DTU

17/11/23

Real-Time Visual and Machine Learning Systems

Concurrency

- Prioritizing Work and Hand-ins
- Walkthrough of the GPU hand-in
- Scattering and gathering
- Walkthrough of the vector add code
- Did everything make sense in the last lecture?
- Hand-in 2 – Analysis

- Concurrency - part 1

Concurrency – Hand-in

- Just like the GPU hand-in, should be handed in at the same time as your project (January 15th) at the latest
- Put it in a folder in your course GitHub repo and write it in Markdown
- IMPORTANT – The point isn't the answer, the point is the attempt
- Two Parts
  - Attempt to analyze a code template which uses an immediate mode GUI, parallelism and some rendering. Don't worry about the graphics, just try to decipher what a box diagram of the template looks like. How many threads are there, who owns what data, at which granularities and when is each part doing its job?
  - Choose 3+ points worth of papers/blog posts from a list and write at least 10 lines of your interpretation of the content per point.

Concurrency

- A series of easy-to-grasp concepts, they are much harder in the wild

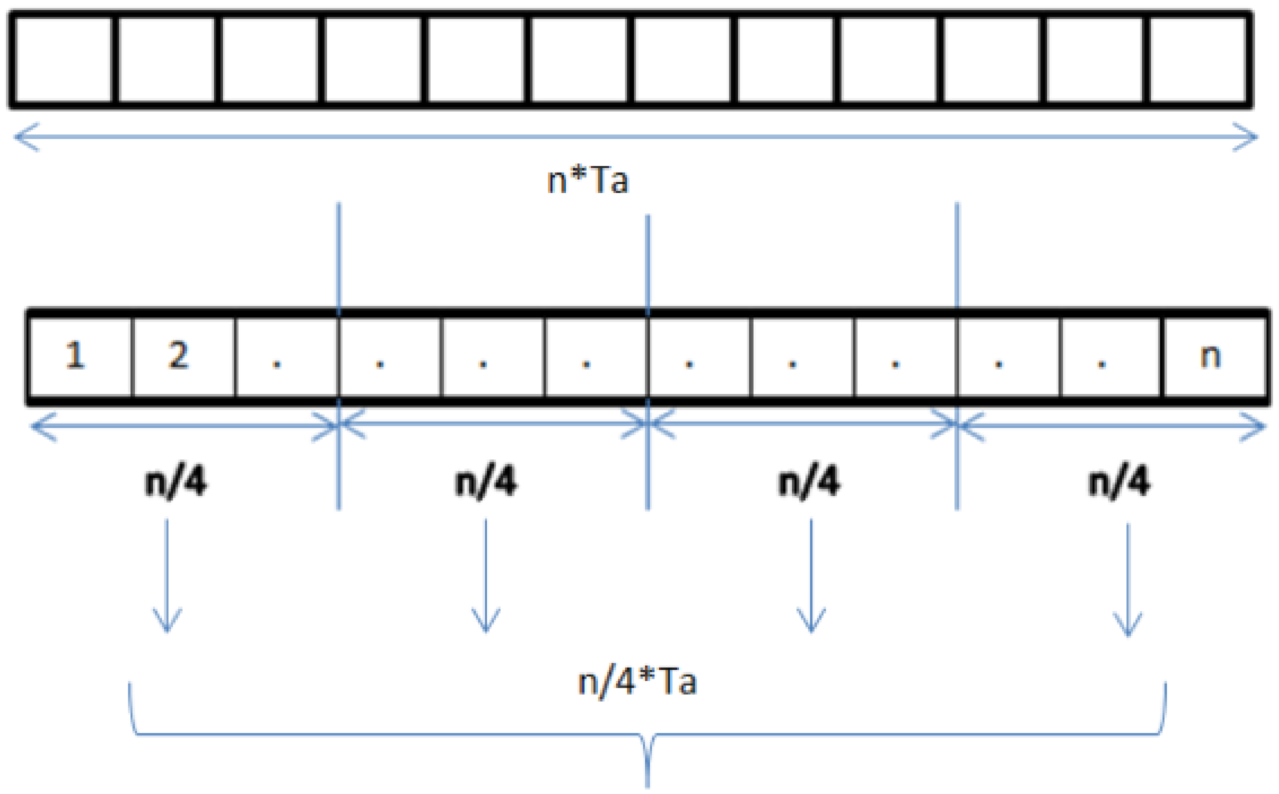
- Image credit

Concurrency – Data Parallelism

- Data parallelism in this case means that our task has few enough interdependencies that we can schedule multiple threads to handle individual subsections of the data without needing synchronization and communication
- Rayon! Drop-in replacement of iterators – iter() => par_iter()
- Work Stealing - The data supplied through the iterator is distributed to threads, if one thread is finished with its workload before the others, work is transferred from one thread to another

