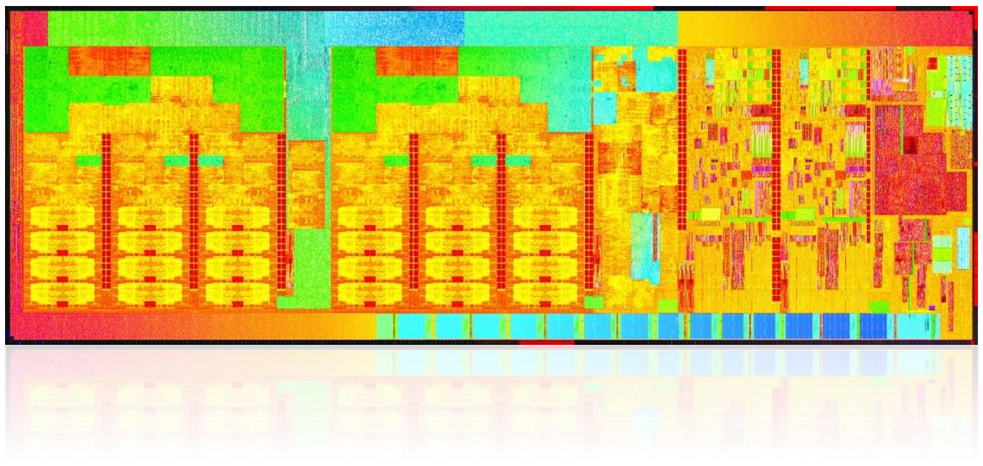
The Compute Architecture of Intel® Processor Graphics Gen8

Version 1.1



External Revision History

Ver	Date	Comment
1.0	9/10/2014	IDF-2014 release of "The Compute Architecture of Intel Processor Graphics Gen8" – Stephen Junkins
1.1	7/15/2015	Updated to include Intel Iris diagrams, speeds/feeds, text references, and other minor edits. – Stephen Junkins

1 CONTENTS

2 AUDIENCE

Software, hardware, and product engineers who seek to understand the architecture of Intel® processor graphics Gen8. More specifically, the architecture characteristics relevant to running compute applications on Intel processor graphics.

This Gen8 whitepaper updates much of the material found in “The Compute Architecture of Intel Processor Graphics Gen7.5” so that it can stand on its own. But where necessary, specific architecture changes for gen8 are noted.

3 INTRODUCTION

Intel’s on-die integrated processor graphics architecture offers outstanding real-time 3D rendering and media performance. However, its underlying compute architecture also offers general purpose compute capabilities that approach teraFLOPS performance. The architecture of Intel processor graphics delivers a full complement of high-throughput floating-point and integer compute capabilities, a layered high bandwidth memory hierarchy, and deep integration with on-die CPUs and other on-die system-on-a-chip (**SoC**) devices. Moreover, it is a modular architecture that achieves scalability for a family of products that range from cellphones to tablets and laptops, to high end desktops and servers.

3.1 WHAT IS INTEL PROCESSOR GRAPHICS?

Intel processor graphics refers to the technology that provides graphics, compute, media, and display capabilities for many of Intel’s SoC products. At Intel, architects colloquially refer to Intel processor graphics architecture as simply “**gen**”, short for generation. A specific generation of the Intel processor graphics architecture may be referred to as “gen6” for generation 6, or “gen7” for generation 7, etc. The branded products Intel HD graphics 4600, Intel Iris™ graphics 5100, and Intel Iris Pro graphics 5200 are all derived from instances of Intel processor graphics gen7.5 architecture. Intel HD graphics 5300, Intel Iris graphics 6100, and Intel Iris Pro graphics 6200 are examples of processor products whose graphics components are based on the Intel processor graphics gen8 architecture. This whitepaper focuses on just the compute architecture aspects of Intel processor graphics gen8. For shorthand, in this paper we use the term **gen8 compute architecture** to refer to just those compute components. This paper also briefly discusses the instantiation of Intel processor graphics gen8 in the Intel Core™ i5 Processor and also in the Intel Core M Processor for low power form factors.

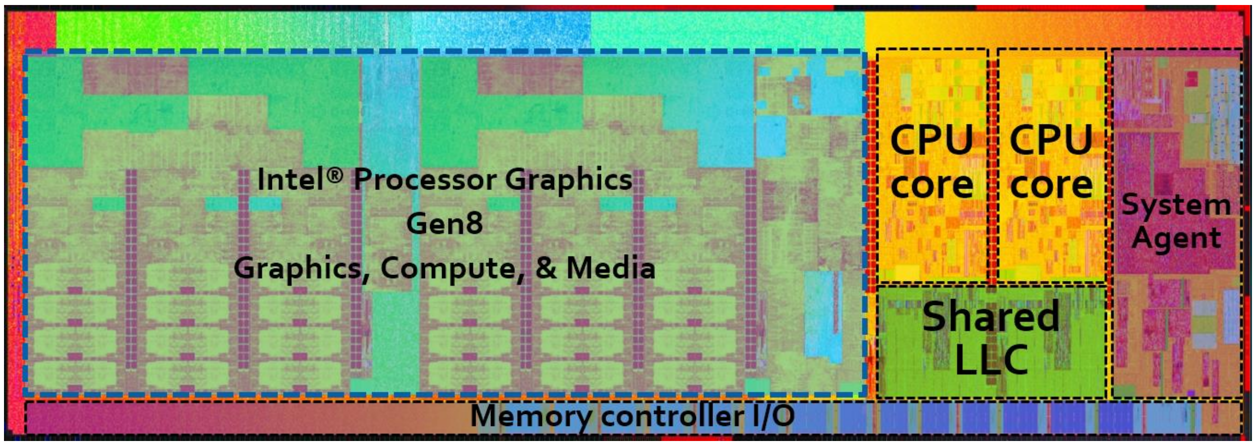


Figure 1: Silicon die architecture layout for an Intel Core™ i5 processor. This SoC contains 2 CPU cores, outlined in orange boxes. Outlined in the blue dashed box, is Intel Iris graphics 6100. It is a two-slice instantiation of Intel processor graphics gen8 architecture.

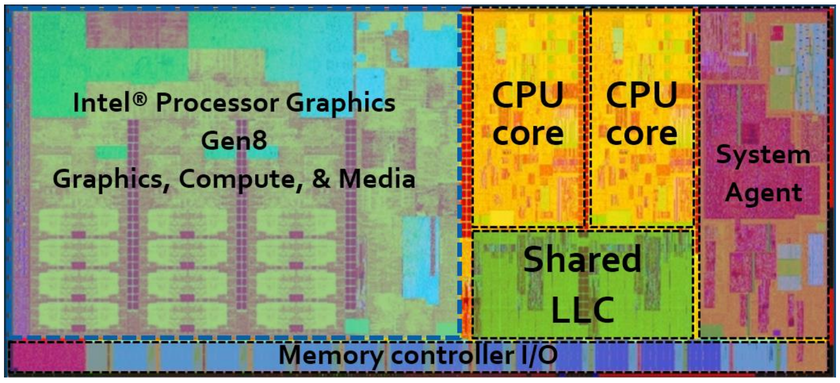


Figure 2: Silicon die architecture layout for a low power Intel Core M processor for tablets and 2-in-1 devices. This SoC contains 2 CPU cores, outlined in orange boxes. Outlined in the blue dashed box, is Intel HD graphics 5300. It is a one-slice instantiation of Intel processor graphics gen8 architecture.

4 SoC Architecture

This section describes the SoC architecture of which Intel processor graphics is a component.

