Optimization Techniques for the Intel MIC Architecture. Part 3 of 3: False Sharing and Padding

Andrey Vladimirov

Colfax International

August 8, 2015

Abstract

This is part 3 of a 3-part educational series of publications introducing select topics on optimization of applications for Intel's multi-core and manycore architectures (Intel Xeon processors and Intel Xeon Phi coprocessors).

In this paper we discuss false sharing, highighting the situations in which it may occur, and eliminating it with the help of data container padding.

For a practical illustration, we construct and optimize a micro-kernel for binning particles based on their coordinates. Similar workloads occur in Monte Carlo simulations, particle physics software, and statistical analysis.

Results show that the impact of false sharing may be as high as an order of magnitude performance loss in a parallel application. On Intel Xeon processors, padding required to eliminate false sharing is greater than on Intel Xeon Phi coprocessors, so target-specific padding values may be used in real-life applications.

Table of Contents

1. Introduction

This paper completes our 3-part series of educational publications on performance optimization in applications for Intel Xeon Phi coprocessors. First part [1] discussed thread parallelism. Second part [2] focused on data parallelism and automatic vectorization. We achieved good performance results with thread-parallel and vectorized code in Part 2. In this part, we re-visit the parallelization technique that we used in Part 1 and consider an alternative implementation, in which false sharing of cache lines occurs and negatively impacts performance. We use this alternative method to demonstrate a common pitfall in multi-threaded applications and optimization techniques available in this case.

Optimization of multi-threading and vectorization on Intel Xeon processors and Intel Xeon Phi coprocessors requires controlling a wide array of aspects, including exposing sufficient parallelism, regularizing vectorization pattern, binding threads to cores in a favorable order, minimizing synchronization, and many other. An extensive discussion of these techniques is presented in our book, "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors", Second Edition [4]. This paper demonstrates how to eliminate false sharing and obtain a highly parallel implementation of parallel reduction with an alternative method. For code download, see the landing page of this paper [3].

2. Motivating Example: Particle Binning

For an illustration we will use the same example as in [1] and [2]: binning. Suppose that we have data coming from a simulation or from an experiment on particles moving in a cylindrical particle detector (see Figure 1). Particle positions are reported in polar coordinates, and our interest is to bin these particles into bins defined in Cartesian coordinates. Workloads like this occur in particle physics (for example, to detect particle tracks — see [5]), in Monte Carlo simulations and also in statistics where data transformation and binning takes place.

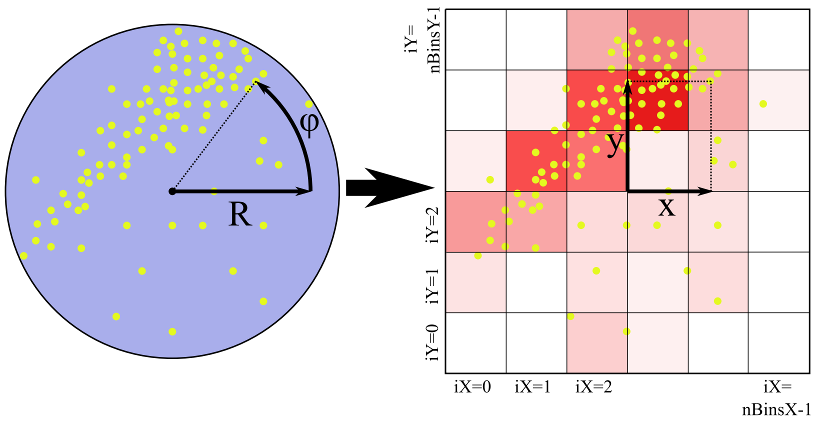

Figure 1: Workload illustration: take polar particle coordinates and compute particle counts in bins on the Cartesian grid.