Concurrency – Data Parallelism

- let double_data: Vec<i32> = data.iter().map(|x| x * 2).collect();

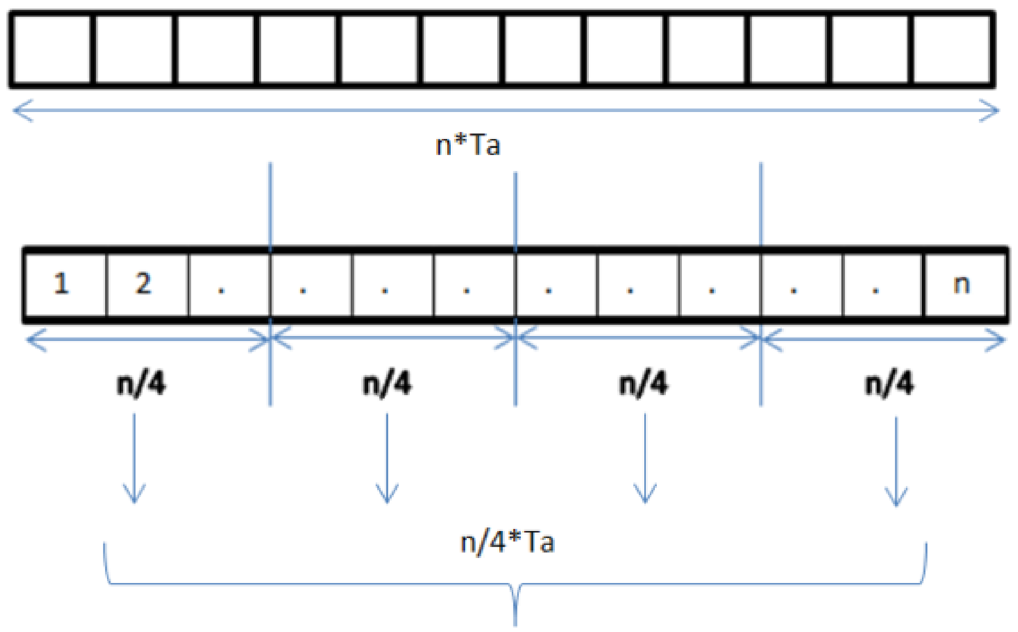


- let double_data: Vec<i32> = data.par_iter().map(|x| x * 2).collect();

Concurrency – Data Parallelism

This would require a lot more work if we had interdependencies

In the case of multiple reads from the original data, we would be fine if we don't do our calculations in-place



