Geekbench 6
Benchmark Internals

Introduction **4**
Platform Support **5**
Architecture Support **5**
Compiler Support **6**
CPU Benchmark **7**
Runtime 7
Multi-Threading 7
Instruction Sets 8
x86 Instruction Sets 8
ARM Instruction Sets 9
Scores 10
Productivity Workloads 11
File Compression 12
Navigation 13
HTML5 Browser 14
PDF Render 15
Photo Library 16
Developer workloads 17
Clang 18
Text Processing 19
Asset Compression 20
Machine Learning Workloads 21
Object Detection 22
Background Blur 23
Image Editing 24
Object Remover 25
Horizon Detection 26
Photo Filter 27
HDR 28
Image Synthesis 29
Ray Tracer 30
Structure from Motion 31

CPU Benchmark Characteristics **32**
Instructions Per Cycle 33
Branch Prediction Miss Rate 34
Working Set Size 35
Cache Misses **36**
Single-Core Workload Cache Miss Rates 37
Multi-Core Workload Cache Miss Rates 38
GPU Benchmark **39**
API Support 39
Runtime 39
Scores 40
Machine Learning Workloads 41
Background Blur 42
Face Detection 43
Image Editing Workloads 44
Horizon Detection 45
Edge Detection 46
Gaussian Blur 46
Image Synthesis Workloads 47
Feature Matching 48
Stereo Matching 49
Simulation 50
Particle Physics 51

Introduction

This document outlines the workloads included in the Geekbench 6 CPU Benchmark and GPU Benchmark suites.

CPU Benchmark scores are used to evaluate and optimize CPU and memory performance using workloads that include data compression, image processing, and machine learning. Performance on these workloads is important for a wide variety of applications including web browsers, image editors, and developer tools.

GPU Benchmark scores are used to evaluate and optimize GPU Compute performance using workloads that include image processing, computational photography, computer vision, and machine learning. Performance in these workloads is important for a wide variety of applications including cameras, image editors, and real-time renderers.

Platform Support

Platform	Minimum Version	Comment
Android	Android 10	
iOS	iOS 15	
Linux	Ubuntu 18.04 LTS	CentOS, RHEL support TBD
macOS	macOS 11	
Windows	Windows 10	

Architecture Support

Platform	Architectures	Comment
Android	AArch64, x64	
iOS	AArch64	
Linux	AArch64, x64	
macOS	AArch64, x64	
Windows	x64	

Compiler Support

Geekbench 6.0 is built using the following compilers:

Platform	Compiler	Comment
Android	Clang 14	Clang provided by NDK r25c
iOS	Clang 15	
Linux	Clang 15	
macOS	Clang 15	
Windows	Clang 15	

Geekbench 6.1 is built using the following compilers:

Platform	Compiler	Comment
Android	Clang 16	
iOS	Clang 16	
Linux	Clang 16	
macOS	Clang 16	
Windows	Clang 16	

CPU Benchmark

Runtime

Geekbench 6 groups CPU workloads into two sections:

1. Single-Core Workloads
2. Multi-Core Workloads

Each single-core workload has a multi-core counterpart, and vice versa. Each section is grouped into two subsections:

1. Integer Workloads
2. Floating-Point Workloads

Geekbench inserts a pause (or gap) between each workload to minimize the effect thermal issues have on workload performance. Without this gap, workloads that appear later in the benchmark would have lower scores than workloads that appear earlier in the benchmark.

The default gap in Geekbench 6.0 is 2 seconds.

The default gap in Geekbench 6.1 and later is 5 seconds.

Multi-Threading

Geekbench 6 uses a “shared task” model for multi-threading, rather than the “separate task” model used in earlier versions of Geekbench. The “shared task” approach better models how most applications use multiple cores.

The "separate task" approach used in Geekbench 5 parallelizes workloads by treating each thread as separate. Each thread processes a separate independent task. This approach scales well as there is very little thread-to-thread communication, and the available work scales with the number of threads. For example, a four-core system will have four copies, while a 64-core system will have 64 copies.

The "shared task" approach parallelizes workloads by having each thread processes part of a larger shared task. Given the increased inter-thread communication required to coordinate the work between threads, this approach may not scale as well as the "separate task" approach.

Instruction Sets

Each Geekbench 6 build targets a base instruction set. The base instruction set informs the compiler which instructions it can safely use when generating code.

Some platforms may include multiple builds that target different base instruction sets. In this case, Geekbench 6 selects the build with the most advanced base instruction set supported on the system to measure performance. For x86 processors, Geekbench 6 uses SSE2 and AVX2 as the base instruction sets. For ARM processors, Geekbench 6 uses ARMv8 as the base instruction set.

