MULTITHREADING AND THE GPU PIPELINE
WHAT IS MULTITHREADING?

- Occurs on a single processing unit
  - Multiple virtual threads executed concurrently using shared resources

- Not the same thing as parallel execution (i.e. multiple cores/multiple processing units execute their tasks in parallel)

- Note: hyper-threading is where the OS sees CPU as two logical cores increasing independent instructions
TYPES OF PARALLELISM

- Task parallelism
  - Distributes multiple tasks (jobs) across cores to be performed in parallel

- Data parallelism
  - Distributes data across cores to have sub-operations performed on that data to facilitate parallelism of a single task

- Note: Parallelism is frequently accompanied by concurrency (i.e. multiple cores still have multiple threads operating on the data)
  - We will conflate these two concepts for simplicity in the remaining slides
EMBARRASSINGLY PARALLEL WORKLOADS

- Workloads that can be easily separated into parallel subtasks are called “embarrassingly parallel”

- Some examples:
  - Monte Carlo analysis
  - Numerical integration
  - Graphics rendering
  - Discrete Fourier transforms
  - etc...

- What makes a problem easy to parallelize?
Communication and Dependencies

- Any workload that is not embarrassingly parallel will have associated overhead
  
  - Threads need to communicate
  
  - Threads need to wait on other threads to complete

- Thread management is additional overhead
  
  - Creating and destroying threads is expensive

- Naive parallelization can increase, rather than decrease, execution time
RACE CONDITIONS

- Occur when program behavior is dependent on timing of multiple threads or processes
  - Outcome of execution is non-deterministic
- Hard to identify and debug
  - Requires parallel thinking
  - Behavior is only reproducible some of the time
- Thread-safety indicates that access patterns by threads will not result in a race condition
  - Naive implementation can increase, rather than decrease, execution time
Many development operations in a game engine can be highly parallelized

- Light building
- Level of detail generation
- Code Compilation
- Package building
- etc...

Build times *extremely* important in development

- Full builds of large games can easily take overnight or multiple days
NOT JUST COMPILING...

- C++ compilation is a slow process (particularly if .h files modified) but hardly the bulk of build times

- Working with geometry is extremely time consuming
  - Pre-baking global illumination
  - Performing mesh decimation

- Fortunately these operations can be offloaded to render farms and efficiently parallelized
MULTITHREADING FOR GAMEPLAY

- Many operations within a game can be parallelized
- Some built-in UE4 threads:
  - Gameplay thread manages objects
  - Rendering thread handles graphics (always a frame or two behind the gameplay thread)
  - Audio and audio mixer threads handle playing of audio and mixing of audio respectively (note that they are two separate threads)
  - Physics substepping handled on its own thread
- These are notably task parallel, making them easier to distribute across cores/threads
- What is we want data parallelism?
POOLS AND SCHEDULERS

- Thread pools manage threads to reduce the destruction and creation of workers.
- Job schedulers allocate tasks or subtasks to worker threads to reduce under-utilization of threads.
- At least some thought must be put into both of these to effectively parallelize a job.
CREATING YOUR OWN THREADS

- **FRunnable** is an interface for objects that are run on an arbitrary thread
  - Implement `Init()`, `Run()`, and `Stop()`
  - Use in conjunction with **FQueuedThreadPool** to determine number of threads needed for the task
- **FNonAbandonableTask** used for running non-blocking, asynchronous tasks that cannot be abandoned
  - Other flavors of asynchronous tasks available
WHEN TO THREAD?

- When you are not performant
  - Avoid premature optimization
  - And remember: poor parallelization is worse than no parallelization
- Tasks that are well-suited:
  - Asynchronous loading of assets
  - Calculations that are readily parallelizable
  - Tasks that can be pulled off the main game loop safely
GPUS AND PARALLELISM

- GPUs (Graphical Processing Units) are designed for **throughput architecture**

- Relatively simple cores but a lot of them in parallel!
SHADERS

- Small arbitrary programs that run on GPU
- Massively parallel
- Four kinds: vertex, geometry, tessellation, fragment
**VERTEX SHADER**

- Runs in parallel on every **vertex**
  - No access to triangles or other vertices
- Performs operations such as vertex transformations
  - e.g. apply 4x4 matrices to each vertex
Controls amount of **tessellation** per patch

- Lower poly models can be subdivided into higher resolution models
- Values calculated for generated vertices
- Level of detail (LOD) controllable within the shader pipeline
- Optional
GEOMETRY SHADER

- Takes a primitive and outputs multiple primitives
- Not optimized for subdivision (tessellation shader’s job)
- Ability to work on entire primitive
- Optional
FRAGMENT SHADER

- Runs in parallel on each fragment (pixel) of the rasterized data
  - Can only access neighboring pixel values via textures
- Writes color and depth values per pixel
  - Finalizes appearance of pixels
MODERN GPU CHARACTERISTICS

- Homogeneous programmable cores for all programmable stages
- Relatively few special purpose texture units
- Even fewer fixed function units
- Task parallel at pipeline level
SIMD

- Single instruction, multiple data
- Large vectors of data that have the same operation applied to individual elements in parallel
- Based on old super computing techniques but has regained popularity in modern architectures (both CPU and GPU)
- Same thing is done in parallel for all fragments/verts/etc
- SIMD amortizes instruction handling over multiple ALUs
MULTIPLE TYPES OF PROCESSING

- GPUs do more than shading
  - Allow execution of more than one program
- Replicate SIMD processors for different SIMD computations in parallel

8 programs running in parallel, 128 threads in parallel
PROBLEMS?

- What situations does this throughput style of architecture not handle well?
BRANCHING AND STALLING

- Threads stall when next instruction depends on previous instruction’s result
- Pipeline dependencies
- Memory latency
- How to handle these?
MULTITHREADING

- We can assume there are more threads (scheduled computations) than processors
- Threads with similar code executed in “warps” to maintain minimal divergence
- Interleaving warp execution keeps hardware busy when an individual warp stalls
Threads 1-8

Threads 9-16

Threads 17-24

Threads 24-36

Stall

Waiting

Ready

Extra latency

Extra latency

Stall

Waiting

Ready

Extra latency

Extra latency

Stall
GPGPU

- Can do operations on the GPU besides graphics
  - Heavily used in scientific computing and machine learning
- Potentially useful in games for highly parallel calculations (e.g. physics and AI)
  - Depends on the graphical demands of the game
- Upfront versus amortized costs of sending data between cpu and gpu