Investigating Web Mining on WebGPU: A JavaScript API for Accelerated Graphics and Compute

Tal Derei, Marc Ferdinandusse, Xinhao Liu

Abstract

Web Browsers are increasingly using hardware acceleration to perform GPU-accelerated 3D rendering and General-Purpose GPU (GPGPU) computing. Accessing the large-number of computing units residing in an integrated / discrete Graphics Processing Unit (GPU) requires low-level Graphics APIs: 3D graphic libraries like DirectX, OpenCL, OpenGL, WebGL, Metal, CUDA, and WebGPU that expose the capabilities of physical GPU hardware to the web. Browser-based cryptocurrency mining platforms and applications heavily rely on the efficiency of these graphics APIs to facilitate pooled-mining operations off-loaded to the GPU. These APIs achieve this by tapping into the host systems GPU to accelerate mining. In this paper we investigate the feasibility of running traditional mining workloads like *Ethash's* Proof-of-Work (PoW) consensus algorithm and block validation on the WebGPU standard **[12]**. For testing purposes, we abstract these workloads as matrix multiplication and SHA-256/512 hashing subroutines. This serves as a generalized proof-of-concept for mining on WebGPU. Native CPU and GPU benchmarks were compared against the performance using WebGPU, GPU.js (JavaScript Acceleration library for GPGPU), and CUDA.

In our initial testing, we found an average 8x speedup when enabling the WebGPU API to run the matrix multiplication and 4x speedup for hashing workloads on CUDA, compared to native CPU performance. Overall, web mining on WebGPU proved to be significantly more work than originally anticipated, and our attempts weren't entirely successful.

1 Overview

WebGPU is the successor to the WebGL/2 family of JavaScript APIs, supporting native 3D graphics capabilities on the browser. WebGPU is based on Vulkan's standardized API, a lower-level API compared to OpenGL, DirectX 11, and Metal for parallel tasks. Vulkan benefits from higher performance in compute-intensive workloads, offering a more balanced CPU/GPU distribution **[3]**.

Unlike Vulkan, WebGPU implements a custom WebGPU Shader Language (WGSL) specification for both its native source-code and compiler target. Shader languages like WebGL/OpenGL's GLSL, DirectX's HLSL, and WebGPUs WGSL are graphics programming languages written with C-like syntax, representing the actual code that's injected onto the GPU for execution **[2]**. These shaders compile to SPIR-V intermediate language, the binary format for its compilation target developed by Khronos Group, that the GPU can understand **[4]**.

In the context of browser-based crypto- mining, the mathematics underpinning these mining algorithms need to be rewritten in WGSL and packaged into shaders. The WebGPU shaders represent code-blocks that are then executed by compute units inside GPU hardware. Our project benchmarks parallel mining and hashing operations for Ethereum on Chrome Canary (beta version of Chrome) using the WebGPU API. The API enables webpages to use the system's GPU to perform data-parallel computations and render 3D objects (enabled for NVIDIA GPUs with proprietary NVIDIA drivers). By leveraging the fact that Ethereum's network implements a classical Proof-of-Work consensus mechanism, our goal was to benchmark the performance of Ethash on a variety of different GPUs with different parallel hash functions. *Keccak-256/512* (variant of SHA3 standard) is the default hash function used by Ethash. Our proposal includes swapping out the Keccak-256/512 hasher with **(1)** *KangarooTwelse* = Keccak-based parallel hash mode that uses AVX intrinsics and SIMD instructions, and **(2)** *Blake3* = cryptographic hash function faster than SHA-256/512.

The motivation behind implementing PoW Ethash on WebGPU are two-fold: SHA3 hashing is an embarrassing parallel problem that's computationally (not-memory) bound. Therefore it would benefit from GPU acceleration through the WebGPU API, rather than performing CPU-based hashing. Additionally, a key-feature of WebGPU are "Compute Shaders" which help optimize memory-bound programs. And since Ethash introduces memory-hardness properties that make the hash functions ASIC/FPGA resistant, we hypothesized that running Ethash on WebGPU would be more computationally efficient.