Geekbench 6 workloads may also use functions that target instruction sets extensions above and beyond those supported by the base instruction set. These functions are written using intrinsics and are guarded by runtime checks to ensure they only run on supported processors.

x86 Instruction Sets

Geekbench 6 workloads may use functions written using the following instruction set extensions on x86 processors:

Instruction Set	Description
AES-NI	Accelerates AES encryption and decryption functions
VAES	Accelerates AES encryption and decryption functions
SHA-NI	Accelerates SHA1 cryptographic hash functions
AVX	Generic floating-point 256-bit SIMD instruction set
AVX2	Generic 256-bit SIMD instruction set
AVX-512	Generic 512-bit SIMD instruction set
AVX-VNNI	Accelerates quantized machine learning workloads
AVX512-VNNI	Accelerates quantized machine learning workloads
AMX	Accelerates quantized machine learning workloads

ARM Instruction Sets

Geekbench 6 workloads may use functions written using the following instruction set extensions on ARM processors:

Instruction Set	Description
ARMv8 AES	Accelerates AES encryption and decryption functions
ARMv8 SHA1	Accelerates SHA1 cryptographic hash functions
NEON	Generic 128-bit SIMD instruction set
NEON FP16	Generic 128-bit SIMD instruction set with support for 16-bit floats
DOTPROD	Accelerates image processing and machine learning workloads
I8MM	Accelerates quantized machine learning workloads
SME	Accelerates machine learning workloads

Scores

Geekbench 6 scores are calibrated against a baseline score of 2,500 (which is the score of a Dell Precision 3460 with a Core i7-12700 processor). Higher scores are better, with double the score indicating double the performance.

Geekbench 6 provides two composite scores: single-core and multi-core. These scores are computed using a weighted arithmetic mean of the subsection scores. The subsection scores are computed using the geometric mean of the scores of the workloads contained in that subsection.

Subsection	Weight
Integer	65%
Floating Point	35%

Productivity Workloads

Productivity workloads measure how well your CPU handles common operations critical to everyday tasks, including data compression, image compression, web browsing, and 2D graphics.

File Compression

The File Compression workload compresses and decompresses a file using different compression formats. It models use cases where users and software apps compress files to reduce data and bandwidth (such as compressing photos and files when sending emails).

This workload compresses and decompresses the Ruby 3.1.2 source archive (a 75 MB archive with 9,841 files) using the LZ4 and ZSTD compression codecs with high and low compression ratios. It also verifies the compressed and decompressed output files using the SHA1 hash function.

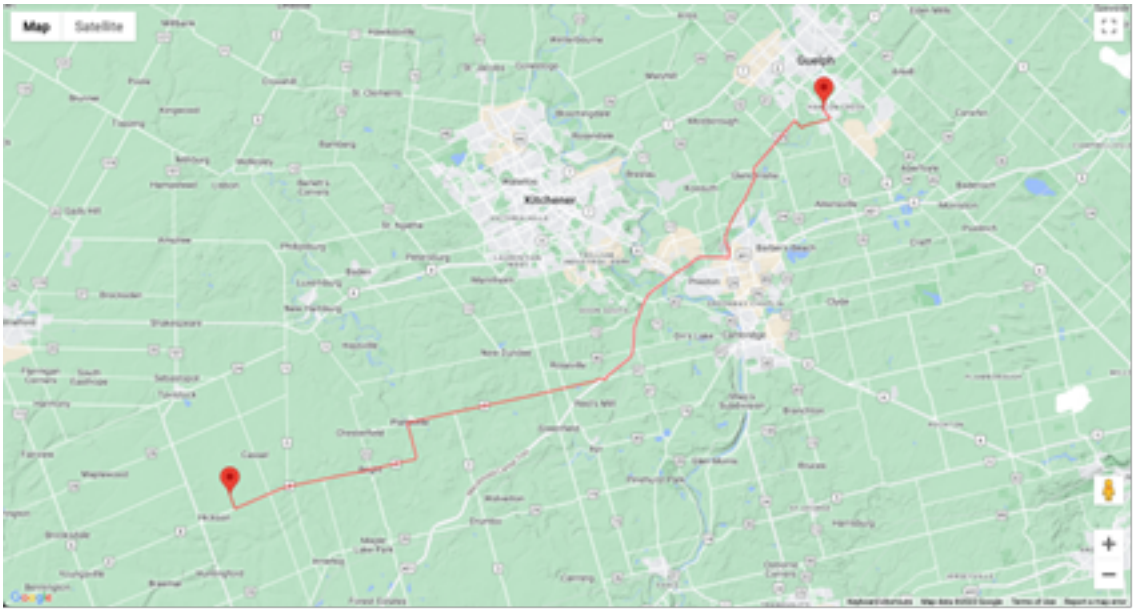
The files are stored using an in-memory encrypted file system.

