International Science Group
ISG-KONF.COM

TECHNICAL AND COMPUTER SCIENCES: MODERN CHALLENGES, INNOVATIVE SOLUTIONS AT THE CROSSROADS OF DISCIPLINES AND MULTIDISCIPLINARY ANALYSIS

Collective monograph

ISBN 979-8-90214-595-0
DOI 10.46299/ISG.2026.MONO.TECH.1
BOSTON(USA)-2026

ISBN – 979-8-90214-595-0
DOI – 10.46299/ISG.2026.MONO.TECH.1

Technical and computer sciences: modern challenges, innovative solutions at the crossroads of disciplines and multidisciplinary analysis

Collective monograph

Boston 2026

Library of Congress Cataloging-in-Publication Data

ISBN – 979-8-90214-595-0
DOI – 10.46299/ISG.2026.MONO.TECH.1

Authors – Smirnova O., Votinov M., Ajdid Y., Klimko Y., Koshchii I., Mospan D., Khvalin D., Грабовський О., Кисельова О., Зіангірова Л., Жеребцова Л., Кузнєцова Л., Stoliarova T., Третяк В., Воронін В., Коломійцев О., Кривчун В., Крамар О., Andrushchak I., Rozum R., Bobko O., Romanyuk O., Prunenko D., Dolya O.

REVIEWER

Ivan Katerynchuk – Doctor of Technical Sciences, Professor, Honoured Worker of Education of Ukraine, Laureate of the State Prize of Ukraine in Science and Technology, Professor of the Department of Telecommunication and Information Systems of Bohdan Khmelnytskyi National Academy of the State Border Guard Service of Ukraine.

Kostiantyn Dolia – Doctor of Engineering, Department of automobile and transport infrastructure, National Aerospace University “Kharkiv Aviation Institute”.

Published by Primedia eLaunch
https://primediaelaunch.com/

Collection of scientific articles published is the scientific and practical publication, which contains scientific articles of students, graduate students, Candidates and Doctors of Sciences, research workers and practitioners from Europe and Ukraine. The articles contain the study, reflecting the processes and changes in the structure of modern science.

The recommended citation for this publication is:
Technical and computer sciences: modern challenges, innovative solutions at the crossroads of disciplines and multidisciplinary analysis: collective monograph / Smirnova O., Votinov M., Ajdid Y. – etc. – International Science Group. – Boston : Primedia eLaunch, 2026. 403 p. Available at : DOI – 10.46299/ ISG.2026.MONO.TECH.1

TABLE OF CONTENTS

SECTION 9. SOFTWARE ENGINEERING

DOI: 10.46299/ISG.2026.MONO.TECH.1.9.1

9.1 Analysis of graphics pipeline performance scaling weaknesses

Considering the specifics of rendering three-dimensional images, graphics processors are computing systems whose architecture is oriented towards parallel computing in real time. Due to the increase in the number of cores, the complexity of the graphics pipeline and the use of SIMD architecture [276], the performance of graphics cards has grown much faster than the performance of central processors. The main goal was to involve hardware parallelism [277], which provides greater performance when rendering complex scenes.

However, practical experience with graphics engines and the results of numerous experimental studies indicate that performance increases are not linear. More complex scenes or increased image quality have a lower performance gain than expected given the hardware capabilities. This is evidence of the presence of system limitations of the graphics pipeline that cannot be eliminated by increasing computing power.

Typically, the reasons for poor scaling efficiency are attributed to individual hardware limitations, such as memory bandwidth, the number of final merge blocks, or limitations of the rasterization subsystem. However, this approach analyzes bottlenecks separately, i.e. without considering the interconnection between the stages of the graphics pipeline. As a result, optimizations are aimed at local improvements in one of the stages.

This paper proposes a systematic approach to analyzing the performance of the graphics pipeline. Bottlenecks are considered because of the action of various factors. Special attention is paid to the fragment shader, which in many practical scenarios becomes the main performance limitation. It is shown that the poor parallelization efficiency at this stage is due not so much to the hardware characteristics of the GPU, but to the performance of many redundant calculations.

The purpose of this article is to analyze the weaknesses of scaling the performance of the graphics pipeline, considering the interaction of its stages and the nature of the calculations performed. The paper examines the main sources of performance degradation at different stages of rendering, investigates the impact of logical redundancy on the effectiveness of parallelization, and justifies the feasibility of software approaches to reducing redundant calculations as a key direction for increasing performance.

9.1.1. General architecture of the graphics pipeline

Modern graphical software interfaces implement a step-by-step model of graphic data processing, within which the image formation process is divided into a certain sequence [278]. Each stage performs a specific function and passes the results to the next, forming a graphic pipeline. Shown in Figure 1.

The initial stage is the generation of rendering commands on the CPU side. This step prepares and passes commands from the CPU to the graphics card, including geometry descriptions, pipeline states, resource bindings, and primitive display calls. Although this stage does not perform any direct graphics calculations, it determines the order and structure of further data processing.

The next stage is the vertex stage, which processes individual vertices of primitives. Typical operations of this stage are geometric coordinate transformations, calculation of auxiliary attributes, and data preparation for the formation of graphic primitives. The vertex stage is characterized by a high degree of data processing independence and lends itself well to parallelization.

Next, the formation of graphic primitives and their clipping by the volume of visibility occurs. At this stage, the vertices are grouped into primitives - triangles, while primitives that completely or partially go beyond the visibility area are modified or discarded.

The step of rasterization transforms geometric primitives into a set of fragments that correspond to potentially painted pixels on the screen. This step determines the

coverage of pixels by primitives and interpolates attributes in screen space. Rasterization is a key step that connects the geometric description of a scene with the discrete nature of the image.

Fragment shading performs color and other parameter calculations for each generated fragment. This stage applies lighting models, material properties, texture sampling, and other calculations that directly shape the visual quality of the image.

The fragment shading results then go through depth testing, color blending, and writing to output buffers. These operations determine which fragments make it into the final image, as well as the order and how they are combined with the data already in the frame buffer.

The final stage is the formation of a frame in the frame buffer, which is displayed on the screen or transmitted for further processing. As a result of passing through all these stages, the final image is formed, which is the result of the joint work of the software and hardware components of the graphics pipeline.

The described model is generalized and does not take into account specific optimizations or extensions of specific graphical programming interfaces, but it provides the necessary basis for further analysis of the weaknesses and limitations of scaling the performance of graphics cards.

