Optimization Techniques for the Intel MIC Architecture. Part 2 of 3: Strip-Mining for Vectorization

Andrey Vladimirov

Colfax International

June 26, 2015

Abstract

This is part 2 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 data parallelism. Our focus is automatic vectorization and exposing vectorization opportunities to the compiler.

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

The optimization technique discussed in this paper leads to code vectorization, which results in an order of magnitude performance improvement on an Intel Xeon processor. Performance on Xeon Phi compared to that on a high-end Xeon is 1.4x greater in single precision and 1.6x greater in double precision.

Table of Contents

1. Introduction

This paper continues our 3-part series of educational publications on performance optimization in applications for Intel Xeon Phi coprocessors. First part [2] discussed thread parallelism. In this part, we focus on some aspects of data parallelism and vectorization.

Optimization of vectorization on Intel Xeon processors and Intel Xeon Phi coprocessors requires exposing the vectorization opportunities, data structures designed for contiguous memory access, data alignment, regularization of vectorization pattern, and compiler hints for pointer disambiguation, alignment guarantees and loop count. An extensive discussion of these techniques is presented in our book, “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, 2nd Edition” [3]. In this paper, we demonstrate two techniques: exposing vectorization opportunities with strip-mining and data alignment.

2. Motivating Example: Particle Binning

For an illustration we will use the same example as in [1]: 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 [4]), in Monte Carlo simulations and also in statistics where data transformation and binning takes place.

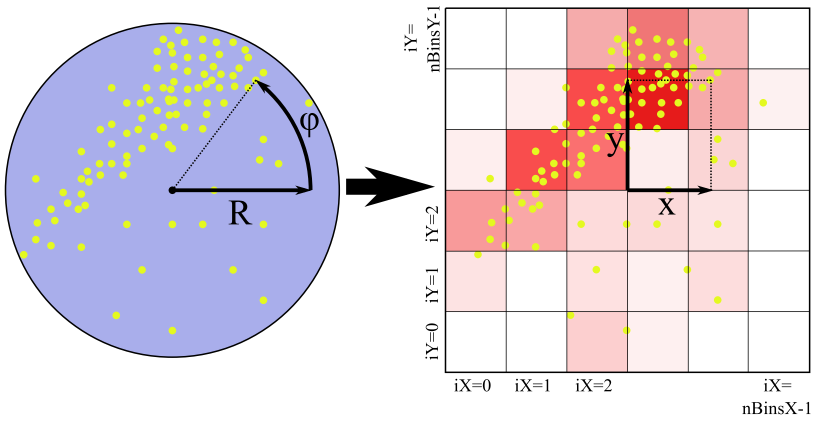

Figure 1: Workload illustration: take polar particle coordinates and compute particle counts in bins on the Cartesian grid.

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.

1

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.

In part 1 [2] we implemented multi-threading in this code and the result is shown in Listing 3.

```
void BinParticles_1(const InputDataType  & inputData, BinsType & outputBins) {
  // Thread-parallel implementation. Race conditions are avoided by using
  // parallel reduction with thread-private containers
#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 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 thread-private counter:
      threadPrivateBins[iX][iY]++;
    }
    // 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: Binning code. Parallel and correct implementation using reduction with thread-private containers.

3. Vectorization

Parallel code in Listing 3 uses OpenMP to distribute the calculation across multiple cores of the processor that it is running on. This level of parallelism is absolutely essential for extracting performance out of multi- and manycore processors. Indeed, in our previous publication [2], we reported that going from serial code in Listing 2 to parallel code in Listing 3 leads to speedups of 25x and 100x on an 24-core Intel Xeon and a 61-core Intel Xeon Phi processor, respectively.

At the same time, the parallel code shown above does not handle another level of parallelism: vectorization inside of every core. Cores of our Intel Xeon E5-2697 v2 processor support 256-bit wide AVX2 instructions, and cores of the Intel Xeon Phi 7120P coprocessor support 512-bit wide IMCI instructions. These are SIMD instructions which can perform a *S*ingle *I*nstruction on *M*ultiple *D*ata elements in one operation. That is, each core of a Xeon processor can perform an addition, multiplication, sine or cosine calculation, or type conversion on 8 single precision or 4 double precision numbers at a time. Each core of a Xeon Phi can do the same on 16 single precision or 8 double precision numbers. However, the calls to these short vector instructions are not automatic, i.e., if the software does not call for these instructions, hardware will not automatically apply them.

3.1. Automatic Vectorization

Intel compilers are able to automatically vectorize loops and certain other constructs. Automatic vectorization is enabled at the default optimization level `-O2`. To see the result of automatic vectorization, we can request an optimization report from the Intel compiler. For an example, see code in Listing 4.