2 Background

Key to our project are several main technologies which we will share more detail about in this section. These include the WebGPU API and Ethash PoW algorithm.

2.1 WebGPU Framework

The WebGPU framework falls into 5 distinct categories: GPU Adapters, GPU Memory Space, Bind Group Layouts and Bind Groups, Compute Shaders, and Execution Pipeline. Each of these components plays a role in the execution of a program.

2.1.1 GPU Adapters

A WebGPU "adapter" (GPUAdapter) describes the physical properties of a given GPU, such as its name, extensions, and device limits. Adapters are objects that encapsulate the underlying GPU. A WebGPU "device" (GPUDevice) represents a logical connection to the WebGPU adapter **[1]**. Devices are root owners of all WebGPU objects created from it, and all GPU interaction is done through the device object. The invocation of both adapters and devices is what allows access to the physical GPU hardware **[1]**.

2.1.2 GPU Memory Space

WebGPU supports multiple memory models for CPU and GPU processes: Regular ArrayBuffer, Mappable GPU Buffer, Non-mappable GPU Buffer (or texture), and Shared Memory. For the purposes of this project, we'll focus on Mappable GPU Buffers. A "GPUBuffer" represents a block of memory that can be used in GPU operations. The importance of these buffers cannot be understated: WGSL code is packaged into shaders, and shaders are wrapped in GPUBuffers to be offloaded to the GPU for execution **[1]**.

2.1.3 Bind Group Layouts and Bind Groups

"Bind Groups" in WebGPU defines a set of resources to be bound together in a group, and how the resources are used in shader stages. Each bind group contains an array of buffers or textures, which are passed to the shader as specified by the bind group layout **[6]**. Bind Group Layout defines the input/output interface expected by the shader, and Bind Groups represents the actual input/output buffers for a shader. It's effectively mapping an array of buffers to shader **[6]**. These bind groups are used in the execution pipeline discussed later in the following sections.

2.1.4 Compute Shaders

Unique to WebGPU is the concept of Compute Shaders: mini programs executing on the GPU that only perform computations, as opposed to rendering vertices and drawing triangles. Those are known as fragment / vertex shaders **[5]**. Compute shaders are buffered using GPUBuffers.

2.1.5 Execution Pipeline

WebGPU's graphics pipeline describes all the actions the GPU will perform when acting on a set of data. WebGPU incorporates two pipeline variants: "Compute Pipelines" and "Render Pipelines". Compute Pipeline is an object that describes the compute operation (compute shaders) to be performed on the GPU. Render Pipelines in contrast perform rendering workloads on a set of vertices **[8]**.

2.2 Ethash

Ethash is the planned PoW algorithm for Ethereum 1.0. The general route that the algorithm takes is as follows, cited directly from the whitepaper **[7]**:

(1) There exists a seed which can be computed for each block by scanning through the block headers up until that point.

(2) From the seed, one can compute a 16 MB pseudorandom cache. Light clients store the cache.

(3) From the cache, we can generate a 1 GB dataset, with the property that each item in the dataset depends on only a small number of items from the cache. Full clients and miners store the dataset. The dataset grows linearly with time.

(4) Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache.

3 Design

The design of our system is broken down into our original proposal and revised proposal. There were various challenges during the implementation stage that required us to rethink our initial design.

3.1 Original Proposal

3.1.1 Design

We set up a **GETH** client running with JSON RPC and curl enabled. GETH (Go Ethereum's standalone CLI client for running full client nodes on the Ethereum network) allows users to perform transactions and interact with smart contacts in Ethereum. GETH also includes a default CPU-based miner. The idea was to treat GETH like a server that distributes the PoW (i.e. the puzzle), and allow the browser to be the client that does the mining using a custom Ethash algorithm (written in Javascript and modified to work on the WebGPU API) GPU acceleration.

