1 Synchronization Mechanism

Lock & Conditions

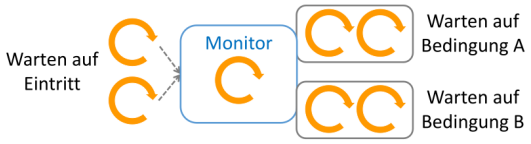

Figure 1: Lock Conditions

```
public class WarehouseWithLockCondition
    {
    public WarehouseWithLockCondition(
        int capacity, boolean fair) {
        lock = new ReentrantLock(fair);
        nonEmpty = lock.newCondition();
        nonFull = lock.newCondition();
    }

    @Override
    public void put(int amount) throws
        InterruptedException {
        lock.lock();
        try {
            // nonFull.await();
            // nonEmpty.signalAll();
        } finally {
            lock.unlock();
        }
    }

    @Override
    public void get(int amount) throws
        InterruptedException { /* code */
        }
}
```

Read-Write Locks

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

CountdownLatch

```
CountDownLatch waitForAll = new
    CountDownLatch(this.CARS);
protected void test() throws
    InterruptedException {
    waitForAll.countDown();
    waitForAll.await();
}
```

Cyclic Barrier

```
var gameRound = new CyclicBarrier(5);

/* 5 different players / threads */
while (true) gameRound.await();
```

Rendez-Vous

- Without exchange: `new CyclicBarrier(2`
- With exchange: `Exchanger.exchange(som`

2 Thread Pool

Java

```
var threadPool = new ForkJoinPool();
Future<Integer> future = threadPool.
    submit(() -> {
        int value = 1;
        return value;
});
Integer i = future.get();

class MyTask extends RecursiveTask<
    Integer> {
    boolean finished = false;
    @Override
    protected Integer compute() {
        if (finished) return 1;
        var left = new MyTask();
        var right = new MyTask();
        left.fork();
        right.fork();
        return left.join() + right.join
            ();
    }
}
```

.NET

```
Task task1 = Task.Run(() => { /* Do some
     stuff */ });
task1.Wait(); // blocking

// Task with return value
Task task2 = Task.Run(() => { return
    3;});
Console.Write(task.Result); // blocking

// Task with Sub Tasks
Task.Run(() => {
    var left = Task.Run(() => Count(
        leftPart));
    var right = Task.Run(() => Count(
        rightPart));
    int result = left.Result + right.
        Result;
    return result;
});

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

// Parallel Loop
Parallel.ForEach(list, file => Convert(
    file));
```

```
// Parallel For - only if iterations are
    independend
Parallel.For(0, array.Length, i =>
    DoComputation(array[i]));
```

3 Async

Continuation Exception in Fire & Forget are ignored. To handle exception you have to wait synchrony for finishing the task.

```
Task.Run(task1).
    ContinueWith(task2).
    ContinueWith(task3).Wait();
Task.WhenAll(task1, task2).
    ContinueWith(continuation).Wait();
Task.WhenAny(task1, task2).
    ContinueWith(continuation).Wait();
```

4 Memory Modell

Atomicity Single reads / writes are atomic for:

- primitive data types until 32 bits
- object references
- long and double only with the `volatile` keyword

Visibility Java guaranties the following visibility:

- changes before release are visible at acquire
- changes until write are visible at read
- the thread sees the correct start values and Join the output of the thread
- initialization of final variables (only relevant if you get the object from a data race!)

Ordering The order of the visibility is the same as in visibility. Additional:

- synchronization instructions are never reordered to each other
- Lock/Unlock, volatile, thread start / join are never reordered
- if everything is a synchronization mechanism than we talk about total order

5 GPU

latency how long does it take to execute a single instruction / operation

throughput number of instructions / operations completed per second

Arithmetic Intensity

$$
\begin{aligned}
&t_c > t_m \\
&\frac{ops}{BW_c} > \frac{bytes}{BW_m} \\
&\frac{ops}{bytes} > \frac{BW_C}{BW_m} \\
&\text{Arithmetic intensity} = \frac{BW_c}{BW_m}
\end{aligned}
\tag{1}
$$

Thermilogy

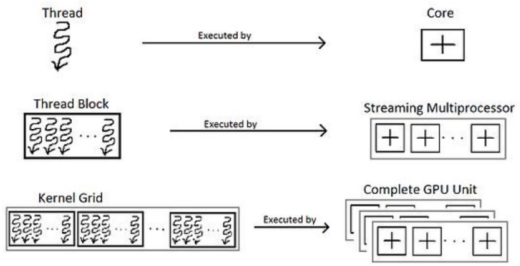

Figure 2: CUDA concepts on the GPU

Classification

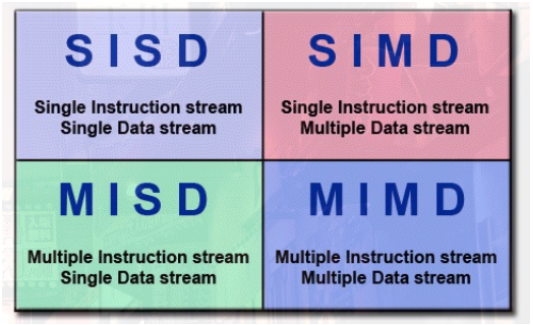

Figure 3: Flynn's Classical Taxonomy

6 Laws

Amdahls's law

$$
\text{SpeedUp} = \frac{1}{s + \frac{p}{N}} \tag{2}
$$

Gutafson's law

s serial part
p parallel part
N number of processes

$$
\text{SpeedUp} = s + p \cdot N = s + (1 - s) \cdot N \tag{3}
$$