For our specific problem, assume that the raw particle data comes in the form of a structure containing arrays `r` and `phi`. These arrays contain the radii and the polar angles, respectively, of each particle. Our task is to compute the output data, which is a 2-dimensional array containing the counts of particles in the respective bins on a 2-dimensional Cartesian grid. Listing 1 illustrates the data types.

```
// Input data structure: arrays of particle coordinates in polar system
struct InputDataType {
  int numDataPoints; // Size of arrays r and phi. Using n=2^27
  FTYPE* r;    // Array of radii
  FTYPE* phi; // Array of polar angles
};

// Output data structure: counts of particles in Cartesian grid bins
typedef int BinsType[nBinsX][nBinsY]; // Using nBinsX = nBinsY = 10
```

Listing 1: Data structures: counts of particles in bins on the Cartesian grid. `FTYPE` is `real` for single precision and `double` for double precision implementation.

A non-optimized scalar C code that performs such binning is shown in Listing 2.

```
void BinParticlesReference(InputDataType  & inputData, BinsType & outputBins) {
  // Reference implementation: scalar, serial code without optimization

  // Loop through all particle coordinates
  for (int i = 0; i < inputData.numDataPoints; i++) {
    // Transforming from cylindrical to Cartesian coordinates:
    const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
    const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);

    // Calculating the bin numbers for these coordinates:
    const int iX = int((x - xMin)*binsPerUnitX);
    const int iY = int((y - yMin)*binsPerUnitY);

    // Incrementing the appropriate bin in the counter:
    outputBins[iX][iY]++;
  }
}
```

Listing 2: Non-optimized binning code. `FTYPE`, `SIN` and `COS` are preprocessor macros set to either `float`, `sinf` and `cosf` for single precision, or `double`, `sin` and `cos` for double precision implementation.

The binning code without optimization is a reference implementation that produces correct results, but uses none of the capabilities of Intel architecture related to parallelism. The code is serial (i.e., uses only one thread and therefore is capable of using only one logical processor) and scalar (i.e., does not use vector instructions for data parallelism).

In Part 1 we implemented multi-threading in this code, and after the work in Part 2, the code also became partially vectorized. The result of our earlier optimization efforts is shown in Listing 3. The key to thread scalability of this implementation is the usage of thread-private containers for temporary storage of binning results (lines 11 and 40), and the aggregation (reduction) of data across these containers into a shared container with mutexes (lines 44-49).

In this paper, we will produce an implementation alternative to Listing 3. Our goal will be to match the performance obtained in Part 2 using a different structure of data containers.

```
void BinParticles_4(const InputDataType  & inputData, BinsType & outputBins) {

  // Thread-parallel, vectorized implementation with alignment hints
  // with reduction using thread-private containers

  const int STRIP_WIDTH = 16;

#pragma omp parallel
  {
    // Declare thread-private containers for bins
    BinsType threadPrivateBins;
    for (int i = 0; i < nBinsX; i++)
      for (int j = 0; j < nBinsY; j++)
        threadPrivateBins[i][j] = 0;

    // Loop through all bunches of particles
#pragma omp for
    for (int ii = 0; ii < inputData.numDataPoints; ii += STRIP_WIDTH) {

      int iX[STRIP_WIDTH] __attribute__((aligned(64)));
      int iY[STRIP_WIDTH] __attribute__((aligned(64)));

      const FTYPE* r   = &(inputData.r[ii]);
      const FTYPE* phi = &(inputData.phi[ii]);

      // Vector loop
#pragma vector aligned
      for (int c = 0; c < STRIP_WIDTH; c++) {
        // Transforming from cylindrical to Cartesian coordinates:
        const FTYPE x = r[c]*COS(phi[c]);
        const FTYPE y = r[c]*SIN(phi[c]);

        // Calculating the bin numbers for these coordinates:
        iX[c] = int((x - xMin)*binsPerUnitX);
        iY[c] = int((y - yMin)*binsPerUnitY);
      }

      // Scalar loop
      for (int c = 0; c < STRIP_WIDTH; c++)
            threadPrivateBins[iX[c]][iY[c]]++;

    }

    // Reduction outside the parallel loop
    for(int i = 0; i < nBinsX; i++)
      for(int j = 0; j < nBinsY; j++) {
#pragma omp atomic
        outputBins[i][j] += threadPrivateBins[i][j];
      }
  }

}
```

Listing 3: Particle coordinate binning code. Parallel and correct implementation using reduction with thread-private containers and relying on strip-mining and loop splitting to enable automatic vectorization.