3.1.2 Problem

GETH already implements its own PoW Ethash algorithm in /consensus/ethash/ folder. Therefore implementing a custom consensus mechanism (Ethash) in GETH requires modifying the GETH core itself — creating a custom GETH client. The modified version of GETH creates a central server that distributes the PoW, and the server listens at a port for any incoming client connections from the browser. Creating a custom GETH client is an enormous undertaking that requires a deep understanding of the client itself. Therefore, we couldn't use the GETH client for the purposes of this project.

3.1.3 Simple Assumption

Rather than building a browser-based Ethereum miner that automatically starts mining when you open the browser (which requires an external client like GETH to feed the browser "work" to do), simply implement the core Ethash algorithm itself to run on the browser using WebGPU and "manually" perform the mining operations. Manually could mean either run Ethash in a while loop and benchmark the performance, or map each function to a button that you can manually control.

Finally, it was time to implement the Ethash codebase to run on WebGPU using GPU adapters and compute shaders. But there were still a number of *challenges* we describe later in the paper.

3.2 Revised Proposal

3.2.1 Challenges With Original Proposal

As we have slowly come to realize, implementing mining algorithms on WebGPU is a hard problem. Trying to Implement the **EthashJS Library** (EthashJS is an javascript-based implementation of the Ethash) on WebGPU was overly optimistic and presented a number of challenges **[11]**:

1. To perform GPU computations on WebGPU, you have to learn two separate languages: **WebGPU Native** and **WGSL** (WebGPU Shader Language). WebGPU Native is used to set up the framework for performing work on the GPU, and WGSL is a shader language for writing the actual code that's executed on the GPU.

2. There are other simpler shader languages like GLSL, HLSL, and SPIR-V (used by WebGL, OpenCL, and Vulkan APIs), but WebGPU doesn't currently support porting external shader languages since WGSL is compiled by a completely different compiler. There are some community maintainers trying to translate GLSL to WGSL, but the beta repositories are unstable. WGSL is ultimately a new shader language specific to WebGPU, and it's difficult to write compute shaders since the documentation is still being developed.

3. The compute shaders are again written in a separate WGSL language. Consequently, WGSL doesn't support importing external javascript libraries! Ethash relies on a variety of external libraries to perform its computations, which are incompatible with WGSL. In order to implement Ethash to run on WebGPU, you would have to translate ALL of the mathematics and hash functions contained in those libraries into WGSL code. This is an enormous task that hasn't been attempted on WebGPU yet.

3.2.2 Revised Proposal

Given all of these challenges, we've decided to shift gears. To compensate for the relative newness of the WebGPU API, we've decided to make the project more about experimenting on this emerging Web API Standard by performing matrix multiplication and hashing operations. These tasks are not trivial, requiring us to write WebGPU Native and WGSL shader code to perform these operations on WebGPU. Therefore, we'll be abstracting the high-level mining operations and focusing on the underlying matrix / vector multiplication and hashing operations. This demonstrates the feasibility of implementing mining algorithms on WebGPU in the future and serves as a proof-of-concept.

4 Implementation

Our implementation took place over several iterations. This section will outline steps taken to complete our goal, and details of our implementation. We will talk through the process of benchmarking different matrix multiplication and hashing implementations, challenges faced in this process, and our eventual decision to exclude WebGPU hashing from final testing.

4.1 Matrix Multiplication

Matrix multiplication workloads were benchmarked for two different implementations: WebGPU API and GPU.js.

4.1.1 WebGPU Program

A matrix multiplication framework was written in WebGPU, including WGSL shaders to perform the mathematical computations remotely on the GPU. The code being referenced is "*gpu_compute.js*". The process begins with initializing a WebGPU context, and then calling navigator.gpu.requestAdapter() to access the GPU. The function call returns a javascript promise which asynchronously resolves with a GPU adapter. Adapter.requestDevice() then returns a promise that resolves with a GPU device.

