Snappy UIs With WebAssembly and Web Workers

Published: 2023-08-07

Our web app allows users can change the length of a song or find loops present in it for infinite listening, remixing, or for their next next video edit or performance. After uploading a song, there is an initial server-side analysis step after which the audio can be manipulated completely in the browser. Users can alter the desired length or mark sections of audio to prefer or avoid which will regenerate results. To make these manipulations responsive and snappy, computations happen client-side and do not need another network call (which would introduce additional latency). To make this possible, we rely on using a WebAssembly binary executed inside of a Web Worker (several of them actually, running in parallel). In this post, we will go over some more details of how this works.

Running Fast(er) With WebAssembly

WebAssembly allows web developers to use low-level code delivered in a binary format that runs at speeds you would not be able to achieve with just JavaScript. WebAssembly (or Wasm for short) describes a low-level assembly-like language that can be targeted from a higher-level language like C, C++, or Rust. The nice thing is that you get to choose which language you want to use, as long as you can find a way to compile it down to a Wasm binary!

We use AssemblyScript which allows us to write our high-level code in TypeScript (strongly typed JavaScript). Since the rest of our client-side code is also written in TypeScript (which in turn gets compiled to “regular” JavaScript), this allows us to share code between the code delivered as part of our UI (using JS) and the binary (using Wasm). We get to share type definitions of the data being passed in and out of the compiled binary and some interop that makes it easier to pass data back and forth. While code written with AssemblyScript looks very similar to TypeScript, it needs some modifications before being able to be compiled with AssemblyScript (by defining more granular types for example).

AssemblyScript code might look something like the following:

```
export function compute(array: StaticArray<f64>, target: i32): Resul
    const total = 10;
    const sum = 0;

    for (let i = 0; i < total; i++) {
        sum += array[i];
    }
    return { sum };
}
```

Here, we are writing a function that sums up the first 10 items from `array` and returns it wrapped in an object. In our code, we return more than just one item inside of this object. The nice thing with AssemblyScript is that it will take care of properly passing this structured data across the JS/Wasm boundary.

Keeping Things Responsive With Web Workers

While the Wasm binary is faster than a JavaScript implementation would be, it still takes a non-neglible time to run. If we were to just call the `compute` function inside of our Svelte front-end, we would run it in the main thread and lock up the UI, causing a bad user experience as the webpage would appear frozen. To fix this, we run that code inside of a Web Worker. Doing this will allow us to run the search algorithm in a separate thread so the main thread is available to respond to the user.

In our case, to perform a search, a worker needs some context on the structure of the song. We first initialize a worker with the song analysis results from the server (this requires us to send data from the main thread into the worker thread), then ask it to execute a search with the target length. Since this context only changes when the user changes the song they are working on, we can reuse a worker given we are editing the same song and just need to vary the target duration or the preferences.

Since that code runs in separate execution context in a separate thread, how can we implement cross-thread communication? Using the Channel Messaging API's `postMessage` method! This method allows us to send a message across the thread boundary.

Our code that manages the worker looks something like the following:

```
import Worker from "./worker?worker";

const worker = new Worker();

// intialize the worker
worker.postMessage({
    type: "populate",
    data: new Float64Array(searchContext),
});
// wait for worker to be initialized (see "loaded" event below)

function search(
    callback: (ratio: Result) => void,
    target: number,
    onProgress?: (ratio: number) => void,
) {
    worker.onmessage = ({ data: { type, data } }) => {
        if (type === "finish") {
            callback(data as Result);
            // check if there are waiting tasks
        } else if (type === "progress") {
            onProgress?.(data);
        }
    };

    // execute search
    worker.postMessage({
        type: "process",
        data: { target },
    });
}
```

The `worker.ts` referenced above (the `?worker` suffix is a Vite feature) has some code to handle the messages from the main thread and copy the search context into its local memory and then run our WebAssembly code when needed:

