software.intel.com

Putting Your Data and Code in Order: Data and layout - Part 2

16-20 minutos

In this pair of articles on performance and memory covers basic concepts to provide guidance to developers seeking to improve software performance. These articles specifically address memory and data layout considerations. Part 1 addressed register use and tiling or blocking algorithms to improve data reuse. The paper begins by considering data layout first for general parallelism – shared memory programming with threads and then considers distributed computing via MPI as well. This paper expands those concepts to consider parallelism, both vectorization (single instruction multiple data SIMD) as well as shared memory parallelism (threading), and distributed memory computing. Lastly, this article considers data layout array of structure (AOS) versus structure of arrays (SOA) data layouts.

The basic performance principle emphasized in Part 1 is: reuse data in register or cache before it is evicted. The performance principles emphasized in this paper are: place data close to where it is most commonly used, place data in contiguous access mode, and avoid data conflicts.

Shared Memory Programming with Threads

Let's begin by considering shared memory programming with threads. Threads all share the same memory in a process.

There are many popular threading models. The most well-known are Posix* threads and Windows* threads. The work involved in properly creating and managing threads is error prone. Modern software with numerous modules and large development teams makes it easy to make errors in parallel programming with threads. Several packages have been developed to ease thread creation, management and best use of parallel threads. The two most popular are OpenMP* and Intel® Threading Building Blocks. A third threading model, Intel® Cilk™ Plus, has not gained the adoption levels of OpenMP and Threading Building blocks. All of these threading models create a thread pool which is reused for each of the parallel operations or parallel regions. OpenMP has an advantage of incremental parallelism through the use of directives. Often OpenMP directives can be added to existing software with minimal code changes in a step-wise process. Allowing a thread runtime library to manage much of the thread maintenance eases development of threaded software. It also provides a consistent threading model for all code developers to follow, reduces the likelihood of some common threading errors, and provides an optimized threaded runtime library produced by developers dedicated to thread optimization.

The basic parallel principles mentioned in the introductory paragraphs are place data close to where it will be used and avoid moving the data. In threaded programming the default model is that data is shared globally in the process and may accessed by all threads. Introductory articles on threading emphasize how easy it is to begin threading by applying OpenMP to do loops (Fortran*) or for loop (C). These methods typically show good speed up when run on two to four cores. These methods frequently scale well to 64 threads or more. Just as frequently, though, they do not, and in some

of those cases where they do not it is a matter of following a good data decomposition plan. This is a matter of designing an architecture of good parallel code.

It is important to explore parallelism at a higher level in the code call stack than where a parallel opportunity is initially identified by developer or software tools. When a developer recognizes that tasks or data can be operated on in parallel consider these questions in light of Amdahls' law: "can I begin the parallel operations higher up in the call stack before I get to this point? If I do this do I increase the parallel region of my code that will then provide better scalability?"

The placement of data and what data must be shared through messages is carefully considered. Data is laid out to so that the data is placed where it is used most and then sent to other systems as needed. For applications represented in a grid, or a physical domain with specific partitions, it is common practice in MPI software to add a row of "ghost" cells around the subgrid or sub-domain. The ghost cells are used to store the values of data sent by the MPI process which updates those cells. Typically ghost cells are not used in threaded software, but just as you minimize the length of the edge along the partition for message passing, it is desirable to minimize the edge along partitions for threads using shared memory. This minimizes the needs for thread locks (or critical sections) or for cache usage penalties associated with cache ownership.

Large multi-socketed systems, although they share global memory address space, typically have non-uniform memory access (NUMA) times. Data in a memory bank closest to another socket takes more time to retrieve, or has longer latency, than data located in bank closest to the socket where the code is running. Access to close memory has a shorter latency.

