Xeon Phi Performance Optimization

Table of Contents

1 Scalability

2 Workload Balance

3 Barriers

4 False Sharing

5 Vectorization

5.1 Array notation

5.2 Elemental functions

5.3 Directives and pragmas

6 Memory and cacheing

6.1 Addressing

- Code runs best when data are accessed in sequential address-order from memory. In many cases, data structure transformation from AoS (array of structures) to SoA (structure of arrays) is needed for this.

```
// AoS                    // SoA
typedef struct {          typedef struct {
  int data1;                int data1[100];
  int data2;                int data1[100];
  ...                       ...
} struct_t;               } struct_t;
struct_t AoS[100];        struct_t SoA;
```

- Consider accessing `data1` and `data2` in a loop:

```
for (...) {               for (...) {
  ... AoS[i].data1;         ... SoA.data1[i];
}                         }
```

- Note that there are many fields in `AoS` (e.g. `data1` through `data100`), the distance between `AoS[0].data1` and `AoS[1].data1` is at least 100 integers apart. We cannot exploit the cache in this case.
- However, consider the following case:

```
for (...) {               for (...) {
  ... AoS[i].data1;         ... SoA.data1[i];
  ... AoS[i].data2;         ... SoA.data2[i];
  ...                       ...
  ... AoS[i].data100;       ... SoA.data100[i];
}                         }
```

Check which of AoS and SoA would be better.

6.2 Prefetching

- Consider:

```
for (...) {
  s = data1[i];
  t = data2[i];
  ...
  ... = s + t;
}
```

- The instruction for `s + t` has to wait inside the pipeline until `data1` and `data2` values are read in.
- Intel compiler performs prefetching but users can give pragmas/directives or use intrinsic commands to specify it (if statically calculable).
- Prefetches typically should be for a *future iteration of the loop*, not the current iteration of the loop.

6.3 Blocking

6.4 Tiling

6.5 Bandwidth