Chapter 2 - Solutions

1. Zero times faster, in fact it may even run slower. I have neglected to add the `loop` clause or directive to the `parallel` directive or the loop itself, so the loop will be run in its entirety on all the threads in the GPU. It will not be split across threads, so no performance benefit can be achieved. The correct code would be something like this:

```
#pragma acc parallel loop
for(i=0;i<2359296;i++){
   answer[i] = part1[i] + part2[i] * 4;
}
```

or

```
#pragma acc parallel
{
#pragma acc loop
for(i=0;i<2359296;i++){
   answer[i] = part1[i] + part2[i] * 4;
}
}
```

2. The code is undertaking a reduction operation in the loop, the adding of 1 on to `passing_count` at each iteration. Because I'm using a parallel loop the compiler will parallelise the loop, assuming it is independent and therefore safe, but it means that `passing_count` will not have the correct value at the end of the loop. To fix this you need to add a `reduction` clause on to either the parallel directive, like this:

```
passing_count = 0
!$acc parallel loop reduction(+:passing_count)
do i=1,10000
   moving(i) = (left(i) - right(i)) * 2
   left(i) = left(i) + 0.6
   right(i) = right(i) - 3
   passing_count = passing_count + 1
end do
!$acc end parallel loop
```

3. The outer loop, which is the one being parallelised across the GPU, does not have many iterations to distribute to hardware resources. However, the lower loop levels have many more iterations. Therefore you could move the loop to be parallelised to one of the inner loops, you could parallelise more than one loop and distribute them to different parts of the hardware (i.e. gang, worker, vector) or you could use the collapse directive to make the loop iteration space to be split across the hardware much larger, i.e.:

```
#pragma acc parallel loop collapse(3)
for(i=0;i<8;i++){
  for(j=0;j<320;j++){
    for(k=0;k<320;k++){
     new_data[i][j][k] = (old_data[i][j][k] * 2) + 4.3;
    }
  }
}
```