From there, we allocated buffered memory that will later be dispatched to the GPUs memory space. Calling device.createBuffer() creates a GPU buffer object in a mapped state. Then calling unmap() allows the GPU to take control -- and prevents race conditions where GPU/CPU access memory at the same time.

Then comes the process of writing the WGLS compute shader logic, and calling device.createShaderModule() creates the actual compute shader module to be run on the GPU. Then we set up the compute pipeline, which is an object that actually describes the compute operation we're going to perform. Calling the function device.createComputePipeline() creates the pipeline, and it takes two arguments: (1) bind group layout and (2) compute stage for compute shader.

Then WebGPU requires defining Bind Group Layouts and Bind Groups for mapping the shaders to appropriate buffer objects. Then we call createCommandEncoder() for creating a compute pass encoder used to encode GPU commands that perform matrix multiplication, and associate the pipeline with the encoder. Finally, we dispatch() the kernel function (i.e. shaders) to the GPU. Reading the data buffers back from the GPU involves copying data between buffers.

We then compared the native GPU performance on WebGPU to CPU performance by writing a cpu-based matrix multiplication program. The code referenced in this subsection is "*cpu_compute.js*"

4.1.2 GPU.js Program

GPU.js is a JavaScript Acceleration library for GPGPU – allowing you to access the GPU directly from the Javascript runtime. We referenced the Github repository: https://github.com/gpujs/gpu.js/. Benchmarking GPU.js against WebGPUs performance yielded interesting results described in the next section.

4.2 Hashing

The initial objective involved translating the SHA3 hash function written in C (~ 300 lines of source code) into WGSL **[11]**.

4.2.1 WebGPU Hasher

Hashing on WGSL presented a number of challenges. WebGPU only supports a 32-bit floating point (f32) or integer (u32 / i32) representation of values / bytes. All other types are not useful for bitwise operations. Javascript was a blocker when trying to convert any string input into bytes and then input it into WebGPU as an array of integers. In addition, since WebGPU is a new API, and its main purpose is to render 3D objects on the web, there is little reference when it comes to performing bit array operations.

Even performing simple XOR operations, bit shifting, and padding proved to be difficult. Consequently, we decided to benchmark a CPU and CUDA-based hasher instead. These hashing programs operate on 3 MB of data contained in the *Data/* folder, comprising 20k block headers and nonces from the bitcoin blockchain. In addition to hashing block headers and nonces, these programs perform block validation by comparing the hash of the concatenation (of the header and nonce) to the expected hash in order to validate the blockchain block by block.

4.2.2 CUDA Program

The parallelized CUDA program was written in CUDA C to run on the GPU. CUDA is a proprietary, closed-source API developed by Nvidia for developing parallel code on Nvidia GPUs. In the program, we (1) define the kernel "__global__ void kernel(unsigned char *d, int d_len, unsigned char *out)" that will run our SHA256 function. Cuda kernels execute functions (from host CPU) on device (GPU). (2) Synchronize the GPU device "cudaDeviceSynchronize()". (3) Allocate memory in the GPU with "cudaMallocManaged((void **)&d_c, d_len)". (4) Copy memory buffers from the host machine onto the GPU with "cudaMemcpy(d_c, data, d_len, cudaMemcpyHostToDevice)". (5) Call the kernel to execute the SHA256 function with "kernel<<<1, 1>>>(d_c, d_len, digest_c)". (6) Resynchronize the GPU device. (7) Copy the results back from the GPU device onto the host machine "cudaMemcpy(digest, digest_c, SHA256_DIGEST_LENGTH,cudaMemcpyDeviceToHost)". (8) Finally free previously allocated memory "cudaFree(d_c)".

The CUDA program performs parallel hashing on the number of threads defined in the Kernel, and further executes the *validate() API for* validating blocks in the bitcoin blockchain by comparing the hashes to an expected, correct hash.

