Solutions

1. The semantics of a "vector" loop says that all loop iterations for a specific statement in the loop are executed simultaneously, or at least give the same results as though they were executed simultaneously. Since simultaneous execution is the fastest possible execution, the iterations of the vector loop are going to be executed faster than the iterations of any loop contained within it. This effectively interchanges the vector loop to the innermost position. This can create semantics that are at best confusing and at worst incorrect.

   To illustrate, consider the following

   ```
   for(i=1; i<2; i++)
      for(j=1; j<2; j++)
         a[i][j] = a[i][j] * x;
   ```

   Consider first, when the outer loop (the "i" loop) is marked as vector. Execution will proceed as follows:

   a. The "i" loop starts, and since it is "vector" and of length 2, 2 iterations immediately start.
   b. Each iteration of the "i" loop, on encountering the "j" loop, initiates execution of that loop.
   c. The assignment will then be executed for iteration pairs (i=1, j=1) and (i=2, j=1), since each of the "i" iterations has spawned a "j" loop that is starting at 1.
   d. Each "j" loop proceeds to the next iteration, executing the assignment for (i=1, j=2) and (i=2, j=2).

   Note what has happened. The original execution order ignoring the directives is (i=1, j=1), (i=1, j=2), (i=2, j=1), (i=2, j=2). The new execution order is [(i=1,j=1), (i=2,j=1)], [(i=1,j=2), (i=2,j=2)], where [] indicates simultaneous execution. Applying the vector directive has effectively interchanged the loops: in the original, all the iterations of a "j" loop associated with an iteration of the "i" loop execute before the next iteration of the "i" loop. In the transformed loops, all the iterations of an "i" loop associated with an iteration of the "j" loop execute before the next iteration of the "i" loop.

   Now consider what happens if a "gang" directive is applied to the inner loop with a "vector" outer loop.

a. The “i” loop starts, and since it is “vector” and of length 2, two iterations immediately start.
b. Each iteration of the “i” loop, on encountering the “j” loop, initiates execution of that loop. However, since it is a “gang” loop, firing off the “j” loop does not mean initiating a single iteration. Initiating a “gang” loop is effected by unleashing multiple processors on the loop’s iterations, and the processors attack the iterations in an indeterminate order.
c. Even though the “i” iterations fire off the “j” loop simultaneously, the fact that the “j” iterations execute indeterminately mean that all iterations, “i” and “j”, execute in an indeterminate order. If the “j” loop associated with the i=2 iteration happens to start before the “j” loop associated with the i=1 iteration, then all the work associated with the “i=2” iteration can complete before the work for “i=1” even begins.

In other words, making a loop contained within a “vector” loop have indeterminate semantics causes both loops to execute indeterminately. At best, these are confusing semantics. It is possible for worse. Consider the following loop:

```
for(i=1; i<m; i++)
   for(j=1; j<n; j++)
      a[i-1][j+1] = a[i][j] * x;
```

While the “i” loop does carry a dependence, that dependence arises because the loop stores to a location that it has earlier fetched. Since simultaneous semantics means that all fetches are performed before any stores, this dependence will in fact be satisfied by “vector” and if the “j” loop is iterated sequentially, the semantics of the loops remain the same if the “i” loop is iterated in vector. Written in their original order, only the “i” loop carries a dependence, so the “j” loop can be executed with indeterminate semantics. (Try executing the “i” loop sequentially and the “j” loop as a gang, worker, or vector loop, and you will find the same results are computed.). However, putting a “vector” on the “i” loop changes that. By essentially interchanging that loop to the innermost position, the “j” loop now carries a dependence, and if the “j” loop is executed as a “gang” loop while the “i” loop is executed in vector, incorrect results are computed. This change in semantics is subtle, and not easily understood by most.

Finally, from an efficiency viewpoint, it is almost always desirable to have the gang loop be as far outward as possible and vector loops be as far inward as possible. The key to efficiency for gang and worker loops is to amortize the overhead across as much computation as possible. Since vector loops are ideally simultaneously, by definition they are going to be executed as innermost.

Summarizing, there are at least three bad things that can happen by allowing a gang or worker loop within a vector loop:

a. The loops are probably executing less efficiently that is possible.
b. The semantics of this can be confusion.
c. That confusion can help contribute to an incorrect transformation.