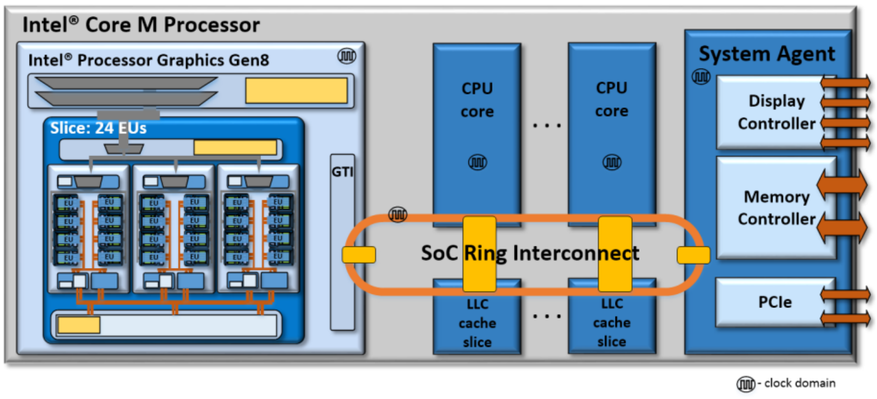


Figure 3: An Intel Core M processor SoC and its ring interconnect architecture.

4.1 SoC Architecture

The Intel Core M processors are complex SoCs integrating multiple CPU Cores, Intel processor graphics, and potentially other fixed functions all on a single shared silicon die. The architecture implements multiple unique clock domains, which have been partitioned as a per-CPU core clock domain, a processor graphics clock domain, and a ring interconnect clock domain. The SoC architecture is designed to be extensible for a range of products, and yet still enable efficient wire routing between components within the SoC.

4.2 Ring Interconnect

The on-die bus between CPU cores, caches, and Intel processor graphics is a ring-based topology with dedicated local interfaces for each connected “agent.” This **SoC ring interconnect** is a bi-directional ring that has a 32-byte wide data bus, with separate lines for request, snoop, and acknowledge. Every on-die CPU core is regarded as a unique agent. Similarly, Intel processor graphics is treated as a unique agent on the interconnect ring. A **system agent** is also connected to the ring, which bundles the DRAM memory management unit, display controller, and other off-chip I/O controllers such as PCI Express. Importantly, all off chip system memory transactions to/from CPU cores and to/from Intel processor graphics are facilitated by this interconnect, thru the system agent, and through the unified DRAM memory controller.

4.3 Shared LLC

Some SoC products include a shared **Last Level Cache** (**LLC**) that is also connected to the ring. In such SoCs, each on-die core is allocated a slice of cache, and that cache slice is connected as a unique agent on the ring. However, all of the slices work together as a single cache, albeit a shared and distributed cache. An address hashing scheme routes data requests to the cache slice assigned for its address. This distributed LLC is also shared with Intel processor graphics. For both CPU cores and for Intel processor graphics, LLC seeks to reduce apparent latency to system DRAM and to provide higher effective bandwidth.

4.4 Optional EDRAM

Some SoC products include embedded DRAM (**EDRAM**), bundled into the SoC’s chip packaging. For example, the Intel processor graphics gen7.5-based Intel Iris Pro 5200 and the Intel processor graphics gen8 based Intel Iris Pro 6200 products bundle a 128 megabyte EDRAM. The EDRAM operates in its own clock domain and can be clocked up to 1.6GHz. The EDRAM has separate buses for read and write, and each are capable of 32 byte/EDRAM-cycle. EDRAM supports many applications including low latency display surface refresh. For both CPU architecture and the compute architecture of Intel processor graphics gen8, EDRAM further supports the memory hierarchy by serving as a large “victim cache” behind LLC. Compute data first populates LLC. Cacheline victims that are evicted from LLC will spill into the EDRAM. If later reads/writes occur to cachelines stored in EDRAM, they are quickly reloaded into LLC, and read/writing then proceeds as usual.

5 The Compute Architecture of Intel Processor Graphics Gen8

5.1 Key Changes in Intel Processor Graphics Gen8

Intel processor graphics gen8 includes many refinements throughout the micro architecture and supporting software. It also includes several major new features and changes over Intel processor graphics gen7.5. To briefly summarize, these changes include:

- Gen8's micro-architecture throughput for 32-bit integer computation has doubled.
- Gen8 has added native 16-bit floating-point support to the execution units.
- For some gen8-based products, the write bandwidth from GTI has doubled.
- Coherent shared virtual memory between CPU cores and Intel processor graphics gen8 has been implemented, enabling seamless sharing of pointer rich data structures.
- For many gen8-based products, 8 execution units are now instantiated per subslice. This can improve compute throughput as data port and sampler are now shared by fewer execution units. (Gen 7.5 was 10 execution units per subslice.)
- For many gen8-based products, 3 subslices are now instantiated per slice. This enables new product configurations, and instantiates more samplers per slice, and more concurrent memory interfaces to L3 and SLM. (Gen 7.5 was 2 subslices per slice.)
- Gen8 has increased the L3 data cache capacity and improved local bandwidth between EUs and L3 data cache.

5.2 Modular Design for Product Scalability

The gen8 compute architecture is designed for scalability across a wide range of target products. The architecture's modularity enables exact product targeting to a particular market segment or product power envelope. The architecture begins with compute components called execution units. Execution units are clustered into groups called subslices. Subslices are further clustered into slices. Together, execution units, subslices, and slices are the modular building blocks that are composed to create many product variants based on Intel processor graphics gen8 compute architecture. Some example variants are shown in Figure 7 and in Figure 8. The following sections describe the architecture components in detail, and show holistically how they may be composed into full products.

5.3 Execution Unit (EUs) Architecture

The foundational building block of gen8 compute architecture is the **execution unit**, commonly abbreviated as **EU**. The architecture of an EU is a combination of simultaneous multi-threading (**SMT**) and fine-grained interleaved multi-threading (**IMT**). These are compute processors that drive multiple issue single instruction multiple data arithmetic logic units (**SIMD**, **ALUs**) pipelined across multiple threads, for high-throughput floating-point and integer compute. The fine-grain threaded nature of the EUs ensures continuous streams of ready to execute instructions, while also enabling latency hiding of longer operations such as memory scatter/gather, sampler requests, or other system communication.