4.2.3 CPU Program

The single-threaded, sequential algorithm written in C runs on the CPU, and servers as a control / baseline. The program performs the same hashing and block validation as the CUDA program. Experimental results indicate significant slowdowns in execution time compared to the CUDA implementation.

5 Testing and Results

The following section outlines our benchmark suites, the experimental results we collected, and some observations and evaluations.

5.1 Platforms

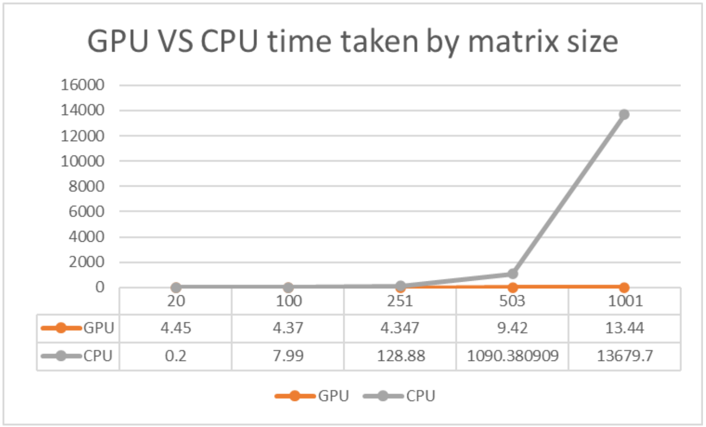
Several platforms were used to benchmark the efficiency of our implementations. These platforms include the ASUS ROG Zephyrus and a Sandbox Linux machine. The ASUS ROG Zephyrus benchmarked the WebGPU and GPU.js matrix multiplication workloads in an isolated docker container. The docker container exposes the underlying GPU drivers.

CPU: AMD Ryzen 9 5900HS
GPU: NVIDIA GeForce RTX 3070
RAM: 16GB LPDDR4 3200MHz

The Sandbox Linux machine uses a workstation-class P1000 Nvidia Quadro, and supports the CUDA runtime / toolkit.

CPU: Intel(R) Xeon(R) CPU E3-1220
GPU: NVIDIA Quadro P1000
RAM: 32GB LPDDR4 2800MHz

5.2 Performance Plots



	20	100	251	503	1001
GPU	4.45	4.37	4.347	9.42	13.44
CPU	0.2	7.99	128.88	1090.380909	13679.7

Figure 1: Plot of the Average Execution Time of the GPU and CPU. x-axis is Matrix Size, y-axis is Execution Time. Average of 50 Iterations for Each Run.

