
DTU

23/11/23

Real-Time Visual and Machine Learning Systems

Concurrency – Part 2

- GPU Hand-in
  - Any Questions?
  - Common problems?
    - Bind groups and unused elements
- Analysis Hand-in
  - Any Questions?
- Projects
  - Any Questions?
  - Section/Communication Hours

Concurrency – Part 2 – Branchless Programming

- Not going to lecture on it, to not overwhelm. But if you are interested in performance, I wrote a page on it, will also be moved to Real-Time Systems after the course is done
- In optimizing both single threaded code and GPU code, the same principles apply

Concurrency – Part 2 – Green Threads

- Software defined threads (extremely lightweight, no hardware support) mapped unto hardware threads
- Good for lots of threads doing very little work
- Sometimes we can create our own virtualized version of what is happening in hardware with quite a nice effect, such as memory arenas
- Is not a Rust language feature – is implemented by libraries in software

Concurrency – Part 2 – Async

- Put something in motion
- Do something else in the meantime

  – or
- Wait for it to come back

You have already seen this in the wgpu code. Interactions with the GPU are async. We can block on and synchronize with the GPU, or don’t bother waiting and move on.

Is sort of related to coroutines – takes a closure (anonymous function), can yield and resume explicitly

Concurrency – Part 2 – Async

main.rs | convolution.rs | utility.rs | vector_add.rs

src > utility.rs > are_vectors_equivalent

```rust
}

pub async fn initialize_gpu() -> Option<GPUHandles> {
    // Instantiates instance of wgpu
    let instance: Instance = wgpu::Instance::new(wgpu::InstanceDescriptor {
        backends: wgpu::Backends::all(),
        dx12_shader_compiler: Default::default(),
    });

    // `request_adapter` instantiates the general connection to the GPU
    let adapter: Adapter = instance
        .request_adapter(&wgpu::RequestAdapterOptions {
            power_preference: wgpu::PowerPreference::HighPerformance,
            compatible_surface: None, // We aren't doing any graphics
            force_fallback_adapter: false,
        })
        .await
        .expect("Failed to find a usable GPU!");
```

Concurrency – Part 2 – Async

```
15  fn main() {
16      // Initialize the env_logger to get usueful messages from wgpu.
17      env_logger::init();
18
19      // Is there a compatible GPU on the system?
20      // Use pollster::block_on to block on async functions.
21      // Think of it like this - this is a function which
22      // uses the GPU. With block_on() we are insisting
23      // on waiting until all the interaction with the GPU
24      // and the tasks set in motion on the GPU are finished.
25      if !pollster::block_on(fut: self_test()) {
26          panic!("Was unable to confirm that your system is compatible with this sample!");
27      }
28
29      // Keep track of the handles to central stuff like device and queue.
30      let handles: GPUHandles = pollster::block_on(fut: initialize_gpu()).expect(msg: "Was unsucces
```

Concurrency – Part 2 – Async

main.rs ● | convolution.rs 1 | vector_add.rs

src > main.rs > main

```rust
fn main() {
    // Initialize the env_logger to get usueful messages from wgpu.
    env_logger::init();

    // Is there a compatible GPU on the system?
    // Use pollster::block_on to block on async functions.
    // Think of it like this - this is a function which
    // uses the GPU. With block_on() we are insisting
    // on waiting until all the interaction with the GPU
    // and the tasks set in motion on the GPU are finished.
    if !pollster::block_on(fut: self_test()) {
        panic!("Was unable to confirm that your system is compatible with this sample!");
    }

    // Keep track of the handles to central stuff like device and queue.
    let future: impl Future<Output = Option<…>> = initialize_gpu();
    let handles: GPUHandles = pollster::block_on(fut: initialize_gpu()).expect(msg: "Was unsuccesful i

    assert!(vector_add(&handles));
    assert!(convolution(&handles));
    assert!(matrix_multiplication(&handles));
```

Concurrency – Part 2 – Async

Sort of related to coroutines – takes a closure (anonymous function), can yield and resume explicitly.