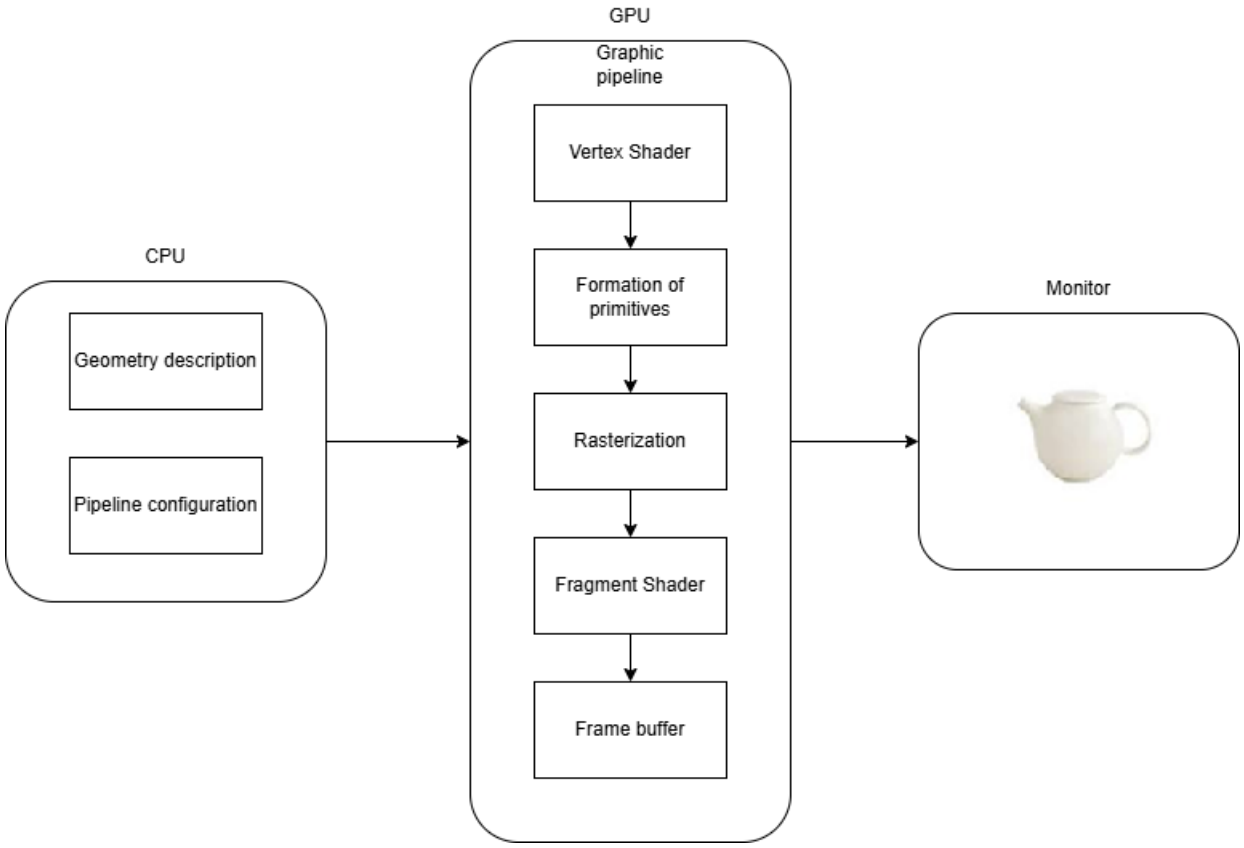

Figure 1. General architecture of the rendering process.

9.1.2. Parallelism in the graphics pipeline

The high performance of modern graphics processors is achieved primarily due to massive parallelism, which is implemented at almost all stages of the graphics pipeline. The main model for performing calculations is the SIMD/SIMT approach, the essence of which is that many threads execute the same instructions operating on different data sets [276]. This model is especially effective for tasks with a regular structure and the absence of dependencies between data elements.

Within the graphics pipeline, parallelism is implemented both at the data processing level and at the level of specialized hardware blocks. Vertex and fragment shaders are implemented as programmable stages that operate on arrays of independent elements – vertices and fragments, respectively. Each such stage is capable of simultaneously processing thousands of elements, which creates the impression of almost unlimited parallelism.

Along with the programmable stages, fixed hardware blocks play a significant role, in particular, rasterization, attribute interpolation, depth testing, and result fusion modules. These components are optimized to perform specific operations and provide high throughput at minimal cost. At the same time, their functionality is rigidly specified and not subject to software control [279].

Despite the availability of significant computing resources, the real efficiency of parallelization depends significantly on the nature of the calculations being performed. Theoretical parallelism, which is determined by the number of available computing blocks and threads, does not always translate into effective parallelism, which manifests itself in the form of a proportional increase in performance. The main factors that reduce the efficiency of parallel execution are uneven load, thread divergence, competition for shared resources, and the need to serialize individual operations.

A feature of the graphics pipeline is that its different stages have different natures of parallelism. For example, the vertex stage is characterized by high regularity of calculations and a minimum number of dependencies, which provides close to linear scaling. On the other hand, the fragment stage, despite the formal independence of fragments, often suffers from logical redundancy of calculations, spatial unevenness and interaction with memory and final merge blocks, which reduces the real efficiency of parallelization.

Thus, the presence of massive hardware parallelism does not guarantee a corresponding increase in performance. To correctly analyze the bottlenecks of the graphics pipeline, it is necessary to distinguish between theoretical parallelism, which is determined by the GPU architecture, and actual parallelism, which is formed by the nature of the algorithms and the organization of calculations at different stages of the pipeline. This distinction is key for further analysis of bottlenecks.

9.1.3. Analysis of bottlenecks at different stages of the graphics pipeline

The performance of a graphics pipeline is determined not by individual processing stages, but by their interaction within the entire pipeline. Even with a high

level of parallelism at each stage, the overall system performance can be significantly limited due to uneven load distribution, serialization of individual operations, or redundant calculations arising from the peculiarities of the rendering organization.

Unlike the localized approach, in which bottlenecks are analyzed separately, for example, only at the fragment shader or memory bandwidth level, in this paper, bottlenecks are considered as a systemic phenomenon that is formed due to the cascading action of several stages of the graphics pipeline. Limitations that arise in early stages can lead to the generation of excess work in later stages, amplifying the overall effect of performance degradation.