Concurrency – Data Parallelism

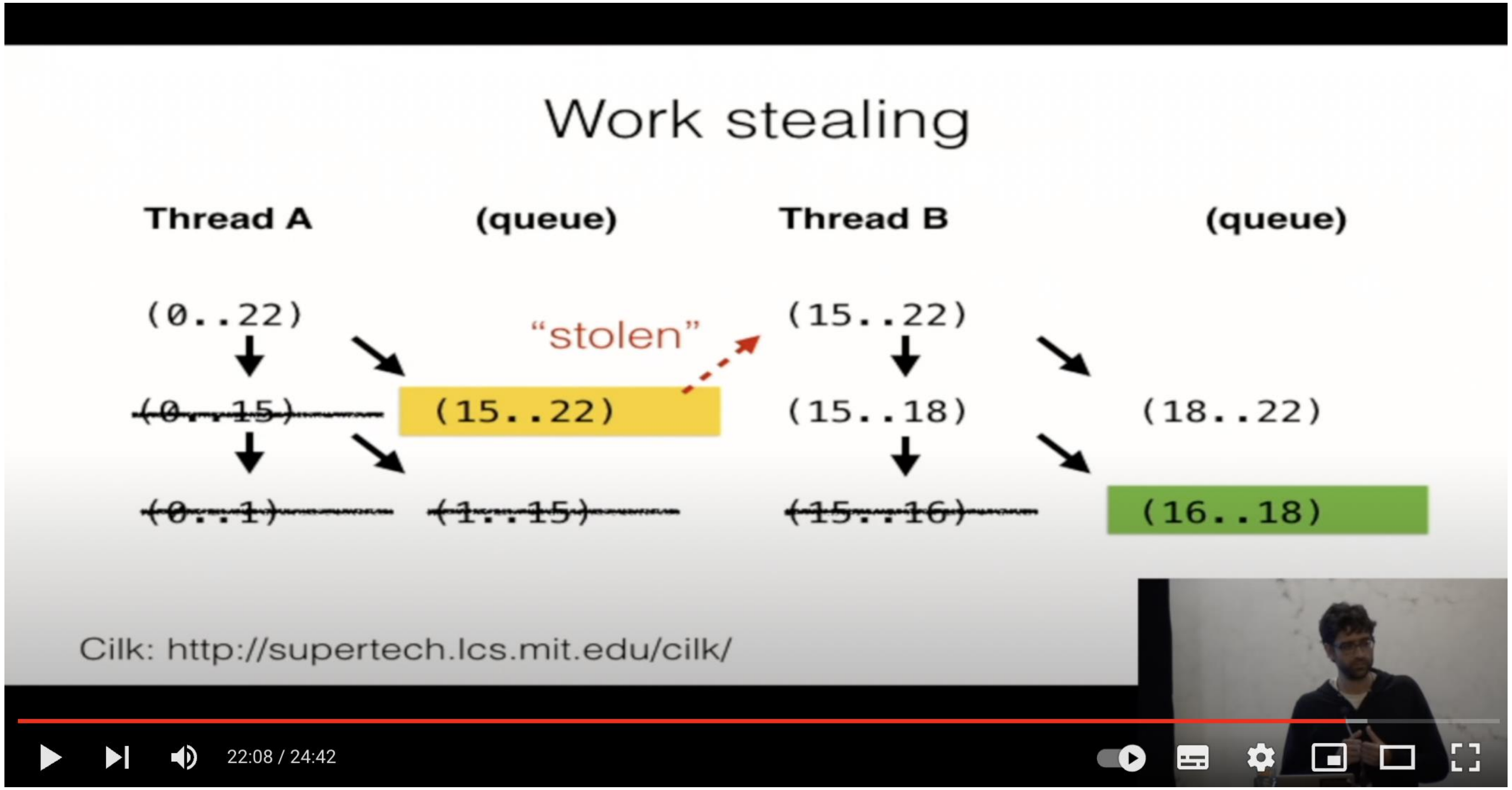


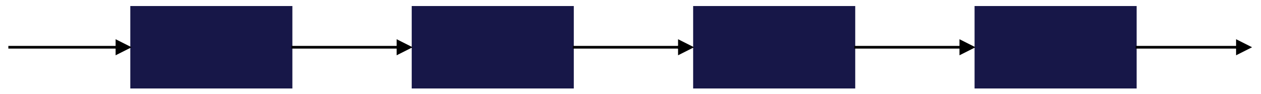
Image Link

Concurrency – Data Parallelism

- Think of work stealing as a thread pool and a scheduler.
- The scheduler must work to distribute and keep track of jobs, it must solve an optimization problem, the more work you give it to solve, the slower it will be.
- Having many small tasks requires lots of work from the scheduler but will also result in poor performance per chunk of work as you will likely see poor caching and branch mispredictions (See s10_branchless_programming).
- You can also benefit from sorting the input data before giving it to Rayon.

Concurrency – Data Parallelism

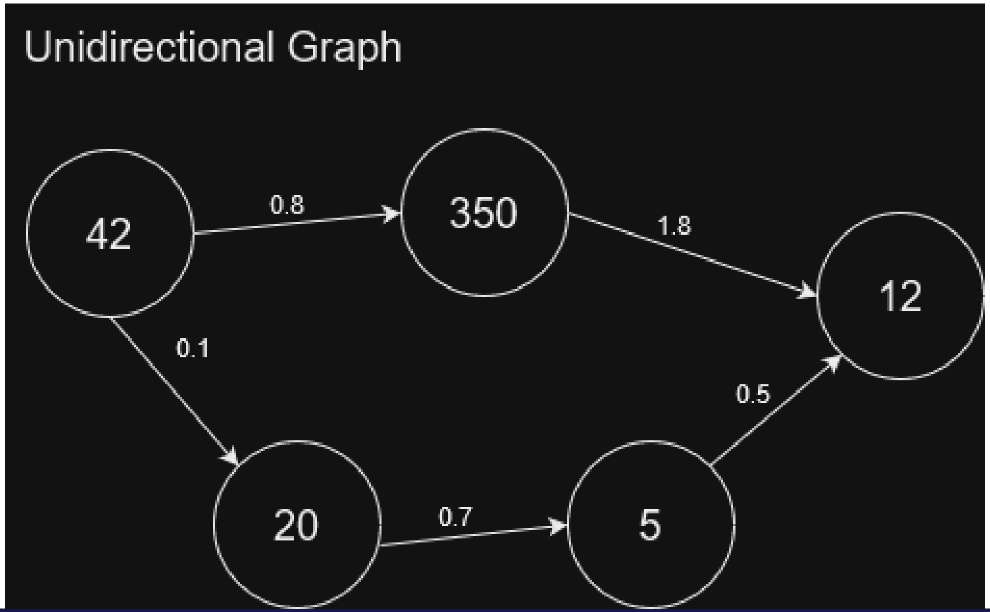
- Rayon is not strictly data parallel; we can still do interprocess communication and synchronization, but it will be just as hard as other parallelism (Arc<Mutex<DATA>>)
- Doing multiple small pockets of parallelism is easy, expect a large cost for scheduling



- If you can formulate your system in a certain way, and your problem is parallel enough, you can also formulate your program in a gathering-oriented fashion and each thread can live happily on its own if writes are thread local – path tracer.