3. Thread-Private versus Global Container

Even though the solution in Listing 3 has satisfactory performance, it may be inconvenient in some applications in terms of data container structure. Indeed, we allocate thread-private storage on the stack of each thread, and once we exit the parallel region, these thread-private containers are gone.

```
#pragma omp parallel
  {
    ContainerType threadPrivateContainer; // Thread-private containers on stacks of each thread
    // ...
  }
// Outside the parallel region, all instances of threadPrivateContainer are gone
```

Listing 4: Thread-private containers are created on the stack of each thread.

We may reasonably want to have access to the thread-private data after the parallel region has ended. Reasons for wishing to retain thread-private data in complex applications may be various:

- our application may re-use the thread-private data in an additional data processing stage, for example, for statistical analysis to give us insight into load balancing across threads;
- we may want to perform reduction (aggregation) of the thread-private data from a separate parallel region — for example, in order to implement tree reduction instead of linear reduction;

To retain data outside the parallel region, we can create a global container for thread-private data, in which each thread has a "compartment" for its data. An example of such data structure is given in Listing 5. An added benefit of this approach is that if thread-private data size is large, we may allocate the global container on the heap and do it only once in the life of the application.

```
const int numberOfOpenMPThreads = omp_get_max_threads();
ContainerType globalContainer = new ContainerType[numberOfOpenMPThreads]; // Global container
#pragma omp parallel
  {
    const int numberOfThisThread = omp_get_thread_num(); // Query current thread ID
    ContainerType& threadPrivateContainer =
                            (ContainerType&) globalContainer[numberOfThisThread]; // ''compartment''
    // ...
  }
// Outside the parallel region, all data in globalContainer is still available
```

Listing 5: Global container with a "compartment" for each thread, allocated on the heap available outside the parallel region.

Specifically, in our binning application, two approaches to data layout may be taken. In one, the innermost array dimension maps threads (we will call it "threads-first layout"). In the other, outermost array dimension maps threads ("threads-last layout"). We will study the efficiency of these layouts.

```
// Approach one: threads-first global container of bin counts:
int threadsFirstGlobalContainer[nBinsX][nBinsY][numberOfOpenMPThreads];

// Approach two: threads-last global container of bin counts:
int threadsLastGlobalContainer[numberOfOpenMPThreads][nBinsX][nBinsY];
```

Listing 6: Threads-first and threads-last layouts for global container.

4. False Sharing

Before we study the effect of having a global container on performance, it will be helpful to discuss a situation that occurs in cache-coherent architectures called false sharing. False sharing degrades application performance, so it is important to learn to recognize situations in which it occurs and how to eliminate it.

False sharing occurs when two or more threads access the same cache line (but different data elements in it), and one of these accesses is a write.

Here is an explanation of what it means. Intel processors cache memory in 64-byte blocks called cache lines. Cache lines are always aligned on 64-byte addresses, so any given virtual memory address always maps to a specific cache line. Different cores can have copies of the same cache line, and when one core modifies a cache line, all other cores must re-fetch this line to maintain cache coherency. This introduces serialization into a parallel application and stresses the memory subsystem, degrading performance.

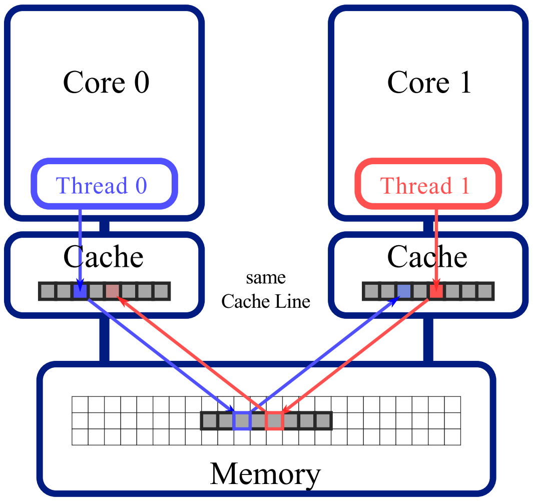

Figure 2: False sharing illustration.