File Compression uses instructions that accelerate AES encryption and decryption (AES-NI and VAES on x86 processors, ARMv8 AES on ARM processors) and that accelerate SHA1 cryptographic hash functions (SHA-NI on x86 processors, ARMv8 SHA on ARM processors).

Navigation

The Navigation workload generates directions between a sequence of locations. It models the use case of users asking for directions from a navigation app (such as Google Maps in offline mode).

This workload uses Dijkstra's algorithm to calculate 24 different routes on two OpenStreetMap maps — one for a small city (Waterloo, Ontario) and one for a large city (Toronto, Ontario).



Example of a route generated by the Navigation workload

HTML5 Browser

The HTML5 Browser workload opens various web pages using a web browser. It models the use case of a user browsing the web with a browser (such as Chrome and Safari).

This workload uses a headless browser and opens, parses, lays out, and renders text and images for web pages based on popular websites (such as Ars Technica, Instagram, and Wikipedia).

The HTML5 Browser workload uses the following libraries:

- Google Gumbo as the HTML parser
- litehtml as the CSS parser, layout, and rendering engine
- FreeType as the font engine
- Anti-Grain Geometry (AGG) as the 2D rendering graphics library
- libjpeg-turbo and libpng as the image codecs

HTML5 Browser renders 8 pages in single-core mode and 32 pages in multi-core mode.

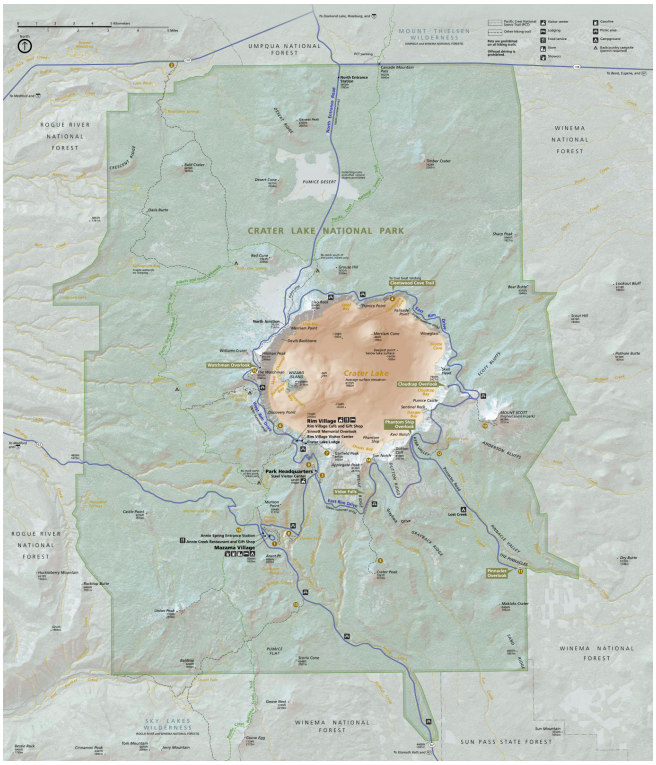
PDF Render

The PDF Render workload opens complex PDF (Portable Document Format) documents using PDFium, Google Chrome's PDF renderer. It models the use case of a user opening PDFs in a browser.

This workload renders PDFs of park maps from the American National Park Service (sizes from 897 KB to 1.5 MB) that contain large vector images, lines and text.

The maps from the American National Park Service include:

- Crater Lake Park Map (Size: 897 KB)
- Golden Gate Area Map (Size: 1.5 MB)
- Lewis and Clark Park Map (Size: 288 KB)
- Mount Rainier Park Map (Size 949 KB)



Crater Lake Park Map

PDF Render renders 4 PDFs in single-core mode and 16 PDFs in multi-core mode.

Photo Library

The Photo Library workload categorizes and tags photos based on the objects that they contain. This lets users search their photos by keyword in image organizer apps (such as Adobe Lightroom, Apple Photos, and Google Photos).

The workload uses MobileNet 1.0 to classify photos and a SQLite database to store the photo metadata (including their tags).

