NOT A REPLACEMENT — A DIVISION OF LABOR

BROWSER×GPU 2026

Compute and rendering — the state of every layer, mapped in one continuous view.

- From WASI and runtimes to MoonBit-family languages, organized layer by layer
- A real map of implementations: Wasmtime, WasmEdge, Dawn, wgpu, and more
- Technology-selection guidance for both browser and native

By asopitech

Contents

1 Browser × GPU 2026: The Road So Far and Where We Stand Now

This book surveys the browser's high-performance computing stack in 2026: WebAssembly, WebGL, WebGPU, WASI, the Component Model, runtimes, toolchains, and WASM-native languages such as MoonBit. Chapter 1 traces the browser–GPU bridge from plugins to WebGPU. Later chapters explain its execution model and implementation flow, then compare the frameworks and languages available for browser and native development.

1.1 Chapter 1: The Road to the Browser–GPU Bridge

Browser GPU access developed through plugins, Canvas, WebGL, and WebGPU. The sequence explains how the browser came to validate and execute work submitted by a Web page.

1.1.1 How the Bridge Was Built

1.1.2 When plugins carried GPU-heavy work

Early Web standards focused on documents and basic interaction. Developers used native applications or browser plugins for applications and games involving 3D rendering or video processing. Plugins extended the browser, and users installed and updated them separately from Web pages. Plugin hangs, crashes, and vulnerabilities encouraged browser vendors to support these uses through standard Web features.

1.1.3 Canvas became the page's drawing surface

Canvas created a standard place for rendered output inside a page. Its 2D context supported graphs, image editing, and games. Demand for 3D rendering and GPU programs led to WebGL as an additional Canvas context.

1.1.4 WebGL created the first standard bridge

WebGL 1.0, released in 2011, exposed an OpenGL ES 2.0-derived drawing model and GLSL shaders to JavaScript through Canvas. It made plugin-free 3D distribution practical and became a foundation for Three.js, maps, data visualization, and Web games. WebGL 2.0 expanded toward the OpenGL ES 3.0 feature set.

In defining WebGL, browser vendors treated code and data supplied by a Web page as untrusted input and brought GPU use within the browser's sandbox. Buffer-range validation, restrictions around cross-origin images and video, shader validation, and context loss after GPU resets were designed into the bridge. This approach continues in later browser GPU APIs.

1.1.5 WebGPU responded to the next GPU generation

During the 2010s, Metal, Direct3D 12, and Vulkan made resources and commands more explicit, while browser use cases increasingly needed GPU computation for image processing, simulation, and machine learning. WebGPU was designed for this setting. It extends the Web's bridge to modern rendering and compute; the current model follows below.

```
JavaScript / WebAssembly
  → WebGL / WebGPU
  → browser validation and translation
  → Metal / Direct3D 12 / Vulkan
  → driver → GPU
```

The browser translates GPU requests, validates resources, and schedules execution. The remaining chapters follow those operations and identify the runtimes and frameworks that implement them.

The bridge controls GPU access and data isolation. The browser limits a page to its permitted resources and validates calls to the GPU driver.

- **Memory isolation.** The browser limits shader access to buffers and textures connected through bind groups. It validates ranges, usages, and access types, shielding uninitialized GPU memory and other workloads' data.
- **Origin isolation.** Reading pixels from an image or video owned by another origin can disclose information. WebGL and WebGPU inherit Canvas origin rules and CORS controls around texture use, shader analysis, and readback.
- **Execution continuity.** Very expensive work, driver defects, and GPU resets can disrupt a tab or the browser. Validation, process isolation, and device/context-loss notifications contain that impact and give the application a chance to rebuild resources.

The same boundary affects performance. The browser validates and translates resources before execution, then submits work in command batches. WebGPU represents these operations through pipelines and command buffers. Applications can reduce transfers and CPU readbacks by keeping dependent stages on the GPU.

1.1.6 An Abstract Model of the Bridge

The browser has three roles in GPU execution.