It is important to distinguish false sharing from data races, also called race conditions (see Figure 3). Races occur when multiple threads access not just the same cache line, but the same data elements within that cache line, for writing. Data races, if not protected by mutexes, cause incorrect and unpredictable results. See Part 1 [1] for more information on data races and mutexes. False sharing, in contrast, does not bias results, it only degrades performance.

Effects of false sharing on performance depend on how often it occurs. As we will see soon, order of magnitude performance degradation is a very realistic scenario. Generally, to eliminate false sharing, the programmer must rearrange data structures so that different threads do not access the same cache lines for writing. This may require transposing the data structures and padding them.

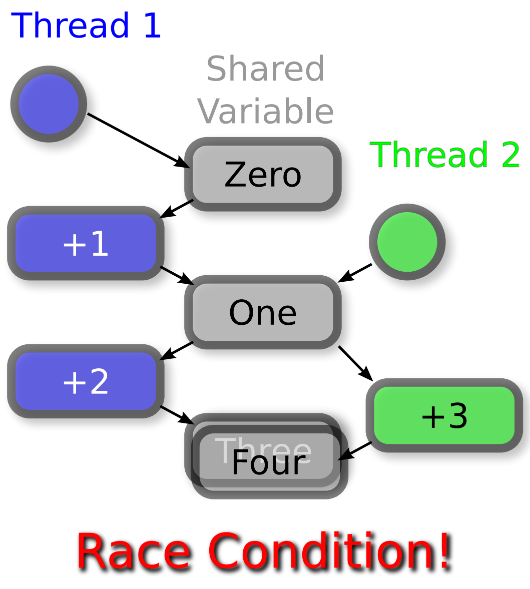

Figure 3: Data race illustration.

5. Binning with Global Containers

5.1. Threads-first Container Layout

To begin our exploration of false sharing, let's implement binning with the threads-first data container layout. The relevant code snippets are shown in Listing 7.

```
  int globalBins[nBinsX][nBinsY][nThreads]; // Global container in threads-first layout
#pragma omp parallel
  {
    const int iThread = omp_get_thread_num(); // Find this thread's own region in global container

    // ...(part of the code skipped)

      // Using global container to write into each thread's private compartment
      for (int c = 0; c < STRIP_WIDTH; c++)
              globalBins[iX[c]][iY[c]][iThread]++;
  }
```

Listing 7: Threads-first layout.

It is obvious that in this case false sharing will be very severe. In C and C++, multi-dimensional arrays are contiguous in memory in the last array index, which in this case maps threads. So the pattern of memory access by different threads is such that each 64-byte cache line containing 16 integers will be continuously bombarded for writes by 16 different threads. This corresponds to Case #1 in Figure 4.

