.
9	31:25	**Reserved** Format: MBZ
	24:16	**Local Start Y** Format: U9 Starting vertical position of the local loop.
	15:9	**Reserved** Format: MBZ
	8:0	**Local Start X** Format: U9 Starting horizontal position of the local loop.
10	31:25	**Reserved** Format: MBZ
	24:16	**Local End Y** Format: U9 Ending vertical position of the local loop.

MEDIA_OBJECT_WALKER			
	15:9	**Reserved**	
		Format:	MBZ
	8:0	**Local End X**	
		Format:	U9
		Ending horizontal position of the local loop.	
11	31:26	**Reserved**	
		Format:	MBZ
	25:16	**Local Outer Loop Stride Y**	
		Format:	S9
		Vertical stride of the local outer loop, in 2's complement.	
	15:10	**Reserved**	
		Format:	MBZ
	9:0	**Local Outer Loop Stride X**	
		Format:	S9
		Horizontal stride of the local outer loop, in 2's complement.	
12	31:26	**Reserved**	
		Format:	MBZ
	25:16	**Local Inner Loop Unit Y**	
		Format:	S9
		Vertical stride of the local inner loop, in 2's complement.	
	15:10	**Reserved**	
		Format:	MBZ
	9:0	**Local Inner Loop Unit X**	
		Format:	S9
		Horizontal stride of the local inner loop, in 2's complement.	
13	31:25	**Reserved**	
		Format:	MBZ
	24:16	**Global Resolution Y**	
		Format:	U9
		Vertical resolution of the global loop.	
	15:9	**Reserved**	
		Format:	MBZ
	8:0	**Global Resolution X**	
		Format:	U9
		Horizontal resolution of the global loop.	
14	31:26	**Reserved**	
		Format:	MBZ
	25:16	**Global Start Y**	
		Format:	S9

MEDIA_OBJECT_WALKER		
		Starting vertical location of the global loop, in 2's complement.
	15:10	**Reserved** Format: MBZ
	9:0	**Global Start X** Format: S9 Starting horizontal location of the global loop, in 2's complement.
15	31:26	**Reserved** Format: MBZ
	25:16	**Global Outer Loop Stride Y** Format: S9 Vertical stride of the global outer loop, in 2's complement.
	15:10	**Reserved** Format: MBZ
	9:0	**Global Outer Loop Stride X** Format: S9 Horizontal stride of the global outer loop, in 2's complement.
16	31:26	**Reserved** Format: MBZ
	25:16	**Global Inner Loop Unit Y** Format: S9 Vertical stride of the global inner loop, in 2's complement.
	15:10	**Reserved** Format: MBZ
	9:0	**Global Inner Loop Unit X** Format: S9 Horizontal stride of the global inner loop, in 2's complement.
17..n	31:0	**Inline Data**

1.8.9 GPGPU_OBJECT

This command is modified from the MEDIA_OBJECT command.

GPGPU_OBJECT		
Length Bias:		2

Programming Notes
If the threads spawned by this command are required to observe memory writes performed by threads spawned from a previous command, software must precede this command with a command that performs a memory flush (e.g., MI_FLUSH).

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 1h Format: OpCode
	23:16	**SubOpcode** Default Value: 04h MEDIA_ Format: OpCode
	15:9	**Reserved** Project: All Format: MBZ
	8	**Predicate Enable** Project: All Format: U1 If set, this command is executed (or not) depending on the current value of the MI Predicate internal state bit. This command is ignored only if PredicateEnable is set and the Predicate state bit is 0.
	7:0	**DWord Length** Format: =n Total Length -2 There are 4 DW needed to specify the Thread Group ID and the execution mask. **Value** / **Name** / **Description** 6h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1)
1	7	**Shared Local Memory Fixed Offset** This bit, if set, specifies that the offset into the 64k Shared Local Memory for the current thread group is specified by software in the Shared Local Memory Offset field. **Value** / **Name** / **Description** 0 / Thread Groups Offset / Offset to start of segment determined by hardware based on concurrently running thread groups. 1 / Shared Local Memory Offset / Offset to start of the Shared Local Memory segment supplied in Shared Local Memory Offset
	6	**Reserved** Project: All

GPGPU_OBJECT

		Format: MBZ
	5:0	**Interface Descriptor Offset** Format: U5 This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors.In VLD mode, this field is ignored by hardware.
2	31:28	**Shared Local Memory Offset** Format: U4 **Description** / **Project** If the Shared Local Memory Fixed Offset is set, this field provides the offset to the start of the Shared Local Memory for this thread group. The value of this field is multiplied by 4k to get the starting address. All threads in the thread group must have the same value. Offset must be aligned with Shared Local Memory Size of the thread group.
	27:25	**Reserved** Project: All Format: MBZ
	24	**End of Thread Group** Project: This bit indicates that this dispatch is the last for the current thread group.
	23:20	**Reserved** Project: All Format: MBZ
	19	**Reserved** Project: Format: MBZ
	18:17	**Half-Slice Destination Select** This field selects the half slice that this thread must be sent to. **Value** / **Name** / **Description** 10b / Half-Slice 1 / Cannot be used in products without a Half-Slice 1. 01b / Half-Slice 0 / 00b / Either Half-Slice / Hardware will choose the slice based on load.
	16:0	**Indirect Data Length** Format: U17 in bytes This field provides the length in bytes of the indirect data. A value zero indicates that indirect data fetching is disabled – subsequently, the Indirect Data Start Address field is ignored. Thread IDs This field must have the same alignment as the Indirect Object Data Start Address. It must be DQWord (32-byte) aligned. As the indirect data are sent directly to URB, range is limited to 496 DW. When both inline and indirect data are fetched for this command, the total size in 8-DW registers must be less than or equal to 63 (with both inline data length and indirect data length rounded up to 8-DW aligned).
3	31:0	**Indirect Data Start Address** This field specifies the Graphics Memory starting address of the data to be loaded into the kernel for

