Memory Layout Transformations

By AmandaS (Intel) (https://software.intel.com/en-us/user/334297), published on November 25, 2013

Compiler Methodology for Intel® MIC Architecture (/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture)

Memory Layout Transformations

Overview

This chapter examines a useful user code transformation: moving from data organized in an Array of Structures (AoS) to an organization of Stucture of Arrays (SoA). This transformation allows the compiler to access data more efficiently on the processor.

Topics

One class of user code changes for efficiency involves eliminating the use of memory gather-scatter operations. Such irregular memory operations can both increase latency and bandwidth usage, as well as limit the scope of compiler vectorization. Some applications may benefit from a data layout change that converts data structures written in an Array of Structures (AOS) representation to a Structure of Arrays (SOA) representation. This helps prevent gathers when accessing one field of the structure across the array elements, and helps the compiler vectorize loops that iterate over the array. Note that such data

transformations have to be done at the program-level (by the user) taking into account all the places where those data structures are used. Doing it at just a loop-level will involve costly transformations between the formats before and after the loop.

AOS to SOA conversion: A common optimization that helps prevent gathers and scatters in vectorized code is to convert data structures from Array-Of-Structures (AOS) to Structure-Of-Array (SOA) representation. Keeping separate arrays for each structure-field keeps memory accesses contiguous when vectorization is performed over structure instances. AOS structures require gathers and scatters, which can impact both SIMD efficiency as well as introduce extra bandwidth and latency for memory accesses. The presence of a hardware gather-scatter mechanism does not eliminate the need for this transformation - gather-scatter accesses commonly need significantly higher bandwidth and latency than contiguous loads.

The SOA form does come at the cost of reducing locality between accesses to multiple fields of the original structure instance (thus increasing TLB (Translation Look-aside Buffer) pressure for example). Depending on the data-access pattern in the original source-code (whether or not they involve accesses to multiple fields of the structurewithin a loop, whether or not all structure instances are traversed or a subset), it may be preferable to consider a conversion to **Array-Of-Structure-Of-Arrays (AOSOA)** instead. The idea here is to get the benefit of locality at the outer-level and also unit-stride at the innermost-level. The inner-array can be a small multiple of vector-length here to take advantage of unit-stride full-vectors. Each structure instance will have multiple fields laid out this way (in small arrays). And at the outer-level, you can have an array of the (now larger) structures. In such a layout, the field-accesses will still be close enough to get the benefit of page-locality for nearby structure-instance accesses in the original code.

Here is a simple example that shows the difference between these forms:

AOS Form (Array of Structs):

```
struct node {
  float x, y, z;
};
struct node NODES[1024];
```

```
float dist[1024];
for(i=0;i<1024;i+=16){
   float x[16],y[16],z[16],d[16];
   x[:] = NODES[i:16].x;
   y[:] = NODES[i:16].y;
   z[:] = NODES[i:16].z;
   d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
   dist[i:16] = d[:];
}
```

SOA Form (Struct of Arrays):

```
struct node1 {
  float x[1024], y[1024], z[1024];
}
struct node1 NODES1;

float dist[1024];
for(i=0;i<1024;i+=16){
  float x[16],y[16],z[16],d[16];
  x[:] = NODES1.x[i:16];
  y[:] = NODES1.y[i:16];
  z[:] = NODES1.z[i:16];
  d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
  dist[i:16] = d[:];
}
```

AOSOA Form (Array of Struct of Arrays OR Tiled Array of Structs):

```
struct node2 {
  float x[16], y[16], z[16];
}
struct nodes2 NODES2[64];

float dist[1024];
for(i=0;i<64;i++){
   float x[16],y[16],z[16],d[16];
   x[:] = NODES2[i].x[:];
```

```
   y[:] = NODES2[i].y[:];
   z[:] = NODES2[i].z[:];
   d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
   dist[i*16:16] = d[:];
}
```

Example that shows how AOS data structures are changed to SOA using Array Notations in source