The feature of the graphics pipeline is the combination of programmable and fixed hardware stages, each of which has its own parallelism and its own limitations. As a result, the overall performance is determined not by the maximum capabilities of individual components, but by the minimum bandwidth between them. This effect is consistent with the classical provisions of the theory of parallel computing, in particular with Amdahl's law, according to which even a small sequential part of the calculations can significantly limit scaling [280].

The analysis begins with the CPU-side instruction generation stage and includes vertex processing, primitive generation, rasterization, fragment shading, as well as the depth testing, color blending, and frame buffer writing stages. For each stage, the typical causes of reduced parallelization efficiency and their impact on overall system performance are identified.

This approach allows not only to identify individual bottlenecks, but also to show their interdependence, which is a prerequisite for substantiating software optimization methods aimed at reducing logical redundancy of calculations and increasing the efficiency of using hardware parallelism of the graphics card.

9.1.3.1. Bottlenecks at the team formation stage

The first potential bottleneck in the graphics pipeline is the stage of generating and transmitting rendering commands from the CPU to the GPU. Although the main

calculations are performed on the GPU, the CPU is responsible for preparing the graphics commands, describing the pipeline states, and binding the necessary resources. The limitations at this stage are often logical in nature and are not directly related to the computing power of the hardware components.

One of the key factors is the overhead associated with the large number of display calls - commands that come from the CPU to the GPU. Each call requires checking the correctness of the states, preparing the parameters and passing the command to the GPU driver. If there is a scene consisting of many small objects or materials, the number of such calls can reach tens or hundreds of thousands per frame, which leads to a significant increase in the load on the CPU without a proportional increase in the amount of useful work on the GPU. A potential solution to this problem is to optimize the scene, namely, to reduce the number of objects to display or to simplify the objects themselves.

An additional source of limitations is serialization within the graphics driver. Even in the case of a multithreaded application, some of the instruction preparation and state management operations remain serialized. This limits the ability to effectively utilize multicore CPUs and leads to a situation when increasing the number of threads on the application side does not provide the expected performance gain.

The limitations of CPU multithreading are closely related to the architectural features of graphics APIs. Traditional programming interfaces such as OpenGL have historically focused on performing most operations in a single context, which makes it difficult to scale instruction preparation. Even in more modern APIs that support multithreaded instruction generation, much of the state and resource management logic remains centralized.

The number and frequency of pipeline state changes and resource bindings also play an important role. Switching shaders, textures, buffers, or rendering parameters requires additional checks and synchronization, which increases the load on the CPU. Without effective grouping of objects by state, such operations can become the dominant factor in reducing performance.

Thus, the performance degradation at the CPU-to-GPU instruction handoff stage is largely software-based and arises from the way the rendering code is structured, the number of rendering calls, and state management, rather than from hardware performance limitations of the GPU. Eliminating or reducing this bottleneck requires not hardware changes, but optimization of the rendering structure and rational organization of instruction preparation.

9.1.3.2. Top-down stage: possibilities and limitations of parallelization

The vertex stage of the graphics pipeline is one of the most parallelized components of modern graphics cards. The main reason for this is the complete independence of vertex processing: each vertex of a primitive can be processed in parallel, without the need for synchronization with other vertices. This property is ideally consistent with the SIMD and SIMT execution models and allows for efficient use of arrays of GPU cores [276].

Typical vertex stage operations include geometric transformations, auxiliary attribute calculations, and data preparation for subsequent stages. These procedures are regular in nature and do not require access to shared resources with strict order requirements. Due to this, vertex shaders typically exhibit near-linear performance scaling with increasing number of available GPU compute units.

In practical scenarios, the vertex stage rarely becomes the main bottleneck of the graphics pipeline. Even in scenes with a lot of geometry, the increase in computational complexity of the vertex shader is usually compensated by the high GPU bandwidth and efficient caching of vertex data. In addition, vertex reuse via indexed buffers further reduces the load on this stage.

The vertex stage can only cause latency under specific conditions. These include extremely high geometry density, complex deformation algorithms such as multiple skinning or morphing, and inefficient vertex access management that results in frequent cache misses. However, even in these cases, the problem is usually local and can be fixed by optimizing the geometric representation or simplifying vertex calculations.

Thus, the vertex stage, despite its importance in the graphics pipeline, is not the key factor limiting the scaling of GPU performance in most practical applications. Its high level of effective parallelism contrasts with the subsequent stages of the pipeline, where the logical and organizational features of data processing led to much more pronounced delays. It is these stages that are of primary interest for further analysis.

9.1.3.3. Constructing Primitives and Clipping

The stage of primitive formation and their visibility clipping is an intermediate link between vertex processing and rasterization. At this stage, individual vertices are grouped into graphic primitives, most often triangles, after which their belonging to the visibility region is checked. Although this stage is implemented in hardware and is characterized by high throughput, it can be a source of indirect performance limitations [279].

One of the key challenges at this stage is the processing of many small triangles. In modern scenes, especially when using detailed models or tessellation algorithms, the size of individual triangles when projected onto the screen can be commensurate with the size of a pixel or even smaller. Such primitives are often called microprimitives. Despite their small area coverage, each of them undergoes a full processing cycle in the graphics pipeline.

The peculiarity of microprimitives is that they create significant overhead in stages that do not scale proportionally to the coverage area. Primitive generation, boundary condition calculations, and preparation for rasterization are performed regardless of how many pixels will ultimately be painted. As a result, a significant amount of computational work is spent on primitives that make minimal or zero contribution to the final image.

The negative impact of this phenomenon is not so much at the primitive assembly stage itself, but at the subsequent stages of the graphics pipeline. Many small triangles lead to an increase in the number of rasterization calls, fragment generation, and fragment shader launches. Even if an individual primitive covers only one or a few

pixels, it causes additional load on the fragment stage, the depth testing system, and the result merging blocks.

Thus, primitive construction and clipping are sources of cascading performance improvements, in which local redundancy at an early stage leads to a disproportionate increase in computational load at later stages of the pipeline. In this case, the bottleneck is not localized directly at the primitive generation stage but manifests itself in the form of a decrease in the efficiency of parallelization of fragment shading and result recording operations.

