# introduction

**Motivation:** Processors haven’t been getting faster for a long time, but you get more cores per chip. Parallelization is required for acceleration and is must be programmed first.

**Parallelism:** Better CPU utilization, More responsive programs, Fair division of CPU resources between different tasks.

Diagram

Description automatically generated

**Levels of parallelism:**

Timeline

Description automatically generated with medium confidence

**Concurrency vs Parallelism:**

|  |  |
| --- | --- |
| Concurrency | Parallelism |
| Decomposition of a program into several sub-programs, which run simultaneously on several processors | Simultaneously or interleaved (time shared) execution that access shared resources. |
| Goal: Faster programs | Goal: Simpler programs |

**Chart

Description automatically generated** **A picture containing icon

Description automatically generated**

## multithreading

**Process:** A process is a program under execution. Process context is everything that an OS needs to run a program correctly. Own address space per Process.

**Process Pros and Cons:**

|  |  |
| --- | --- |
| Pros | Cons |
| Process Isolation | Communication overhead |
| Responsive | New Process creation is expensive |
| Problem with one process doesn’t corrupt another process | Context switching is slow |
|  | Slow process termination |

**Thread:** Parallel sequence within a program/process. Sharing the same address space in the process, but separate stack and registers. Exists within a process and uses the process resources, one process runs at least one thread by default (for the main()). Is lightweight because most of the overhead has been accomplished through the creation of its process.

Text, application, chat or text message

Description automatically generatedLogo

Description automatically generated Graphical user interface, application

Description automatically generated

**Multithreads:** Changes made by one thread to shared system resources (ex. writing into a file) will be seen by all other threads. Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer (else undesired effects)

Diagram

Description automatically generated

**Thread Implementations:** Nowadays always kernel-level threading

|  |  |
| --- | --- |
| User-Level Threads (green threads) | Kernel-Level-Threads (native threads) |
| Scheduled by runtime library or virtual machine | Implemented in the kernel (multi-core exploitation) |
| Implemented in the process (program runtime system) | Native thread implementation can automatically assign work to processors |
| Managed by user |  |

**Thread Scheduling:** Run more threads than processors. Scheduler “assigns” threads to processor to work on.

Diagram

Description automatically generated

**Processor Multiplexing:** Processor executes instructions from multiple threads alternately in partial sequences. Illusion of parallelism of multiple threads even with only one processor

Waterfall chart

Description automatically generated

**Context switch:**

|  |  |
| --- | --- |
| Synchron: Waiting for condition | Asynchron: Timing |
| Thread waiting for condition | Prevent a thread from permanently occupying processor |
| Queues itself as waiting and gives processor free | After some defined time the thread should release the processor |

**Multi-Tasking:** Mostly preemptive

|  |  |
| --- | --- |
| Cooperative | Preemptive |
| Threads must explicitly initiate context switches synchronously at the scheduler at intervals | Scheduler can interrupt the running thread asynchronously via timer interrupt |
| Scheduler cannot interrupt running thread | Time-Sliced Scheduling: Each thread has the processor for maximum time interval |

## java threads

Java is a single process system. Java Virtual Machine (JVM) is a process in the OS. JVM creates a thread at startup, which calls the main method. Programmer can start more threads.

**JVM Termination:** JVM runs as long as threads are running, not after main method is done. Exception when threads are marked as daemon. JVM doesn’t wait for daemon threads (e.g. Garbage Collector). Daemon threads abort uncontrolled at JVM end.

**JVM Threads:** Realized by the Thread class and the Runnable interface.

public static void main(String[] args) {  
 var a = new Thread(() -> *multiPrint*("A"));  
 a.start();  
}  
static void multiPrint(String label) {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(label + ": " + i);  
 }  
}

**OS Perspective:** Java threads are just like any application threads. Scheduling of the threads is handled by the operating system (as well as thread priorities).

**Non-determinism:** Threads run in any order without any precautionary rules (interleaved or parallel). Many JVMs execute System Outputs without interlocking or interleaving. But it is not specified in the JVM specification - Nowhere is it written that it allows concurrency.

**Thread Join:** Wait for a thread to finish

**Passive thread:** Static methods of the Thread class (e.g Thread.sleep(100) or Thread.yield())

**Interrupt:** Threads can be interrupted from outside (e.g myThread.interrupt()). For cooperative cancelling. An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

**InterruptedException:** Thrown when a thread is waiting, sleeping, or otherwise occupied, and is interrupted, either before or during the activity.