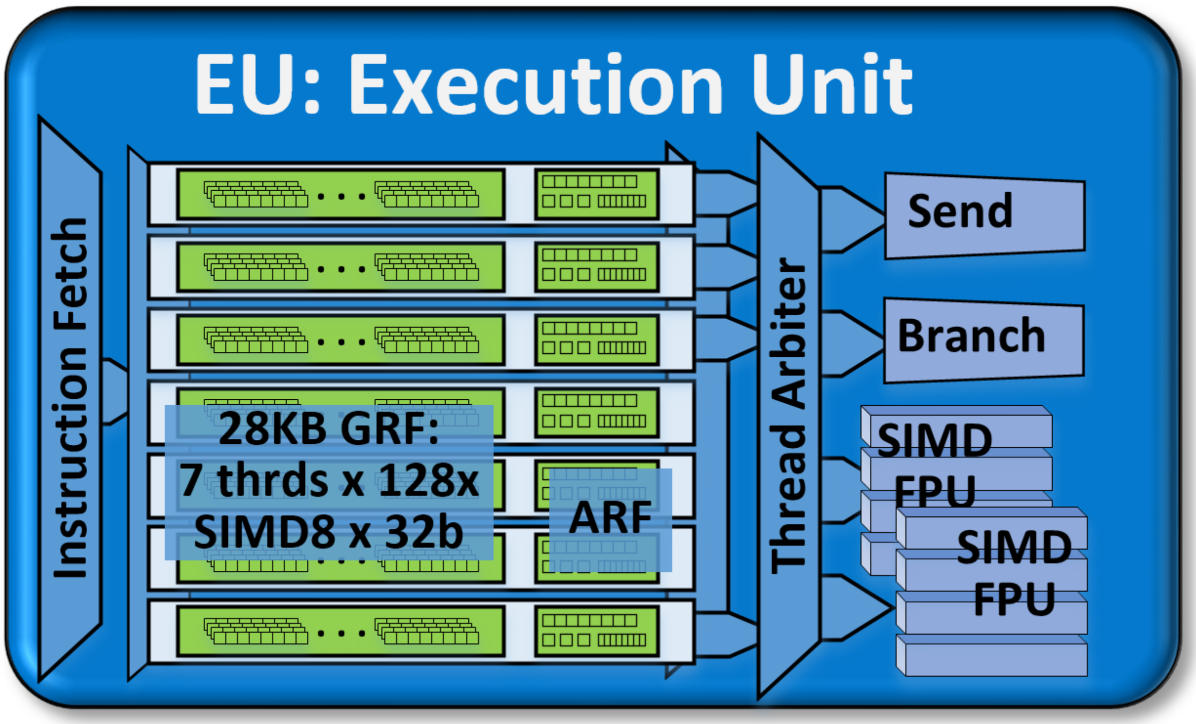


Figure 4: The Execution Unit (EU). Each gen8 EU has seven threads. Each thread has 128 SIMD-8 32-bit registers (GRF) and supporting architecture specific registers (ARF). The EU can co-issue to four instruction processing units including two FPUs, a branch unit, and a message send unit.

Product architects may fine-tune the number of threads and number of registers per EU to match scalability and specific product design requirements. For gen8-based products, each EU thread has 128 general purpose registers. Each register stores 32 bytes, accessible as a SIMD 8-element vector of 32-bit data elements. Thus each gen8 thread has 4 Kbytes of general purpose register file (**GRF**). In the gen8 architecture, each EU has seven threads for a total of 28 Kbytes of GRF per EU. Flexible addressing modes permit registers to be addressed together to build effectively wider registers, or even to represent strided rectangular block data structures. Per-thread architectural state is maintained in a separate dedicated architecture register file (**ARF**).

5.3.1 Simultaneous Multi-Threading and Multiple Issue Execution

Depending on the software workload, the hardware threads within an EU may all be executing the same compute kernel code, or each EU thread could be executing code from a completely different compute kernel. The execution state of each thread, including its own instruction pointers, are held in thread-specific ARF registers.

On every cycle, an EU can co-issue up to four different instructions, which must be sourced from four different threads. The EU's **thread arbiter** dispatches these instructions to one of four functional units for execution. Although the issue slots for the functional units pose some instruction co-issue constraints, the four instructions will be independent, since they are dispatched from four different threads. It is theoretically possible for just two non-stalling threads to fully saturate the floating-point compute throughput of the machine. More typically all seven threads are loaded to deliver more ready-to-run instructions from which the thread arbiter may choose, and promote the EU's instruction level parallelism.

5.3.2 SIMD FPUs

In each EU, the primary computation units are a pair of SIMD floating-point units (**FPUs**). Although called FPUs, they support both floating-point and integer computation. These units can SIMD execute up to four 32-bit floating-point (or integer) operations, or SIMD execute up to eight 16-bit integer or 16-bit floating-point operations. The 16-bit float (half-float) support is new for gen8 compute architecture. Each SIMD FPU can complete simultaneous add and multiply (MAD) floating-point instructions every cycle. Thus each EU is capable of 16 32-bit floating-point operations per cycle: (add + mul) x 2 FPUs x SIMD-4. Also new for gen8, *both* FPUs now support native 32-bit integer operations. Compared to gen7.5, gen8 effectively doubles integer computation throughput within each EU. Finally, one of the FPUs provides extended math capability to support high-throughput transcendental math functions and double precision 64-bit floating-point.

In each EU, gen8 compute architecture offers significant local bandwidth between GRF registers and the FPUs. For example, MAD instructions with three source operands and one destination operand are capable of driving 96 bytes/cycle read bandwidth, and 32 bytes/cycle write bandwidth locally within every EU. Aggregated across the whole architecture, this bandwidth can scale linearly with the number of EUs. For gen8 products with multiple slices of EUs and higher clock rates, the aggregated theoretical peak bandwidth that is local between FPUs and GRF can approach multiple terabytes of read bandwidth.

5.3.3 Branch and Send Units

Within the EUs, branch instructions are dispatched to a dedicated **branch unit** to facilitate SIMD divergence and eventual convergence. Finally, memory operations, sampler operations, and other longer-latency system communications are all dispatched via "send" instructions that are executed by the message passing **send unit**.

5.3.4 EU ISA and Flexible Width SIMD

The EU Instruction Set Architecture (**ISA**) and associated general purpose register file are all designed to support a flexible SIMD width. Thus for 32-bit data types, the gen8 FPUs can be viewed as *physically* 4-wide. But the FPUs may be targeted with SIMD instructions and registers that are *logically* 1-wide, 2-wide, 4-wide, 8-wide, 16-wide, or 32-wide.

For example, a single operand to a SIMD-16 wide instruction pairs two adjacent SIMD-8 wide registers, logically addressing the pair as a single SIMD-16 wide register containing a contiguous 64 bytes. This logically SIMD-16 wide instruction is transparently broken down by the microarchitecture into physically SIMD-4 wide FPU operations, which are iteratively executed. From the viewpoint of a single thread, wider SIMD instructions do take more cycles to complete execution. But because the EUs and EU functional units are fully pipelined across multiple threads, SIMD-8, SIMD-16, and SIMD-32 instructions are all capable of maximizing compute throughput in a fully loaded system.

The instruction SIMD width choice is left to the compiler or low level programmer. Differing SIMD width instructions can be issued back to back with no performance penalty. This flexible design allows compiler heuristics and programmers to choose specific SIMD widths that precisely optimize the register allocation footprint for individual programs, balanced against the amount of work assigned to each thread.

5.3.5 SIMD Code Generation for SPMD Programming Models

Compilers for single program multiple data (**SPMD**) programming models such as Renderscript, OpenCL™[1], Microsoft DirectX* Compute Shader, OpenGL* Compute, and C++AMP, generate SIMD code to map multiple **kernel instances**[2] to be executed simultaneously within a given hardware thread. The exact number of kernel instances per thread is a heuristic driven compiler choice. We refer to this compiler choice as the dominant **SIMD-width** of the kernel. In OpenCL and DirectX Compute Shader, SIMD-8, SIMD-16, and SIMD-32 are the most common SIMD-width targets.

On gen8 compute architecture, most SPMD programming models employ this style of code generation and EU processor execution. Effectively, each SPMD kernel instance appears to execute serially and independently within its own SIMD lane. In actuality, each thread executes a SIMD-width number of kernel instances concurrently. Thus for a SIMD-16 compile of a compute kernel, it is possible for SIMD-16 x 7 threads = 112 kernel instances to be executing concurrently on a single EU. Similarly, for a SIMD-32 compile of a compute kernel, 32 x 7 threads = 224 kernel instances could be executing concurrently on a single EU.

For a given SIMD-width, if all kernel instances within a thread are executing the same instruction, then the SIMD lanes can be maximally utilized. If one or more of the kernel instances chooses a divergent branch, then the thread will execute the two paths of the branch separately in serial. The EUs branch unit keeps track of such branch divergence and branch nesting. The branch unit also generates a “live-ness” mask to indicate which kernel instances within the current SIMD-width need to execute (or not execute) the branch.