Currently an unstable feature in Rust, available in C++ and Go. Good for handling large amounts of input events

```
use coroutine::asymmetric::*;

let mut coro = Coroutine::spawn(|coro, val| {
    println!("Inside {}", val);
    coro.yield_with(val + 1)
});

println!("Resume1 {}", coro.resume(0).unwrap());
println!("Resume2 {}", coro.resume(2).unwrap());
```

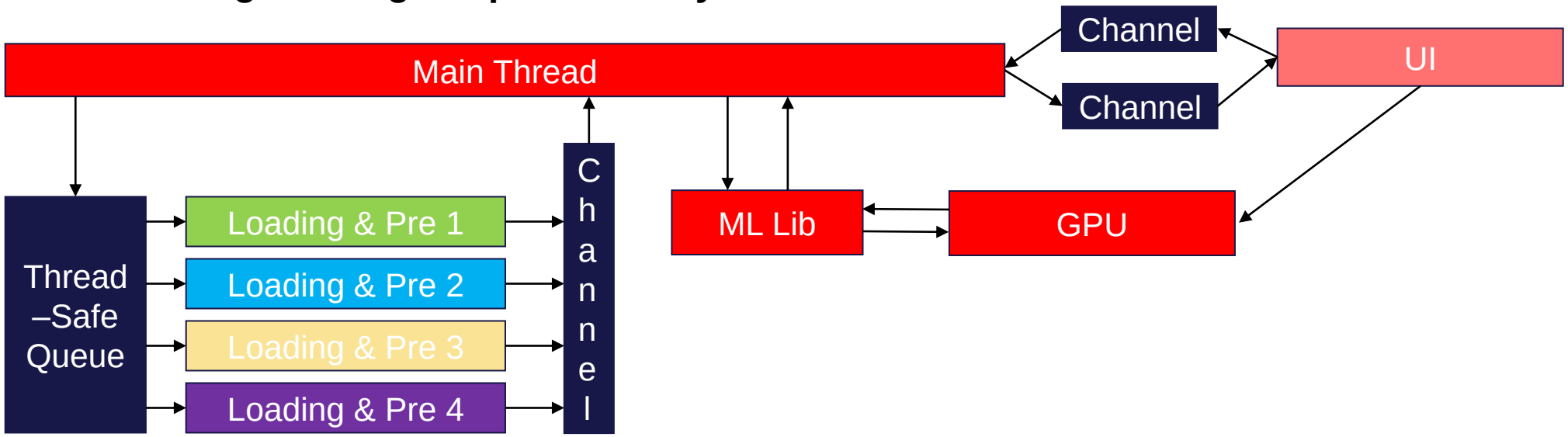
Image credit

Concurrency – Part 2 – Async

- More commonly seen in the handling of disk and networking interactions
- Very common in web servers handling 10000’s of web requests while running on <128 cores
- Will commonly use green threads
- Green threads require a runtime to handle the administration of green threads and the mapping/scheduling onto the actual threads
- Examples include tokio and async-std

Concurrency – Part 2 – Async

- Async in Rust is hard to contain – only async functions are allowed to be called in an async fashion
- Async functions return a handle (a Future<T>), which is a promise that at some point the function will be ready to return T
- We can either block directly on that future and refuse to move on until the Future is ready to return
- **Where might be a good place for async?**



Concurrency – Part 2 – Events

- Not exactly concurrency, will be moved to m5 Real-Time Systems once the course is done
- Events
- Events and subscribers/listeners

Concurrency – Part 2 – Events

- Decode and distribute events
- Can quickly become a tangled mess if you venture far outside of decoding and distributing through channels
- Let’s look at an event loop

Concurrency – Part 2 – GUI

- Thematically would have made more sense at the end, but this is needed for the analysis hand-in
- Retained mode & immediate mode
- Integrated with rendering
- Immediate mode is easiest for prototyping, retained is better for performance
- Let’s look at a bit of GUI

Concurrency – Part 2 – GUI

- Which was made with egui
- egui also has an online demo, go into the underlying code to find out how to use the widgets
- Separation of GPU usage and GUI
- You can also find easy to use visualization libraries like rerun.io from the maker of egui

Concurrency – Part 2 – GUI

- You can also find easy to use visualization libraries like rerun.io from the maker of egui
- Some other frameworks for apps (retained mode) – tauri and dioxus
- Other tools can be found at areweguiyet.com
- Each framework is a skill you might have to learn and structure your code around unless you separate the GUI element completely from the rest of your application