# monitor synchronization

**Thread Synchronization:** Non-determinism, Threads run arbitrarily interleaved/parallel without precaution, Concurrent execution often must be limited for correct behaviour (Synchronisation = restriction of concurrency)

**Thread Communication:** Threads communicate primarily by sharing access to fields and the objects reference fields refer to. Extremely efficient but poses a threat of thread interference or memory consistency error.

**Critical section:** Code which is only allowed to be executed by 1 thread at a time. Concurrent and parallel execution of instructions not allowed.

**Mutual exclusion:** Correct implementation is not trivial. Must also take care of weak Memory Consistency. Different threads should NOT have inconsistent views of what should be the same data.

**synchronized:** Body of the method is a critical section. Mutually exclusive execution

class BankAccount {  
 private int balance = 0;  
 public synchronized void deposit(int amount) {  
 this.balance += amount; *// Critical section* }  
 *// or*

public void deposit(int amount) {  
 synchronized (this) {  
 this.balance += amount; *// Critical section* }  
 }  
}

**Guarantees of synchronized:** Impossible for two invocations of synchronized methods on the same object to interleave. When thread t1 is executing a synchronized method for an object, another thread t2 that invokes synchronized methods for the same object is blocked (suspend execution) until t1 is done with the object. When a synchronized method exits, it guarantees that changes to the state of the object are visible to all threads.

**Behind the scene:** Every object has a Lock (Monitor-Lock) and a maximum of 1 thread can acquire the lock. synchronized acquires the lock of the object. By entry is the lock set - otherwise wait till the lock is free (released). Exit releases the lock.

**Recursive Locks:** Same thread can acquire the same Monitor-Lock multiple times through recursive calls. Lock will be free by the last release.

synchronized void limitedDeposit(int amount) {  
 if (amount + balance <= limit) {  
 deposit(amount);  
 }  
}  
synchronized void deposit(int amount) { … }

## monitor locks

**Monitor:** Internal mutual exclusion. Only one thread operates at a time in Monitor. All not-private methods are synchronized, private Variables. Threads can wait in Monitor for condition to be fulfilled. Threads can wake up/signal waiting threads via signalling.

Graphical user interface, text, application, chat or text message

Description automatically generated

**Example:** wait() temporarily releases the Monitor-Lock so that other threads can run

Graphical user interface, text, application, chat or text message

Description automatically generatedclass BankAccount {  
 private int balance = 0;  
  
 public synchronized void withdraw(int amount) {  
 while (amount > balance) {  
 wait();}  
 balance -= amount;  
 } *// Release monitor* public synchronized void deposit(int amount) {  
 balance += amount;  
 notifyAll(); *// Wake up all the waiting threads in monitor inner waiting area* }  
}

**Wakeup signal:** Signalling a condition/thread in Monitor. notify() signals any waiting thread in the inner waiting room. notifyall() wakes up all threads that wait in the monitor.

**Accidental wakeup:** A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. Rarely occur in practice, but applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied.

**Single Notify:** Only one semantic condition (Uniform Waiters) and change applies to only one (One-In/One-Out)

**Drawbacks of Monitor:** Java Monitor can be inefficient. With many different waiting conditions, waiting room is inefficient. Fairness-Problem with simple notify(). Inefficient with notifyAll() und wait() in loops (starvation). No shared Lock (Read-Write Lock)

## semaphore

Allocation of limited number of free resources. A semaphore maintains a set of permits. It is in essence a counter of free resources. A semaphore is coupled with operations to adjust that record safely. Any synchronization ideas can be implemented. Semaphores are relatively low-level. In Buffer we want to avoid inefficient notifyAll().

**Semaphore methods:**

|  |  |
| --- | --- |
| acquire() => P() | release() => V() |
| Acquire a permit | Free a permit |
| Wait if none is available (counter <= 0) | Increment counter |

There is no requirement that a thread that releases a permit must have acquired that

permit by calling acquire().