Introducing the concept of cascading performance degradation is important for further analysis, as it helps explain why optimizing individual stages, such as the fragment shader, does not always lead to the expected performance increase. Eliminating or mitigating such limitations requires considering the relationship between the stages of the graphics pipeline, rather than an isolated approach to optimization.

9.1.3.4. Rasterization as a generator of redundant work

The rasterization stage performs the transformation of geometric primitives into a set of fragments corresponding to discrete screen elements. At this stage, the coverage of pixels by primitives is determined, and data is also prepared for subsequent fragment shading. Rasterization is implemented by hardware fixed blocks of the GPU and is characterized by high throughput. However, it is at this stage that a significant part of the redundant work is formed, which subsequently becomes a source of performance limitations.

Coverage generation is a geometric operation that determines which pixels or screen samples are covered by each primitive. An important feature of this process is that it is independent of the complexity of the fragment shader. Each triangle, no matter how large or how prominent it is in the final image, triggers the process of generating fragments for all covered pixels.

The consequence of this approach is the phenomenon of overdrawing - multiple drawings of the same pixel with different primitives. In scenes with complex geometry or a significant number of overlaps, the number of generated fragments can exceed the number of pixels in the final image by many times. In this case, hardware rasterization does not distinguish between fragments that will eventually become visible and those that will be discarded at later stages of depth testing or overlapped by other primitives.

An additional factor in the increase in the number of fragments is the use of multi-sample anti-aliasing. In this mode, each pixel is represented by several samples, for which the coverage is determined separately. Although MSAA improves the visual quality of the image, it also leads to a multiple increase in the number of operations associated with the generation of coverage, depth testing and recording of results. Thus, even with the same scene geometry, the overall load on the later stages of the pipeline can increase significantly.

A key feature of rasterization is that the number of generated fragments is determined primarily by the geometric properties of the scene, rather than directly by the quality or complexity of the final image. Two visually similar frames can require radically different amounts of computation at the fragment stage, depending on the number of overlaps, the size of the primitives, and the order in which they are processed. In this sense, rasterization acts as a mechanism that converts local geometric features of the scene into a global increase in computational load.

Thus, rasterization, despite its hardware efficiency, is a key generator of redundant work in the graphics pipeline. It creates a stream of fragments, the number of which can significantly exceed the minimum required to form the final image. It is this stream of redundant fragments that enters the fragment stage, where logical redundancy of calculations becomes the main factor in reducing the efficiency of parallelization.

9.1.3.5. Fragment stage as a concentrator of parallelization constraints

The fragment stage of the graphics pipeline is the stage where the consequences of the limitations formed in the previous stages converge. It is here that the excess number of fragments generated during the rasterization process is transformed into a significant amount of computational load. Despite the high level of hardware parallelism, fragment stage often becomes the main bottleneck when rendering complex scenes [281].

It is important to emphasize that the basic operations required to run the fragment shader are performed in hardware. Attribute interpolation is performed by fixed hardware blocks based on barycentric coordinates calculated during rasterization. Thus, the fragment shader receives ready-made, perspectively correctly interpolated values and does not waste computing resources on geometric transformations or interpolation.

Formally, fragments are independent processing objects, which perfectly correspond to the SIMD and SIMT execution models. Each fragment can be processed in parallel, without the need to synchronize with other fragments, which creates the prerequisites for a high level of parallelization. However, in practice, this formal independence does not guarantee efficient use of hardware resources.

The main reason for the decrease in the efficiency of parallelization at the fragment stage is the logical redundancy of calculations. A significant part of the fragments for which the fragment shader is run either does not enter the final image or has a minimal contribution to its formation. At the same time, a full set of calculations is performed for each such fragment, including texture sampling, lighting calculations, and other operations that are computationally expensive.

An additional factor is the divergence of threads within SIMD and SIMT groups. The presence of conditional transitions depending on material parameters or texture sample results, which in turn leads to different fragments in the same group executing different branches of code. In such conditions, hardware parallelism is reduced, since

execution occurs sequentially for each branch, even if many threads are formally started.

The load on the memory subsystem also has a significant impact on performance. Fragment shaders actively use texture fetches and access buffers, which reduces memory bandwidth. Even with efficient caches, memory access delays can lead to idle computing units, reducing the real level of parallelism. In this case, the bottleneck occurs not due to a lack of computing resources, but due to waiting for data.

Thus, the fragment stage acts as a concentrator of parallelization limitations, in which the consequences of excessive fragment generation, execution divergence, and high memory load accumulate. It is important to note that these limitations are mainly logical in nature and are associated with the way the calculations are organized, and not with the imperfection of the GPU hardware implementation. This makes the fragment stage a key point for applying software optimization methods aimed at reducing logical redundancy and increasing the efficiency of parallel execution.

9.1.3.6. Depth testing, color mixing, and frame buffer recording

After the fragment shader is executed, the processing results go through the stages of depth testing, color mixing, and writing to the frame buffer. These stages, often combined under the general name ROP – Raster Operations [279], play a crucial role in the formation of the final image. Although they are implemented in hardware and optimized for high throughput, it is at this stage that specific limitations arise.

The early depth testing mechanism is usually considered an important optimization that allows fragments to be cut off before the fragment shader is executed. However, the effectiveness of early-z depends significantly on the nature of the fragment code and the order in which primitives are processed. The use of fragment discard operations, modification of depth values, or complex conditional constructs can lead to early testing being disabled. In such cases, fragments undergo a full processing cycle, and depth checking is performed only after the fragment shader has completed, which significantly increases the number of redundant calculations.

Color blending operations are another source of parallelization limitations. Which, unlike most computational steps, requires accessing existing pixel values in the frame buffer and combining them with new results. If multiple fragments simultaneously access the same pixel, it becomes necessary to serialize the write operations to ensure the correctness of the result. Thus, even with many parallel threads, some operations are inevitably performed sequentially.

An additional drawback of hardware is the limited number of ROP blocks in the GPU. The number of such blocks is significantly less than the number of computing cores, which leads to a situation where the overall performance of writing results to memory does not scale proportionally to the increase in computing resources. In scenes with many transparent objects or with intensive use of color blending, it is the ROP blocks that can determine the maximum throughput of the entire pipeline.

The use of multi-sample anti-aliasing MSAA further exacerbates these limitations through write amplification. Each pixel is represented by multiple samples, which require separate depth testing, blending, and writing operations. As a result, the number of memory access operations and the load on ROP blocks increases exponentially, which further reduces the efficiency of parallelization.