5.4 Subslice Architecture

In gen8 compute architecture, arrays of EUs are instantiated in a group called a **subslice**. For scalability, graphics product architects can choose the number of EUs per subslice. For most gen8-based products, each subslice contains 8 EUs. Each subslice contains its own **local thread dispatcher** unit and its own supporting instruction caches. Given these 8 EUs with 7 threads each, a single subslice has dedicated hardware resources and register files for a total of 56 simultaneous threads. Each subslice also includes a sampler unit and a data port memory management unit. Compared to the gen7.5 design which had 10 EUs per subslice, this gen8 design reduces the number EUs sharing each subslice’s sampler and data port. From the viewpoint of each EU, this has the effect of improving effective bandwidth local to the subslice.

[1] *OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.*
[2] We use the generic term *kernel instance* as equivalent to OpenCL *work-item*, or DirectX Compute Shader *thread*.

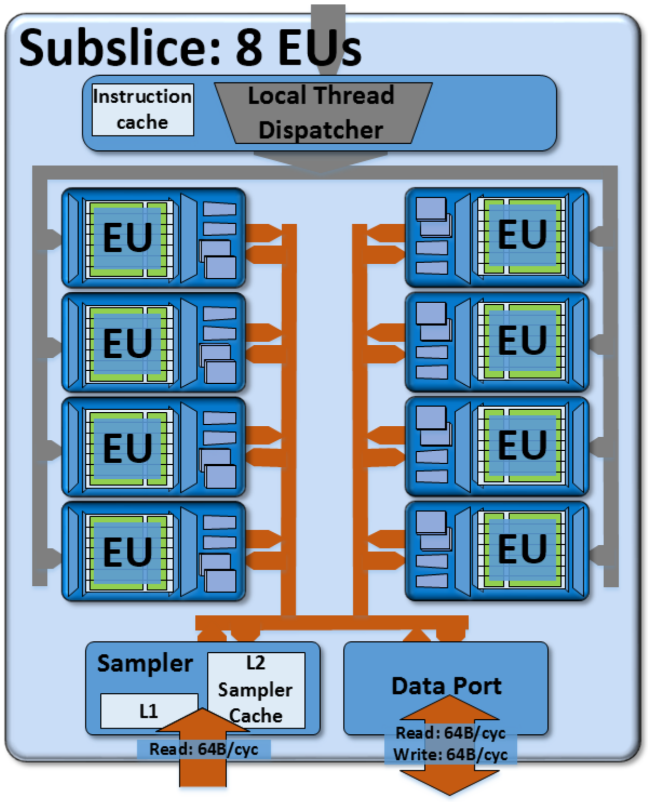


Figure 5: The Intel processor graphics gen8 subslice, containing 8 EUs each. The subslice also instantiates sampler and data port units per subslice.

5.4.1 Sampler

The **sampler** is a read-only memory fetch unit that may be used for sampling of tiled (or not tiled) texture and image surfaces. The sampler includes a level-1 sampler cache (**L1**) and a level-2 sampler cache (**L2**). Between the two caches is dedicated logic to support dynamic decompression of block compression texture formats such as DirectX BC1-BC7, DXT, and OpenGL compressed texture formats. The sampler also includes fixed function logic that enables address conversion on image (u,v) coordinates, and address clamping modes such as mirror, wrap, border, and clamp. Finally, the sampler supports a variety of sampling filtering modes such as point, bilinear, tri-linear, and anisotropic.

5.4.2 Data Port

Each subslice also contains a memory load/store unit called the **data port**. The data port supports efficient read/write operations for a variety of general purpose buffer accesses, flexible SIMD scatter/gather operations, as well as shared local memory access. To maximize memory bandwidth, the unit dynamically coalesces scattered memory operations into fewer operations over non-duplicated 64-byte cache line requests. For example, a SIMD-16 gather operation against 16 unique offset addresses for 16 32-bit floating-point values, might be coalesced to a single 64-byte read operation if all the addresses fall within a single cacheline.

5.5 Slice Architecture

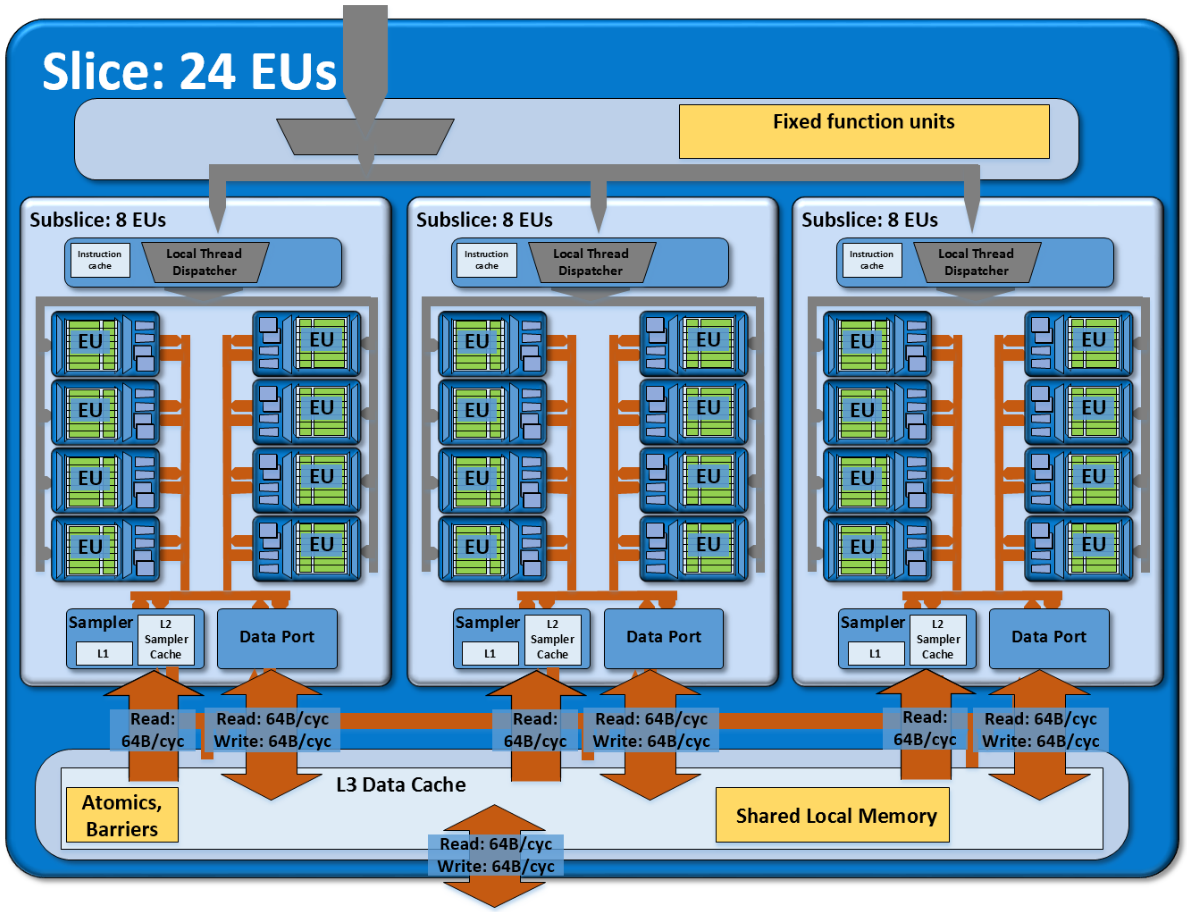


Figure 6: The Intel processor graphics gen8 slice, containing three subslices for a total of 24 EUs. The slice adds supporting L3 cache, shared local memory, atomics, barriers, and other fixed function units.