Diagram

Description automatically generated

**Types of semaphore:**

|  |  |
| --- | --- |
| General Semaphore | Binary Semaphore |
| Counts from 0 to n | Counter 0 or 1 |
| For limited resources/quotas etc. | For mutual exclusion (1 = open, 0 = locked) |

**Fair semaphore:** Uses FIFO-waiting queue for fairness. The longest waiting thread gets the permit (Aging mechanism) slower compared to not guaranteed fair variant. Default is not guaranteed to be fair.

**Example:** uppperLimit shows how many places are free/empty/read. lowerLimit shows how many places are full/written. upperLimit.counter + lowerLimit.counter = Capacity

private Queue<T> queue = new LinkedList<>();  
private Semaphore upperLimit = new Semaphore(Capacity, true);  
private Semaphore lowerLimit = new Semaphore(0, true);  
private Semaphore mutex = new Semaphore(1, true);  
  
public void put(T item) throws InterruptedException {  
 upperLimit.acquire();  
 mutex.acquire();  
 queue.add(item);  
 mutex.release();  
 lowerLimit.release();  
}  
public T get() throws InterruptedException {  
 lowerLimit.acquire();  
 mutex.acquire();  
 T item = queue.remove();  
 mutex.release();  
 upperLimit.release();  
 return item;  
}

## lock and conditions

Monitor with multiple waiting list and conditions. Is independent of Java Monitor. Specific synchronization primitive over API.

A picture containing diagram

Description automatically generated

**Primitives:**

* Lock-Object: Lock for entry in the monitor (Outer/entry waiting list)
* Condition-Object: Wait & Signal for a specific condition (Inner waiting room). Many conditions pro Lock possible

|  |  |  |
| --- | --- | --- |
| Lock |  | Condition |
| lock() | 0..\* | await() |
| unlock() |  | signal() |
|  |  | signalAll() |

**ReentrantLock:** A reentrant mutual exclusion Lock like implicit monitor lock. A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. It is recommended to always immediately follow a call to lock with a try block.

**Condition:** Condition factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations.

**Condition.await():** Causes the current thread to wait until it is signalled or interrupted. The lock associated with this Condition is atomically released and the current thread becomes disabled for thread scheduling purposes and lies dormant until one of four things happens:

* Some other thread invokes the signal() method for this Condition
* Some other thread invokes the signalAll() method for this Condition
* Interrupted
* Spuriously woken up

Throws an InterrupedException if the current thread has its interrupted status set on entry to this method or is interrupted while waiting and interruption of thread suspension is supported. The lock must be freed in case of an interrupt in the finally block.

private Queue<T> queue = new LinkedList<>();  
private Lock monitor = new ReentrantLock(true);  
private Condition nonFull = monitor.newCondition();  
private Condition nonEmpty = monitor.newCondition();  
  
public void put(T item) throws InterruptedException {  
 monitor.lock();  
 try {  
 while (queue.size() == Capacity) {  
 nonFull.await();  
 }  
 queue.add(item);  
 nonEmpty.signal();  
 } finally {  
 monitor.unlock();  
 }  
}  
public T get() throws InterruptedException {  
 monitor.lock();  
 try {  
 while (queue.size() == 0) {  
 nonEmpty.await();  
 }  
 T item = queue.remove();  
 nonFull.signal();  
 return item;  
 } finally {  
 monitor.unlock();  
 }  
}

## readwrite lock

Mutual exclusion is unnecassary for reading only threads/code blocks. Allow parallel reading access (Reader). Mutual exclusion for write access (Writer)

|  |  |  |
| --- | --- | --- |
| Parallel | Read | Write |
| Read | Ja | Nein |
| Write | Nein | Nein |

**Example:**

ReadWriteLock rwLock = new ReentrantReadWriteLock(true);   
rwLock.readLock().lock();   
*// read-only accesses*rwLock.readLock().unlock();  
rwLock.writeLock().lock();   
*// write (and read) accesses*rwLock.writeLock().unlock();

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Lock |  |  |
|  |  | lock() |  |  |
| ReadWriteLock |  | unlock() |  |  |
|  |  |  |  | Condition |
|  |  | Lock |  | await() |
|  |  | lock() | 0..\* | signal() |
|  |  | unlock() |  | signalAll() |

# concurrency hazards

Concurrent programming carries the risk of new types of programming errors that do not exist with single-threading. Very difficult to find by tests as they can occur infrequently.

## race condition and data races

**Race Condition:** Program behaviour depends on relative timing or interleaving of multiple threads or processes. Non-deterministic behaviour which often causes a data race. Multiple threads access shared resources without sufficient synchronization.