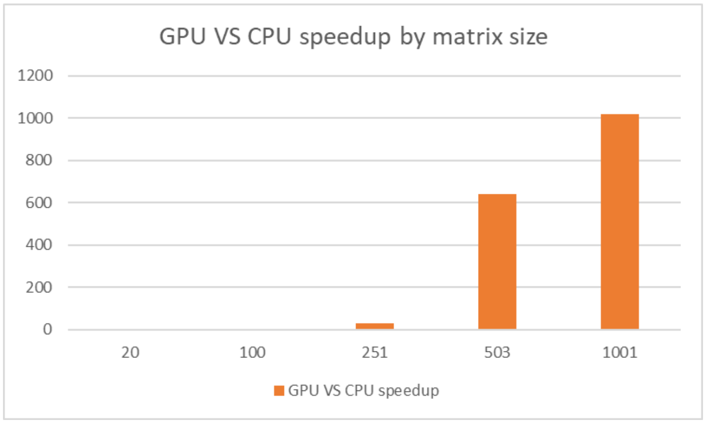
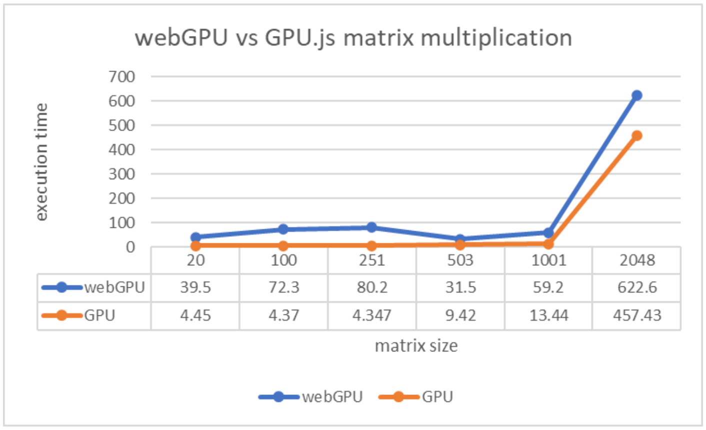
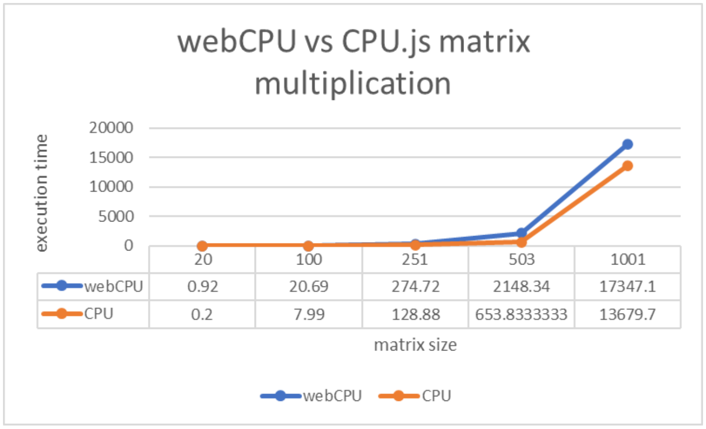


Figure 2: Relative Speedup of the GPU vs. CPU in the GPU.js Implementation



	20	100	251	503	1001	2048
webGPU	39.5	72.3	80.2	31.5	59.2	622.6
GPU	4.45	4.37	4.347	9.42	13.44	457.43

Figure 3: Matrix Multiplication for WebGPU API vs. GPU.js on the GPU. Average of 50 Iterations for Each Run.



	20	100	251	503	1001
webCPU	0.92	20.69	274.72	2148.34	17347.1
CPU	0.2	7.99	128.88	653.8333333	13679.7

Figure 4: Matrix Multiplication for WebGPU API vs. GPU.js on the CPU. Average of 50 Iterations for Each Run.

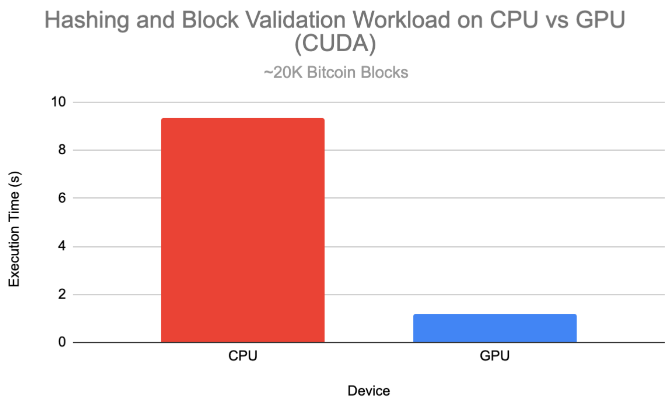


Figure 5: CPU vs GPU (CUDA) Hashing and Block Validation

5.3 Evaluation

Fig. 1. shows the time taken by both the cpu and the GPU for a variety of different matrix sizes. The matrix multiplication execution times shows us when the overhead of invoking the GPU is outweighed by the speedup of the GPU which starts to become clear with a matrix size of 251x251. This information helps inform how we look at the overhead and which benchmarks will be the most telling for the GPU. Fig. 2. shows us the speedups of the GPU so that we can clearly see at what size the GPU becomes more viable than the CPU. These two plots will serve as a source of information for the analysis of the next two plots to see how the overhead might be different for webGPU. Fig. 3. plot shows the execution times for both the webGPU and GPU.js implementations. As you can see in the plot the webGPU has a significant slowdown compared to GPU.js when the matrix size is small. In some of the smallest cases it is even slower than the CPU implementation. Because this speedup shrinks as the matrix gets larger it seems that the webGPU has a larger overhead than the GPU.js.