Subslices are clustered into **slices**. For most gen8-based products, 3 subslices are aggregated into 1 slice. Thus, a single slice aggregates a total of 24 EUs[3]. Aside from grouping subslices, the slice integrates additional logic for thread dispatch routing, a banked level-3 cache, a smaller but highly banked shared local memory structure, and fixed function logic for atomics and barriers. Additional fixed function units support the media and graphics capability, but are not discussed here.

5.5.1 Level-3 Data Cache

For gen8-based products, the level-3 (**L3**) data cache capacity has been increased from gen7.5 to a range of 384 Kbytes to 576Kbytes total per slice. Each application context has flexibility as to how much of the L3 memory structure is allocated: 1) as application data cache, 2) as system buffers for fixed-function pipelines, and 3) as shared local memory. For example, 3D rendering contexts often allocate more L3 as system buffers to support their fixed-function pipelines, instead of as shared local memory. For compute application contexts on gen8 compute architecture, the typical allocation is 384 Kbytes per slice as application data cache. (For gen7.5 compute architecture, the amount was 256 Kbytes per slice.)

[3] Note some gen8-based products may enable fewer than 24 EUs in a slice.

As with previous generations, gen8 products with multiple subslices will instantiate multiple L3 cache partitions. These cache partitions aggregate together and act as a single larger capacity monolithic cache. Cachelines are 64 bytes and are uniformly distributed across the entire aggregate cache. Every sampler and every data port are given their own separate memory interface to the L3. The interface between each data port and the L3 data cache enables both read and write of 64 bytes per cycle. Thus a slice containing three subslices, each with a unique data port, will have an aggregate bandwidth of 192 bytes per cycle. For accesses that miss the L3 cache, the L3 fill logic can read and write system memory data at 64 bytes per cycle.

All data in and out of the samplers and data ports flows through the L3 data cache in units of 64-byte wide cachelines. This includes read and write actions on general purpose buffers. It also includes sampler read transactions that miss the level-1 (L1) and level-2 (L2) sampler caches. L3 cache bandwidth efficiency is highest for read/write accesses that are cacheline aligned and adjacent within a cacheline. Compute kernel instructions that miss the subslice instruction caches flow through the L3 cache.

5.5.2 Shared Local Memory

Shared local memory[4] is a structure within the L3 complex that supports programmer managed data for sharing among EU hardware threads within the same subslice. The read/write bus interface between each subslice and shared local memory is again 64-bytes wide. Latency wise, access to shared local memory is similar to accessing the L3 data cache. However, the shared local memory itself is more highly banked than the L3 data cache. The shared local memory banking can yield full shared local memory bandwidth for access patterns that may not be 64-byte aligned or that may not be contiguously adjacent in memory. For gen8-based products, 64 Kbytes of shared local memory are dedicated and available per subslice. Note that shared local memory is not coherent with other memory structures.

SPMD programming model constructs such as OpenCL's *local* memory space or DirectX Compute Shader's *shared* memory space are shared across a single work-group (thread-group). For software kernel instances that use shared local memory, driver runtimes typically map all instances within a given OpenCL work-group (or a DirectX11 threadgroup) to EU threads within a single subslice. Thus all kernel instances within a work-group will share access to the same 64 Kbyte shared local memory partition. Because of this property, an application's accesses to shared local memory should scale with the number of subslices.

At first glance, it may seem like gen8 instantiates an equivalent share local memory capacity per subslice as did gen7.5: 64 Kbytes per subslice. However, recall that gen8 has 8 EUs per subslice, whereas gen7.5 had 10. The gen8 EUs/subslice ratio means that fewer EUs will be simultaneously sharing the 64 Kbytes, resulting in modestly improved local bandwidth between EUs and their subslice's shared local memory.

5.5.3 Barriers and Atomics

Each slice within gen8 compute architecture bundles dedicated logic to support implementation of barriers across groups of threads. This barrier logic is available as a hardware alternative to

[4] We use the term *shared local memory* to indicate the hardware memory structure that supports the software address space that OpenCL refers to as *work-group local memory,* and that DirectX Compute Shader refers to as *thread-group shared memory*.

pure compiler-based barrier implementation approaches. The gen8 logic can support barriers simultaneously in up to 16 active thread-groups per subslice.

Each slice also provides a rich suite of atomic read-modify-write memory operations. These operations support both operations to L3 cached global memory or to shared local memory. gen8-based products support 32-bit atomic operations.

5.5.4 64-Byte Data Width

A foundational element of gen8 compute architecture is the **64-byte data width**. Recall that the EU register file is composed of 128 32-byte registers (SIMD-8 x 32-bit). But recall also that operands to SIMD-16 instructions typically pair two such registers, treating the pair as a single 64-byte SIMD-16 register. Observe:

- A SIMD-16 instruction can source 64-byte wide operands from 64-byte wide registers.
- The data for such 64-byte wide registers are read and written from L3 over a 64-byte wide data bus.
- Within the L3 data cache, each cacheline is again 64-bytes wide.
- Finally the L3 cache's bus interface to the SoC shared LLC is also 64-bytes wide.

5.6 Product Architecture

Finally, SoC product architects can create product families or a specific product within a family by instantiating a single slice, or groups of slices. Members of a product family might differ primarily in the number of slices. These slices are combined with additional front-end logic to manage command submission, as well as fixed function logic to support 3D, rendering, and media pipelines. Additionally the entire gen8 compute architecture interfaces to the rest of the SoC components via a dedicated unit called the graphics technology interface (GTI).

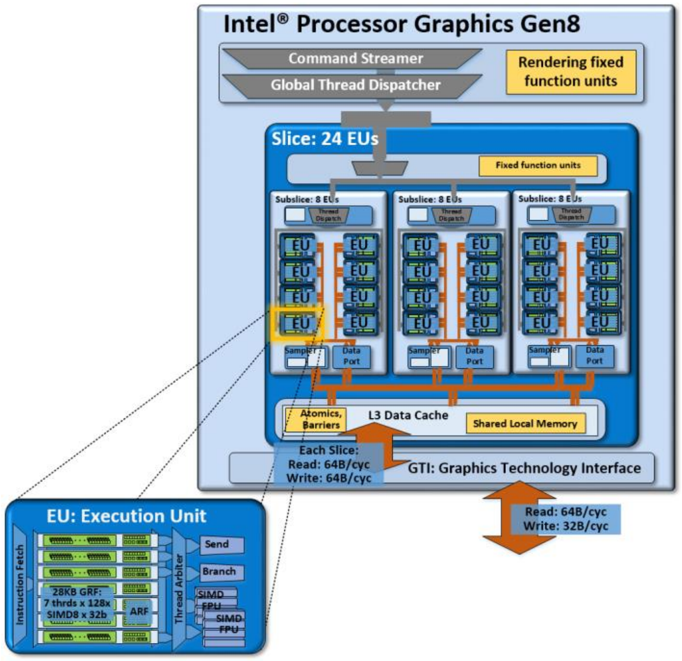


Figure 7: A potential product design that instantiates the compute architecture of Intel processor graphics gen8. It is composed of a single slice with three subslices, for a total of 24 EUs. The Intel Core M processor with Intel HD graphics 5300 instantiates such a design.