**Data Races:** Occurs when two or more threads in a single process access the same memory location concurrently, and at least one of the accesses is for writing, and the threads are not using any exclusive locks to control their accesses to that memory.

**Race Condition without Data Race:** Critical Section is not protected. Data Races is eliminated using synchronization but no synchronization over larger blocks.

**Combinations:**

|  |  |  |
| --- | --- | --- |
|  | Race Condition | No Race Condition |
| Data Race | Erroneous behaviour | Program works, but formally incorrect |
| No Data Race | Erroneous behaviour | Correct behaviour |

**Synchronize everything:** May not help, as a race condition can happen even when synchronize is used. Synchronization is relatively expensive (Optimization hindrances)

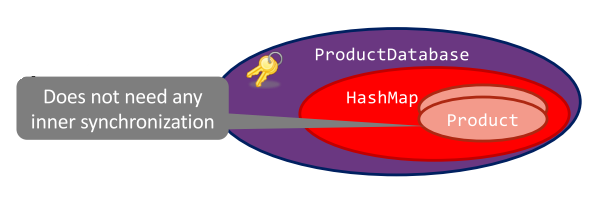
**Avoid Synchronization:** Can be avoided by using read-only objects or confine an object to only 1 thread at a time.

**Confinement:** Structure guarantees that only one thread accesses an object at any point of time

* Thread Confinement: Object only accessible via reference from thread
* Object Confinement: Object encapsulated in other synchronized objects

**Example:**

class ProductDatabase {  
 final HashMap<String, Product> pMap = new HashMap<>();  
 synchronized void addProduct(String name, String det) {  
 pMap.put(name, new Product(det)); *// encapsulated*  
 }  
 synchronized String getProductDetails(String name) {  
 return pMap.get(name).getDetails(); *// encapsulated*  
 }  
 public synchronized void notifySale(String name) {  
 productMap.get(name).increaseSales();  
 }  
}



**Thread Safety:** Thread Safety it the avoidance of data races. When no sharing is intended, give each thread a private copy of the data. No thread safety problem. When sharing is important, provide explicit synchronization to make certain that the program behaves in a deterministic manner (Synchronized, Monitor Locks, Semaphore, etc.)

## deadlock

**Deadlock scenario:**

|  |  |  |  |
| --- | --- | --- | --- |
| Thread 1 | | Thread 2 | |
| synchronized (listA) {  synchronized (listB) {  listB.addAll(listA);  } } | | synchronized (listB) {  synchronized (listA) {  listA.addAll(listB);  } } | |
| Thread 1 | Thread 2 | | Locked Objects |
| synchronized(listA) |  | | listA |
|  | synchronized(listB) | | listA, listB |
| synchronized(listB)  => blocked |  | | listA, listB |
|  | synchronized(listA)  => blocked | | listA, listB |

**Deadlocks:** Threads lock each other, prohibiting both from running. Programs with potential deadlock are not considered correct as they can suddenly block each other.

**Deadlock avoidance:** Introduce linear blocking order. Lock nested only is ascending order. Or use coarse granular locks where ordering does not make sense

Graphical user interface, diagram, text

Description automatically generated Diagram

Description automatically generated

## starvation

A thread never gets chance to access a resources although the resource is free from time to time (no deadlock or livelock) as other threads constantly overtake the thread and snatch the resource. Starvation is a Liveness/ Fairness Problem

**Starvation avoidance:** Fair synchronization constructs where longer waiting threads with fulfilled condition take precedence (aging mechanisms). Enable fairness in Java Semaphore, Lock & Condition, Read-Write Lock.

**Parallelism Correctness Criteria:**

* No Race Conditions: Critical sections on shared resources are sufficiently synchronized (Safety)
* No Deadlocks: Threads cannot lock each other indefinitely (Safety)
* No Starvation: If a thread waits for a condition it should be able to progress after a certain amount of time if the condition is met enough times (Liveness)

# thread pools

**Tasks:** Tasks define potentially parallel work packages. Purely passive objects describing the functionality. Can run in parallel, but they don’t have to.