```
// File try-vector.cc
#include <cmath>
void ComputeSine(double* x, int n) {
  for (int i = 0; i < n; i++) // This loop is a target for automatic vectorization
    x[i] = sin(x[i]); // The sine function can be processed with vectors
}
```

```
vega@lyra% icpc -qopt-report -c try-vector.cc # Compiling...
icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location
vega@lyra% cat try-vector.optrpt # Viewing the optimization report
...
LOOP BEGIN at try-vector.cc(4,3)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

Listing 4: Code with vectorizable loop and result of automatic vectorization by the Intel C++ compiler.

To have the loop automatically vectorized means that the compiler will represent the loop in `i` as a loop with a stride of 4, 8, 16, or whatever is the vector width in the target architecture. Inside of every iteration, data will be loaded from memory into the 4-, 8- or 16-wide vectors, the program of the loop will be executed in each respective vector lane, and the result will be stored back in memory. The compiler will automatically handle cases where the number of iterations is not a multiple of the vector width. In these cases, a remainder loop will be implemented. Similarly, the compiler will handle architectural restrictions on the alignment of data using a peel loop when necessary (see Section 4.4). For more details on automatic vectorization, see our book [3].

1

3.2. Vectorization Opportunity

The binning code has a single loop that goes over all particle coordinates (line 14 in Listing 3). We found a way to parallelize this loop across cores, because different particles can be processed independently as long as parallel reduction is implemented correctly (see Part 1). The same loop, in principle, can be parallelized across vector lanes. This is the loop targeted for vectorization in the binning code:

```
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 thread-private counter:
  threadPrivateBins[iX][iY]++;
}
```

Listing 5: Binning code. Parallel and correct implementation using reduction with thread-private containers.

If this loop was vectorized, then the calculation of trigonometric functions, multiplication, addition and type conversion can all be done with vectors as in shown in Figure 2.