This workload performs the following steps for each photo:

1. Decompress the photo from a compressed JPEG file.
2. Store photo metadata (e.g., file name, photo resolution, etc) into a SQLite database. The database is pre-populated with metadata for over 70,000 photos.
3. Generate a preview thumbnail (using bilinear scaling) and encode it as a JPEG file.
4. Generate an inference thumbnail (using centre crop and bilinear scaling).
5. Run an image classification model on the inference thumbnail. The model is based on MobileNet 1.0 and the model weights are quantized.
6. Store image classification tags in the SQLite database.

The Photo Library workload operates on 16 photos in single-core mode and 64 photos in multi-core mode.

Photo Library uses instructions that accelerate quantized machine learning workloads (AVX-VNNI,AVX512-VNNI, and AMX on x86 processors, DOTPROD, I8MM, and SME on ARM processors).

Developer workloads

Developer workloads measure how well your CPU handles typical developer tasks such as processing text files, compiling code, and compressing assets.

Clang

The Clang workload uses the Clang compiler to compile the Lua interpreter, a popular open-source language interpreter. It models the use case of developers building their code and the just-in-time compiling that general users can encounter on their systems (such as JIT compilation for scripting Java and compilation for shading languages in GPU drivers).

This workload uses the musl libc as the C standard library for the compiled files.

The Clang workload compiles 8 files in single-core mode and 96 files in multi-core mode.

Text Processing

The Text Processing workload loads numerous files, parses the contents using regular expressions, stores metadata in a SQLite database, and finally exports the content to a different format. It models typical text processing tasks that manipulate, analyze, and transform data to reformat it for publication and to gain insights.

The input and output files are stored using an in-memory encrypted file system.

The workload is implemented using a mix of Python and C++. The workload uses the Python 3.9.0 interpreter and processes 190 Markdown files as its input.

Text Processing uses instructions that accelerate AES encryption and decryption (AES-NI and VAES on x86 processors, ARMv8 AES on ARM processors).

Asset Compression

The Asset Compression workload compresses 3D textural and geometric assets using a variety of popular compression codecs (ASTC, BC7, DXT5). It models standard content compression pipelines, such as those used by game developers.

This workload prepares 16 texture images and geometry files for distribution using the following codecs and encounters: ASTC, BC7, DXTC, and Draco.

The workload uses bc7enc for its BC7 and DXTC implementations and Arm ASTC Encoder for its ASTC implementation.

Machine Learning Workloads

Machine Learning workloads measure how well your CPU handles recognizing objects in images and scenes.

Object Detection

The Object Detection workload uses machine learning to detect and classify objects in photos and then highlight them in the photo.

The workload uses the convolutional neural network (CNN) MobileNet v1 SSD to detect and classify objects in photos. The photos are pre-sized to meet the model input dimensions (300 X 300 px).

This workload performs the following steps:

1. Loads the photo.
2. Extract objects from the photo using MobileNet v1 SSD.
3. Generates a confidence or detection score that represents the accuracy of the detection.
4. Draws a bounding box around the objects and outputs the confidence score.

Input

Output with bounding box, confidence score

Object Detection uses instructions that accelerate quantized machine learning workloads (e.g. AVX-VNNI, AVX512-VNNI, and AMX on x86 processors, DOTPROD, I8MM, and SME on ARM processors).

Object Detection processes 16 photos in single-core mode and 64 photos in multi-core mode.

Background Blur

The Background Blur workload separates the background from the foreground in a video stream and blurs the background. It models background blurring features in video conferencing apps (such as Zoom, Slack Huddles, and Microsoft Teams).

This workload uses DeepLabV3+ as its network and blurs 10 frames from a 1080p video stream.

Input

Output

Background Blur uses SIMD and machine learning instruction sets (AVX, AVX2, and AVX-512 on x86 processors, NEON and SME on ARM processors) to accelerate machine learning functions. Since the Background Blur machine learning model uses 32-bit floating point weights (i.e., is not quantized) so the Background Blur workload cannot use quantized machine learning instructions or 16-bit floating point instructions to accelerate its machine learning functions.

Background Blur also uses generic SIMD instruction sets (AVX2 on x86 processors, NEON and NEON FP16 on ARM processors) to accelerate image processing functions.

Image Editing

Image editing workloads measure how well your CPU handles making simple and complex image edits.

