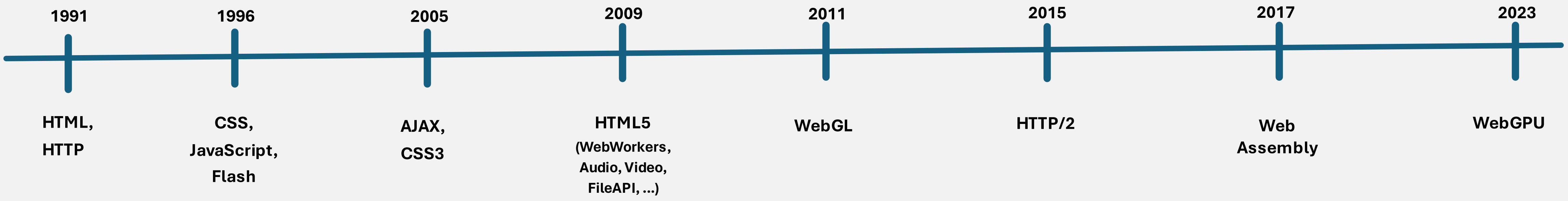
USING WEBASSEMBLY AND WEBGPU TO BUILD THE NEXT GENERATION OF DATA-INTENSIVE APPLICATIONS IN THE BROWSER