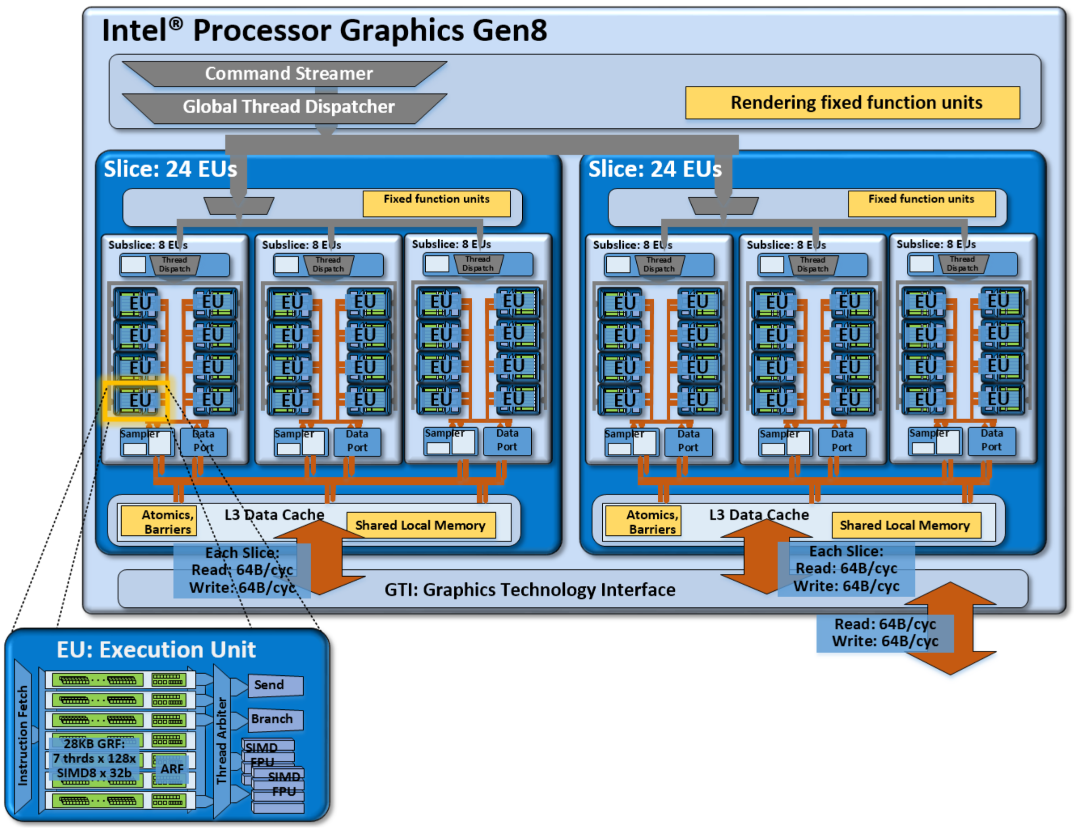


Figure 8: Another potential product design that instantiates the compute architecture of Intel processor graphics gen8. This design is composed of two slices, with three subslices each, for a total of 48 EUs. The Intel Core i5 processor with Intel Iris graphics 6100 instantiates such a design.

5.6.1 Command Streamer and Global Thread Dispatcher

As its name implies, the **command streamer** efficiently parses command streams submitted from driver stacks and routes individual commands to their representative units. For compute workloads, the **global thread dispatcher** is responsible for load balancing thread distribution across the entire device. The global thread dispatcher works in concert with local thread dispatchers in each subslice.

The global thread dispatcher operates in two modes. For compute workloads that *do not depend* on hardware barriers nor on shared local memory, the global thread dispatcher may choose to distribute the workload over all available subslices to maximize throughput and utilization. Given the unit's global visibility, it is able to load balance across all the execution resources. For compute workloads that *do depend* upon hardware barriers or shared local memory, the global thread dispatcher will assign thread-group sized portions of the workload to specific subslices. Such an assignment ensures localized access to the barrier logic and shared local memory storage dedicated to each subslice.

5.6.2 Graphics Technology Interface (GTI)

The graphics technology interface, or simply **GTI**, is the gateway between gen8 compute architecture with the rest of the SoC. The rest of the SoC includes memory hierarchy elements such as the shared LLC memory, the system DRAM, and possibly embedded DRAM. GTI facilitates communication with the CPU cores, and possibly with other fixed function devices such as camera imaging pipelines. GTI also implements global memory atomics that may be shared between Intel processor graphics gen8 and CPU cores. Finally GTI implements power management controls for Intel processor graphics gen8 and interfaces between the GTI clock domain and the (usually different) SoC clock domains.

The bus between each slice that is interfaced to GTI is capable of 64-bytes per cycle read and 64-bytes per cycle write. Internally, GTI has memory request queues to maintain memory order and manage differing latencies on individual memory requests. For instances of Intel processor graphics gen8, the bus between GTI and LLC has two possible configurations. Like gen7.5, one gen8 configuration is capable of 64-bytes per cycle read and 32-bytes per cycle write. A second configuration (new for gen8) is capable of 64-bytes per cycle read and 64-bytes per cycle write. Like other aspects of the architecture, this bus width is also scalable, and SoC designers may configure it for specific products.

5.7 MEMORY

5.7.1 Unified Memory Architecture

Intel processor graphics architecture has long pioneered sharing DRAM physical memory with the CPU. This unified memory architecture offers a number of system design, power efficiency, and programmability advantages over PCI Express hosted discrete memory systems.

The obvious advantage is that **shared physical memory** enables **zero copy** buffer transfers between CPUs and gen8 compute architecture. By zero copy, we mean that no buffer copy is necessary since the physical memory is shared. Moreover, the architecture further augments the performance of such memory sharing with a shared LLC cache. This architecture benefits performance, conserves memory footprint, and indirectly conserves system power not spent needlessly copying data. Shared physical memory and zero copy buffer transfers are programmable through the buffer allocation mechanisms in APIs such as OpenCL 1.0+ and DirectX 11.2+.

5.7.2 Shared Memory Coherency

A new feature for gen8 compute architecture is global memory coherency between Intel processor graphics and the CPU cores. SoC products with Intel processor graphics gen8 integrate new hardware components to support the recently updated Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d) specification. This specification extends the Intel VT, which generally addresses virtual machine to physical machine usage models and enables virtual machine monitor implementation. In particular, the recent Intel VT-d specification extensions define new page table entry formats, cache protocols, and hardware snooping mechanisms for shared memory between CPU cores and devices such as Intel processor graphics.

These new mechanisms can be used to maintain memory coherency and consistency for fine-grained sharing throughout the memory hierarchy between CPU cores and devices. Moreover, the same virtual addresses can be shared seamlessly across devices. Such memory sharing is application-programmable through emerging heterogeneous compute APIs such as the shared virtual memory (SVM) features specified in OpenCL 2.0. The net effect is that pointer-rich data-structures can be shared directly between application code running on CPU cores with application code running on Intel processor graphics, without programmer data structure marshalling or cumbersome software translation techniques.

Within Intel processor graphics, the data port unit, the L3 data cache, and GTI have all been upgraded to support a new globally coherent memory type. Reads or writes originating from Intel processor graphics to memory typed as globally coherent drive Intel VT-d specified snooping protocols to ensure data integrity with any CPU core cached versions of that memory. Conversely, any reads or writes that originate from the CPU cores against globally coherent memory, will drive snooping protocols against gen8's L3 cache. As shown in Figure 9, the sampler's L1 and L2 caches as well as the shared local memory structures are not coherent.