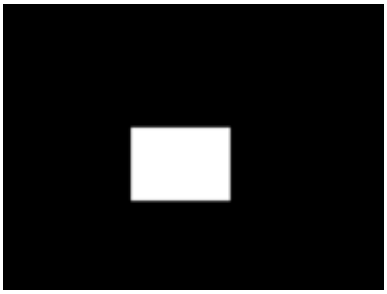
Object Remover

The Object Remover workload removes an object from a photo and automatically fills in the gap left behind. It models content-aware fill and magic eraser features in photo editing apps (such as Adobe Photoshop and Google Photos).

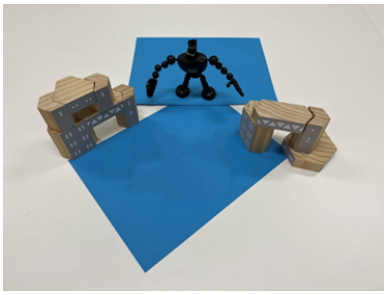
Given a 3 MP image with an undesirable region (indicated via a mask image), this workload removes the region and uses an inpainting scheme to reconstruct the gap left behind.

Object Remover uses the iterative PatchMatch Inpainting approach discussed in Barnes et al.'s (2009) "PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing".

Input



Mask



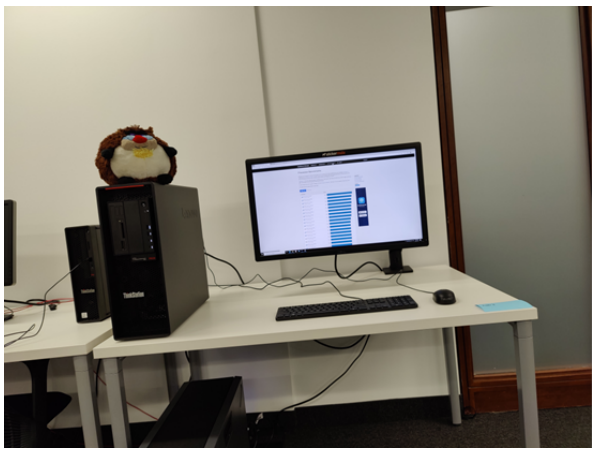
Output

Horizon Detection

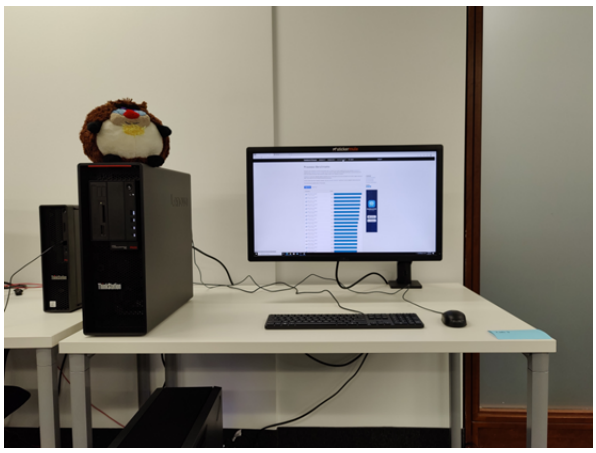
The Horizon Detection workload detects and straightens uneven or crooked horizon lines in photos to make the photos look more realistic. It models horizon line correctors in photo editing apps (such as Adobe Lightroom, Google Gallery, and Apple Photos).

This workload reduces the detail in the photo using the Canny edge detector and applies the Hough transform to detect the horizon line. It then rotates the image so that the horizon line is level in the photo.

This workload uses a 48 MP photo as its input.



Input



Output

Photo Filter

The Photo Filter workload applies filters to photos to enhance their appearance. It models common filters available in social media and photo editing apps (such as Instagram and Adobe Lightroom).

This workload applies the following effects to 10 photos: colour and blur filters, level adjustments, cropping and scaling, and image compositing. The photos range in size from 3 MP to 15 MP.

Example Input

Example Output

Photo Filter uses generic SIMD instruction sets (AVX2 on x86 processors, NEON and NEON FP16 on ARM processors) to image processing functions.

HDR

The HDR workload blends 6 SDR (Standard Dynamic Range) photos to create a single HDR (High Dynamic Range) photo that is more colourful and vibrant than any of the individual SDR photos. It models HDR features that are standard in modern smartphone camera apps (such as Google Camera and Apple Camera).

This workload creates a 16 MP HDR image from six 16 MP SDR photos.

HDR uses a recovery process and radiance map construction that is based on the methodology described by Debevec and Malik (1997) "Recovering High Dynamic Range Radiance Maps from Photographs". It also uses a tone mapping algorithm that is based on Reinhard and Devlin's (2005) "Dynamic Range Reduction inspired by Photoreceptor Physiology."