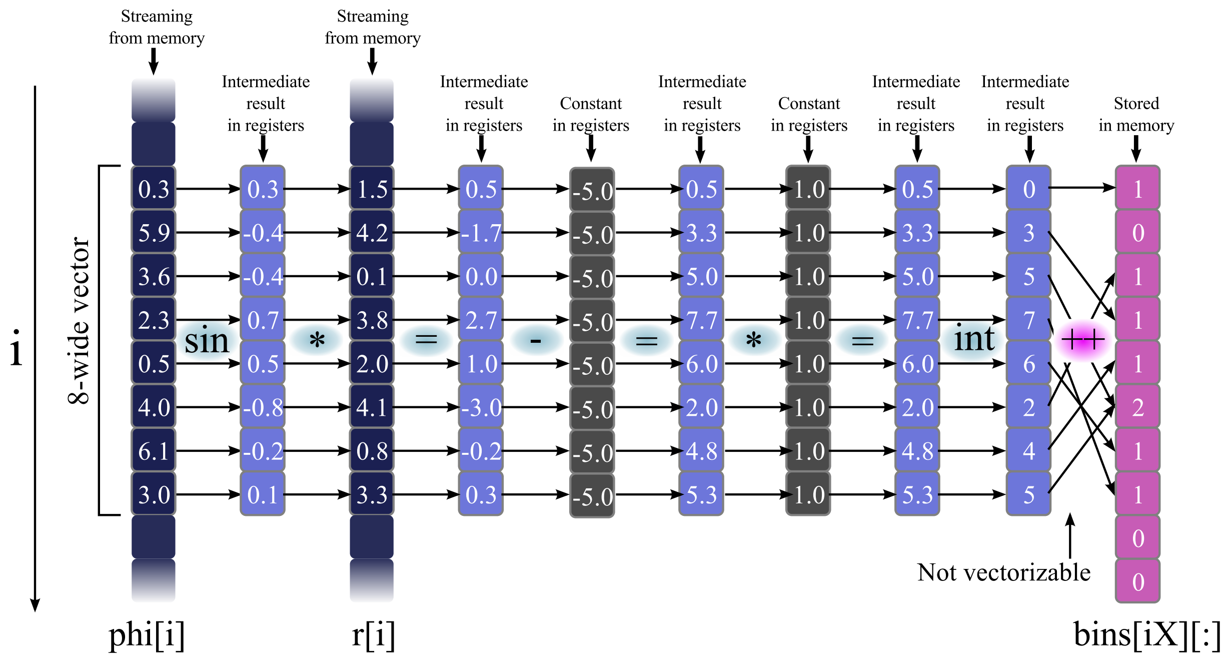

Figure 2: Vectorization opportunity in the binning code.

All calculation steps up until the increment of the bin counters (the ++ operation) can be vectorized by processing a block of values `phi[i]` and `r[i]` in a vector. However, the increment cannot be vectorized because it does not have SIMD semantics: `iY` can have scattered or colliding values in different vector lanes, and the same is true of `iX`. [1]

It is obvious to us that after computing a vector of values `iX` and `iY` the processor must switch from vector instructions to scalar instructions to increment the bins. Let's see what happens when the compiler processes this loop. We can use `-qopt-report=5` to increase the report verbosity.

```
vega@lyra% icpc -c main.cc -qopenmp -qopt-report=5
icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location
vega@lyra% cat main.optrpt
...
LOOP BEGIN at ../code/main.cc(123,5)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed FLOW dependence between threadPrivateBins line 134
                  and threadPrivateBins line 134
   remark #15346: vector dependence: assumed ANTI dependence between threadPrivateBins line 134
                  and threadPrivateBins line 134
LOOP END
```

Listing 6: Optimization report of the binning code.

It appears that the compiler is not able to automatically vectorize this loop. The optimization report indicates FLOW and ANTI dependencies in the line of code that corresponds to line 11 in Listing 5. In this case, the dependencies are due to the fact that values of `iX` and `iY` have unpredictable dependence on the loop counter `i`, and therefore SIMD semantics does not apply.

Even though the compiler refused to vectorize this loop, this is not the end of the story. Automatic vectorization is still possible. We just need to expose the opportunity for vectorization. In the next section we will use a ubiquitous programming technique called strip-mining to transform the code into a form where the compiler can recognize safe paths for automatic vectorization.

4. Optimization

4.1. Loop Splitting

The compiler refused to vectorize a loop because a non-vectorizable operation is present in it. How about this idea: let's split this loop into two loops, one of which will contain only vector operations and the other will contain only scalar operations as in Listing 7.

```
for (int i = 0; i < inputData.numDataPoints; i++) { // This loop can be vectorized
  const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
  const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);
  const int iX = int((x - xMin)*binsPerUnitX);
  const int iY = int((y - yMin)*binsPerUnitY);
}

for (int i = 0; i < inputData.numDataPoints; i++) // This loop will remain scalar
  threadPrivateBins[iX][iY]++; // But of course, this is not going to compile
```

Listing 7: Optimization idea: split the loop to have one vector and one scalar loop.