**Thread Pool:** Workerthreads grab queued tasks from queue and execute them. Any task must complete execution before worker thread is safe to grab another (Exception subtask)

Diagram

Description automatically generated

**Future<T>:** Represents a future result that is to be completed. Acts as a proxy for the result that is possibly not available yet because the task has not finished

Future<T> ft = threadPool.submit(...); *// Launch task async.*  
T result = ft.get(); *// Blocks until task is terminated*

**Example:** Parallel counting

**Timeline

Description automatically generated**

var left = threadPool.submit(() -> count(leftPart));  
var right = threadPool.submit(() -> count(rightPart));  
result = left.get() + right.get();

## .net task parallel library (tpl)

Efficient default thread pool. Multiple abstraction layers such as Task Parallelization (use tasks explicitly), Data Parallelization (parallel statements and queries using tasks implicitly) and Asynchronous Programming.

Task task = Task.Run(() => { *// task implementation* });  
task.Wait(); *// blocks until task is finished*

Task<int> task = Task.Run(() => {  
 int total = ... *// some calculation return total;*});   
Console.Write(task.Result); *// Blocks until task is done*

**Nested tasks:** Tasks can launch and/or wait for sub-tasks

var task = Task.Run(() => {  
 var left = Task.Run(() => Count(leftPart));  
 var right = Task.Run(() => Count(rightPart));  
 return left.Result + right.Result;  
});

**Parallel Statements:** Execute independent statements potentially in parallel. Implicit task barrier at the end.

Parallel.Invoke(  
 () => MergeSort(l, m),   
 () => MergeSort(m, r)  
);

**Data Parallel For Loops:** Execute loop-bodies potentially in parallel. Queue a task for each item in the sequence. Implicit task barrier at the end

|  |  |
| --- | --- |
| Data Parallel Foreach | Data Parallel For |
| Parallel.ForEach(  list,   file => Convert(file) ); | Parallel.For(0,   array.Length,   i => DoStuff(array[i]) ); |
|  | |

**Parallel LINQ:** Parallelization of Language-Integrated Query like Java Stream API. Parallel Processing of collection operations.

**Example:** Add .AsParallel() behind collection to enable parallelism. Add .Ordered() as optional function to guarantee order (slower).

from book in bookCollection.AsParallel()  
 where book.Title.Contains("Concurrency")  
 select book.ISBN;

**Work stealing:**

Diagram

Description automatically generated

**Thread Injection:** TPL adds new worker threads at runtime. Hill Climbing Algorithm measures throughput & varies number of worker threads. Avoids deadlock with task-dependencies

# asynchronous programming

**Kinds of Asynchronisms:**

* Caller-centric (pull): Caller waits for the task end and gets the result
* Callee-centric (push): Task hands over the result directly to successor task

**Continuations:** Define task whose start is linked to the end of the predecessor task

|  |  |
| --- | --- |
| Task Continuation | Multi Continuation |
| Task.Run(t1)  .ContinueWith(t2)  .ContinueWith(t3); | Task.WhenAll(t1, t2)  .ContinueWith(cont); Task.WhenAny(t1, t2)  .ContinueWith(cont); |
|  |  |

**CompletableFuture:** Modern asynchronous programming in Java. Multi-Continuation with CompletableFuture.allOf(fut1, fut2) and CompletableFuture.any(fut1, fut2)

|  |  |  |
| --- | --- | --- |
|  | void Op | Return value |
| Async call | runAsync | supplyAsync |
| Continuation | thenAccept | thenApply |

## non blocking gui

Only a special UI-Thread is allowed to access UI-Components. UI-Thread loop to process the event queue.

Diagram

Description automatically generated

**GUI Premise:** No long operations in UI events (or else blocks UI) and no access to UI-Elements by other threads (or else incorrect).

**Non-Blocking GUI:**

* Classic: Fragmentation of program logic. Chain of dispatches (UI Thread/Worker Thread)
* Async/await: More readable Code. Otherwise fragmented code can be written in a single statement sequence. Fragmentation into several UI-events happens behind the scenes

**Async/Await:** Keyword async for methods. Caller may not be blocked during the entire execution of the async method. Keyword await for tasks. „Non-blocking wait“ on task-end / result.

var url = textBox.Text;  
var text = await DownloadAsync(url);  
label.Context = text;

async Task<string> DownloadAsync(string url){  
 var web = new HttpClient();  
 string result = await web.GetStringAsync(url);  
 return result;  
}

**Execution Model:** Async method runs partly synchronous, partly asynchronous. Caller executes synchronously as long as there is no blocking await. Subsequently, the method runs asynchronously

Diagram

Description automatically generatedasync Task<string> DownloadAsync(string url){  
 var web = new HttpClient();  
 Task<string> task = web.GetStringAsync(url);  
 string text = await task;  
 return text;  
}

**Mechanism:** Compiler dissects method into segments. First segment before await is synchronously executed and segment after await runs after completion of the task.