Input (Short Exposure)

Input (Long Exposure)

Output

Image Synthesis

Image synthesis workloads measure how well your CPU handles creating artificial images.

Ray Tracer

Ray tracing is a rendering technique used to generate photorealistic images by modelling how light rays interact with objects in a virtual scene. It models the rendering processes employed in 3D rendering software (such as Blender, Maxon Cinema 4D, and Chaos Corona).

This workload renders the Blender BMW scene using a custom ray tracer built with the Intel Embree ray tracing library.

Structure from Motion

Structure from Motion is a technique that generates 3D geometry from multiple 2D images. Augmented Reality (AR) systems use techniques like Structure from Motion to understand real-world scenes and integrate computer-generated graphics into these scenes.

The Structure from Motion workload takes nine 2D images of the same scene and constructs an estimate of the 3D coordinates of the points that are visible in both images.

Structure from Motion uses generic SIMD instruction sets (AVX2 on x86 processors, NEON and NEON FP16 on ARM processors) to image processing functions.

CPU Benchmark Characteristics

Geekbench 6 CPU workload performance depends on a wide range of processor and memory subsystems. These dependencies are characterized by the following characteristics.

The data was collected from a Dell Precision 3460 workstation with an Intel Core i5-12500 processor running Ubuntu 22.04 LTS.

Instructions Per Cycle

Instructions per Cycle, or IPC, is a measure of the effective instruction throughput of a processor, which correlates with higher performance. It is measured as the number of instructions executed for a workload divided by the number of cycles used for that workload.

Workload	Single-Core IPC	Multi-Core IPC
File Compression	2.0	0.8
Navigation	1.1	0.7
HTML5 Browser	2.7	1.5
PDF Renderer	3.5	2.2
Photo Library	3.0	1.9
Clang	1.9	1.2
Text Processing	3.9	3.6
Asset Compression	2.7	1.8
Object Detection	3.7	1.3
Background Blur	2.9	1.3
Horizon Detection	2.1	1.3
Object Remover	2.2	1.3
HDR	3.0	1.5
Photo Filter	2.5	1.0
Ray Tracer	2.4	1.7
Structure from Motion	2.9	1.5

Branch Prediction Miss Rate

Branch prediction miss rate is a measure of how frequently a system incorrectly predicts a code branch, which results in lower performance. It is measured as a percentage of total branches. Whenever a code path branches into multiple cases, a system executing that code attempts to predict which case will be true in order to pre-fetch data or pre-execute instructions. When operating on well-ordered data, systems can correctly predict (or “hit”) branches more frequently, improving performance.

Workload	Single-Core Branch Miss Rate	Multi-Core Branch Miss Rate
File Compression	3.4	3.3
Navigation	5.6	5.9
HTML5 Browser	0.5	0.5
PDF Renderer	1.1	1.2
Photo Library	1.3	1.6
Clang	3.0	3.6
Text Processing	0.4	0.4
Asset Compression	2.0	2.0
Object Detection	0.2	0.2
Background Blur	0.2	0.3
Horizon Detection	2.8	2.8
Object Remover	0.1	0.2
HDR	0.4	0.4
Photo Filter	0.4	0.5
Ray Tracer	1.0	1.0
Structure from Motion	0.7	0.7

Working Set Size

Working Set size is a measure of the amount of memory that a program uses, either by reading from it or writing to it. It is expressed in bytes, but operates by measuring memory used at the granularity of pages. Programs with a large working-set size run on systems with small caches can encounter more cache misses, where the program cannot find the data it needs in a cache and must query a larger cache or main memory. Cache misses negatively impact performance.

Workload	Single-Core	Multi-Core
File Compression	105.5 MB	418.2 MB
Navigation	190.6 MB	190.8 MB
HTML5 Browser	299.4 MB	1195.8 MB
PDF Renderer	433.7 MB	1609.3 MB
Photo Library	225.9 MB	922.7 MB
Clang	25.8 MB	112.7 MB
Text Processing	54.7 MB	81.8 MB
Asset Compression	63.8 MB	108.0 MB
Object Detection	342.4 MB	1287.4 MB
Background Blur	269.5 MB	304.9 MB
Horizon Detection	803.5 MB	816.2 MB
Object Remover	207.5 MB	208.6 MB
HDR	747.4 MB	846.6 MB
Photo Filter	1039.2 MB	1169.3 MB
Ray Tracer	120.7 MB	112.9 MB
Structure from Motion	168.5 MB	871.9 MB

Cache Misses

Cache miss rates measure how frequently a program fails to find data in a processor's cache when attempting to read from or write to memory.