Thus, the depth testing, color mixing, and rasterization stages form a bottleneck where hardware limitations combine with logical requirements for correctness of calculations. These limitations cannot be overcome solely by increasing the number of computational units and require software approaches aimed at reducing the number of conflicting writes and redundant fragments that pass to the final stages of the pipeline.

9.1.4. Logical redundancy of calculations as a systemic cause of reduced productivity

Analysis of individual stages of the graphics pipeline shows that most of the limitations in scaling graphics card performance do not arise from a lack of hardware resources or architectural imperfections. Modern graphics processors provide an extremely high level of parallelism in both programmable and fixed stages of the

pipeline. However, in practice, this parallelism is used inefficiently, which is due to the performance of a significant number of logically redundant calculations.

Logical redundancy is the execution of calculations that are formally correct from the point of view of the graphics algorithm, but do not have a proportional impact on the formation of the final image. Such calculations cannot be eliminated by hardware without changing the rendering logic, since from the GPU's point of view they are an integral part of the standard graphics pipeline.

Unlike classic hardware bottlenecks, where performance is limited by bandwidth or the number of functional blocks, logical redundancy manifests itself as a mismatch between the amount of work performed and its utility. It is this mismatch that leads to a situation where an increase in the number of computing resources does not provide a proportional increase in performance.

9.1.4.1. Classification of logically redundant calculations

Within the graphics pipeline, several main types of logical redundancy can be distinguished, each of which affects parallelization efficiency differently.

Spatial redundancy arises from high correlation between neighboring fragments. In most cases, the values of color, lighting, or material parameters change smoothly in screen space, but the calculations are performed independently for each fragment. This leads to multiple repetitions of almost identical operations within SIMD and SIMT groups.

Geometric redundancy is associated with the generation of fragments that do not make it into the final image. Redrawing, overlapping primitives, and microprimitives cause fragment shaders to be run on fragments that are later rejected by the depth test or overlapped by other objects.

Perceptual redundancy occurs when calculations are performed with a precision or frequency that exceeds the ability of human vision to perceive. Minor differences in color or lighting that are not discernible to the user still require a full computational cycle.

Temporal redundancy occurs when calculations are repeated from frame to frame without significant changes in the input data. Even with minor scene dynamics, most of the fragmented calculations are performed anew, which leads to duplication of work in time.

9.1.4.2. The impact of logical redundancy on parallelization efficiency

Logical redundancy directly affects the efficiency of parallelizing computations on GPUs. Although fragment shaders are formally independent and can execute in parallel, the redundant nature of the computations results in hardware parallelism being used to process data with low efficiency.

In such conditions, increasing the number of computational units leads to an increase in the amount of work performed, but not to an increase in useful results. This manifests itself in the form of sublinear scaling of performance and saturation, when further increase in hardware resources does not give a noticeable increase in performance.

In addition, logical redundancy amplifies the negative effects of thread divergence and memory contention. As a result, hardware delays hiding mechanisms lose their effectiveness, and the actual level of parallelism becomes much lower than theoretically possible.

9.1.4.3. Logical redundancy as a system issue

The key feature of logical redundancy is its systemic nature. It is not localized at one stage of the graphics pipeline but is formed gradually due to the interaction of several stages. Redundant geometry at the vertex stage leads to the generation of microprimitives, which, in turn, increase the load on rasterization and fragment shading. The results of this redundant work further increase the load on the depth testing blocks and ROPs.

Thus, bottlenecks observed in the later stages of the pipeline are only a manifestation of a deeper problem – a mismatch between the structure of calculations

and the real information needs of the rendering task. Their elimination requires not point optimizations, but a review of the principles of organizing calculations in the graphics pipeline.

This is what makes software methods for reducing logical redundancy a key area for improving parallelization efficiency, which will be discussed in the next section.

9.1.5. Software approaches to optimizing the graphics pipeline

As shown in the previous sections, the limitations of performance scaling are systemic in nature and are largely due to logical redundancy of calculations. Since these limitations are formed at the level of organization of the rendering process, their elimination or mitigation is possible primarily through software approaches. This section considers the main classes of methods aimed at reducing the amount of redundant work and increasing the efficiency of using hardware parallelism.

9.1.5.1. Methods for reducing the number of fragments

One of the most effective ways to reduce the load on the fragment stage is to reduce the number of fragments for which the shader is run. Such methods include methods that allow you to cut off fragments early in the graphics pipeline.

Using a depth pre-pass allows the depth buffer to be filled before the main rendering, thus ensuring that the early-z engine works efficiently. As a result, fragments that are guaranteed not to make it into the final image are cut off before expensive calculations are performed.

Further reductions in the number of fragments processed can be achieved by ordering primitives by depth, as well as various geometry culling techniques, including frustum culling and occlusion culling. While these approaches do not change the structure of the fragment shader, they significantly reduce the amount of input data for later stages of the pipeline.

9.1.5.2. Methods for reducing the number of fragment shading calls

One of the most effective ways to reduce logical redundancy in computations is to reduce the frequency of fragment shader execution. Unlike methods aimed at cutting off fragments, this approach assumes that not every fragment requires a full computational cycle to achieve acceptable visual quality. Thus, optimization is achieved by adaptively reducing the number of fragment shader runs without significant degradation of the result.

Adaptive shading involves varying the frequency or complexity of fragment calculations based on scene properties or local image characteristics. Adaptation criteria can include color, depth, normal gradients, or temporal motion. In areas with low spatial or temporal variability, the fragment shader can be executed less frequently, and the results of the calculations are reused for neighboring fragments.

This approach directly reduces spatial and perceptual redundancy, as computations are performed only where they have a noticeable impact on the final image. However, adaptive shading requires additional control logic and can complicate the implementation of the rendering algorithm.

Checkerboard rendering is a special case of downsampling of fragmented shading, in which calculations are performed on only a subset of pixels, usually in a checkerboard pattern. The remaining pixels are restored by interpolation or reconstruction based on neighboring values or data from previous frames.

This approach allows you to almost halve the number of fragment shader runs while maintaining the overall image structure. Checkerboard rendering effectively reduces the computational load and pressure on the memory subsystem, but its use requires careful control of artifacts, especially in rapidly changing scenes.