Markus Tremmel

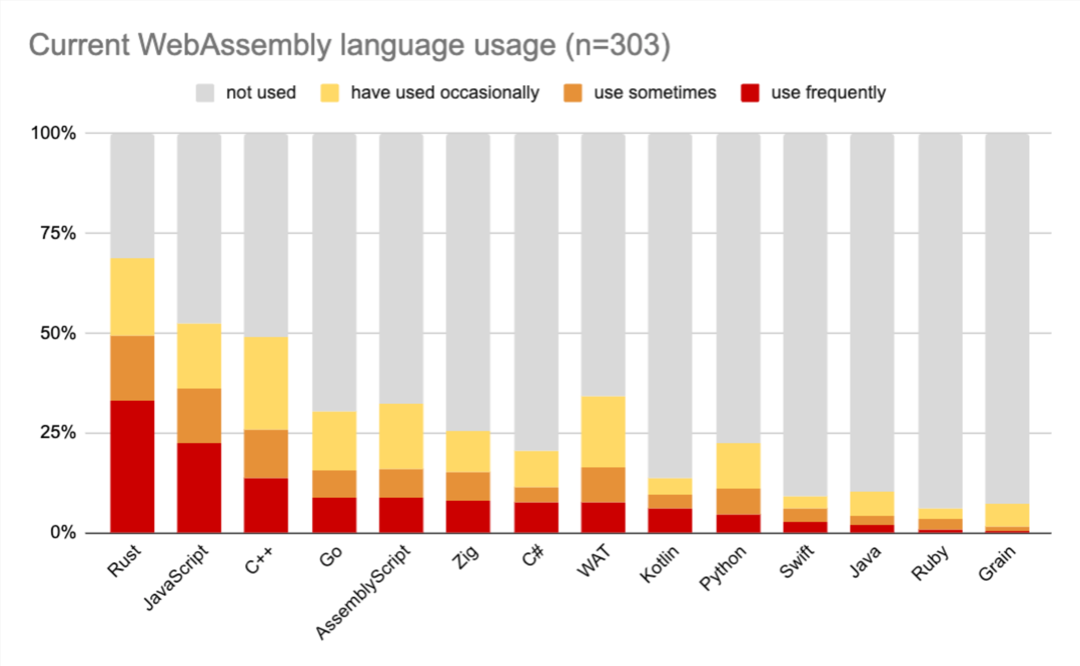
ROUGH TIMELINE OF WEB TECHNOLOGIES

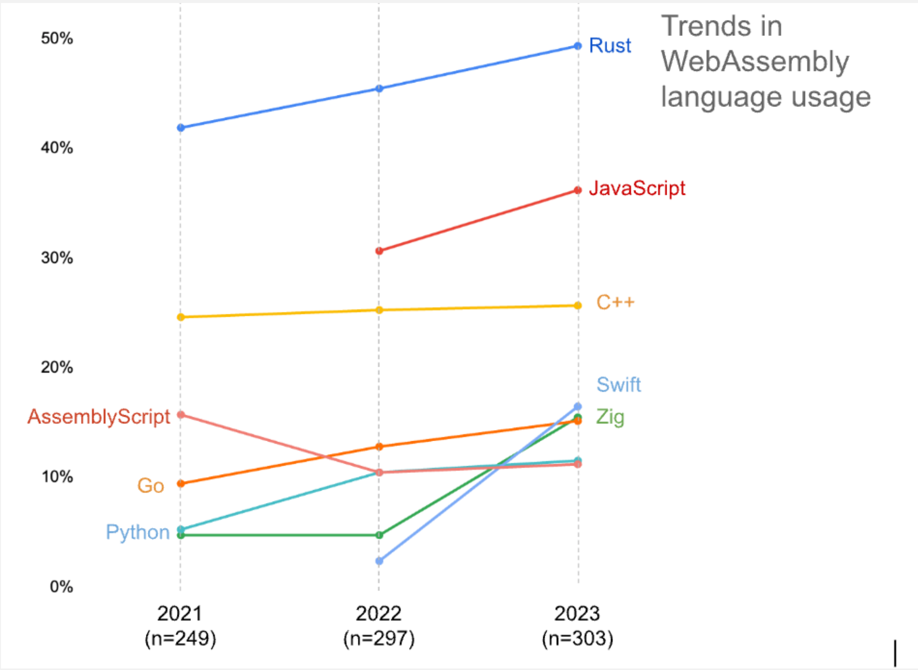


WEBASSEMBLY

- WebAssembly (Wasm) is an universal low level bytecode that runs on a stack-based virtual machine in web browsers (and servers/cloud) at near-native speeds
- It is not primarily intended to be written by hand, rather it is designed to be an portable compilation target for source languages such as Rust, AssemblyScript (Typescript-like), Emscripten (C/C++), Blazor WASM (C#), etc.
- WebAssembly only provides numeric data types (i32, i64, f32, f64, v128) and lacks direct access to the Web API (network, DOM, …)
- Three main reasons why people use WebAssembly
  - Use other languages then JavaScript in the browser
  - Migrate (legacy) native apps to the web -> AutoCAD, Figma, Photoshop, Google Earth, …
  - Improve performance for data-intensive applications

WEBASSEMBLY LANGUAGES





Source: The State of WebAssembly 2023

WHY RUST?

- Rust lacks a runtime and therefore doesn’t include extra bloat like a garbage collector which enables small wasm sizes -> faster pages load
- Great tooling support
    - Wasm-pack -> for building, testing, and publishing Rust-generated WebAssembly modules
    - Wasm-bindgen -> Rust crate for high-level interactions between Rust and JavaScript
    - Other useful Rust crates: js-sys, web-sys, …

WEBASSEMBLY WITH WASM-PACK

- Create new project: **wasm-pack new app**
- Build WASM module: **wasm-pack build --release --target web**
  - The wasm-pack build command creates the files neccessary for JavaScript interoperability and for publishing a package to npm
  - This involves compiling the code to wasm and generating a pkg folder

```
pkg
    .gitignore
    app.d.ts
    app.js
    app_bg.wasm
    app_bg.wasm.d.ts
    package.json
    README.md
```

app.js → **JavaScript Bindings**

app_bg.wasm → **WebAssembly Module**

Rust WebAssembly

```rust
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn add(a: f32, b: f32) -> f32{
    a + b
}

#[wasm_bindgen(js_name = showMessage)]
pub fn show_message() {
    alert( s: "Test Message");
}
```

JavaScript Glue Code

```html
<script type="module">
    // Importing WASM as a JS module requires us to call an init function provided by the default export.
    // This is planned to be changed in the future.
    import { default as init, add, showMessage } from "./pkg/app.js";

    const module = await init();
    const sum = add(1,2);
    showMessage();
```

WEBASSEMBLY PERFORMANCE

WebAssembly is not a silver bullet for performance but can enable:

- Faster startup times
- Predictable performance -> no unpredictable garbage collection pauses
- Vectorization based on SIMD instructions (not available in JS)
    - SIMD (Single Instruction Multiple Data) enables efficient data parallelism by performing the same action on multiple data elements concurrently
    - The WebAssembly SIMD proposal defines a portable, performant subset of SIMD operations that are available across most modern architectures