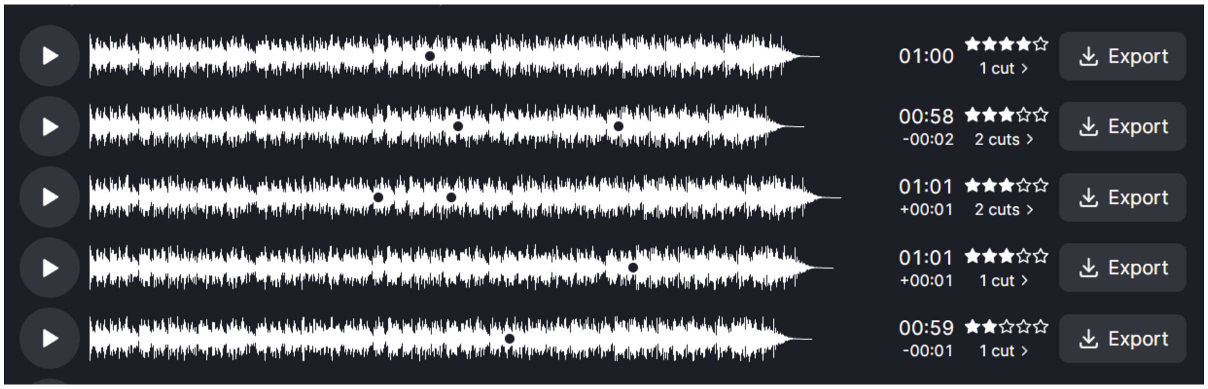
```
let cachedData: Float64Array;

onmessage = async ({ data: { type, data } }: { data: { type: string;
    const { compute } = await import("./wasm/assembly");

    if (type === "populate") {
        cachedData = data;

        postMessage({
            type: "loaded",
        });
    } else if (type === "process") {
        const { target } = data;

        postMessage({
            type: "finish",
            data: compute(cachedData, target),
        });
    }
};
```

Note: as pointed out on Hacker News, if you are running Firefox, the dynamic import above requires Firefox 113 or newer.

As mentioned before, keeping a copy of the search result is possible since the analysis context required is shared across searches. This avoids some overhead when the user initializes a new search (by altering the target length or region preferences) by avoiding having to copy it in for every search.

Reusing Workers

In Mofi, users get to choose from multiple results of generated songs. Since these searches can run in parallel and we want to get results shown to the user as quickly as possible, we run multiple instances of Web Workers at the same time. This allows us to have results “trickle in” as each worker finishes processing.



We don’t want to run too many workers at once, however: after we reach the number of physical cores, the cost of switching between threads starts to outweight the benefits of parallelization and computation incurs a performance hit. To manage this, we use a pool of workers. When a new request comes in to look for a result, we try to find a worker that will handle the request:

- If we haven’t filled the pool of workers yet, we have space to create one, so we initialize a new instance of the worker and populate it with data. Once it has finished initialization, we can send over the request.
- If there is an idle worker in our pool, we take one and let it perform the search. The advantage of this is that reusing the worker allows us to skip initialization since it already has the context needed to perform the search.
- If there is no idle worker, we add the request along with a callback to a list of waiting tasks. Once a worker finishes a task, the scheduler takes the next task off of the queue and processes it.

Having a pool of workers also let’s us stop all searches by terminating all workers:

```
export function terminate() {
    waitingTasks = [];
    busyWorkers.forEach((worker) => worker.terminate());
    busyWorkers = [];
}
```

This happens when the user issues a new search. At this point, all running workers are looking for outdated results so we don’t need them anymore.

Tip: Progress Updates

A worker can only return data once completed, but we also want to show a progress indicator even when its task has not finished. To do this, inside of the worker code, we create a global `onProgress` helper function that allows us to send a message to the main thread like this:

```
(globalThis as any).onProgress = (ratio: number) => {
    postMessage({
        type: "progress",
        data: ratio,
    });
};
```

Then, inside the AssemblyScript code, we can add this:

```
@external("env", "onProgress")
export declare function onProgress(ratio: number): void;
```

which allows us to "import" the function and call it inside of the Wasm binary and send incremental updates before returning the final result:

```
+ import { onProgress } from "./glue";

export function compute(array: StaticArray<f64>, target: i32): Resul
    const total = 10;
    const sum = 0;

    for (let i = 0; i < total; i++) {
        sum += array[i];
+       onProgress(i / total);
    }

    return { sum };
}
```

This is a simplified example and this loop would finish fast, but in the case of long-running loop iterations the progress updates will be more beneficial.

Wrapping Up

Using WebAssembly and Web Workers, we are able to shift the iterative aspect of Mofi's audio manipulation to the client to make the experience more responsive and snappy. This project was exciting for me to build because it got me using novel web technologies to build a performant application that mostly runs in the browser. In the future, I hope to be able to analyze the audio in the browser too, but it looks difficult to do at this stage.

Thank you for reading, and if you haven’t already, please give Mofi a try!

Try Mofi →
More blog entries →