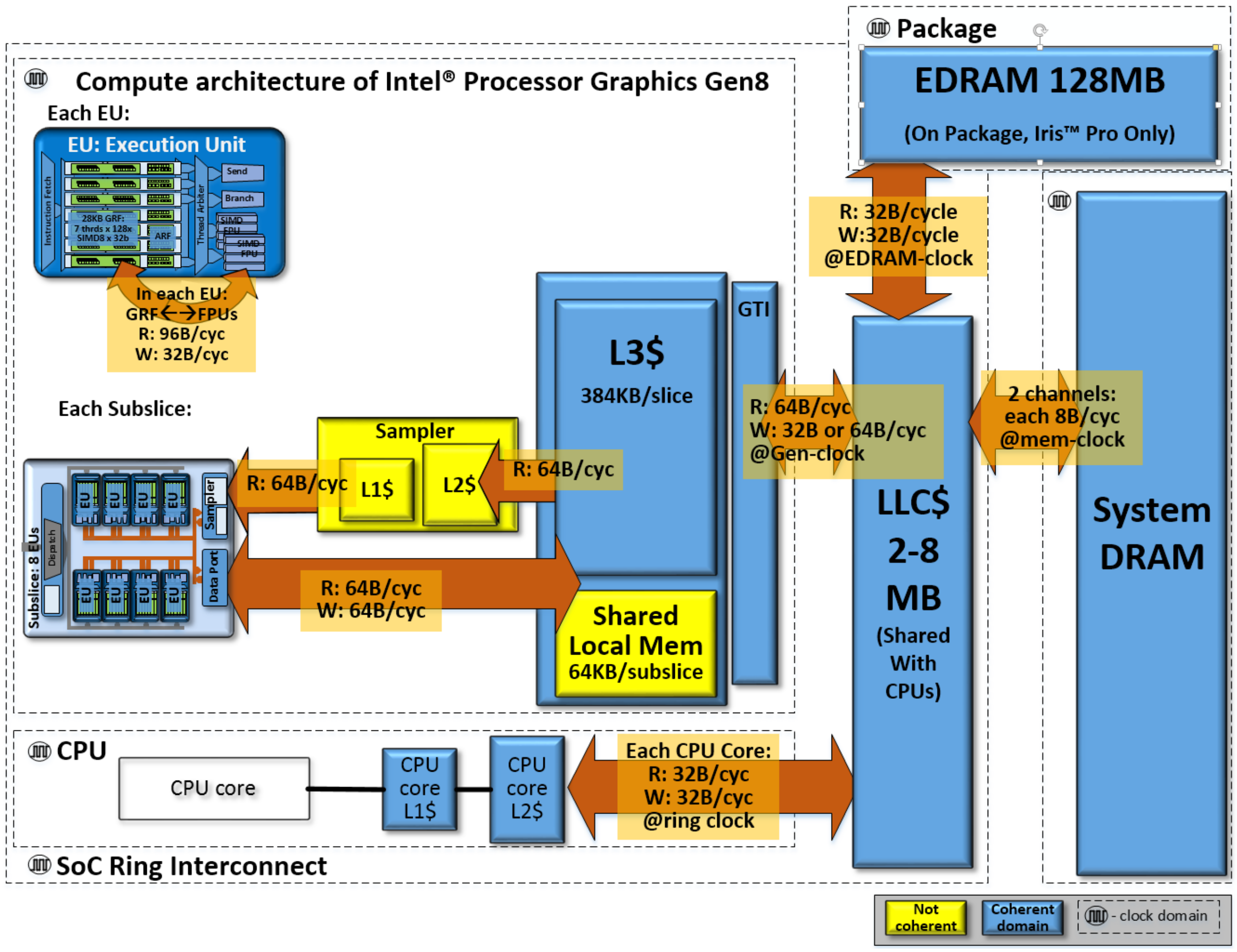


Figure 9: A view of the SoC chip level memory hierarchy and its theoretical peak bandwidths for the compute architecture of Intel processor graphics gen8.

5.8 Architecture Configurations, Speeds, and Feeds

The following table presents the *theoretical peak throughput* of the compute architecture of Intel processor graphics for two example product SKUs. For each product, peak throughput is aggregated across the entire architecture. Theoretical peak throughput for other products based on gen8 architecture follow a similar pattern to the Intel Iris Pro derivations.

	Intel HD Graphics 5300	**Intel Iris Pro graphics 6200**	**Intel Iris Pro Derivation, notes**
Configurations:			
Execution Units (EUs)	24 EUs	48 EUs	8 EUs x 3 subslices x 2 slices
Hardware Threads	168 HW threads	336 HW threads	48 EUs x 7 threads
Concurrent kernel instances (e.g. num OpenCL work-items or DirectX Compute Shader "threads")	5376 kernel instances	10,752 kernel instances	336 HW threads * SIMD-32 kernel compile
Level-3 data cache (L3$) size	384-576 Kbytes	768-1024 Kbytes	2 slices x 384-512 Kbytes /slice
Max Shared Local Memory size (in aggregate)	192 Kbytes	384 Kbytes	2 slices x 3 subslices x 64 Kbytes /subslice
Last Level Cache (LLC$) size	4-8 Mbytes	4-8 Mbytes	Generally, 2MB/CPU core, depending on product configuration
Package embedded DRAM size	n/a	128 Mbytes	
Peak Compute Throughput			
32b float FLOPS	384 FLOP/cycle	768 FLOP/cycle, or 883.0 GFLOPS @1.15GHz.	48 EUs x (2 x SIMD-4 FPU) x (MUL + ADD)
64b double float FLOPS	96 FLOP/cycle	192 FLOP/cycle, or 220.8 GFLOPS @1.15GHz	48 EUs x SIMD-4 FPU x (MUL + ADD) x ½ throughput
32b integer IOPS	192 IOP/cycle	384 IOP/cycle Or 441.6 GIOPS@1.15GHz	48 EUs x (2 x SIMD-4 FPU) x (ADD)

6 Example Compute Applications

The following images provide a few visual examples of the kinds of compute applications and algorithms that have been accelerated on Intel processor graphics.

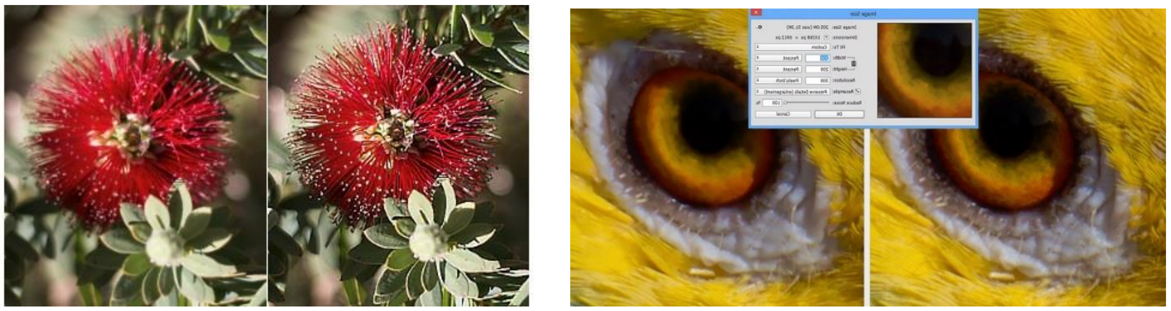

Before After Before After

Figure 10: Intel processor graphics acceleration in Adobe Photoshop, via OpenCL. In the left pair of images, Adobe Photoshop's "Smart Sharpen" feature uses Intel processor graphics to efficiently analyze images to maximize clarity and minimize visual noise and halos. In the right pair of images, Adobe Photoshop's "Intelligent upsampling" feature use Intel processor graphics to accelerate upsampling operations that preserve detail and sharpness without introducing visual noise. Images courtesy of Anita Banerjee and Ilya Albrekht.