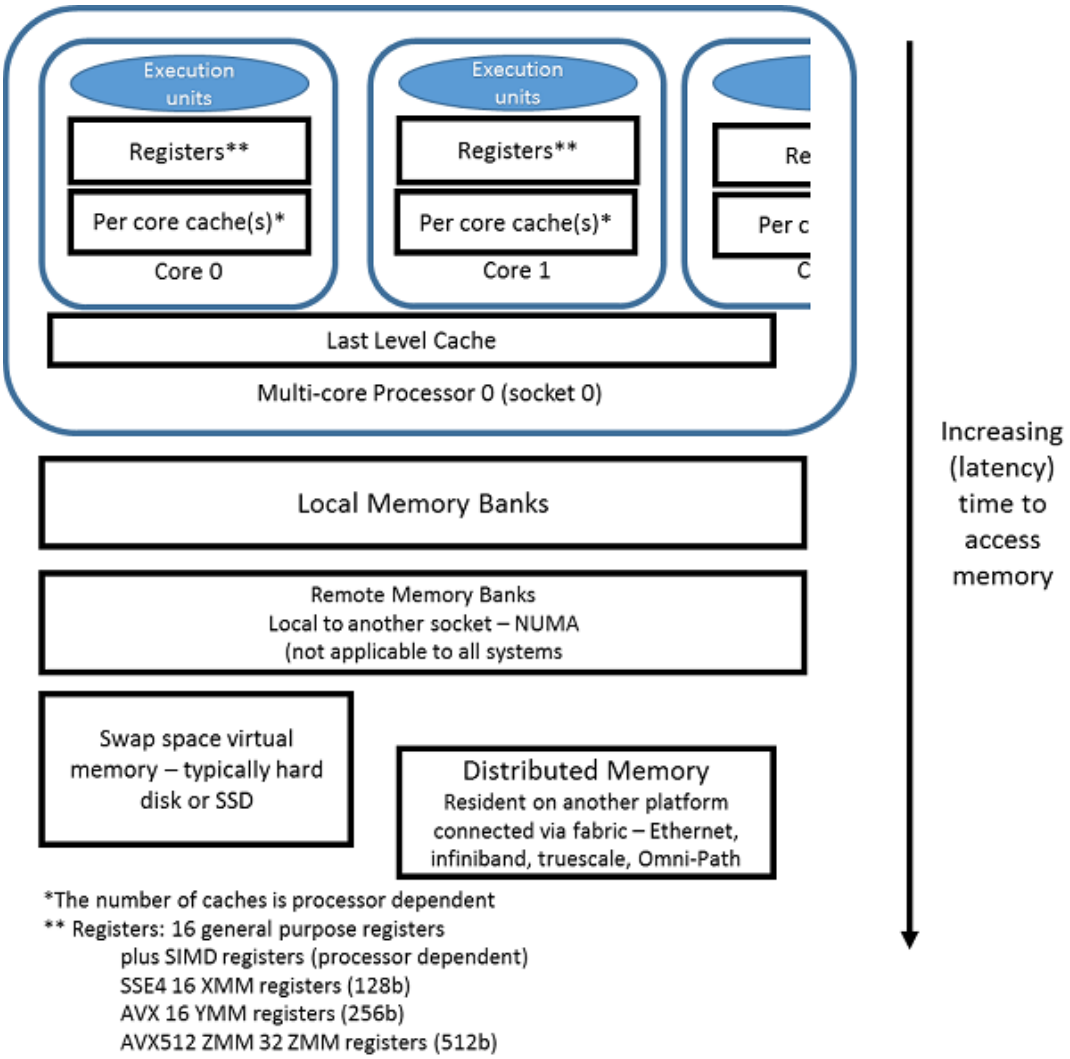


Figure 1. Latency memory access, showing relative time to access data.

If one thread allocates and initializes data, that data is usually placed in the bank closest to the socket that the thread allocating and initializing the memory is running on (Figure 1). You can improve performance having each thread allocate and first reference the memory it will predominately use. This is usually sufficient to ensure that the memory is closest to the socket the thread is running on. Once a thread is created and active, the OS typically leaves threads on the same socket. Sometimes it has been beneficial to explicitly bind a thread to a specific core to prevent thread migration. When data has a certain pattern it is beneficial to assign, bind or set affinity of the threads to specific cores to match this pattern. The Intel

OpenMP runtime library (part of Intel® Parallel Studio XE 2016) provides explicit mapping attributes which have proven useful for Intel® Xeon Phi™ coprocessor.