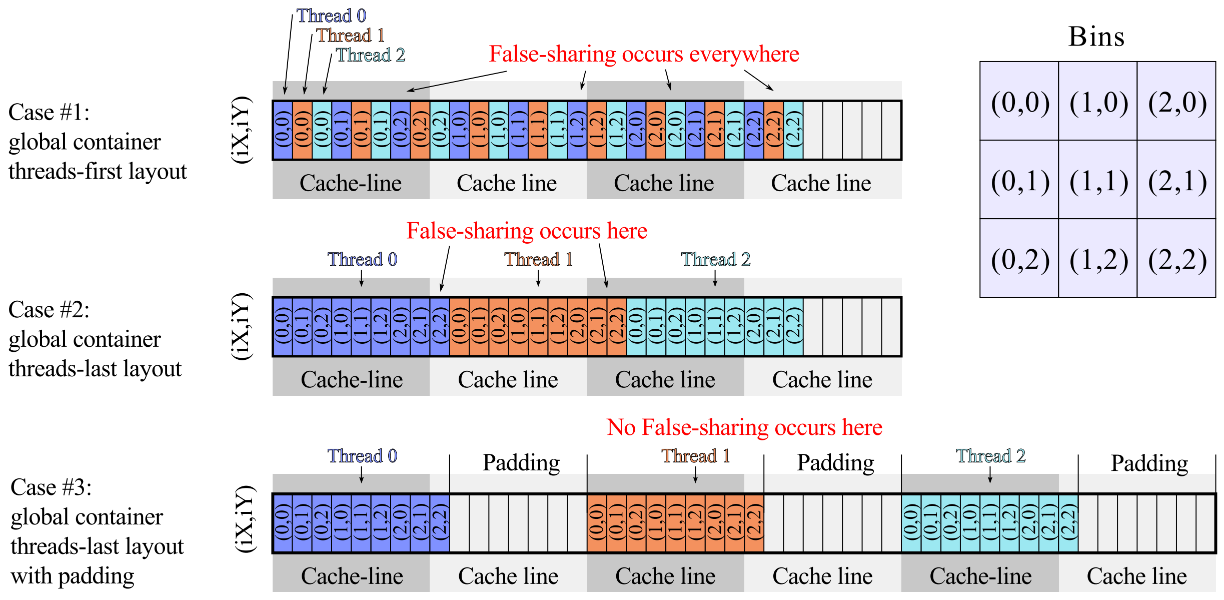

Figure 4: Pattern of access to data by different threads for threads-first, threads-last and padded threads-last container layouts.

Figure 4 illustrates the container layout for the case of `nBinsX==nBinsY==3` and 3 threads operating on the data set. For Case #1, bin `(0,0)` is represented by 3 consecutive entries in the data container, which are reserved for writing by threads 0, 1, 2, respectively. The same goes for bins `(0,1)`, `(0,2)`, etc. You can easily see how this layout generalizes to a larger container and a greater number of threads.

5.2. Threads-last Container Layout

Instead of putting areas for different threads consecutively in memory, we could transpose the container as shown in Listing 8. This layout corresponds to Case #2 in Figure 4. Here, thread 0 has a contiguous copy of the 2-dimension bins array to write in to, thread 1 has its own copy, and so on.

```
  int globalBins[nThreads][nBinsX][nBinsY]; // Global container in threads-last layout
#pragma omp parallel
  {
    const int iThread = omp_get_thread_num(); // Find this thread’s own region in global container

    // ...(part of the code skipped)

      // Using global container to write into each thread’s private compartment
      for (int c = 0; c < STRIP_WIDTH; c++)
              globalBins[iThread][iX[c]][iY[c]]++;
  }
```

Listing 8: Threads-last layout.

Obviously, this layout will relieve pressure on the cache because we will not have up to 16 different threads accessing the same cache line. At the same time, false sharing may still occasionally occur if the size of each thread’s compartment is not a multiple of the cache line width. For example, for our 3x3 container in Figure 4, bin `(2,2)` for thread 0 spills into the next cache line written to by thread 1.

Furthermore, even if the container size *was*, in fact, a multiple of 64 bytes, some false sharing could still occur because caches in Intel Xeon processors have aggressive prefetchers, which may fetch not just the requested cache line, but some cache lines in its neighborhood trying to preempt access to them. See Section 6.3 for a proof of this possibility. The resolution to both problems is discussed in the next section.

5.3. Padded Container

Finally, to prevent false sharing from happening, we may use the thread-last layout, but pad the container with some free space between the compartments reserved for different threads. A possible implementation for this is shown in Listing 8 and illustrated in Figure 4 as Case #3.

```
  const int containerSize = sizeof(BinsType);
  const int paddingBytes = 64; // Just one of the possible padding values
  const int paddedSize = (containerSize - containerSize%64) + paddingBytes;
  typedef char PaddedBinsType[paddedSize];
  const int nThreads = omp_get_max_threads();
  PaddedBinsType globalBins[nThreads];

#pragma omp parallel
  {
    const int iThread = omp_get_thread_num(); // Find this thread’s own region in global container
    BinsType& myBins = (BinsType&)(globalBins[iThread]);
    // ...(part of the code skipped)

      for (int c = 0; c < STRIP_WIDTH; c++)
              myBins[iX[c]][iY[c]]++;
  }
```

Listing 9: Padded threads-last layout.