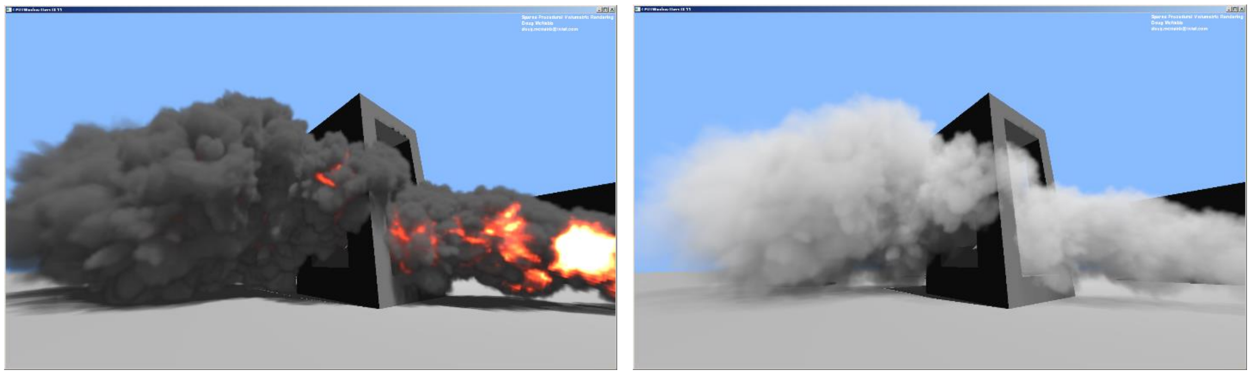


Figure 11: Interactive Real-time volumetric rendered 3D smoke effect implemented using DirectX11 Compute Shader on Intel processor graphics. Image courtesy Doug McNabb.

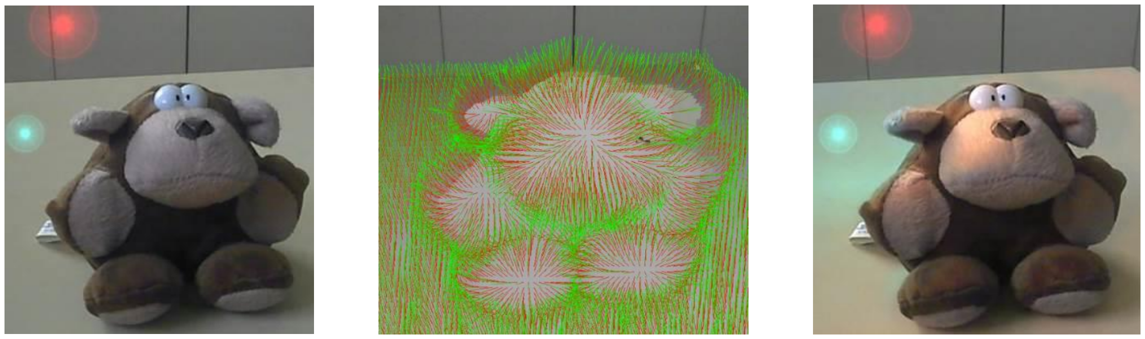

Raw video frame & virtual lights Computed surface normals Video frame with relighting

Figure 12: Interactive dynamic relighting of a real-time video feed. A localized surface polynomial approximation solver are implemented in OpenCL and applied to real-time depth captures. Derived surface normals and lighting computations can then synthetically re-light the scene based on virtual light sources. Images courtesy of Konstantin Rodyushkin. [See an interactive example video.]

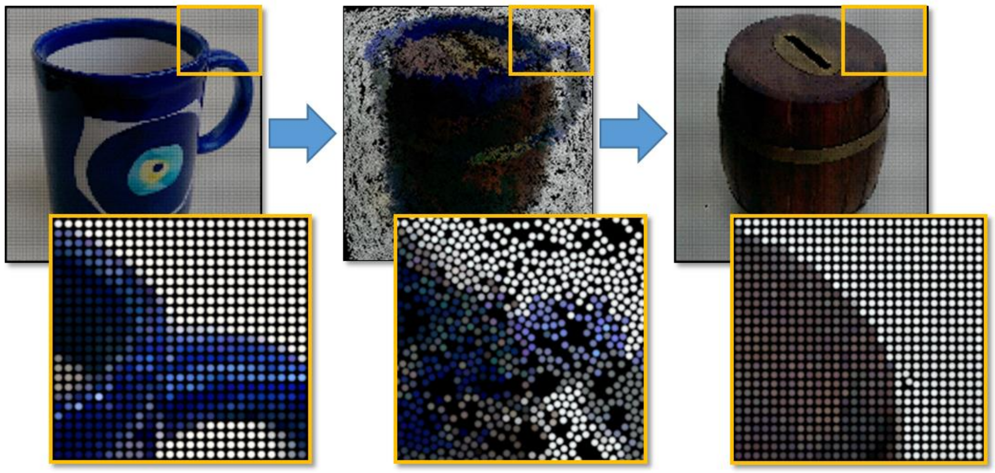


Figure 13: Crowd simulation-based transition effect between two photos (or videos). Particles carry colors of the source image and then change the colors while moving to form the destination image. Dynamic particle collision detection and response are calculated with UNC's RVO2 library ported to OpenCL 2.0 and running on Intel processor graphics. Intel processor graphics and OpenCL 2.0's Shared Virtual Memory enable passing the original pointer-rich data structures of the RVO2 library directly to Intel processor graphics "as is". Neither data structure redesign nor fragile software data marshaling is necessary. Images courtesy of Sergey Lyalin and UNC. More info: *http://gamma.cs.unc.edu/RVO2/.*

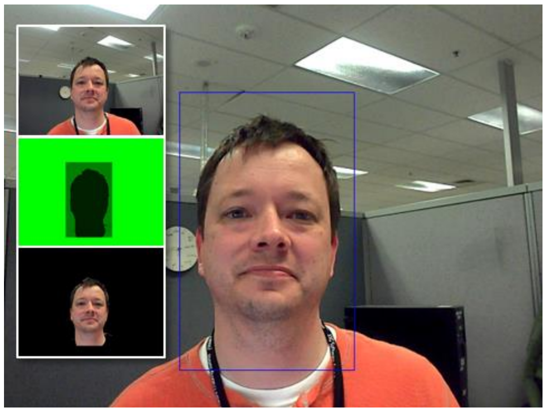


Figure 14: OpenCV Face detection algorithm, accelerated via OpenCL and optimized for Intel processor graphics. Image courtesy Aaron Kunze.

7 Acknowledgements

Intel processor graphics architecture, products, supporting software, and optimized applications are the results of many years and the efforts of many engineers and architects, too many to list here. Also many reviewers contributed to this document. Thank you all. A particular thank-you to Jim Valerio, Murali Sundaresan, David Blythe, and Tom Piazza for their technical leadership and their support in writing this document.

8 More Information

- The Compute Architecture of Intel® Processor Graphics Gen7.5
- Intel® Iris™ Graphics Powers Built-in Beautiful
- About Intel® Processor Graphics Technology
- Open source Linux documentation of Gen Graphics and Compute Architecture
- Intel® OpenCL SDK
- Optimizing Heterogeneous Computing for Intel® Processor Graphics, IDF 2014 Shenzhen
- Intel® 64 and IA-32 Architectures Software Developers Manual
- Intel® Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices
- Intel® Virtualization Technology for Directed I/O - Architecture Specification

9 NOTICES