Cache miss rates are measured as a percentage of total accesses to that cache. For example, if the requested data is not in the L1D cache or the L2 cache but is in the L3 cache, then this is recorded as both an L1D cache miss and an L2 cache miss.

Single-Core Workload Cache Miss Rates

Workload	L1I Miss	L1D Miss	L2 Miss	L3 Miss
File Compression	0.0	2.1	16.1	6.7
Navigation	0.0	5.1	19.3	23.9
HTML5 Browser	0.9	2.5	6.8	31.6
PDF Renderer	1.0	1.4	9.7	52.3
Photo Library	0.2	2.9	5.8	44.9
Clang	7.0	2.4	5.5	2.0
Text Processing	0.5	0.9	6.8	4.7
Asset Compression	0.1	1.6	0.6	12.8
Object Detection	0.1	5.7	9.0	22.2
Background Blur	0.0	13.6	0.5	25.6
Horizon Detection	0.0	5.3	3.8	78.6
Object Remover	0.0	13.3	47.1	8.3
HDR	0.0	2.9	19.1	70.5
Photo Filter	0.1	7.5	1.8	78.3
Ray Tracer	0.2	0.3	17.9	73.6
Structure from Motion	0.1	2.4	20.4	40.7

Multi-Core Workload Cache Miss Rates

Workload	L1I Miss	L1D Miss	L2 Miss	L3 Miss
File Compression	0.1	3.4	25.9	46.3
Navigation	0.0	5.7	26.2	38.9
HTML5 Browser	1.5	3.1	8.6	60.9
PDF Renderer	0.9	1.6	8.5	69.9
Photo Library	0.1	3.3	4.8	60.0
Clang	9.3	3.6	7.7	13.3
Text Processing	0.8	0.6	11.1	8.0
Asset Compression	0.2	1.6	1.3	9.0
Object Detection	0.3	5.5	10.3	58.1
Background Blur	0.3	13.5	4.6	2.9
Horizon Detection	0.1	5.6	4.9	63.2
Object Remover	0.2	15.7	44.0	8.2
HDR	0.1	1.7	34.9	89.6
Photo Filter	0.2	8.9	4.5	84.7
Ray Tracer	0.2	0.3	16.7	74.3
Structure from Motion	0.1	3.3	18.5	55.3

GPU Benchmark

API Support

API	Minimum Version	Comments
Metal	3.0	
OpenCL	1.2	
Vulkan	1.2	

Geekbench 6 adds a new API abstraction layer, called Thorium, that sits above all supported Compute APIs. Thorium provides an interface that can express everything necessary when writing Compute workloads. By using Thorium, the host code for a workload only needs to be implemented once.

Thorium is designed and built with performance in mind. The interface is lightweight, with most of the code being simple wrappers around the Compute APIs.

Thorium is also designed so that no compute framework is at a disadvantage. For example, high-performance Vulkan code uses command buffers recorded into a queue, whereas OpenCL has no recording mechanism. The Vulkan implementation of thorium::Queue uses command buffers for maximum performance, and the OpenCL implementation uses asynchronous dispatches and events. The behaviour is the same from the host perspective while still being performant on both frameworks.

Runtime

Geekbench inserts a pause (or gap) between each workload to minimize the effect thermal issues have on workload performance. Without this gap, workloads that appear later in the benchmark would have lower scores than workloads that appear earlier in the benchmark.

The default gap in Geekbench 6.0 is 2 seconds.

The default gap in Geekbench 6.1 and later is 5 seconds.

Scores

Geekbench 6 scores are calibrated against a baseline score of 2,500 (which is the score of a Dell Precision 3460 with a Core i7-12700 processor). Higher scores are better, with double the score indicating double the performance.

Geekbench GPU provides one composite score that is computed using the geometric mean of the scores of all the workloads.

Machine Learning Workloads

Machine Learning workloads measure how well your GPU uses machine learning algorithms to perform object recognition tasks such as identifying objects and blurring backgrounds in photos.

Background Blur

Background blur separates the background from the foreground in a video stream and blurs the background. It models background blurring features in video conferencing apps (such as Zoom, Slack Huddles, and Microsoft Teams).

This workload uses DeepLabV3+ as its network and blurs a frame from a 1080p video stream.

Input

Output

Face Detection

Face detection is used to locate faces in images. Face detection is used in applications such as photography or video conferencing for autofocusing.

