Intel® UHD Graphics Open Source

Programmer's Reference Manual

For the 2020 Intel Core™ Processors with Intel Hybrid Technology based on the "Lakefield" Platform

Volume 7: Memory Cache

April 2021, Revision 1.0

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

Customer is responsible for safety of the overall system, including compliance with applicable safety-related requirements or standards.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Table of Contents

Memory Cache

This section describes the GFX L3 Cache, which is a large storage that backs up various L1 and L2 caches inside the design. It supports a simple, way-based partitioning for segregating the cache among groups of clients. It also has a provision to dedicate a (programmable) section of the storage for the GFX Unified Return Buffer.

L3 Cache

This is the PRM volume describing the L3/URB/SLM. Much of the design is similar to prior generations, with important changes to enhance performance.

Overview

In order to cater to the bandwidth demands, the L3 cache is organized as multiple independent banks which can be accessed concurrently. The cache arrays are clocked at 2X the base clock to achieve the bandwidth.

The L3 cache and URB data form a single contiguous memory space across all the banks and sub banks in the design. The vision is to build a compute scalable cache where with each additional compute block, both the size and bandwidth of L3 Cache are scaled while maintaining the monolithic cache concept. Each added bank becomes a part of a unified cache rather than an independent localized segment. The concept is to be able to keep a single copy of a line and service all requesters via distributing their accesses over many physical cache banks. The L3 cache can operate concurrently in the non-IA-coherent GFX virtual address space as well as the IA-coherent address space.

- Each logical bank consists of:
    - The Data Array - This array stores the actual data
    - The Tag Array - This array stores the tags for the cachelines above
    - The LRU Array - This array stores information that helps determining the cache line that will be evicted when a fill arrives for a set
    - The State Array - This array stores the cache state information (MESI States of the cache lines and some additional internal information)
    - The SuperQ Buffer - This array stores data temporarily on the way in or out of the data array for each access that is in progress
    - the Atomic Processing Units - This unit houses the ALU and associated logic to perform atomic operations on the data
- The rest of the support logic around L3 consists of:
    - The SuperQ : This is the main scheduler of the micro-sequences to be followed for each access to the cache
    - Ingress/Egress queues: These queues buffer incoming accesses and outgoing data on their way into or out of the cache
    - CAM structures: These structures are used to maintain coherency in cases of proximal accesses to the same address

- Crossbars: These are used for routing the data to and from the various sub-banks/arrays

Note : A portion of the L3 cache can be allocated as a Unified Return Buffer (URB) region

Bandwidth and Throughput Capability

Bandwidth and throughput

The table below shows the throughput capability for the L3/URB per bank. Note that L3$ and URB share the same pipelines, so the throughput for L3$ and URB is shared as well. Total system throughput is derived by multiplying the throughput numbers below by the total number of L3 Banks supported by a product. There is an additional limitation on throughput based the source of the transactions. A single client (e.g. an HDC or sub-slice) can only issue one 64-byte read or write per clock.

Memory Type	Read Throughput (Bytes/Clock)	Write Throughput (Bytes/Clock)	Atomic Throughput (32b Ops/Clock)
L3$ or URB	2*64	1*64	10

L3 Bank Configuration

Each L3 bank is configured as described below.

- Each bank consists of a 320KB data-array organized as 80 logical ways
- Up to 64 ways representing 256KB, tagged for L3$, remaining ways treated as dedicated URB.
- The minimum URB size per L3 Bank is 64KB, but this can be programmed up to 128KB.
- 64B Cacheline storage per cell.
- 64B@2x clock interface Data Buffer for the fill/write path and 64B@2x clock Read/Evict path to Data Buffer.
    - The total Array bandwidth is 128B for each core cycle. This bandwidth can be utilized as:
        - Two independent 64B read per cycle
        - One 64B read and one 64B write per cycle
        - One 64B write per cycle
- Data protection via ECC.
- 1b LRU implementation for selecting the line to be replaced.