6. Performance

6.1. System Configuration

All of the benchmarks presented in this section were taken on a Colfax ProEdge™ SXP8600 workstation based on a dual-socket Intel Xeon E5-2697 v2 processor (12 cores per socket, 24 physical cores with two-way hyper-threading). The Intel Xeon Phi coprocessor benchmarks presented in this section were taken using native execution on 7120P Xeon Phi coprocessor (61 physical cores with four-way hardware-threading) installed in that system. The Xeon Phi benchmarks are taken using the coprocessor alone, i.e., the CPU was not computing in tandem with the coprocessor. For compilation we used the Intel C++ compiler version 15.0.3.187 on a CentOS 7.0 Linux OS with Intel MPSS 3.5.

6.2. Performance Benchmarks

Figure 5 and 6 reports the performance benchmarks of the binning algorithm with different global container layout in single precision and in double precision.

For each benchmark the binning workload was repeated 10 times, and the average of the last 8 iteration is reported. We exclude the first two iterations because they tend to be much slower due to various initialization overheads on both Xeon CPUs and Xeon Phi coprocessors. They do not represent sustained performance that is achieved if the application has a long execution time. Variance in most cases was less than 1%, so we report results with two significant figures and without error bars. The performance is measured in MP/s, where 1 MP/s is 10^6 particles binned per second.

The four sets of bars in this Figures 5 and 6 correspond to three versions of the code:

1. "*Threads-first Layout*" is the code from Listing 7 corresponding to Case #1 in Figure 4,
2. "*Threads-last Layout*" is the code from Listing 8 corresponding to Case #2 in Figure 4,
3. "*Threads-last with Padding*" is the code from Listing 9 corresponding to Case #3 in Figure 4,
4. "*Reference (Part 2)*" is the reference implementation from Part 2 shown in Listing 3.

As we can see from the difference between the "threads-first" layout and the reference implementation, severe false sharing degrades performance on the Xeon processor by a factor of 30 in single precision and a factor of 15 in double precision. On the Xeon Phi coprocessor, the corresponding performance penalties are a factor of 17 and a factor of 10, respectively.

Once we transposed the storage container and started using the "threads-last" layout, performance recovered by a large factor. Still, the remaining false sharing events allowed Xeon to perform only at 50% of the reference implementation speed in single precision and 70% in double precision. On Xeon Phi, the effect of false sharing was less: the code achieved over 90% of the reference implementation performance.

Finally, with padding ("threads-last with padding" case), we recovered nearly 100% of the performance of the reference implementation. The conclusion is that with the appropriate data layout and appropriate padding, using a global container can give the same performance as using thread-private variables, and at the same time allow us to use heap allocation and retain access to the thread-private data outside of the parallel region.

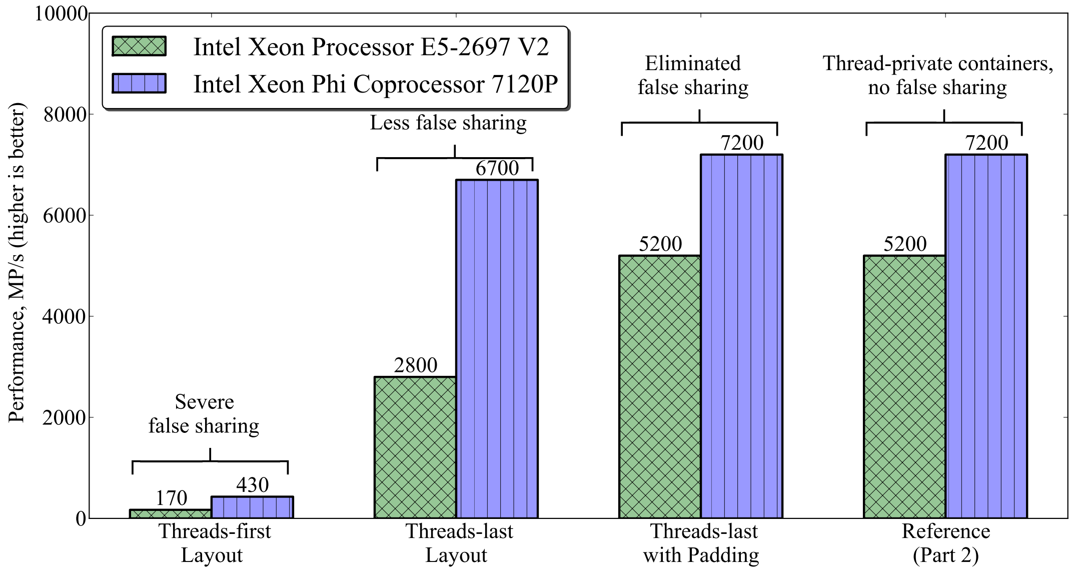

Figure 5: Benchmarks of the binning algorithm in single precision.