1. **Capability broker.** The page requests a `GPUAdapter`, then a `GPUDevice` within the capabilities and limits the browser chooses to expose.
2. **Validator and translator.** The browser checks WGSL source, buffer ranges, texture formats, resource bindings, and command ordering. It then

maps the WebGPU model to native backends such as Metal, Direct3D 12, and Vulkan.

3. **Scheduler.** The page records work into a command encoder. The browser submits the completed command buffer through a queue and manages asynchronous completion and device loss.

```
Application layer       JavaScript / WASM expresses the intended work
        ↓
WebGPU abstraction      shaders, resources, pipelines, and commands
        ↓
Browser implementation  validation, state tracking, translation, process boundaries
        ↓
Native execution        OS GPU API, driver, and GPU
```

The API handles code, data, and execution plans as separate objects. WGSL defines the program; buffers and textures provide inputs and outputs; a pipeline combines code with fixed-function state; a command buffer records the work.

1.2 Chapter 2: The Browser–GPU Execution Model and Dispatch Flow

The browser returns a `GPUAdapter` representing an available GPU. The application obtains a `GPUDevice` from it to create buffers and shaders. The device's queue submits recorded commands for execution.

The page supplies WGSL source through `createShaderModule()`. Pipeline creation combines a selected entry point with its resource layout and fixed-function state. Bind groups connect `GPUBuffer`, `GPUTexture`, and `GPUSampler` objects to the shader's declared `@group` and `@binding` slots. A shader accesses resources bound through a compatible layout.

Browser implementations translate this model to native APIs. Chromium's Dawn validates and maps WebGPU to backends such as D3D12, Metal, and Vulkan, using Tint for WGSL translation. Firefox's WebGPU implementation is built around wgpu and Naga. The exact intermediate representation and compilation timing are implementation details; the portable application contract is WebGPU and WGSL.

After device loss, the application recreates GPU resources from its source data.

1.2.1 From WGSL Source to GPU Dispatch

For a compute kernel, the application follows this sequence.

```
WGSL source
  → createShaderModule()
  → createComputePipeline()
  → upload input to GPUBuffer
  → connect resources with a bind group
```

```
  → record dispatch in a command encoder
  → queue.submit()
  → GPU executes shader invocations
  → read back when the CPU needs the result
```

`dispatchWorkgroups()` launches many invocations of a WGSL `@compute` entry point. The number of workgroups selected by JavaScript and the shader's `@workgroup_size` determine the logical execution grid. Invocations in one workgroup coordinate through workgroup memory and barriers. Synchronization across workgroups requires separate dispatches or passes.

The JavaScript side records work before it submits it:

```
const encoder = device.createCommandEncoder();
const pass = encoder.beginComputePass();
pass.setPipeline(pipeline);
pass.setBindGroup(0, bindGroup);
pass.dispatchWorkgroups(Math.ceil(elementCount / 64));
pass.end();
device.queue.submit([encoder.finish()]);
```

CPU memory, WASM linear memory, and GPU buffers are normally distinct memory domains. `queue.writeBuffer()` transfers prepared input into GPU memory. Results can remain there for subsequent compute and render passes. A `mapAsync()` readback creates a synchronization point and may cost more than the kernel itself. A typical flow is `CPU → GPU → Compute A → Compute B → Render`.

Workers and OffscreenCanvas keep the UI thread responsive; SharedArrayBuffer can reduce CPU-side copies when cross-origin isolation is configured. For WebCodecs frames, decoded 3D assets, and AI tensors, trace format conversions, copy counts, and GPU-to-CPU readback points.

1.3 Chapter 3: WASM Runtimes and Host Execution

WASM executes in browser engines and in standalone runtimes embedded in applications. This chapter covers runtime implementations, host APIs, and the use cases they serve.

1.3.1 Layers of the WASM Ecosystem

Wasmer and other WASM-related implementations serve different layers:

```
Applications / plugins / serverless
        │
        ▼
Higher-level frameworks
  Spin / wasmCloud / Extism / Wasmer Edge
        │
        ▼
```