Size of L3 Bank and Allocations

General notes

The LKF L3 Cache has been divided into following client pools:

- URB: Local memory space
- DC: Data Cluster Data type
- Color : Color cache allocation
- Z : Depth cache allocation

In addition to these sub-groups, a collection of groups are generated to bundle multiple clients under the same allocation set:

- Read-Only (RO) Clients: Inst/State, Constants & Textures (I/S/C/T)
- All L3 Clients (a.k.a "Rest of L3"): DC, Inst/State, Constants & Textures
- Unified Tile Cache : Color/Z combined

Multi-Bank Allocation Options with Tile Cache and Command buffer support

The L3 does not support the highly banked SLM mode of operations. Out of the total L3 space per bank, a minimum of 64Kbytes will be earmarked for URB storage. The rest can be used as tagged L3 cache. L3 Cache allocation is done on a per way basis. The Bank programming options allow a varied set of configurations to be programmed in the L3. Broadly, the number of ways allocated to the various sections (URB, DC, RO, Z, C and the command buffer) is selectable. Each of these sections can be allocated any number of ways out of the total ways. Apart from that, in lieu of the DC and RO sections a single unified “Rest of L3” section can be used. Similarly, in lieu of separate Z and C partitions in the L3, a single Unified Tile Cache programming allows all Z and C streams to share a common section of the L3.

L3 Space allocation can only be changed when the GPU pipeline is completely flushed. To guarantee that following two events need to be executed prior to the inline register updates to L3 allocation registers:

1. PIPECONTROL FLUSH, CS Stall set, with HDC Flush set, RO cache invalidation set if required (This flush command ensures the workload is completely drained, Datapipe is completely flushed followed by initiation of RO cache invalidation. Doesn’t ensure RO cache invalidation is complete)
2. PIPECONTROL FLUSH , CS Stall set, With HDC flush. (This flush ensures any prior RO cache invalidation in progress to be complete before processing flush for this command, this will avoid RO cache invalidation colliding with the command to change the allocations.)

LKF Configuration

For LKF1, L3 size has been reduced to 320KB banks. The configuration has been adjusted as following.

L3 Allocation programming (KBytes per bank)								
URB	**Rest** **(DC+RO)**	**DC**	**RO** **(I/S/C/T)**	**Z**	**Color**	**Unified** **Tile Cache**	**Command** **Buffer**	**Sum**
64 to 320 in increments of 4KB	0 to 256 in increments of 4KB	0 to 256 in increments of 4KB	0 to 256 in increments of 4KB	0 to 256 in increments of 4KB	0 to 256 in increments of 4KB	0 to 256 in increments of 4KB	0 to 256 in increments of 4KB	320

However, several restrictions apply. It is not allowed to allocate the entire cache to DC (Data space) with 0KB for reads. However, it is allowed for entire cache to be programmed as RO or Rest. If a cycle is received by L3 when the allocation is 0 for its section, it will convert it into an un-cacheable. cycle. The register programming section details all the requirements from the hardware angle. Also, though the underlying hardware supports varied programming options, only the following configurations are recommended & validated.

L3 Allocation programming (KBytes per bank)									
Config	**URB**	**Rest (DC+RO)**	**DC**	**RO (I/S/C/T)**	**Z**	**Color**	**Unified Tile Cache**	**Command Buffer**	**Sum**
0 (def)	128	128	0	0	0	0	0	0	256
1	128	80	0	0	0	0	96	16	320
2	96	0	32	80	48	48	0	16	320
3	64	0	0	112	64	64	0	16	320
4	64	0	0	48	0	0	192	16	320
5	64	256	0	0	0	0	0	0	320
6	64	128	0	0	0	0	128	0	320
7	64	112	0	0	0	0	128	16	320
8	128	192	0	0	0	0	0	0	320

Programming Notes:

- *Config#0 was kept to keep compatible driver mode*
- *Config#7 is recommended when PTBR is enabled*
- *Config#6 is recommended when Caching of C/Z is enabled but PTBR is NOT enabled*
- *Config#5 is recommended when neither Caching of C/Z nor PTBR is enabled*
- Additionally, if the command buffer section is not allocated, then the state cache redirection to the command buffer section (through bit[10] of register 0x731C) is not allowed.

Note : The number of L3 Banks will vary for different products and SKUs. The number of banks supported for each product is defined in the Configurations section. The total amount of L3$ and URB supported by a product can be calculated by multiplying the number of banks by the values in the above tables.