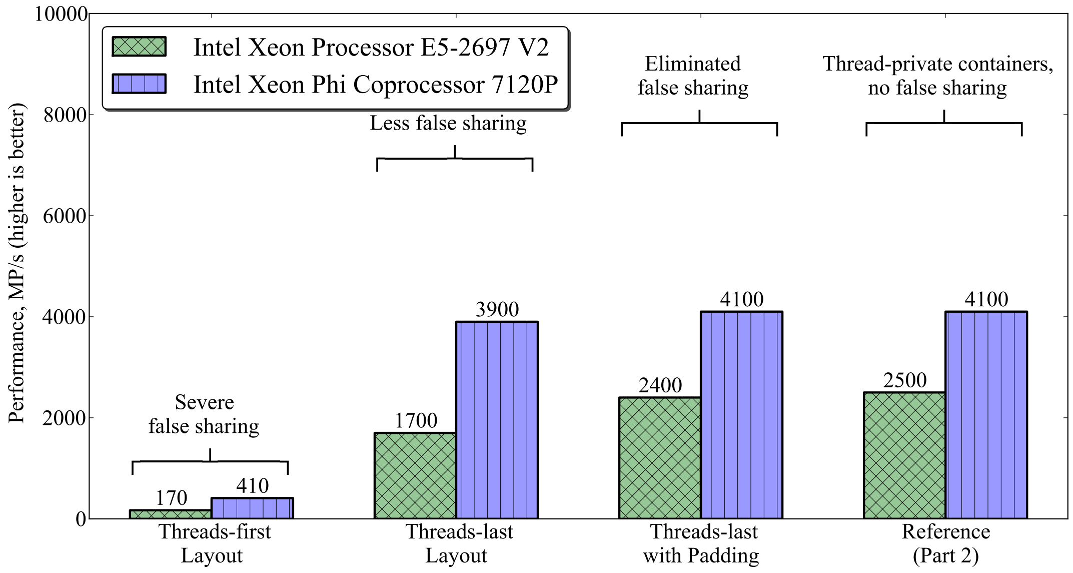

Figure 6: Benchmarks of the binning algorithm in double precision.

6.3. How Much Padding is Enough?

The padded containers used for the case “Threads-last with padding” have a padding value of 512 bytes on Intel Xeon processors and 64 bytes on Intel Xeon Phi coprocessors. These are minimum values of padding that allow performance to recover to the optimal case. They were obtained empirically by benchmarking the code with different values of padding. Results of that tuning are shown in Figure 7.