Of course, this is not going to work in this exact form. In fact, it will not even compile because `iX` and `iY` are not defined in the second loop. However, the compiler will have one clean vector loop that it can vectorize. This will potentially speed up the processing of the arithmetic needed for coordinate transformation and index calculation. Now let's fix the code and implement loop splitting correctly. For logical correctness, we will have to carry an array of indices `iX` and `iY` from one loop to another.

```
int iX[inputData.numDataPoints]; // Temporary storage container for iX
int iY[inputData.numDataPoints]; // Temporary storage container for iY

for (int i = 0; i < inputData.numDataPoints; i++) { // This loop can be vectorized
  const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
  const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);
  iX[i] = int((x - xMin)*binsPerUnitX);
  iY[i] = int((y - yMin)*binsPerUnitY);
}

for (int i = 0; i < inputData.numDataPoints; i++) // This loop will remain scalar
  threadPrivateBins[iX[i]][iY[i]]++;
```

Listing 8: Loop splitting refined: have a temporary storage container to carry indices from one loop into the other.

The implementation in Listing 8 is formally correct, but it, too, has problems. We know that `numDataPoints` is of order 2^{27}, so arrays `iX` and `iY` combined have 1 GiB of data. This means that, first, we need to increase the stack size limitation of our application to over 1 GiB, or use heap allocation for `iX` and `iY`. With a large stack, we may run out of memory because each thread has its own stack. Heap allocation inside the function is inefficient, so we would have to move `iX` and `iY` to the scope of function caller. Second, the amount of data in `iX` and `iY` is comparable to that in `r` and `phi`. So now, in addition to reading particle data, we need to read and write a comparable amount of data for indices, which may as much as triple our memory traffic.

Looks like with loop splitting we are on the right track in terms of exposing vectorization, but we need to do something to avoid unnecessary memory overhead.

4.2. Strip-Mining

We can resolve the memory overhead problem using strip-mining. This is a programming technique that transforms a single loop into two nested loops. The outer loop strides through "strips" of the iteration space, and the inner loop operates on the iterations inside the strip ("mining" it). Listing 9 demonstrates a generic strip-mining transformation.

```
// Original loop:
for (int i = 0; i < n; i++)
  { /* ... do work */ }

// Strip-mining converts the original loop into two nested loops:
const int STRIP_WIDTH=64;
for (int ii = 0; ii < n; ii += STRIP_WIDTH)
  for (int i = ii; i < ii + STRIP_WIDTH; i++)
    { /* ... do work */ }
```

Listing 9: Illustration of the strip-mining loop transformation.

This transformation is useful in several cases:

1. Expose data parallelism to enable automatic vectorization;
2. Allow vectorization to co-exist with multi-threading;
3. As a basis for other techniques such as loop tiling (see, e.g., [5] or [3]).

Strip size must usually be chosen as a multiple of the vector length in order to facilitate the vectorization of the inner loop. The optimal value for `STRIP_WIDTH` is a tuning parameter that must usually be determined experimentally. Furthermore, if the iteration count `n` is not a multiple of the strip size, then the programmer must implement a remainder loop for `n%STRIP_WIDTH` iterations at the end of the loop.

Strip-mining can help us to reduce the memory footprint of the loop-split code. Applying strip-mining to the code in Listing 8, we get to the following:

```
  for (int ii = 0; ii < inputData.numDataPoints; ii += STRIP_WIDTH) {
    int iX[STRIP_WIDTH], iY[STRIP_WIDTH]; // STRIP_WIDTH=16

    const FTYPE* r   = &(inputData.r[ii]);   // Find the current strip
    const FTYPE* phi = &(inputData.phi[ii]);

    for (int c = 0; c < STRIP_WIDTH; c++) {      // Vector loop
      // 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);
    }

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

Listing 10: Loop splitting combined with strip-mining.

The code in Listing 10 is more efficient than the original code from Listing 3 because the arithmetically intensive part of it is vectorized. It is also more efficient than Listing 8 because the memory footprint is increased only marginally. Optimization report confirms that vectorization succeeded for the first of the two split loops but failed for the second, as we intended.