Variable Rate Shading is a hardware-supported mechanism that allows you to explicitly specify the frequency of execution of a fragment shader for different regions of the screen. Unlike purely software approaches, it integrates the idea of reducing the frequency of shading directly into the graphics pipeline, allowing you to run a shader for a group of pixels once. This mechanism is a prime example of how the problem of

excessive fragment calculations is recognized not only at the algorithmic but also at the architectural level. At the same time, even with VRS, the logic of determining areas with a reduced frequency of shading remains a software task and requires an analysis of the perceptual significance of the image.

The methods considered have a common ideological basis: not all fragments are equivalent in terms of information value. Reducing the frequency of fragment shading allows to reduce logical redundancy of calculations and increase the efficiency of parallelization without changing the hardware architecture of the GPU. This makes such approaches one of the key tools for optimizing the performance of modern graphics systems.

9.1.5.3. Using compute shaders

The use of compute shaders opens the possibility of implementing alternative approaches to organizing rendering calculations, in which the structure of parallelism is determined programmatically. Unlike the classic graphics pipeline, where parallelism is fixed and tied to the processing of individual vertices or fragments, the compute stage allows for flexible control of the size and composition of thread groups, as well as the order of data processing.

One common approach is tile-based processing, in which the screen or rendering area is divided into small blocks so called tiles that are processed independently. Within a single tile, information about geometry, lighting, or material properties can be aggregated and calculations can be performed at the pixel block level, rather than individual fragments. This approach significantly reduces spatial redundancy, since data common to the region is calculated once and reused for all pixels in the tile.

An extension of the tile-based approach is clustered methods, in which the space is further divided by depth or other parameters of the scene. This allows for more precise localization of calculations and limits them to only those regions where they are really needed. As a result, the number of redundant operations associated with

processing fragments that have similar properties or minimal contribution to the final image is reduced.

The key advantage of the compute stage is the ability to implement controlled parallelism. The developer can explicitly define the size of workgroups, the structure of memory access, and the order of computations. This allows for reduced thread divergence, improved memory access locality, and more efficient use of GPU caches. In such conditions, parallelism becomes not only massive, but also controllable, which significantly increases its efficiency.

At the same time, transferring calculations to the compute stage is accompanied by an increase in the complexity of the software implementation. The developer assumes responsibility for the correct synchronization of flows, the organization of data exchange and ensuring consistency of results. Therefore, the use of compute approaches is appropriate primarily in cases where the gain from reducing logical redundancy exceeds the overhead of complicating the algorithm.

Thus, the compute stage provides tools for rethinking the traditional fragment shading model and creates conditions for implementing algorithms focused on reducing redundant computations. This confirms the thesis that increasing the efficiency of graphics pipeline parallelization is possible through software control over the computation structure, and not solely through increasing hardware resources.

9.1.6. Experimental confirmation

To investigate the impact of the complexity of the fragment stage of the graphics pipeline on the overall rendering performance, an experimental software was developed based on OpenGL 3.3. The program renders a single full-screen triangle, which allows minimizing the impact of the vertex and geometry stages and focusing on the work of the fragment shader.

Vertical synchronization was disabled to eliminate display refresh rate limitations and ensure correct measurement of graphics execution time. All CPU-side

calculations were performed in a single thread, eliminating the influence of multi-threaded frame preparation on the experimental results.

MSI Afterburner was used to monitor GPU hardware parameters such as load and temperature. During the experiments, the GPU temperature did not exceed 66 °C, which indicates the absence of thermal frequency limitation and ensures stability of measurements. In turn, the GPU load consistently exceeded 80%, which indicates high load and no GPU downtime.

The execution time of graphics commands on the GPU t_gpu was measured using OpenGL timer queries GL_TIME_ELAPSED. The total frame generation time t_frame was determined on the CPU side using high-precision timers, and the frame rate per second was calculated as the reciprocal of the average value of t_frame.

Two variants of fragment shaders were used for comparative analysis. The first shader is computationally simple and performs only the writing of a constant color to the frame buffer, which corresponds to the minimum load on the fragment stage. The second shader is computationally intensive and contains a significant number of operations of sampling from the texture, trigonometric functions and accumulation of results in a loop, which allows for a controllably increasing the load on the fragment stage of the GPU graphics pipeline.

Thus, the proposed experimental setup allows us to isolate the impact of the complexity of the fragment shader on the total frame generation time and to investigate the limitations of parallelization efficiency under conditions of dominance of one stage of the graphics pipeline.

Figure 2 shows a triangle rendered using a simple fragment shader. Table 1 contains the timing metrics and frame rates. The GPU load and temperature are shown in Figure 3.

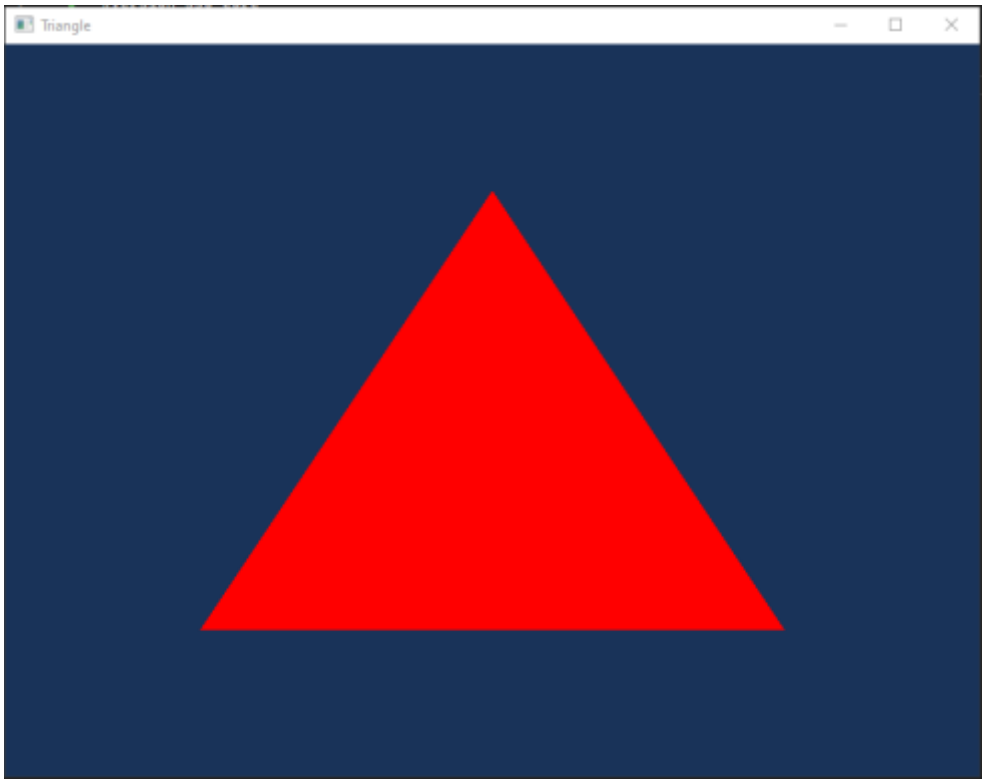

Figure 2. Image of a triangle when using a lightweight shader.