Here is the relevant code snippet from an OpenMP* version of the LIBOR* benchmark where the AOS form (in the original) is converted to use SOA form (modified) using Array Notations:

Original code (LIBOR,AOS form):

```
long tid = (long)(tid_);
long long int chunk = npath/nthreads;
long long int beg = tid*chunk;
long long int end = min(npath, (tid+1)*chunk);
long long int chunk_inner = PATH_BLOCK_SIZE/nthreads // JSP
printf("chunk = %d, chunk_inner = %dn", chunk, chunk_inner);

 #pragma omp parallel
 for(long long int path = beg; path < end; path += chunk_inner){

      #pragma omp for nowait
      for (long long int path_=0; path_ < chunk_inner; path_+=1) {
             int    i, j, k;
             __declspec(align(64)) FPPREC B[nmat], S[nmat], L[n];
             FPPREC sqez, lam, con1, v_scal, vrat;
             FPPREC b, s, swapval;
             FPPREC *zlocal=z+nmat*(tid*chunk_inner+path_);

             for(i=0;i < n;i++) {
                    L[i] = L0[i];
             }
             for(j=0; j < nmat; j++){
                    sqez = SQRT(delta)*zlocal[j];
                    v_scal = ZERO;
```

Modified code (LIBOR*, SOA form):

```
int tid = (int)(tid_);
long long int chunk = (npath%nthreads)?(1 + npath/nthreads):
(npath/nthreads);
long long int beg = tid*chunk;
long long int end = min(npath, (tid+1)*chunk);
long long int chunk_inner = ((PATH_BLOCK_SIZE%(nthreads*SIMDVLEN))?(1 +
PATH_BLOCK_SIZE/(nthreads*SIMDVLEN)):
(PATH_BLOCK_SIZE/(nthreads*SIMDVLEN)))*SIMDVLEN;

 #pragma omp parallel
 for(long long int path = beg; path < end; path += chunk_inner){

      #pragma omp for nowait
      for (long long int path_=0; path_ < chunk_inner; path_+=SIMDVLEN) {
            int    i, j, k;
             __declspec(align(64)) FPPREC B[nmat][SIMDVLEN], S[nmat]
[SIMDVLEN], L[n][SIMDVLEN];
            __declspec(align(64)) FPPREC sqez[SIMDVLEN], lam[SIMDVLEN],
con1[SIMDVLEN], v_scal[SIMDVLEN], vrat[SIMDVLEN], vrat_tmp[SIMDVLEN];
            __declspec(align(64)) FPPREC b[SIMDVLEN], s[SIMDVLEN],
swapval[SIMDVLEN];
             FPPREC *zlocal=z+nmat*(tid*chunk_inner + path_);
            __declspec(align(64)) FPPREC old_output[SIMDVLEN];
```

Take Aways

Although it may seem more logical to organize data in Array of Strutures (AoS), this organization makes it extremely difficult to access the memory for reads (gather) and writes (scatter). This prevents many optimizations by the compiler, most notably it often prevents efficient vectorization. Algorithms with this characteristic are often very poor performers on both Intel® Xeon Architecture and Intel® MIC Architecture.

If possible, it is desirable to reorganize data into a Stucture of Arrays (SoA) organization. Unfortunately, for real world applications this may prove to be a non-trivial amount of effort. However, the benefits of doing such a transformation may be quite significant.

NEXT STEPS

It is essential that you read this guide from start to finish using the built-in hyperlinks to guide you along a path to a successful port and tuning of your application(s) on Intel® Xeon Phi™ coprocessors. The paths provided in this guide reflect the steps necessary to get best possible application performance.

BACK to Preparing for the Intel® Many Integrated Core Architecture (Intel® MIC Architecture) (/en-us/articles/preparing-for-the-intel-many-integrated-core-architecture#next_steps)

For more complete information about compiler optimizations, see our Optimization Notice (/en-us/articles/optimization-notice#opt-en).