```
LOOP BEGIN at ../code/main.cc(176,7)
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at ../code/main.cc(187,7)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
LOOP END
```

Listing 11: Optimization report for the binning code with split loops and strip-mining.

The size of the value `STRIP_WIDTH` was chosen empirically, and the most efficient value turned out to be 16 for all cases: single precision and double precision, Intel Xeon processor and Intel Xeon Phi coprocessor. It means that the loops in `c` have either 1, or 2 or 4 vector iterations, depending on the precision and platform. Importantly, `STRIP_WIDTH` had to be defined as a compile-time constant in the code for the compiler to choose a good vectorization strategy.

4.3. Note on Loop Remainder

Our code assumes that `n` is a multiple of `STRIP_WIDTH`. If this is not the case, the most efficient way to generalize the code is to insert a remainder loop to process `n%STRIP_WIDTH` operations:

```
for (int ii = 0; ii < inputData.numDataPoints; ii += STRIP_WIDTH) {
  /* ... */
  for (int c = 0; c < STRIP_WIDTH; c++) // Vector loop
    { /* ... */ }
}
for (int i = inputData.numDataPoints%STRIP_WIDTH; i < inputData.numDataPoints; i++)
  { /* ... */ }  // Remainder loop
```

Listing 12: Efficient way to implement a remainder loop.

It is also possible to implement a “smart” dynamic upper bound in the loops to avoid redundant code (see code below), however, the reader may verify that this approach dramatically reduces performance.

```
for (int ii = 0; ii <= inputData.numDataPoints; ii += STRIP_WIDTH) {
  /* ... */
  // ``Smart'' loop bounds used when numDataPoints%STRIP_WIDTH != 0
  int cMax = (ii+STRIP_WIDTH < inputData.numDataPoints ? ii+STRIP_WIDTH : inputData.numDataPoints);
  for (int c = 0; c < cMax; c++) // Vector loop
    { /* ... */ }
}
```

Listing 13: Elegant, but inefficient way to deal with loop remainder.

Despite our progress, optimization is not quite done because with vectorization enabled, we need to worry about an architectural requirement of data alignment. This topic is discussed in the next section.

4.4. Data Alignment

When the calculation reaches the loop in line 7 of Listing 10, the core will have to load a block of values of `r` and `phi` into a vector register for subsequent manipulations. Intel Xeon Phi architecture has a restriction: the memory address of this block of values must be a multiple of 64 bytes. In other words, the block must be aligned on a 64-byte boundary. In Intel Xeon architectures, alignment requirements are relaxed, however, having the block begin on a 16-byte boundary (for SSE instructions) or 32-byte boundary (for AVX instructions) may be beneficial for performance.

What happens if at runtime the data address is not aligned? The way that Intel compilers implement automatic vectorization, the code will still work and produce correct results. However, performance may suffer if the data is misaligned, because the compiler will have to implement a peel loop to get to the first aligned element (see Figure 3). Because our loops are short (`STRIP_WIDTH`=16), this peel loop together with the possible remainder loop may take a relatively large amount of time.

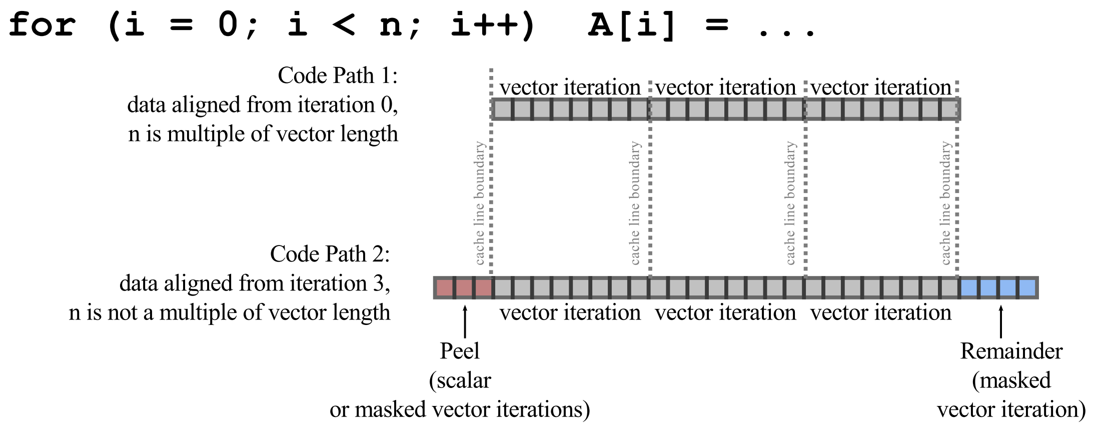

Figure 3: Alignment requirement may necessitate a peel loop.

To avoid situations of misaligned data, we have to allocate the containers `r` and `phi` on an aligned boundary. Because `STRIP_WIDTH` is 16, we will have alignment not only for the first iteration in `ii`, but for all subsequent iterations, because `ii` will always be a multiple of 16.

Aligned allocation can be done with the intrinsic `_mm_malloc` supported by Intel compilers as shown below. To free such data containers, `_mm_free` must be used.