L3 Cache Theory of Operation

Following are the L3/URB clients:

L3 Cache Clients	RW/RO
Data Cluster (i.e., spill/fills, load/stores, Global memory accesses)	RW
Sampler (L2$)	RO
IME (Motion Estimation)	RO
Instruction Cache (I$)	RO
State Arbiter	RO
Constant Cache	RO

URB Clients	RW/RO
Local Thread Dispatcher	RO
SF Backend	RO
Stream Out	RO
Clipper	RO
Geometry shader	RO
Tesselator	RO
VF	RW
Data Cluster	RW

The L3 and the URB are separate address spaces with clients capable of accessing only one or the other with the exception of the data port. L3 access/cacheability is determined via a parameter, part of the surface state or base address programming of L3 clients and is communicated to L3 cache along with the request packet.

L3 Ordering Restrictions

The Super Q will enforce specific ordering requirements on the accesses to the L3/URB, but will still allow out-of-order accesses where possible.

Basic Ordering Requirements

Requirements:

1. The primary purpose of ordering transactions is to ensure coherency and causality for software accesses to memory. For example, if a thread writes to a memory address, it can expect that any subsequent read issued by the same thread to the same address should read back what it wrote. Conversely, if the read is before the write, it shall read back the value prior to the write.
2. Transactions which have hardware constraints must be ordered to avoid conflicts. For example, two FP-Atomic Add operations must be ordered to avoid conflict on a single FP Adder.

L3 Cache Allocation Policy

The L3 cache allocation policy is "Allocate on fill". i.e., a line in the cache storage is allocated only upon receipt of the data from the external memory and not at the time that the cache detects a miss. The "Allocate on Fill" policy eliminates many boundary cases by regulating the entry invalidation at the last phase of the data servicing.

If the data already residing in the allocated entry is not modified, then the incoming FILL will overwrite the location and pipeline moves on. If the allocated entry carries a dirty data, then an eviction is generated, and the dirty data will be moved to the super queue for writing out to memory.

Cache replacement algorithms

The L3 Cache has a selectable algorithm for line replacement. The default algorithm is the 1b LRU scheme.

The pLRU algorithm uses a binary decision tree with (N-1) nodes. Each node is controlled by a single bit. When a way is filled, all the nodes in the tree to reach that way are flipped to point to another way which is essentially as far away as you can get. The new cache-line is guaranteed to be retained for N-2 additional fills to the set. The LRU bits are only updated on Fills.

Programming Note
If pLRU is used, the best performance can be attained via assigning a power-of-2 number of ways to each section. This is to ensure pLRU to distribute the ways w/o hot spotting within that client's group.

An N-way 1b LRU has an N-bit vector for each set. The vector is initialized to all 0's. As each Fill request arrives, the first bit in the vector with a 0 is selected, that way is replaced, and the bit is flipped. Each subsequent fill will search for the first 0 and replace that line and flip its bit. Eventually, when all N bits are 1 and there are no candidates, all bits are cleared, and the first way is selected. Post this, any access to the cacheline that results in a hit will cause the bit to be set. This will deselect that way from getting re-allocated until all the other ways are also accessed.

Memory Object Control State on Cacheability

This 7-bit field is used in various state commands and indirect state objects to define L3/LLC/eDRAM cacheability, memory type, and graphics data type for memory objects.

Note that memory type information from state is used for non-IA compatible paging structures (legacy context). For new context definition where IA compatible (IA32e) paging structures are used, memory typing follows the IOMMU defined structures.

MOCS[6:1] in L3 is used as an index to a set of programmable tables starting with address xB020h. GFX Software can set up the tables as part of the h/w context, and program various index values in surfaces to point to a table that best suits for that particular surface.

L3 Coherency

Coherency is one of the crucial topics within the L3 cache. There are multiple levels of coherency that are checked and ensured via L3. Though the list of domains and flows is dependent on the usage models, the premise is always the same.

The coherency levels:

1. Thread Level Coherency within a Thread Group
2. Thread Group Coherency between multiple domains
3. GPU/IA level coherency

Besides these special domains, some basic producer/consumer models are followed which are listed as:

1. Fixed function as a producer
2. Data Port as a producer

Thread Level Coherency

A given thread group is contained within a sub-slice, where its writes and reads target the L3 for global memory and SLM for shared local memory. Given the shared local memory view is the same for all sub-slice accesses, coherency or data sharing is guaranteed within the thread group. Local syncs are executed up to Data Port boundary and not exposed to L3.