Table 1.

Measurement results when using a lightweight shader

No.	t_gpu, ms	t_frame, ms	FPS
1	0.055264	8.7696	114.03
2	0.055328	8.7183	114,701
3	0.05536	9.7725	102,328
4	0.055296	8.7744	113,968
5	0.055296	8.0474	124,264
6	0.055424	7.2662	137,624
7	0.055296	14.3843	69.5202
8	0.055136	11.3821	87.8572
9	0.055264	7.6877	130,078
10	0.055424	8.9772	111,393
11	0.05536	19.1302	52.2734
12	0.05552	9.2192	108,469
13	0.055584	12.4927	80.0467
14	0.055328	13.9368	71.7525
15	0.055488	8.1731	122,353

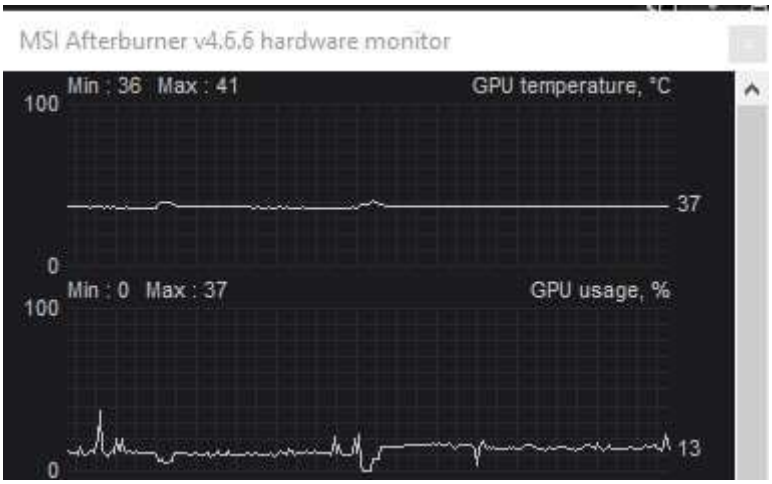

Figure 3. GPU utilization and temperature.

The average execution time of one drawing operation on the GPU is approximately 0.055 ms, the rendering time of one frame is approximately 7.688 ms. At the same time, the number of frames per second is approximately 130. These values indicate that the GPU does almost nothing. The frame is not limited by the GPU, but by overhead: CPU, driver, call to draw, frame replacement, window compositor. That is, GPU ≈ 1% of the frame.

GPU share in frame time:

$$p = \frac{0.055}{7.688} \approx 0.0072\,(0.72\%)$$

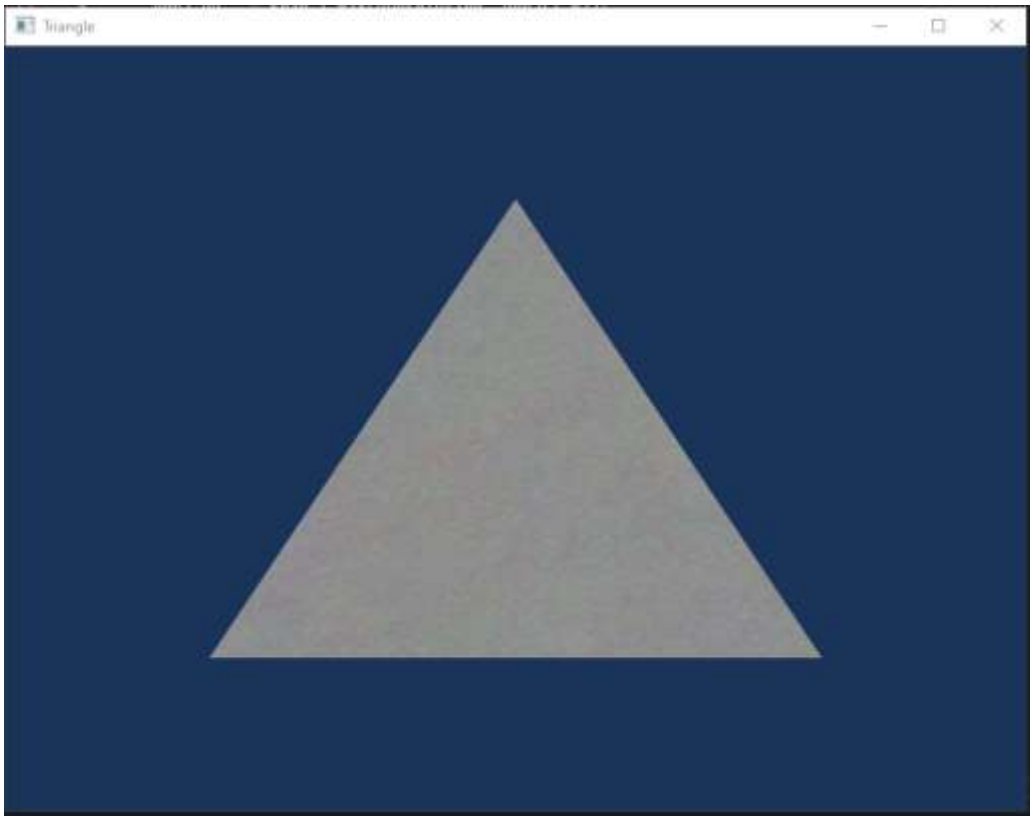

Figure 4. A triangle when using complex shader (white noise. Each pixel is painted many times).

Table 2.