```
rawData.r   = (FTYPE*) _mm_malloc(sizeof(FTYPE)*n, 64);
rawData.phi = (FTYPE*) _mm_malloc(sizeof(FTYPE)*n, 64);
/* ... */
_mm_free(rawData.r);
_mm_free(rawData.phi);
```

Listing 14: Aligned allocation of data on the heap.

When the data containers are allocated, peel loop is not necessary, so that already improves performance. However, the compiler is not aware of the data alignment guarantee and will implement a check for it anyway. To tell the compiler that data in a loop is aligned, the programmer can use

1

`#pragma vector aligned`, which will allow the compiler to drop the check and just start processing the loop with vector instructions. This pragma is illustrated in Listing 15 along with specifier `__attribute__((aligned(64)))`, which aligns a stack array on a 64-byte boundary.

```
  for (int ii = 0; ii < inputData.numDataPoints; ii += STRIP_WIDTH) {

    int iX[STRIP_WIDTH] __attribute__((aligned(64))); // Align temporary containers
    int iY[STRIP_WIDTH] __attribute__((aligned(64))); // on 64-byte boundary

    const FTYPE* r   = &(inputData.r[ii]);   // Guaranteed to be aligned
    const FTYPE* phi = &(inputData.phi[ii]); // on 64-byte boundaries