Thread Group Coherency

Thread groups can be distributed to multiple sub-slices that are physically far from each other. The coherency between thread groups can only be maintained for their global memory accesses. There are two implications of this coherency depending on the mode we are operating at:

1. Non IA-Coherent L3 mode: This is the same methodology that was introduced with the addition of GT4 support. The cross-thread group coherency is maintained via Sync Global which is processed by L3 as well as introducing a WT mode to be able to update global memory with latest data. This mechanism is supported, but not expected to be used.
2. IA-Coherent L3 mode: For the new mode of operation, there is no need to have WT behavior. The data in L3 is already visible to all consumers (i.e. other thread groups of GPU or IA cores). Sync Global has no effect given the data in L3 is already globally visible to all consumers.

GPUIA Level Coherency

In the non-coherent L3 cache mode, data sharing between GPU and IA happens via a s/w controlled flow which requires the internal GPU caches to be flushed in order to make the data visible. Similarly, these caches need to be invalidated when IA produces data. This mode of operation will still be available. Within the L3 cache, non-coherent accesses use “virtual” addresses and are tagged in the L3 Tag array as "Virtual addresses."

Coherent memory access is supported where the L3 cache's contents are visible to IA via snoops. This allows certain streams (i.e. data port) to be GO (Globally Observable) for IA once posted to L3, allowing shorter loop for completions and eliminating the need to do an entire pipeline flush over L3 to get it sync'ed to IA coherent domain. Coherency enhances the general programmability of GPU when cooperating with IA. Within the L3, coherent accesses use a full 48-bit physical address and are tagged in the L3 Tag array as "Physical addresses."

Any context can have both coherent and non-coherent L3 entries. Hence L3 has no register bit that says it is running in coherent vs non-coherent mode. In the operation time, both modes will co-exist simultaneously where some lines in the cache are following a coherent protocol (i.e. physical address) and remaining lines are following non-coherent protocol (i.e. virtual address).

Programming Note
GPU-IA coherency support is removed.

Coherency Usage Models

This section is to give some examples of usage models and high-level handling within the L3 cache. This is specific to L3 cache flows and is not meant to represent the coherency usage models at the system level.

Fixed Func Producing (URB)

Fixed function (FF) clients producing data and slice clients consuming that data is a very common usage model for URB based data sharing. In this mode, FF sends writes to URB and shifts to the next task in their pipeline, rest of the pipeline clients including slice clients will read their content from URB.

Coherency is achieved via delivery of URB write completions from the GO (“Globally observable”) point in the L3 cache fabric. Once the fabric consumes the write and schedules it towards the L3 cache bank, a completion is returned back to producer which enables the consumer. When the consumers start, the data is already at the point of GO where the L3 bank superQ will ensure the ordering of the incoming reads with prior writes.

Fixed Func Producing (Push Constants)

Push constants are also similarly processed; in fact, the target is the same: URB. However, the hardware mechanism of fetching the push constants involves two addresses -- that of the virtual memory pointer to the buffer in memory and a URB offset. This causes the GO point to be pushed later in the cycle to ensure that there are no race conditions and is internally handled by the hardware.

There are additional constraints independent of L3 handling of push constants which are defined as part of the Global Arbitration fabric.

EUs Producing via HDC

Data port is the producer for Global and Shared Local memory data types. The shared local memory is specific to a data port and is confined to the sub-slice. Hence the L3 cache is not involved in the SLM coherency.

For the global memory, coherency is maintained via combination of mechanisms.

1. Completion tracking: Each Data Port tracks its writes towards L3 and waits for them to reach to GO. GO message is given by L3

a. For Non-coherent/virtual addressed Write: Once the ingress queue retires the write in the corresponding node to targeted bank.

b. For Coherent/physical addressed Write: Once the ownership is obtained for the write (i.e. read-for-ownership or invalid-to-modified) which means write is GO with respect to IA cores.

The GO information is used within data port to make sure handle releases are gated until the producers updates are globally visible.

2. Thread Level Flush: EU threads have the capability to push globally tagged data from L3 to next level caches. It is supported but the usage is not recommended.

Programming Note
Support for GPU-IA coherent accesses is removed.