2. There is an obvious counter example to Dr. Grignard's assertion: subscripts that may vary from run to run based on input data or something similar. As a simple example,

```
for(i=1; i<n; i++) {
      index[i] = rand() % n;
}
for(i=1; i<n; i++) {
      a[index[i]] = i;
}
```

The first loop sets an index vector to be a random set of integers; the second the scatters values in a result vector based on that. Obviously, the set of random numbers generated on one run may differ from those generated on another, and as such the indexing patterns may change. Similar, if subscript values are read from a file or other outside data, the results may vary. The fact that one set of subscripts happen to meet the criterion does not mean that others will.

A second "outlier" counter example depends on the definition of "same result". If "same result" means just results that are printed out – in other words, some subset of the entire computation state – it is trivial to come up with a counter example. The situation is more complicated when "same result" means that all variables have the same values when the loop is run forward and backward.

Outliers aside, the focus of the assertion is whether getting the same output with forward execution and backward execution guarantees that simultaneous semantics are satisfied. The text states that a sequential loop in

which no statement depends upon itself, either directly or indirectly, can be executed correctly with simultaneous semantics. The ideal proof, then would be that getting the same results with forwards and backwards execution means no cross-loop memory accesses, which would in turn imply no dependences carried by the loop and therefore no self-dependence and correct vector execution. Unfortunately, while the absence of loop-carried dependence does imply the same results with forward and backward execution, the other direction does not hold. It is possible to have a loop that carries dependences and that yields the same results when executed in the forward and backward direction. However, it is also true that executing a loop backwards "reverses" dependences: fetches that occur before stores now occur after stores; stores that occur after fetches now occur before; and two stores to the same memory location reverse their order of access. If subscripts are monotonic functions of loop induction variables (something many assume to be true and true of most common cases, it is not required), it is easy to show this. For non-monotonic functions, it is possible to come up with counterexamples.

As an example of a loop that will pass the test and fail the result, consider

```
for(i=0; i<8; i++) {
      a[i%2] = i%3;
      b[a[i%2]] = a[i%2];
}
```

Executed in the forward direction, here are the values of b after each iteration:

	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
b[0]	0	0	0	0	0	0	0	0
b[1]	0	1	1	1	1	1	1	1
b[2]	0	0	2	2	2	2	2	2

Reversed

	i=7	i=6	i=5	i=4	i=3	i=2	i=1	i=0
b[0]	0	0	0	0	0	0	0	0
b[1]	1	1	1	1	1	1	1	1
b[2]	0	0	2	2	2	2	2	2

The result is the same at the end. If executed simultaneously, however, all executions of the second statement are executed after the first statement has completed. At that point, a[0] == 0 and a[1] == 1; intermediate results where some values of a are equal to 2 have been overwritten. As a result, b[2] will never get set to anything other than 0.

3. With all the same caveats as the second question, it is fairly easy to give examples showing that giving the same results with forward and backward execution do not map to indeterminate semantics. As one example:

```
for(i=0; i<8; i++) {
      a[i%2] = i%3;
}
```

The values of `a` after each iteration:

	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
a[0]	0	0	2	2	1	1	0	0
a[1]	0	1	1	0	0	2	2	1

Reversed:

	i=7	i=6	i=5	i=4	i=3	i=2	i=1	i=0
a[0]	0	0	0	1	1	2	2	0
a[1]	1	1	2	2	0	0	1	1

The values of a end up the same regardless of the execution order, but if the loop is executed indeterminately, the intermediate values may be stored last, causing incorrect results. A common assumption is that subscripts are monotonic functions of loop induction variables. When that is the case and a loop gives the same results forward and reversed, it can be run correctly with simultaneous or indeterminate semantics.

4. Even though this procedure represents a naturally vector operation, the fact that it has been written generally (which is good coding practice) means that it can be difficult for a compiler to automatically vectorize. The problem is the general stride “incy” through the result vector. Because it is a variable whose value is unknown to the compiler, the compiler must assume that it

can take on any value, including the value "0". The value "0" is a problem, because at that point the operation becomes a reduction rather than a pure vector operation.

a. Without knowledge that "incy" cannot be 0, it is difficult for a compiler to vectorize this loop. The only completely correct way to vectorize it is to create an "if" statement that tests "incy" against 0. A vector version of this statement can be placed n the clause corresponding to "incy" not being equal to 0. The other clause would need either scalar code or a reduction.
b. If the user places an OpenACC "vector" directive around the loop without indicating that the loop is also a reduction, the compiler can and should issue the vector version of the code.