The Face Detection workload uses a machine learning model. It returns the coordinates, along with a confidence score, for each face in an image. The workload uses RetinaFace as its network.

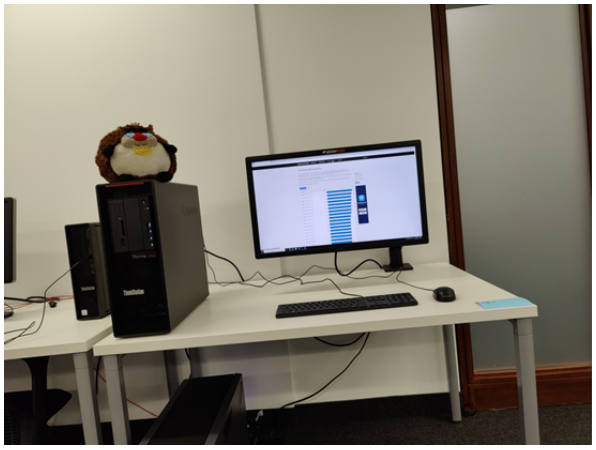
Image Editing Workloads

Image editing workloads measure how well your GPU handles making simple and complex image edits.

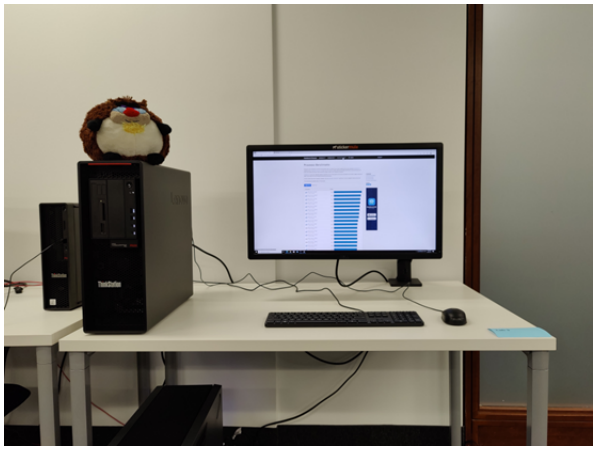
Horizon Detection

Horizon detection locates the horizon line in an image. It is used in image editing and photo retouching applications to automatically level photos.

The Horizon Detection workload uses the Hough transform to identify straight lines in a 24 MP image and decide which of these lines is the horizon line. It then rotates the image, so that the horizon line is horizontal.



Input



Output

Edge Detection

Edge detection is used in image processing and computer vision applications to identify edges in an image. Edge detection produces a sketch-like representation of the image. It is often used as the first stage of more complicated computer vision applications, including feature detection and pattern recognition.

The Edge Detection workload applies the Canny edge detector operator to a 24 MP photo.

Gaussian Blur

Gaussian blur is an image filter used to soften and blur images. It is used in image editing programs to improve the appearance of photos, and to remove fine details before applying other imaging processes and techniques. It is also used in modern user interfaces (for example, to blur background windows to focus user attention).

This workload applies a Gaussian Blur filter that uses a filter diameter of 25 px by 25 px to a 24 MP photo.

Image Synthesis Workloads

Image synthesis workloads measure how well your GPU handles content creation tasks, including image rendering and image processing.

Feature Matching

Feature matching takes two photos and identifies points that are the same in both. It is often used as part of other processes to identify objects in photos and to make 3D reconstructions. For example, Structure From Motion uses Feature Matching to find the initial points to reconstruct into a 3D scene.

The Feature Matching workload uses the Oriented FAST and Rotated BRIEF (ORB) algorithm to match features (or keypoints) between two 6MP images.

Stereo Matching

Stereo matching is used to generate 3D depth maps from two 2D images of the same scene. Camera applications on multi-sensor smartphones use stereo matching to produce depth maps, which in turn are used to power photo filters, to create 3D images, and to improve the quality of augmented reality (AR) applications.

The Stereo Matching workload uses a block-matching algorithm to compute the difference in position of each pixel and to create a depth map. This algorithm compares small groups of pixels in one image to the closest match in the other image, using the Sum of Absolute Differences (SAD) as a measure of similarity.

Simulation

Simulation workloads measure how well your GPU Handles physics simulation tasks.

Particle Physics

Particle Physics is a technique commonly used in games to simulate fluids and smoke.

The Particle Physics workload implements a simulation where particles interact with one another and their environment via elastic collisions. Other particle-particle forces are ignored. The Particle Physics workload uses 4,096 particles in its simulation.