    // Telling compiler to drop checks for alignment
#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; alignment does not matter
    for (int c = 0; c < STRIP_WIDTH; c++)
    threadPrivateBins[iX[c]][iY[c]]++;
  }
```

Listing 15: Final optimized binning algorithm with loop splitting, strip-mining, data alignment and alignment hints.

4.5. Thread Affinity

In Part 1 of this series [2], we implemented thread parallelism using OpenMP. In this paper we are using a multi-threaded code which scales across all threads of either the host CPU, or of the coprocessor. Here, Part 2, in addition to using a multi-threaded code, we tuned parallel performance by fixing the assignment of OpenMP threads to physical cores[1]. This assignment is known as thread affinity. With the Intel OpenMP library, it can be controlled by the environment variable `KMP_AFFINITY`.

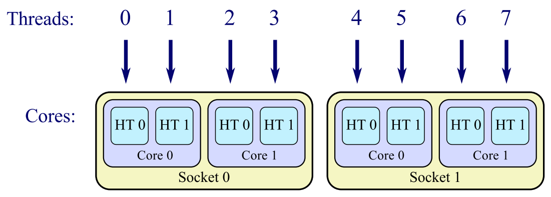

Figure 4: OpenMP thread affinity pattern effected by setting the environment variable `KMP_AFFINITY=compact`.

With affinity set, performance improves because threads do not migrate across cores and the pattern of thread binding combined with the pattern of data sharing may result in better locality of data access. See, e.g., [6] or [3] for more information about the impact of thread affinity on performance.

Empirically we found that the optimal affinity pattern is `compact`, i.e., we place threads with nearby numbers as close to each other as possible as shown in Figure 4. This results in marginal performance improvement on the host CPU and up to 10% improvement on an Intel Xeon Phi coprocessors compared to not stetting affinity.

[1]In hindsight, we should have discussed affinity settings in Part 1.

5. Performance

5.1. 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.

5.2. Performance Benchmarks

Figure 5 and 6 reports the performance benchmarks of the binning algorithms in single precision and double precision. Variance in most cases was in less than 1%, so we report results without error bars with two significant figures. The performance is measured in MP/s, where 1 MP/s is 10^6 particles binned per second.

The three cases reported in the figure correspond to three versions of the code:

1. "*Affinity control*" is the parallel code without vectorization from Listing 3. This is where we left off in Part 1 (plus affinity control which is not really a code optimization but an environment optimization technique),
2. "*Vectorization, alignment*" is the code with vectorization Listing 10 in which data containers are aligned as shown in Listing 14. This is the core result of the present tutorial: enabling vectorization.
3. "*Vectorization, alignment, hints*" is the case where in addition to aligning the containers, we gave a compiler hint of data alignment (`#pragma ivdep`) as shown in Listing 15. This is the final optimized version that tweaks the vectorized code to make vectorization more efficient.

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.

For reference, we also present results from Part 1 ("Baseline code", "Reduction with Atomics", "Reduction with Private Variables"). Results of Part 2 build on top of the success reported in Part 1 and add performance improvement by another large factor.

5.3. Performance Analysis

Figure 5 and 6 report on the entire story. We started with a simple application that produced correct result, but was single-threaded and scalar (not vectorized). This application was quite miserable on Xeon Phi, yielding only 10% of the performance that the Xeon processor could deliver. In Part 1, we implemented multi-threading which gave the application a boost commensurate with the number of cores on each respective platform. However, Xeon Phi was still behind Xeon in total performance. Finally, in