A picture containing graphical user interface

Description automatically generated

**Caller View of Async Method:** Method runs synchronously until await. If await blocks, execution of the calling method continues, the rest of the called method executes

Asynchronously

Chart

Description automatically generated

**Different Execution Scenarios:**

* Case 1: Caller is a «normal» thread (Usual case, e.g Console, TPL). Continuation is executed by a TPL-Worker-Thread
* Case 2: Caller is a UI-thread. Continuation is dispatched to the UI thread and processed by the UI-Thread as event

Diagram

Description automatically generated with low confidence

A picture containing graphical user interface

Description automatically generated

**Async Return Value Types:**

* Void: “Fire and forget”
* Task: No return value, allows waiting for end
* Task<T>: For methods having return value of type T
* No support for ref or out Parameters

**Special rules:** Await operator only allowed in async methods (otherwise syntax error). Async method must contain await (otherwise syntax error)

# memory model

**Lock-Free Programming:** Correct concurrent interactions without using locks. Use guarantees of memory models. Goal is efficient synchronization

**Problems:** Memory accesses are seen in different order by different threads except when synchronized and at memory barriers (Weak Consistency). Optimizations by compiler, runtime system and CPU. Instructions are reordered or eliminated by optimization.

A picture containing text, sign

Description automatically generated

**Java Atomicity Guarantees:** A single read/write is atomic. Primitive data types up to 32 Bit. long and double only with volatile keyword atomic.

**Example without:** The problem with the code is that atomicity doesn’t imply visibility. One thread may not see updates of another thread at all or possibly much later. Results in an endless loop.

A picture containing graphical user interface

Description automatically generatedclass Worker extends Thread {  
 private boolean doRun = true;  
  
 @Override  
 public void run() {  
 while (doRun) {…}  
 }  
  
 public void stopRequest() {  
 doRun = false;  
 }  
}

**Java Visibility Guarantees:**

* Lock Release & Acquire: Memory writes before release are visible after acquire
* Volatile Memory: Memory writes up to and including the write to volatile variable are visible when reading the variable
* Thread/Task-Start and Join: Start: input to thread; Join: thread result
* Initialization of final variables: Visible after completion of the constructor

**Visibility Unlock -> Lock:** In case Lock and Unlock are applied to the same object

Diagram

Description automatically generated

**Visibility of volatile Write -> Read:** In case Read and Write concern the same variable

Diagram

Description automatically generated

**Java Ordering Guarantees:**

* Ordering guarantees given from inter-thread visibility   
  (Unlock -> Lock, Volatile write -> volatile Read) = Partial Order
* In addition: Synchronization operations are never reordered. Lock/Unlock, volatile-accesses, Thread-Start/Join = Total Order

**Rendez-Vous in Java:** Correct in Java, because a and b are volatile

|  |  |
| --- | --- |
| Thread 1 | Thread 2 |
| volatile boolean a = false, b = false | |
| a = true; while (!b) {} | b = true; while (!a) {} |

**Atomic Classes:** Classes for Boolean, Integer, Long, References and Array Elements. Different kinds of atomic operations (addAndGet(), getAndAdd(), etc.).

do {  
 oldValue = var.get(); *// Read current value* newValue = calculateChanges(oldValue);  
} while (!var.compareAndSet(oldValue, newValue)); *// Write only when value hasn’t changed*

**.NET Memory Model:** Difference to Java Memory Model are long/double not atomic with volatile (Atomicity) and there are only half and full fences (Ordering/Visibility)

**.NET Volatile: Half Fence:**

* Volatile Write: Release semantics. Preceding memory accesses are not moved below it
* Volatile Read: Acquire semantics. Subsequent memory accesses are not moved above it

Graphical user interface

Description automatically generated with medium confidence

**.NET Memory Barrier: Full Fence:** Disallows reordering in both directions

Text

Description automatically generated with medium confidence

**Rendez-Vous in .NET:**

|  |  |
| --- | --- |
| Thread 1 | Thread 2 |
| volatile bool a = false, b = false; | |
| a = true; Thread.MemoryBarrier(); while (!b) { } | b = true; Thread.MemoryBarrier(); while (!a) { } |

# gpu

A GPU is a circuit designed to accelerate the creation of images for display. They are efficient at manipulating graphics and processing images due to their parallel structure, making them superior to CPUs for algorithms that process large blocks of data in parallel.