Measurement results when using a heavy shader

No.	t_gpu, ms	t_frame, ms	FPS
1	10.2967	12.1935	82.0109
2	10.2778	11.5432	86.6311
3	10.2874	11.8119	84.6604
4	10.2724	12.4641	80.2304
5	10.2744	11.3832	87.8488
6	10.2702	11.578	86.3707
7	10.2979	12.282	81.42
8	10.2834	11.7582	85,047
9	10.2769	12.4252	80.4816
10	10.2682	12.5718	79.5431
11	10.2766	11.6618	85.7501
12	10.2828	13.2443	75.5042
13	10.2614	11.8154	84.6353
14	10.2649	14.995	66.6889
15	10.2722	12.0007	83.3285

When using a heavy shader, the GPU execution share is on average 10.263 ms, the average frame time is 11.392 ms. At the same time, the approximate number of frames per second is 87.8. Here the fragment stage has become dominant. Approximately 90% of the frame is determined by the GPU.

GPU share:

$$p = \frac{10.263}{11.392} \approx 0.901\,(90.1\%)$$

The difference in t_gpu between shaders:

$$\frac{10.263}{0.055} \approx 186$$

The fragment stage is about 186 times harder, and this is directly visible in the GPU timing.

Let's define the part that is executed outside the GPU.

For a heavy shader this is:

$$t_{other} = t_{frame} - t_{gpu} \approx 11.392 - 10.263 = 1.129 \text{ ms}$$

Even if you make this part perfectly fast, the frame will be:

$$t'_{frame} \approx t_{gpu} = 10.263 \text{ ms}$$

Maximum win:

$$S_{max} \approx \frac{11.392}{10.263} \approx 1.11$$

That is, approximately 11% maximum.

9.1.6.1 Experimental conclusions

Despite the high level of internal parallelization built into the architecture of modern GPUs, experimental results have shown that as the complexity of the fragment shader increases, the corresponding stage of the graphics pipeline becomes dominant in the overall frame generation time. Under such conditions, rendering performance is limited precisely by the fragment stage, and potential optimizations or parallelization at other levels of the system do not lead to a significant reduction in frame time. The results obtained are consistent with the provisions of Amdahl's law and illustrate the fundamental limitations of the effectiveness of parallelization in the graphics pipeline in the mode high GPU load.

Conclusions

The paper conducts a systematic analysis of the weaknesses of graphics pipeline performance scaling and shows that the performance limitations of modern GPUs are not only hardware but also logical in nature.

Unlike the local approach, which focuses on individual stages of the pipeline, the proposed approach considers the graphics pipeline as a holistic system, within which bottlenecks are formed in a cascade due to the interaction of the stages of command generation, geometry processing, rasterization, fragment shading and result recording operations. It is shown that even in the presence of massive hardware parallelism, the scaling efficiency is limited by redundant calculations, thread divergence and competition for memory resources.

Experimental results confirmed that as the complexity of the fragment shader increases, it is the fragment stage that becomes dominant in the frame time structure, and potential off-GPU optimizations have limited effect.

This is consistent with Amdahl's law and demonstrates a fundamental limitation of parallelization under high GPU load. Thus, it is advisable to link the increase in the efficiency of the graphics pipeline not only with the increase in hardware resources, but primarily with software methods for reducing logical redundancy, reducing the number of fragments, adaptive shading, and controlled organization of parallelism. These approaches constitute a promising direction for further research in the field of optimizing the rendering of three-dimensional scenes.

272. United Nations Economic Commission for Europe. (2018). UNECE Regulation No. 51: Uniform provisions concerning the approval of motor vehicles having at least four wheels with regard to their sound emissions. United Nations. https://op.europa.eu/ga/publication-detail/-/publication/09bcd363-67c8-11e8-ab9c-01aa75ed71a1

273. United Nations Economic Commission for Europe. (2017). UNECE Regulation No. 83: Uniform provisions concerning the approval of vehicles with regard to the emission of pollutants according to engine fuel requirements. United Nations. https://eur-lex.europa.eu/eli/reg/2012/83/oj/eng

274. Rozum R.I., Shevchuk O.S., Prohnii P.B. (2022) Optimization of working processes of internal combustion engines with the purpose of improving their environmentality. Modern engineering and innovative technologies. Sergeieva&Co Karlsruhe (Germany) 19. 1, 147-150 DOI: https://doi.org/10.30890/2567-5273.2022-19-01-023

275. Prohnii P.B., Popovych P.V., Shevchuk O.S. et al. (2024) Do analizu vykydiv vid znosu avtomobil′nykh shyn, yak ekolohichnoyi skladovoyi vykorystannya avtomobil′noho transport. Central Ukrainian Scientific Bulletin. Technical sciences, 9(40), part II, 127-136 DOI: https://doi.org/10.32515/2664-262X.2024.9(40).2.127-136

276. Bobko O. L., Romanyuk O. N. Using SIMD, SIMT and VLIW architectures for parallel data processing in video cards // Collection of scientific papers with materials of the VIII International Scientific Conference "Traditional and Innovative Approaches to Scientific Research", Drohobych, January 31, 2025. Vinnytsia: LLC "UKRLOGOS Group, 2025. P. 272-280.

277. Romanyuk O. N., Bobko O. L. Parallelization of rendering at the hardware level // Collection of abstracts of the XII All-Ukrainian Scientific and Practical Conference of Young Scientists “Information Technologies – 2025”, Kyiv, May 15, 2025. Kyiv: Borys Grinchenko Kyiv Metropolitan University, 2025. Pp. 189-191.

278. Romanyuk ON, Bobko OL, Zavalniuk YK, Titova NV, Romanyuk SO, Stakhov OY Analysis of graphics pipelines. Innovation in der modernen Wissenschaft: Innovative Technology, Informatik, Sicherheitssysteme, Verkehrsentwicklung, Architektur und Bauwesen, Physik und Mathematik. Monografische Reihe "Europäische Wissenschaft". 2024. Buch 30. Teil 1. Chap. 4. pp. 71-80.

279. T. Akenine-Möller, E. Haines, N. Hoffman, Real-Time Rendering, 4th ed. Boca Raton, FL, USA: CRC Press, 2018.

280. Romanyuk O. N., Bobko O. L., Snigur A. V. Metric evaluation of rendering parallelism // Scientific works of VNTU. Electronic text data. 2025. Issue 2. URI: https://praci.vntu.edu.ua/index.php/praci/article/view/815 .