Part 2, we implemented vectorization, and the performance experienced significant acceleration. This 2x-3x boost on Xeon illustrates how much performance can be "left on the table" even on CPUs if a parallel application is not thoroughly optimized. Additionally, after optimization, Xeon Phi was able to perform 40 to 60 percent faster than a high-end Xeon of a comparable thermal design power.

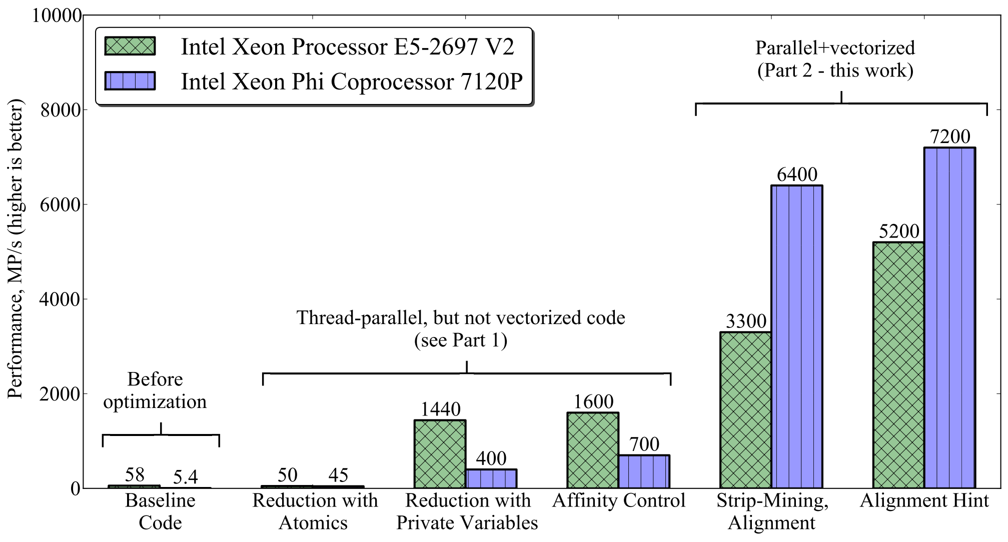

Figure 5: Benchmarks of the binning algorithm in single precision.

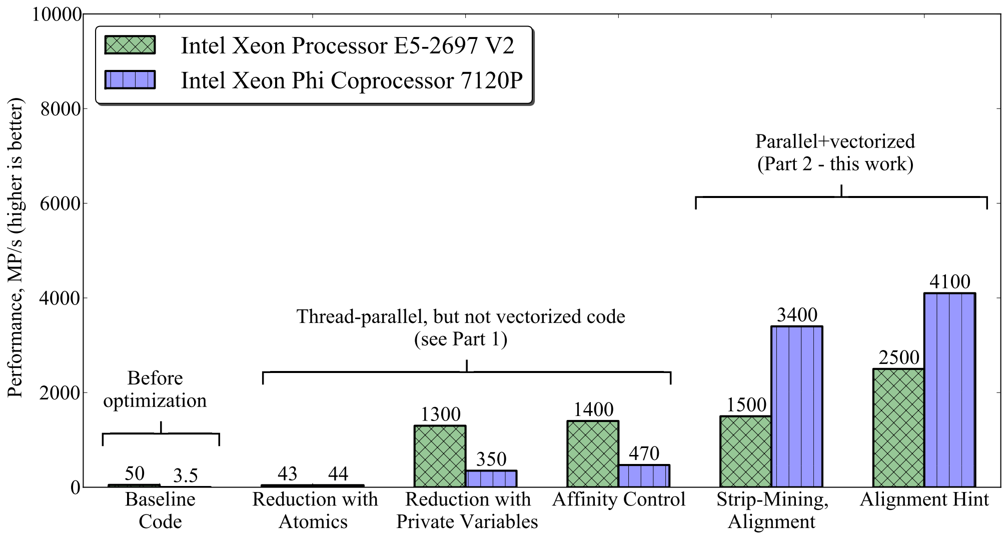

Figure 6: Benchmarks of the binning algorithm in double precision.

1

6. Discussion

6.1. What We Learned

In this work we discussed vectorization, which is another level of parallelism in Intel Xeon processors and Intel Xeon Phi coprocessors. We demonstrated a case (a particle binning application) where automatic vectorization fails because performance-critical loops contain both vectorizable and non-vectorizable operations. To improve application performance, we modified the code in a way that helps compiler in the task of automatic vectorization. Code modifications involved loop splitting combined with the strip-mining technique and data alignment hints. As a result, the data-parallel part of the application (particle coordinate transformation and index calculation) was vectorized, while the sequential part (incrementing bin counters) remained scalar.

The effect of vectorization in our example application was very significant. On an Intel Xeon processor, vectorization increased performance of the parallel code by a factor of 3.3 in single precision and by a factor or 1.8 in double precision. On an Intel Xeon Phi coprocessor, which has wider vectors, performance increased by a factor of 10.3 and 8.7, respectively.

In the end, we can compare the performance of a single Intel Xeon Phi coprocessor 7120P with 61 cores to that of a two-way Intel Xeon CPU E5-2697 V2 with a total of 24 cores. Xeon Phi is is faster than Xeon by 1.4x in single precision and 1.6x in double precision in our binning application.

6.2. What's Next

This paper is part 2 of a 3-part series of tutorials on selected performance optimization techniques. In part 3 we will revisit thread parallelism and experience a close (and victorious) encounter with another enemy of performance: false sharing. Stay tuned!

References

[1] Andrey Vladimirov. Optimization Techniques for the Intel MIC Architecture. Part 2 of 3: Strip-Mining for Vectorization, 2015 *(landing page for this paper)*.
http://colfaxresearch.com/?p=709.

[2] Ryo Asai and Andrey Vladimirov. Optimization Techniques for the Intel MIC Architecture. Part 1 of 3: Multi-Threading and Parallel Reduction, 2015.
http://research.colfaxinternational.com/post/2015/05/29/Techniques-1of3.aspx.

[3] 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.

[4] Parallel Computing in the Search for New Physics at LHC.
http://research.colfaxinternational.com/post/2013/12/02/LHC.aspx.

[5] Andrey Vladimirov. Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors.
http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx.

[6] Rob Farber. Power Profiling Shows Simple Changes To Save Megawatts of Power On Leadership Supercomputers.
http://www.techenablement.com/power-profiling-shows-simple-changes-save-megawatts-power-leadership-supercomputers/.

1