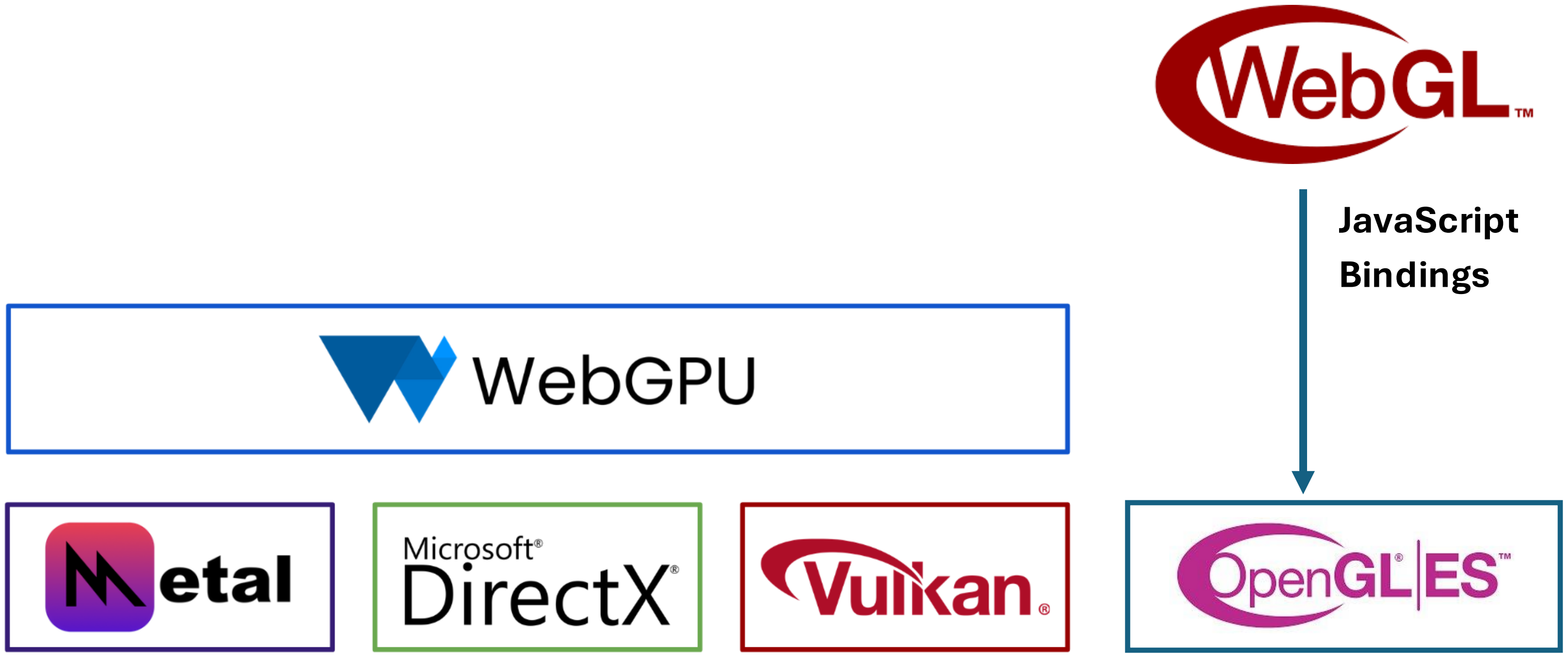
```c
#include <wasm_simd128.h>

void multiply_arrays(int* out, int* in_a, int* in_b, int size) {
  for (int i = 0; i < size; i += 4) {
    v128_t a = wasm_v128_load(&in_a[i]);
    v128_t b = wasm_v128_load(&in_b[i]);
    v128_t prod = wasm_i32x4_mul(a, b);
    wasm_v128_store(&out[i], prod);
  }
}
```