Concurrency – Threads

- So, these threads Rayon are using…
- Threads are expensive to create and destroy
- At most you should target (1-4)xCores
- Even if not currently using them, Rayon can keep the threads around and free them once the application is done using ThreadPool
- We can also create object pools in general – pedestrian system story



Concurrency – Threads

- How you can use threads
  - Spawn
  - Join
- Main thread and child threads
- Green threads (not today)
- In the simplest model all threads are just running their own program, at some point you might join all of them
- The hard part – communication and synchronization

Concurrency – Locks

- Synchronization method
- You either have the lock or you don’t
- A thread can wait at the lock until it can acquire it or it can try the lock and move on
- Once a thread has acquired the lock it can enter the **critical region**

Concurrency – Locks

- It is extremely important that the thread which acquired the lock puts it back
- It is extremely important that you make sure to analyze the critical region to ensure that it puts that lock back, if you do file access, put it in a try region and handle the failure
- If the lock is not put back and other threads are blocking waiting on acquiring the lock you can end up with a **deadlock**
- To optimize performance for locks, minimize the time spent in the critical region and lower the threads-to-locks ratio.

Concurrency – Mutex

- An implementation of a lock
- Mutex stands for Mutual Exclusion
- Software lock around a **critical region**
- In Rust you will find both Mutex and RwLock

Concurrency – Atomics

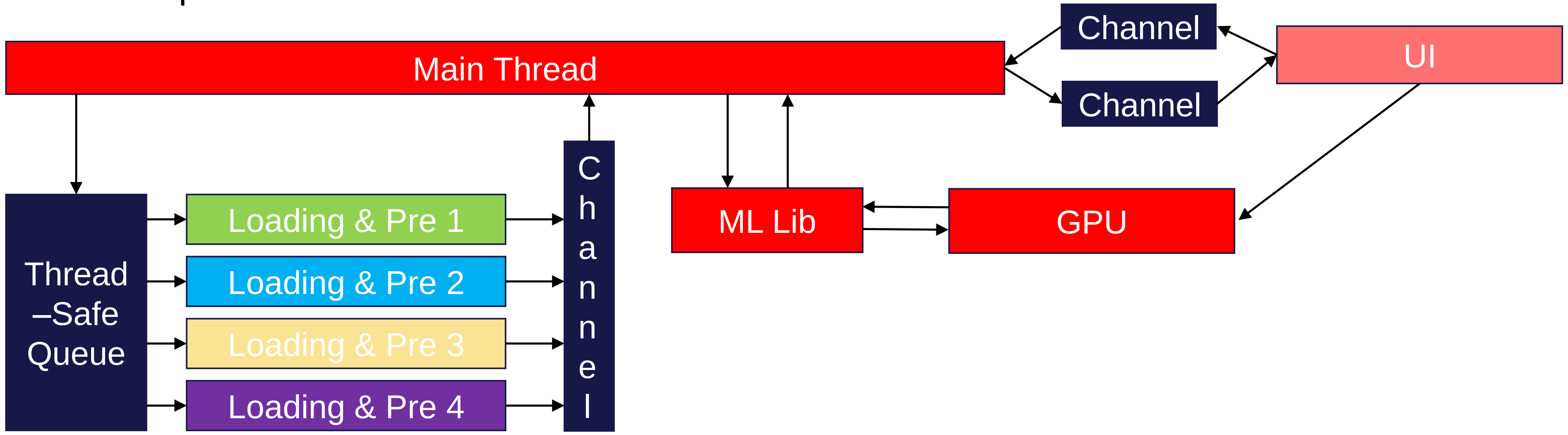
- A very small (usually hardware accelerated) lock wrapping a piece of data (at most 64 bits of data)
- If you don’t have hardware acceleration it will be emulated in software, and quite slow, basically becoming a mutex
- Atomics combine lock, data and operation in a single function call
- Atomics are also usable in your GPU shaders (not relevant for the hand-oin)

Concurrency – Message Passing

- Will focus mainly on channels, but is also commonly seen in MPI
- Ownership of the message is transferred to the channel itself, and acquired on reception by the receiving thread
- mpsc::channel or crossbeam::channel use sender/receiver pairs
- Much like locks you can either block on or try the channel

Concurrency – A bit of structure

- Start writing everything single threaded and optimize that
- If you have some obvious data parallelism, stick with Rayon
- If you have a more heterogenous workload, launch some threads, keep their handles (end the application with joining them) and communicate with channels.
- If you have done that, **maybe** investigate atomics, mutexes and more complicated parallelism



- The loader/preprocessing thread could be using Rayon for preprocessing