C++ AMP Errata

The following is a list of the errata for the book. If you find other errors with the book text then please report them on the O'Reilly Errata page. For issues related to the code samples please file an issue on Codeplex.

Chapter 1: Vectorization (page 8)

The code fragment should read as follows. Note the use of the & operator rather than &&.

```
int CPUInfo[4] = { -1 };
__cpuid(CPUInfo, 1);
bool bSSEInstructions = (CpuInfo[3] >> 24 & 0x1);
```

Chapter 1: OpenMP (page 10)

There is a small technical mistake here. The text should state that the loop is not parallelizable in its current form. This loop is parallelizable but must be rewritten to remove the loop carried dependency.

The developer is responsible for writing a loop that is parallelizable, of course, and this is the truly hard part of the job. For example, this loop is not parallelizable in its current form:

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

Chapter 2: The CPU NBody Classes (page 36)

The code fragment should read as follows. Note the use of the += operator add the current particle's acceleration to acc, rather than using the = operator to set the value.

```
std::for_each(pParticlesIn, pParticlesIn + numParticles, [=, &acc](const ParticleCpu& p)
{
    const float_3 r = p.pos - pos;
    float distSqr = SqrLength(r) + m_softeningSquared;
    float invDist = 1.0f / sqrt(distSqr);
    float invDistCube = invDist * invDist * invDist;
    float s = m_particleMass * invDistCube;
    acc += r * s;
});
```

Chapter 3: array_view<T,N> (page 52)

The code fragment should read as follows. The reference to av.grid has been replaced with av.extent. The grid property was the name used for extent in the beta release of C++ AMP.

```
std::vector<int> v(5);
std::iota(v.begin(), v.end(), 0);
array_view<int, 1> av(5,v);
parallel_for_each(av.extent, [=](index<1> idx) restrict(amp)
{
    av(idx) = av(idx) * 2;
});
av.synchronize();
```

Chapter 4: Tile Barriers and Synchronization (page 74)

The references to variable TS should refer to TileSize. The code fragment should read as follows.

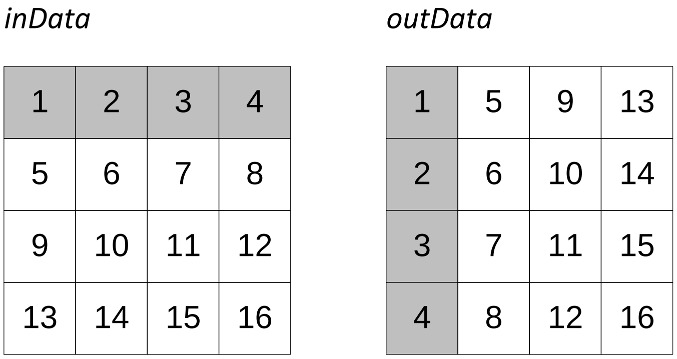
```
for (int i = 0; i < W; i += TileSize)
{

   tile_static float sA[TileSize][TileSize];
   tile_static float sB[TileSize][TileSize];
   sA[row][col] = a(tidx.global[0], col + i);
   sB[row][col] = b(row + i, tidx.global[1]);
   for (int k = 0; k < TileSize; k++)
      sum += sA[row][k] * sB[k][col];
}
```

Chapter 7: Efficient Accelerator Global Memory Access (pages 148-9)

This section does not contain any technical mistakes but feedback from readers suggested that it was not entirely clear. It's been reworded and the second diagram updated to convey the correct meaning.

This difference in execution time is due to the writes to *outData* being uncoalesced. Although the reads from *inData* on each thread are from adjacent memory addresses, the writes to *outData* from consecutive threads occur on different rows.



inData

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

outData

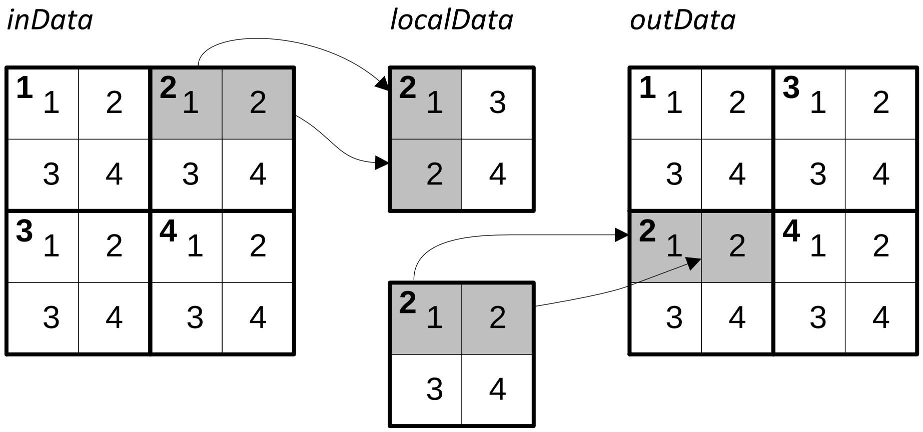
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

This explains the big difference in performance. The threads (numbered in the diagram above) are writing to memory locations that are not adjacent. Remember that C++ AMP, like C and C++, stores multi-dimensional data in row-major order, so the shaded row represents coalesced memory access while the column in *outData* is an uncoalesced access.

It's actually possible to use tile static memory to mitigate this by adding an additional set of copies, as shown in the following example:

```
... Code sample is unchanged. See the original text.
```

Here the kernel is divided into two phases using the familiar tiled kernel pattern introduced in Chapter 4. In the first part of the kernel each thread in the tile copies coalesced data from *inData* in global memory into tile static *localData* and transposes it during the copy to tile static memory. After the barrier—which ensures that all threads have finished the copy—the data is written in a coalesced way back to global memory. Tile static memory has a much higher bandwidth and smaller interface width than global memory, so the penalty for un-coalesced memory accesses is far less. By transferring the matrix elements by means of tile static memory and doing the transpose there, uncoalesced writes to global memory can be eliminated. The diagram shows four tiles (numbered in bold), each with four threads. The memory accesses for the threads in tile 2 are shown shaded. It clearly shows that the writes to *outData* by threads 1 and two in tile 2 are now coalesced.


inData
localData
outData
1 1 2
3 4
2 1 2
3 4
3 1 2
3 4
4 1 2
3 4
2 1 3
2 4
2 1 2
3 4
1 1 2
3 4
3 1 2
3 4
2 1 2
3 4
4 1 2
3 4