The speedup of GPU.js is still present even at larger sizes, however the speedup has been reduced significantly. When looking at figure 4 you can see that the webCPU implementation is slower than the web.js this is most likely due to the fact that CPU.js is the current state of the art of web matrix multiplication and therefore utilizes more parallelism. When we look at the webGPU implementation knowing that the webCPU has a significant slowdown due to the more naive implementation there is reason to believe that the webGPU implementation would also see a significant speedup when altered and fine tuned and optimized like the state of the art implementations it would see performance similar to GPU.js. Fig. 4. shows an approximate 4x speedup for performing hashing and block validation (~20K blocks) on CUDA versus the CPU.

6 Conclusion and Future Research

Future research will expand on the framework laid out in this paper by leveraging the hashing and matrix multiplication that this paper established. The SHA-256/512 hashing that was implemented in this paper is the foundation for several elements of future research. To implement Ethereum mining on webGPU GETH needs to be modified to create a central server that distributes the PoW, and the server listens at a port of the incoming client connections from the browser. Another objective would be to implement a more complex parallel hashing algorithm such as KangarooTwelve with webGPU based on this paper's hashing implementation. For now, we'll need to wait for the WebGPU Standard to become more mature for these goals to come to fruition.

7 References

[1] "Webgpu." *W3C*, 14 Dec. 2021, https://www.w3.org/TR/webgpu/.
[2] Manor, Eytan. "The Story of WebGPU-the Successor to WebGL." *Medium*, Medium, 2 Apr. 2021,https://eytanmanor.medium.com/the-story-of-webgpu-the-successor-to-webgl-bf5f74bc036a.
[3] Gpuweb. "Implementation Status · Gpuweb/Gpuweb Wiki." *GitHub*, https://github.com/gpuweb/gpuweb/wiki/Implementation-Status.
[4] Seguin, Damien. "Graphics on the Web and beyond with WebGPU." *Medium*, Medium, 1 July 2020,https://dmnsgn.medium.com/graphics-on-the-web-and-beyond-with-webgpu-13c4ba049039.
[5] "The Pipeline." *Learn Wgpu*, 7 Dec. 2021, https://sotrh.github.io/learn-wgpu/beginner/tutorial3-pipeline/#vertex-fragment-what-are-those.
[6] Usher, Will. *From 0 to Gltf with WebGPU: Bind Groups*,https://www.willusher.io/graphics/2020/06/20/0-to-gltf-bind-groups.
[7] "Ethash." *Ethereum Wiki*, https://eth.wiki/en/concepts/ethash/ethash.
[8] "GPU.js - GPU Accelerated Javascript." *GPU.js - GPU Accelerated JavaScript*, https://gpu.rocks/#/.
[9] RustyBamboo. "Rustybamboo/Hash-Shader: Sha256 Compute Shader (Kernel) Written in Rust." *GitHub*,https://github.com/RustyBamboo/hash-shader.
[10] Brainhub. "Sha3iuf/sha3.c At Master · Brainhub/sha3iuf." *GitHub*, https://github.com/brainhub/SHA3IUF/blob/master/sha3.c.
[11] Ethereumjs. "Ethereumjs/Ethashjs: Project Is in Active Development and Has Been Moved to the Ethereumjs VM Monorepo." *GitHub*, https://github.com/ethereumjs/ethashjs.