

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?

# 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

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

Currently an unstable feature in Rust, 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());
```

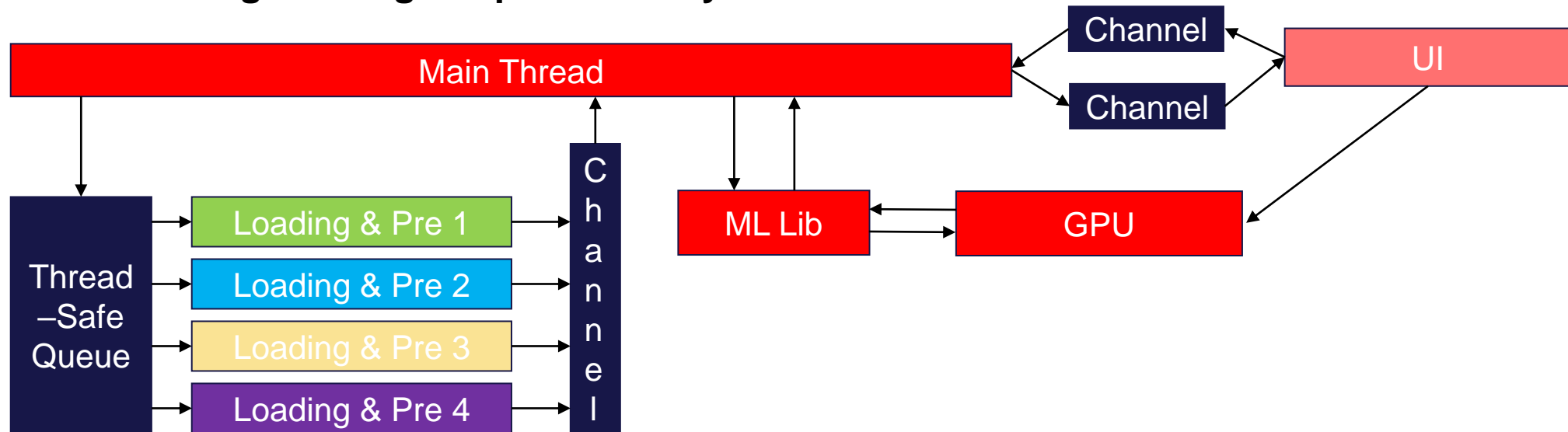
[Image credit](#)

# 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

- Let's look at an [event loop](#)
- Decode and distribute events
- Can quickly become a tangle mess if you venture far outside of decoding and distributing through channels

# Concurrency – Part 2 – GUI

- 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](https://rerun.io) from the maker of egui
- Some other frameworks for apps (retained mode) – [tauri](https://tauri.app) and [dioxus](https://dioxus.net)
- Other tools can be found at [areweguiyet.com](https://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

# Concurrency – Part 2 – Branchless Programming

- I'm not gonna lecture on it, to not overwhelm. But if you are interested in performance, I wrote [a page on it](#)
- In optimizing both single threaded code and GPU code, the same principles apply