5. The statements can be executed simultaneously. Since the dependence runs from the second statement to the first, all that is required in order for the loop to be valid for vector execution is to reverse the order of the two statements

```
for(i=1; i<n; i++) {
      a[i] = b[i] + c[i];
      d[i] = a[i-1] + 1;

}
```

With the reversal, all the values needed by the second statement are computed by the first statement before they are needed, so the same result is computed.

If the user places a "vector" directive around the loop, the user is ensuring that the loop will compute the correct results when executed in vector as is. As a result, the compiler should generate vector code with the statements in their original order. The results will be incorrect, but that is because the user has incorrectly asserted that the loop can be correctly executed in vector in its original order.

6. The loop cannot be executed in gang or worker mode as written. Since on iteration i1 the second statement overwrites a value that was used by the first statement on iteration i1-1. If the loop executes in indeterminate order, there can be iterations where the second statement stores it location a[i1] before iterations i1-1 fetches, leading to incorrect results.

There are at least two ways in which the loop can be transformed for correct gang and worker execution. One possibility, following the lead of the previous question, is to reorder the statements and execute them in individual loops, so that all stores are complete before any fetches are initiated:

```
for(i=1; i<n; i++) {
      a[i] = b[i] + c[i];
}
for(i=1; i<n; i++) {
      d[i] = a[i-1] + 1;

}
```

Each loop can be executed in parallel, with the implied barrier between taking care of any stores.

Another approach is to align the load and store so that they will also occur on the same processor:

```
d[1] = a[0] + 1;
for(i=1; i<n-1; i++) {
      a[i] = b[i] + c[i];
      d[i+1] = a[i] + 1;

}
a[n] = b[n] + c[n];
```

By removing an iteration from either end, the statements are aligned so that the store and the fetch occur on the same iteration without incurring a race condition. The loop may be executed in gang or worker mode.

7. Each of the loops in matrix multiplication can be done in parallel or vector, although the innermost loop is a reduction. As a result, the natural inclination, assuming the loop lengths are long enough to justify parallelism, is to put a different parallel directive around each loop. This doesn't quite work for the reduction loop because OpenACC doesn't support reductions into an array reference. Taking that into account yields something similar to the following:

```
%acc loop parallel gang worker
for(j=0; j<n; j++) {
    %acc loop vector
```

```
    for(i=0; i<m; i++) {
        c[i][j] = 0;
        for(k=0; k<p; k++) {
          c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

If the only transformation is to add directives, this is probably the best you can do. However, minor changes can create dramatically improved performance. First, the "j" loop (outmost loop) has stride one access in variables `c` and `b` in the innermost loop. Stride one access is much better for vector operations, so splitting the "j" loop into two loops, the innermost of which exactly fits the warp length, and moving that loop inside the "i" loop to yield:

```
for(j1=0; j1<n; j1+=warplength) {
    for(i=0; i<m; i++) {
      for(j=j1; j<j1+warplength; j++) {
        c[i][j] = 0;
        for(k=0; k<p; k++) {
          c[i][j] += a[i][k] * b[k][j];
        }
      }
    }
}
```

(Note that there are some cleanup operations needed for the "j1" loop that are not shown; the assumption is that `n` is an exact multiple of warp length.) This transformation with the following directives will result in much better performance:

```
%acc parallel loop gang
for(j1=0; j1<n; j1+=warplength) {
    %acc loop worker
    for(i=0; i<m; i++) {
      %acc loop vector
      for(j=j1; j<j1+warplength; j++) {
        c[i][j] = 0;
        for(k=0; k<p; k++) {
          c[i][j] += a[i][k] * b[k][j];
        }
```

```
        }
      }
}
```

With the second form, the reduction is now performed explicitly into a scalar and can be coded as such:

```
%acc parallel loop gang
for(j=0; j<n; j++) {
    %acc loop worker private(t)
    for(i=0; i<m; i++) {
        t = 0;
        %acc loop vector sum(:t)
        for(k=0; k<p; k++) {
          t += a[i][k] * b[k][j];
        }
        c[i][j] = t;
    }
}
```

Comparing the two loops with no changes other than adding directives, the second form should run faster. The second form has a better balance of parallelism, even though the vector form is a reduction, which is slightly slower than full vector. However, the transformed version of the first loop should beat all versions but achieving the better balance, utilizing stride 1 access, and fully utilizing the vector unit as a pure vector operation.