**High Parallelization:** Offer large number of cores (512, 1024,…) compared to CPU (4, 8,…) but are slower than CPU cores.

**Latency vs Throughput:** Tradeoff between latency and throughput. Increasing throughput by pipelined processing, latency most often also increases.

**Compute bound vs Memory bound:** Performance is defined by memory and compute bandwidth. If compute time is longer, the function is compute bound, if memory time is longer then it is memory bound. If operation is memory-bound, tweaking params to utilize processor more efficiently is ineffective.

**Arithmetic intensity:** , higher AI is better

* More calculations relative to the number of memory accesses -> compute bound.
* Larger time spent on memory accesses relative to the calculations -> memory bound

**SIMD:** Streaming Multiprocessor is in principle SIMD (Single Instruction Multiple Data). Cores perform the same Instructions on different/multiple data elements/memory locations

**Diagram

Description automatically generated**

**NUMA Model:** Non-Uniform Memory Access. No shared/common main memory between CPU and GPU (only explicit). Different Instructionset/Architecture. Compile and design code Code for GPU

**Icon

Description automatically generated**

## cuda

Computer Unified Device Architecture is aparallel Computing platform and API that allows the host program to use certain types of devices/co-processors like graphics processing units (GPUs) for general purpose processing. Software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

**Heterogenous Computing with CUDA:** A typical CUDA program allocates GPU memory, copies data from CPU to GPU, launch kernel on GPU to process the data.

Diagram

Description automatically generated

**Example Vector Addition:**

*//Sequential*

void VectorAdd(float\* A, float\* B, float\* C, int N){  
 for (int i = 0; i < N; i++){  
 C[i] = A[i] + B[i];  
 }  
}

*//Parallelize: N Threads*PrperThread i (i = 0 .. N-1):  
 C[i] = A[i] + B[i];

## cuda compilation flow

**CUDA memory functions:**

* cudaMalloc(\*p, size): Allocates space in Device Global Memory
* cudaFree(\*p): Deallocates space in Device Global Memory
* cudaMemcpy(\*target, \*source, size, destination): Copies data/memory between CPU/GPU

**JIT compilation:** By specifying a virtual code architecture instead of a real GPU, nvcc postpones the assembly of PTX code until application runtime, at which time the target GPU is exactly known. Disadvantage is increased application startup delay, but can be alleviated by letting CUDA driver use a compilation cache

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| virtual compute architecture | NVCC |  |  |  |
| x.cu (device code) |  |  |  |
|  |  |  |  |
| Stage 1  (Ptx Generation) |  |  |  |
|  |  |  |  |
| x.ptx |  |  |  |
| real sm architecture |  |  | CUDA Runtime |  |
| Stage 2  (Cubin Generation) |  |  |  |
|  |  |  |  |
| x.cubin |  | Execute |  |

## cuda execution model

**CUDA Blocks:** Threads are grouped in blocks. Host code can define how many threads each block has (upto a certain limit of 1024 threads). Is observable/accessible from CUDA/Programming model. Threads in one block can interact with each other.

**CUDA Execution Model:** Thread = virtual scalar processor, block = virtual multiprocessor. Blocks must be independent, run to completion, have freedom of order of execution.

**Thread Hierarchie:**

Diagram

Description automatically generated

**CUDA Thread Pool abstraction:** CUDA runtime can choose how to allocate these blocks to multiprocessors as shown with streaming multiprocessors (SMs). For a larger GPU with eight SMs, each SM gets one CUDA block. This enables performance scalability for applications with more powerful GPUs without any code changes.

**Block Grid to Hardware:** CUDA makes guarantees about when/where thread blocks will run. All threads in a block run on the same SM at the same time. Blocks in a kernel finish before any block from a new kernel can be started.