GPGPU_OBJECT		
		processing. This pointer is relative to the Indirect Object Base Address. Hardware ignores this field if indirect data is not present. The start address is a 64-byte aligned address. (Bits 31:29 MBZ) **Value** | **Name** [0,512MB) |
4	31:0	**Thread Group ID X** This is the X coordinate of the group id.
5	31:0	**Thread Group ID Y** This is the Y coordinate of the group id for all channels generated by this command.
6	31:0	**Thread Group ID Z** This is the Z coordinate of the thread group id.
7	31:0	**Execution Mask** This provides a bit per channel enable for the SIMD32 dispatch. The LSB of the Mask enables the execution of SIMD32 channel 0; the remaining bits enable the corresponding channel numbers. SIMD16 and SIMD8 dispatches should use the LSB bits of the mask. Any disabled channel will not read or write data to memory.

1.8.10 GPGPU_WALKER Command

GPGPU_WALKER		
Length Bias:		2
Programming Notes		
If the threads spawned by this command are required to observe memory writes performed by threads spawned from a previous command, software must precede this command with a command that performs a memory flush (e.g., MI_FLUSH).		
DWord	**Bit**	**Description**
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Pipeline** Default Value: 2h Media Format: OpCode
	26:24	**Media Command Opcode** Default Value: 1h Format: OpCode
	23:16	**SubOpcode A** Default Value: 05h Format: OpCode
	15:11	**Reserved** Project: All Format: MBZ
	10	**Indirect Parameter Enable** Project: All Format: U1

If set, the values in DW 4, 6, 8 are ignored and replaced by the current values of the corresponding GPGPU_xxx MMIO registers:

GPGPU_DISPATCHDIMX (instead of DW4)

GPGPU_DISPATCHDIMY (instead of DW6)

GPGPU_DISPATCHDIMZ (instead of DW8)

9 Reserved

Project:	All
Format:	MBZ

8 Predicate Enable

Project:	All
Format:	U1

If set, this command is executed (or not) depending on the current value of the MI Predicate internal state bit. This command is ignored only if **PredicateEnable** is set and the Predicate state bit is 0.

7:0 DWord Length

Project:	All
Format:	=n Total Length - 2

Value	Name	Description
9h	DWORD_COUNT_n **[Default]**	Allowed value is 9

1 7:6 Reserved

Project:	All
Format:	MBZ

5:0 Interface Descriptor Offset

Format:	U5

This field specifies the offset from the interface descriptor base pointer to the interface descriptor which will be applied to this object. It is specified in units of interface descriptors.

2 31:30 SIMD Size

This field determines the size of the payload and the number of bits of the execution mask that are expected. The kernel pointed to by the interface descriptor should match the SIMD declared here.

Value	Name	Description
0	SIMD8	8 LSBs of the execution mask are used
1	SIMD16	16 LSBs used in execution mask
2	SIMD32	32 bits of execution mask used

29:22 Reserved

Project:	All
Format:	MBZ

21:16 Thread Depth Counter Maximum

GPGPU_WALKER

		The maximum value of the thread depth counter. Thread_Depth_Max*Thread_Height_Max*Thread_Width_Max <= 32 for SIMD32, <= 64 for SIMD16 and SIMD8
	15:14	**Reserved** Project: All Format: MBZ
	13:8	**Thread Height Counter Maximum** The maximum value of the thread height counter
	7:6	**Reserved** Project: All Format: MBZ
	5:0	**Thread Width Counter Maximum** The maximum value of the thread width counter
3	31:0	**Thread Group ID Starting X** This is the initial value of the X component of the thread group. When X reaches the maximum value it rolls around to 0, not to this value.
4	31:0	**Thread Group ID X Dimension** The X dimension of the thread group (maximum X is dimension -1)
5	31:0	**Thread Group ID Starting Y** This is the initial value of the Y component of the thread group. When Y reaches the maximum value it rolls around to 0, not to this value.
6	31:0	**Thread Group ID Y Dimension** The Y dimension of the thread group (maximum Y is dimension -1)
7	31:0	**Thread Group ID Starting Z** This is the initial value of the Z component of the thread group
8	31:0	**Thead Group ID Z Dimension** The Z dimension of the thread group (maximum Z is dimension -1)
9	31:0	**Right Execution Mask**
10	31:0	**Bottom Execution Mask**

Revision History

Revision Number	Description	Revision Date
1.0	First 2012 OpenSource edition	May 2012

§§