Invalidation and Flushes

There are two different sources to initiate flushes and invalidations:

- Command Streamer initiated
- EU/thread initiated

Both cases have two modes of operation

- Non-IA coherent L3
- IA coherent L3

In addition, there are side flows that back up the various flows and they do require invalidation/flush sequences to be executed for their purpose. Their behavior has no impact between two modes of L3 operation.

- Global Invalidation
- Power Management Invalidation

Command Streamer Invalidation Flows

The Command streamer can initiate the following flush/invalidation flows.

1. **Top of the pipe invalidations**: These are direct, asynchronous commands from the command streamer unit to L3 in respective slices.
2. **Pipeline flush**: These commands flow through the pipe and are issued through the data port to the L3 cache in each slice.

Non-IA Coherent Flows

This is the traditional flow where the content of L3 needs to be invalidated or flushed similar to previous flows.

Top of the Pipe Invalidations

For this case, the invalidation causes the cache to drain all FIFOs and pipelines, and the node performs all invalidations. Only after completion in all slices will new transactions be sent to the L3.

The Top of the Pipe invalidation is intended for cases where invalidation and completion need to be coordinated between slices. Therefore, each slice performs invalidation, but will not proceed until all slices have completed the invalidation.

Nodes are responsible to serialize the invalidation and use double buffering for each event.

Pipeline Flush

Pipeline invalidation is much simpler and optimized for performance. In this case there is no need to coordinate between slices and the operation is more like a “sync” point. Transactions arriving after the invalidation request will still be queued in the SQ but will only be processed after the invalidation of all candidate lines in the cache is complete. The invalidation acts like a fence.

Nodes are responsible to serialize the invalidation and use double buffering for each event.

IA-Coherent Flows

In principle IA-coherent flow is same from node perspective, the only difference is within the banks given the data that we are trying to flush or invalidate is already coherent with IA. Such case eliminates the need to push any bank content explicitly out to LLC.

All that is needed from the bank is to make sure LSQC content is posted into the bank where it is IA visible for a bank to return flush/invalidation completion status to the node.

EUThread Flows

We continue to support the capability for EUs to be able to perform flush/invalidation events that are not coordinated between other EUs. HDC will relate the request to corresponding L3 and require the entire content of this buffer to be invalidated or flushed to relative coherency domain.

Similar to pipeline flush and top of the pipe invalidations, each node receiving this event will have to communicate with other nodes and make the corresponding traffic un-cacheable before returning the response back to corresponding HDC, allowing progress. The actual invalidation/flush will be deferred and performed once all nodes agree on what needs to be done.

The IA-coherent vs non-IA coherent treatment is same as command streamer flows and still applicable for EU/Thread Flows for invalidation/flush.

Global Invalidation

As part of a mitigation plan to plug holes, a global invalidation flow is introduced where each L3 will get the request from their SARB (or config agent) and again coordinate between the nodes. Once invalidation/flush is performed, completion will be returned back to config agent to clear the flags allowing s/w to observe the end of the global invalidation. Only lines which are tagged as “global” will be flushed in this manner.

Global invalidation will invalidate/flush every line which has its global bit set regardless of whether the line is tagged as virtual or physical (non-coherent or coherent.)

Programming Note	
Context:	Global Invalidation
Global invalidation is supported, but there is no known usage case. It is strongly recommended that Global Invalidation using the global bit in the cache not be used. To achieve coherence between slices, coherent memory is the better choice.	

Power Management Invalidation

In the IA-coherent mode, a standard pipeline flush does not push the modified lines to outside of GT to allow deferred invalidations. Consequently, the L3 content is cleared and deferred events are completed through an interaction with the power management (PM) subsystem.

The PM will message the L3 config agent to request a flush when needed. The main usage mode is prior to entering RC6. The rest of the treatment in the nodes is the same. Once flush/invalidate event is complete, message(s) will be sent back to PM with the completion status.

L3 Cache Error Protection

L3 cache error protection is covered via ECC (SECDED). All accesses are subject to ECC protection where single bit errors are fixed silently. Double bit errors are reported via a register structure and communicated by an interrupt to GFX driver. L3 cache HW is additionally capable of stalling execution upon a double bit error.

L3 Cache and URB

These are the registers for L3/URB/SLM. Some of the content is identical to previous, but there are important changes to enhance performance.

LBCF Registers

L3 SQC registers 1

LNCF Registers

Tile Cache Control Register