GRAPHICS APIS OVERVIEW

GRAPHICS APIS OVERVIEW

GRAPHICS APIS OVERVIEW



WEBGPU

- The integration of WebGL in browser allowed web applications to take advantage of GPUs for rendering high-performance interactive 2D and 3D graphics
- WebGL is based on the OpenGL family of APIs first developed in 1992
- Since that time GPU hardware has evolved significantly which is not reflected in the WebGL API
- WebGPU is the successor to WebGL bringing the advancements of the modern graphics APIs like Direct3D 12, Metal, and Vulkan to the web
- WebGPU also adds support for compute shaders which enables new classes of algorithms to be ported on the GPU known as GPGPU (CUDA, OpenCL)

WEBGPU COMPUTE SHADER – VECTOR ADDITION

```
const numValues = 1_000_000;
const vec1 = new Float32Array(numValues);
const vec2 = new Float32Array(numValues);
for (let i = 0; i < numValues; i++) {
  vec1[i] = i;
  vec2[i] = i;
}
```

Creating the two vectors which will be uploaded on the GPU as Typed Arrays

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();

const vec1Buffer: GPUBuffer = device.createBuffer({
  mappedAtCreation: true,
  size: vec1.byteLength,
  usage: GPUBufferUsage.STORAGE,
});
const vec2Buffer: GPUBuffer = device.createBuffer({
  mappedAtCreation: true,
  size: vec2.byteLength,
  usage: GPUBufferUsage.STORAGE,
});
const resultVectorSize = Float32Array.BYTES_PER_ELEMENT * vec1.length;
const resultVectorBuffer: GPUBuffer = device.createBuffer({
  size: resultVectorSize,
  usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_SRC,
});
```

Creating three buffers on the GPU; two for uploading the data of the vectors and one for storing the results

```
const stagingBuffer: GPUBuffer = device.createBuffer({
  size: resultVectorSize,
  usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.MAP_READ,
});
```

To read data from the GPU, we copy data from an internal, high-performance buffer to a staging buffer and then map the staging buffer to the host machine, so we can read the data back into main memory

WEBGPU COMPUTE SHADER – VECTOR ADDITION

```
const shaderModule = device.createShaderModule({
  code: `
        @group(0) @binding(0) var<storage, read> firstVector : array<f32>;
        @group(0) @binding(1) var<storage, read> secondVector : array<f32>;
        @group(0) @binding(2) var<storage, read_write> resultVector : array<f32>;

        @compute @workgroup_size(64)
        fn main(@builtin(global_invocation_id) global_id : vec3<u32>) {
            if (global_id.x >= arrayLength(&firstVector)) {
                return;
            }

            resultVector[global_id.x] = firstVector[global_id.x]  + secondVector[global_id.x];
         }
        `,
});
```

Shader code in the WebGPU Shading Language (WGSL) for the vector addition which will be exectued on the GPU

```
const computePipeline = device.createComputePipeline({
  layout: "auto",
  compute: {
    module: shaderModule,
    entryPoint: "main",
  },
});
```

Creating the compute pipeline

WEBGPU COMPUTE SHADER – VECTOR ADDITION

```
const vec1Mapped: ArrayBuffer = vec1Buffer.getMappedRange();
new Float32Array(vec1Mapped).set(vec1);
vec1Buffer.unmap();
const vec2Mapped: ArrayBuffer = vec2Buffer.getMappedRange();
new Float32Array(vec2Mapped).set(vec2);
vec2Buffer.unmap();
```

Uploading the js arrays in the created buffers on the GPU

```
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginComputePass();
passEncoder.setPipeline(computePipeline);
passEncoder.setBindGroup(0, bindGroup);
const workGroupSize = 64;
const workgroupCountX = Math.ceil(numValues / workGroupSize);
passEncoder.dispatchWorkgroups(workgroupCountX);
passEncoder.end();

commandEncoder.copyBufferToBuffer(
  resultVectorBuffer,
  0,
  stagingBuffer,
  0,
  resultVectorSize
);

const gpuCommands = commandEncoder.finish();
device.queue.submit([gpuCommands]);
```

Executing the compute pass

```
await stagingBuffer.mapAsync(GPUMapMode.READ);
/* This is real, mapped GPU memory, meaning the data will disappear (the ArrayBuffer will be "detached"), when stagingBuffer gets unmapped */
const arrayBuffer = stagingBuffer.getMappedRange();
console.log(new Float32Array(arrayBuffer));
```

Transferring the data back from the GPU

Questions?