## cuda kernel specification

**Specifying Kernel:**

* VectorAddKernel<<< GRID\_dim, block\_dim>>>(A, B, C): Can be specified via dim3
* struct dim3{x;y;z}: Structure defined for storing block/grid dimensions

**1D Grid:** Simply use example above with 1 Gridsize and N threads

**2D Grid:** VectorAddKernel<<<gridSize(3,3), blockSize(3,3)>>>(d\_A, d\_B, d\_C)

Table

Description automatically generated with medium confidence

**Device limits:** Maximum number of threads in a block is 1024. This is the product of the dimensions xyz. (32, 32, 1) creates a block of 1024 threads. Maximum x/y dim is 1024. For z this is only 64.

**Data Access:** Each kernel decides which data to work on. Problem is that threadIdx.x starts at 0 for each block, so both blocks would calculate same thing. Solution is an index based on blockID and threadID (blockIdx.x \* blockDim.x + threadIdx.x)

Diagram

Description automatically generated

## mapping threads to warps

**Warps:** Blocks are allocated internally in Warps on device. (1 Warp = 32 Threads). Hardware groups threads that execute the same instruction into warps. Several warps constitute a thread block, several thread blocks are assigned to Streaming Multiprocessor (SM) and several SM constitute the whole GPU. Once a thread block is launched on a multiprocessor (SM), all its warps are resident until their execution finishes. Thus, a new block is not launched on an SM until there is sufficient number of free registers and shared memory for all warps of the new block

**Warp Execution:** All threads in a warp execute the same instruction. Branches are executed alternately. SM can fit all warps of a block , but only few are running in parallel (1 to 24).

**Divergence:** Different branches in the same warp. SM executes instruction of one branch and the other threads must wait. (Performance Problem)

**DRAM:** The global memory of a CUDA device is implemented using DRAMs that use parallelism to increase data access rates. When a DRAM location is accessed, many consecutive locations are accessed in parallel due to the sensors provided in each DRAM chip. If an application can make focused use of data from consecutive locations, the DRAMs can supply data at a much higher rate than if a truly random sequence of locations were accessed.

**Branching in Kernel:**

|  |  |
| --- | --- |
| Bad Case | Good Case |
| Divergence in the same warm | Same branch within warp |
| if (threadIdx.x > 1) {…} else {…} | if (threadIdx.x / 32 > 1){…}  else {…} |

**Memory Coalescing:** Thread access patterns are critical for performance. If threads access 32-byte areas, then combined access per area (burst), otherwise expensive individual accesses if threads access different bursts.

## memory model

**CUDA Memory Model Details:** All threads have access to the same global memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory. Each thread has private local memory. The local memory space resides in device memory. Local memory accesses have the same high latency and low bandwidth as global memory accesses.

Diagram

Description automatically generated

**Memory Hierarchie:**

|  |  |
| --- | --- |
| Shared Memory | Global Memory |
| Per Streaming Multiprocessor | «Main Memory» in GPU Device |
| Fast roughly. 4 cycles | slow about 400-600 cycles |
| Visible/accessible/shared between threads in 1 block | Accessible to ALL threads |
| A few KiloBytes | In GigaBytes |
| \_\_shared\_\_ float x; | cudaMalloc() |

**Global vs Registers:** The global memory is off the processor chip and is implemented with DRAM technology, which implies long access latencies and relatively low access bandwidth. The registers correspond to the “register file” of the von Neumann model. It is on the processor chip, which implies very short access latency and drastically higher access bandwidth. In a typical device, the aggregated access bandwidth of the register files is about two orders of magnitude of that of the global memory.

**Registers vs Shared:** Shared memory is on-chip, can be accessed with lower latency and higher bandwidth than global memory. Because of the load store operations that need to be performed, shared memory has longer latecy and lower bandwidth than registers. However, registers are private to a thread. Shared memory is a scratch pad memory shared between threads.

**Constant Memory:** Constant variables are often used for variables that provide input values to kernel functions. Constant variables are stored in the global memory but are

cached for efficient access.

**Reduce global memory traffic:** We have an intrinsic trade-off in the use of device memories in CUDA: global memory is large but slow, whereas the shared memory is small but fast. A common strategy is partition the data into subsets called tiles so that each tile fits into the shared memory. An important criterion is that the kernel computation on these tiles can be done independently of each other.