These types are compact, scatter, and balanced.

- The compact attribute allocates consecutive or adjacent threads to the symmetric multithreading (SMTs) on a single core before beginning to assign threads to other cores. This is ideal where threads share data with consecutively numbered (adjacent) threads.
- The scatter affinity assigns a thread to each core, before going back to the initial cores to schedule more threads on the SMTs.
- Balanced affinity assigns thread of consecutive or neighboring IDs to the same core in a balanced fashion. Balanced is the recommended be starting affinity for those seeking to optimize thread affinity according to the Intel 16.0 C++ compiler documentation. The Balanced affinity setting is only available for Intel® Xeon Phi™ product family. It is not a valid option for general CPUs. When all the SMTs on a Xeon Phi platform are utilized balanced and compact behave the same. When only some of the SMTs are utilized on a Xeon Phi platform the compact method will fill up all the SMTs on the first cores and leave some cores ideal at the end.

  Taking the time to place thread data close to where it is used is important when working with dozens of threads. Just as data layout is important for MPI programs it can be important for threaded software as well.

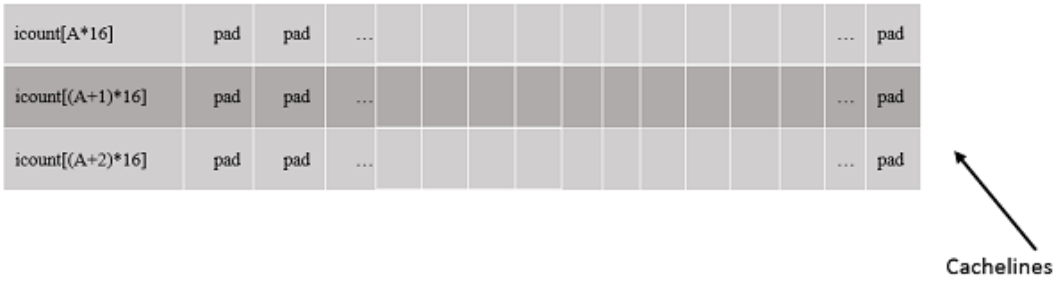
  There are two short items to be considered regarding memory and data layout. These are relatively easy to address, but can have significant impact. The first is false sharing and the second is data alignment. One of the interesting performance issues with threaded software is false sharing. Each thread

data operates on is independent. There is no sharing, but the cache line containing both data points is shared. This is why it is called false sharing or false data sharing; the data isn't shared, but the performance behavior is as though it is shared.

Consider a case where each thread increments its own counter, but the counter is in a one-dimensional array. Each thread increments its own counter. To increment its counter, the core must own the cache line. For example, thread A on socket 0 takes ownership of the cacheline and increments iCount[A]. Meanwhile thread A+1 on socket 1 increments iCount[A+1], to do this the core on socket 1 takes owner ship of the cacheline and thread A+1 updates its value. Since a value in the cacheline is altered, the cacheline for the processor on socket 0 is invalidated. At the next iteration, the processor in socket 0 takes ownership of the cacheline from socket 0 and alters the value in iCount[A], which in turn invalidates the cacheline in socket 1. When the thread on socket 1 is ready to write the cycle repeats. A significant number of cycles are spent to maintain cache coherency, as invalidating cachelines, regaining control and synchronizing to memory that performance can be impacted.

The best solution to this is not invalidate the cache. For example, at the entrance to the loop, each thread can read its count and store it in a local variable on its stack (reading does not invalidate the cache). When the work is completed the thread can copy this local value back into the permanent location (see Figure 2). Another alternative is to pad data so that data used predominately by a specific thread in its own cacheline.

```
1 int iCount[nThreads*16] ;
7       iCount[myThreadId*16]++
```




```
        int local_Count = iCount[myThreadID]
07 ;
12     iCount[myThreadId] = local_Count ;
```

Figure 2.