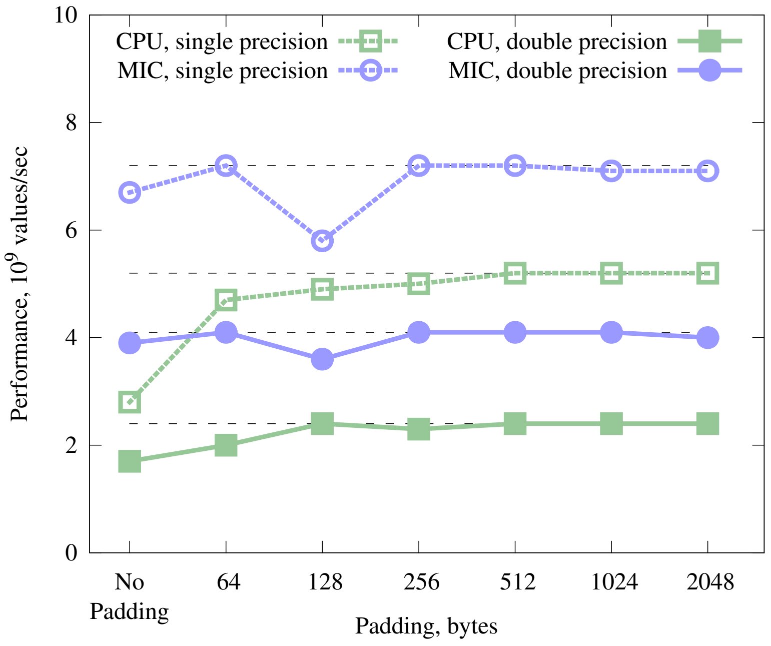

Figure 7: Effect of different padding sizes on the binning algorithm performance.

The figure shows that on Xeon Phi, performance recovers with just 64 byte padding – this is sufficient to eliminate false sharing. However, on Xeon, we need to pad to 512 bytes, which is 8 cache lines, before we achieve the same performance as in the reference case. The reason for such behavior is likely the aggressive nature of prefetching in Xeon processors. When a core requests a cache line, the prefetcher requests neighboring cache lines to be fetched as well in anticipation that the core will explore data in the local neighborhood. This is a good strategy for codes that perform streaming or localized data access, however, for the case where different threads access nearby “compartments” in a data container, it causes false sharing.

To implement different parameter values for Xeon and Xeon Phi, we used the preprocessor macro `__MIC__` as shown in the code listing below. The rest of the code was identical for both architectures.

```
#ifdef __MIC__
  const int paddingBytes =  64; // Padding value for Xeon Phi
#else
  const int paddingBytes = 512; // Padding value for Xeon
#endif
```

Listing 10: Patform-specific tuning parameters in an otherwise identical code for Xeon and Xeon Phi.

Macro `__MIC__` is defined by the Intel compiler when the compiler argument `-mmic` is specified, so the code is cross-compiled for the Intel MIC architecture. It is undefined when the code is compiled for general-purpose Intel processors or by non-Intel compilers.

7. Discussion

7.1. What We Learned

In this tutorial we discussed false sharing – a situation that occurs when multiple threads access for writing memory locations mapped to the same cache line (or to neighboring cache lines). The cautionary example demonstrated here shows that this situation may occur when programmers allocate memory containers for each thread's private data too densely in memory. The performance impact of false sharing can exceed an order of magnitude in Intel Xeon and Intel Xeon Phi processors, so it is important to detect and avoid it. Making sure that each thread accesses only its own cache lines for writing, and additionally padding between threads' write domains helps to avoid false sharing.

This tutorial concludes the 3-part series of tutorials on select topics of optimization for Intel architecture. Below is the complete list of the three parts:

1. Part 1 discusses optimization of multi-threading
2. Part 2 demonstrates optimization of vectorization
3. Part 3 cautions about a common pitfall in multi-threaded applications, false sharing.

Of course, the topics covered in these tutorials are only a subset of the optimization techniques necessary for handling parallelism and memory hierarchy in modern processors. Additional resources listed below may help to find additional information.

7.2. Additional Resources

For additional discussion of parallel programming and optimization for Intel architecture, refer to:

1. Our book, "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors" [4]
2. Upcoming Web-based training such as the HOW (Hands-On Workshop) series
3. Recorded past trainings such as this one
4. Intel Modern Code program

References

[1] Optimization Techniques for the Intel MIC Architecture. Part 1 of 3: Multi-Threading and Parallel Reduction, 2015.
http://colfaxresearch.com/?p=6.

[2] Optimization Techniques for the Intel MIC Architecture. Part 2 of 3: Strip-Mining for Vectorization, 2015.
http://colfaxresearch.com/?p=709.

[3] Optimization Techniques for the Intel MIC Architecture. Part 3 of 3: False Sharing and Padding, 2015 *(landing page for this paper)*.
http://colfaxresearch.com/?p=1231.

[4] Andrey Vladimirov, Ryo Asai, and Vadim Karpusenko. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. Colfax International, 2nd edition, March 2015.
http://www.colfax-intl.com/nd/xeonphi/book.aspx.

[5] Parallel Computing in the Search for New Physics at LHC.
http://colfaxresearch.com/?p=24.