The same false sharing can happen to scalars assigned to adjacent memory location. This last case is shown in the code snippet below:

```
3 declspec(align(64)) int data3;
4 declspec(align(64)) int data4;
```

When a developer designs parallelism from the beginning and minimizes shared data usage, false sharing is typically avoided. If your threaded software is not scaling well, even though there is plenty of independent work going on and there are few barriers (mutexes, critical sections), it may make sense to check for false sharing.

Data Alignment

Software performance is optimal when the data being operated on in a SIMD fashion (AVX512, AVX, SSE4, . . .) is aligned on cacheline boundaries. The penalty for unaligned data access varies according to processor family. The Intel® Xeon Phi™ coprocessors are particularly sensitive to data alignment. On the Intel Xeon Phi platforms data alignment is

very important. The difference is not as pronounced on other Intel® Xeon® platforms, but performance improves measurably when data is aligned to cache line boundaries. For this reason it is recommended that the software developer always align data on 64 Byte boundaries. On Linux* and Mac OS X* this can be done with the Intel compiler option – no source code changes – just use the command line option: `/align:rec64byte`.

For dynamic allocated memory in C, `malloc()` can be replaced by `_mm_alloc(`*`datasize`*`,64)`. When `_mm_alloc()` is used, `_mm_free()` should be used in place of `free()`. A complete article specifically on data alignment is found here: https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.

Please check the compiler documentation as well. To show the affect of data alignment two matrices of the same size were created and both ran the blocked matrix multiply code used in Part 1 of this series. For the first case matrix A was aligned, for the second case matrix A was intentionally offset by 24 bytes (3 doubles), performance decreased by an 56 to 63% using the Intel 16.0 compiler for matrices ranging from size 1200x1200 to 4000x4000. In part 1 of this series I showed a table showing performance of loop ordering that used different compilers, when one matrix was offset there was no longer any performance benefit from using the Intel compiler. It is recommended the developer check their compiler documentation about data alignment and options available so that when data is aligned the compiler makes the best use of that information. The code for evaluating performance for a matrix offset from the cacheline is embedded in the code for Part 1 - the code for this experiment is at: https://github.com/drmackay/samplematrixcode

The compiler documentation will have additional information as well.

To show the effect of data alignment, two matrices of the same size were created and both ran the blocked matrix multiply code used in Part 1. The first matrix was aligned, the second matrix was intentionally offset by 24 bytes (3 doubles), performance decreased by 56 to 63% using the Intel 16.0 compiler for matrices ranging from size 1200x1200 to 4000x4000.

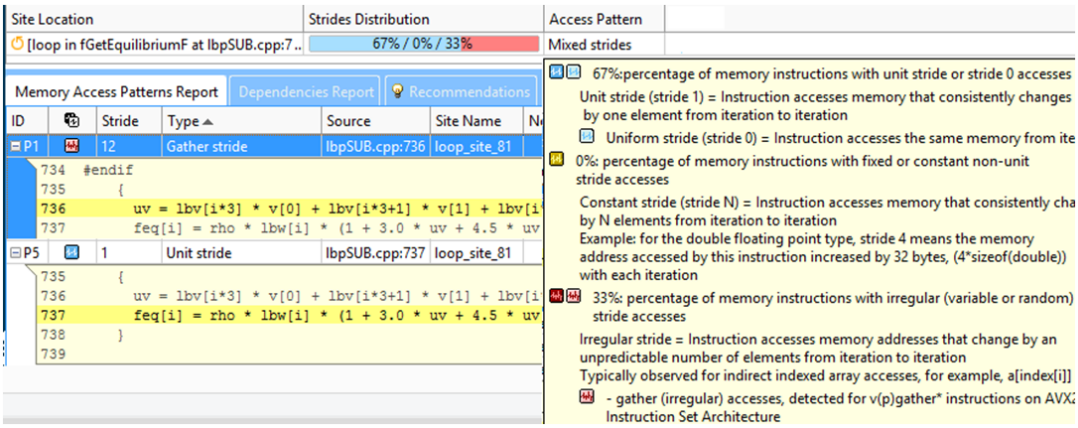
Array of Structure vs. Structure of Array

Processors do well when memory is streamed in contiguously. It is very efficient when every element of a cacheline is moved into the SIMD registers. If contiguous cachelines are also loaded the processors prefetch in an orderly fashion. In an array of structures, data may be laid out something like this:

In this layout the rgb values are laid out contiguously. If the software is working on data across a color plane, then the whole structure is likely to be pulled into cache, but only one value, g (for example), will be used each time. If data is stored in a structure of arrays, the layout might be something like:

When data is organized in the Structure of Arrays and the software operates on all of the g values (or r or b), when a cacheline is brought into cache the entire cache line is likely to be used in the operations. Data is more efficiently loaded into the SIMD registers, this improves efficiency and performance. In many cases software developers take the time to actually temporarily move data into a Structure of Arrays to operate on and then copy it back as needed. When possible it is best to avoid this extra copying as this takes execution time.

Intel (Vectorization) Advisor 2016 “Memory Access Pattern” (MAP) analysis identifies loops with contiguous (“unit-stride”), non-contiguous and “irregular” access patterns:



The “Strides Distribution” column provides aggregated statistics about how frequent each pattern took place in given source loop. In picture above the left two-thirds of the bar is colored blue – indicating a contiguous access pattern, however right one-third is colored red – which means non-contiguous memory access. For codes with pure AoS pattern Advisor can also automatically get specific “Recommendation” to perform AoS -> SoA transformation.

The Access Pattern and more generally Memory Locality Analysis is simplified in Advisor MAP by additionally providing memory “footprint” metrics and by mapping each “stride” (i.e. access pattern) diagnostic to particular C++ or Fortran* objects/array names. Learn more about Intel Advisor at

https://software.intel.com/en-us/get-started-with-advisor and https://software.intel.com/en-us/intel-advisor-xe

Structure of array and array of structure data layout are relevant for many graphics programs as well as nbody (e.g. molecular dynamics), or anytime data/properties (e.g. mass, position, velocity, charge), may be associated with a point or specific body. Generally, the Structure of Arrays is more efficient and yields better performance.

Starting from Intel Compiler 2016 Update 1, AoS -> SoA transformation made simpler by introducing Intel® SIMD Data Layout Templates (Intel® SDLT). Using SDLT the AoS container could be simply redefined in this style:

```
1 SDLT_PRIMITIVE(Point3s, x, y, z)
2 sdlt::soa1d_container<Point3s>
  inputDataSet(count);
```

making it possible to access the Point3s instances in SoA fashion. Read more about SDLT here.

There are several articles written specifically to address the topic of AoS vs SoA. The reader is directed to read one of these specific articles:

https://software.intel.com/en-us/articles/a-case-study-comparing-aos-arrays-of-structures-and-soa-structures-of-arrays-data-layouts

and

https://software.intel.com/en-us/articles/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture
http://stackoverflow.com/questions/17924705/structure-of-arrays-vs-array-of-structures-in-cuda

While in most cases a structure of arrays matches that pattern and provides the best performance, there are a few cases where the data reference and usage more closely matches an array of structure layout, and in that case the array of structure provides better performance.

Summary

In summary here are the basic principles to observe regarding data layout and performance. Structure your code

to minimize data movement. Reuse data while it is in the register or in cache; this also helps minimize data movement. Loop blocking can help minimize data movement. This is especially true for software having a 2D or 3D layout. Consider layout for parallelism – how are tasks and data distributed for parallel computation. Good domain decomposition practices benefit both message passing (MPI) and shared memory programming. A structure of arrays usually moves less data than an array of structures and performs better. Avoid false sharing, and create truly local variable or provide padding so each thread is referencing a value in a different cache line. Lastly, set data alignment to begin on a cacheline.

The complete code is available for download here: https://github.com/drmackay/samplematrixcode

In case you missed Part 1 it is located here.

Apply these techniques and see how your code performance improves.