Test-driving Intel® Xeon Phi™ coprocessors with a basic N-body simulation

Andrey Vladimirov
Stanford University
and
Vadim Karpusenko
Colfax International

January 7, 2013

Abstract

Intel® Xeon Phi™ coprocessors are capable of delivering more performance and better energy efficiency than Intel® Xeon® processors for certain parallel applications[1]. In this paper, we investigate the porting and optimization of a test problem for the Intel Xeon Phi coprocessor. The test problem is a basic N-body simulation, which is the foundation of a number of applications in computational astrophysics and biophysics. Using common code in the C language for the host processor and for the coprocessor, we benchmark the N-body simulation. The simulation runs 2.3x to 5.4x times faster on a single Intel Xeon Phi coprocessor than on two Intel Xeon E5 series processors. The performance depends on the accuracy settings for transcendental arithmetics. We also study the assembly code produced by the compiler from the C code. This allows us to pinpoint some strategies for designing C/C++ programs that result in efficient automatically vectorized applications for Intel Xeon family devices.

Contents

[1]Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

1 What Intel Xeon Phi coprocessors bring to the table

Intel Xeon Phi coprocessor is a symmetric multiprocessor in the form factor of a PCI express device. Intel's James Reinders has defined the purpose of Intel Xeon Phi coprocessors in the following way[2]: "*Intel Xeon Phi coprocessors are designed to extend the reach of applications that have demonstrated the ability to fully utilize the scaling capabilities of Intel Xeon processor-based systems and fully exploit available processor vector capabilities or memory bandwidth.*" It cannot be used as a stand-alone processor. However, up to eight Intel Xeon Phi coprocessors can be used in a single chassis[3]. Each coprocessor features more than 50 cores clocked at 1 GHz or more, supporting 64-bit x86 instructions. The exact number of cores depends on the model and the generation of the product. These in-order cores support four-way hyper-threading, resulting in more than 200 logical cores. Cores of Intel Xeon Phi coprocessors are interconnected by a high-speed bidirectional ring, which unites L2 caches of the cores into a large coherent aggregate cache over 25 MB in size. The coprocessor also has over 6 GB of onboard GDDR5 memory. The speed and energy efficiency of Intel Xeon Phi coprocessors comes from its vector units. Each core contains a vector arithmetics unit with 512-bit SIMD vectors supporting a new instruction set called Intel Initial Many-Core Instructions (Intel IMCI). The Intel IMCI include, among other instructions, the fused multiply-add, reciprocal, square root, power and exponent operations, commonly used in physical modeling and statistical analysis. The theoretical peak performance of an Intel Xeon Phi coprocessor is 1 TFLOP/s in double precision. This performance is achieved at the same power consumption as in two Intel Xeon processors, which yield up to 300 GFLOP/s.

In order to completely utilize the full power of Intel Xeon Phi coprocessors (as well as Intel Xeon-based systems), applications must utilize several levels of parallelism:

1. task parallelism in distributed memory to scale an application across multiple coprocessors or multiple compute nodes,
2. task parallelism in shared memory to utilize more than 200 logical cores,
3. and at last, but definitely not the least, — data parallelism to employ the 512-bit vector units.

The novelty of developer experience in Intel Xeon Phi coprocessor is the continuity of the programming model between Xeon processor and Xeon Phi coprocessor programming:

1. The same C, C++ or Fortran code can be compiled into applications for Intel Xeon processors and Intel Xeon Phi coprocessors. Cross-platform porting is possible with only code re-compilation;
2. The same parallel frameworks are supported by the Intel Xeon and Intel Xeon Phi architectures. MPI can be used to scale in distributed or shared memory with Intel Xeon Phi coprocessors acting as individual cluster nodes. OpenMP, Intel Cilk Plus, Intel Threading Building Blocks, and other shared memory frameworks can be used to split the work homogeneously between the processor or coprocessor cores. The Intel Math Kernel Library provides highly optimized standard functions for both platforms.
3. The same software development tools can be used for both platforms: Intel compilers, Intel VTune Parallel Amplifier XE to profile and optimize the applications, Intel Debugger and Intel Inspector to diagnose critical and logical issues.
4. Similar optimization methods benefit applications on both platforms.

We were fortunate to have early access to Intel Xeon Phi coprocessors. Porting scientific applications to the new architecture has taught us that, even though there are some differences in the architecture of Intel Xeon processors and Intel Xeon Phi coprocessors, it is a general rule that similar methods of optimization benefit both architectures. In this paper, we will share some of the optimization practices that we found effective, and demonstrate their impact on the performance of both architectures. Our starting point is an unoptimized parallel implementation of the general N-body problem. Then we use unit-stride vectorization in order to improve performance. Finally, we relax the floating point arithmetics accuracy control and gain additional speed-up, without sacrificing the code simplicity.

[2]http://software.intel.com/sites/default/files/blog/337861/reindersxeonandxeonphi-v20121112a.pdf
[3]http://www.prweb.com/releases/2012/11/prweb10124930.htm

2 N-body simulation: basic algorithm

The N-body simulation refers to the computational problem of solving the equations of motion of gravitationally or electrostatically interacting particles. These problems are used in astrophysics to model the motion of self-gravitating systems, such as planetary systems, galaxies and cosmological structures, and in molecular physics to model the dynamics of complex molecules and atomic structures.

The general form of particle-particle interaction in N-body problems is given by Equation (1):

$$\vec{F}_i = KC_i \sum_{j \neq i} C_j \frac{\vec{R}_j - \vec{R}_i}{|\vec{R}_j - \vec{R}_i|^3}. \tag{1}$$

Here $\vec{R}_i$ are the position vectors of particles. This equation expresses $\vec{F}_i$, the force on particle i exerted by all other particles. The interaction has inverse-square dependence on distance, i.e., the magnitude of the force exerted by particle j on particle i is inversely proportional to the square of the distance between these particles, $|\vec{R}_j - \vec{R}_i|$. For the gravitational N-body problem, the coupling K is given by the gravitational constant $G \approx 6.674 \cdot 10^{-11}$ m^3 kg^{-1} s^{-2}, and the individual terms C_i and C_j are the mass, in kilograms, for the i-th and j-th particle, respectively. For electrostatical problems K is the Coulomb constant $k_e \approx 8.988 \cdot 10^9$ N m^2 C^{-2} (in the International System of Units, or SI) and the individual terms C_i and C_j are the i-th and j-th charges in Coulombs, respectively.

We assume that all particle masses or charges have the same value: $C_i = C_j$. It is trivial to extend our analysis to the case when C differs from particle to particle. Without loss of generality, we can assume $K = 1$ and $C_i = 1$. Indeed, it is always possible to choose a system of units in which this is true. In the following expressions, $\vec{R}$, $\vec{V}$ and t are scaled quantities in this special system of units, rather than in SI.

Using the forward Euler method for the integration of differential equations, we can express the components of particle velocity $\vec{v}_i$ at the end of the simulation time step Δt as

$$v_{x,\,i}(t+\Delta t) = v_{x,\,i}(t) + \Delta t \sum_{j\neq i} \frac{(x_j - x_i)}{\left[(x_j-x_i)^2 + (y_j-y_i)^2 + (z_j-z_i)^2\right]^{\frac{3}{2}}}, \tag{2}$$

$$v_{y,\,i}(t+\Delta t) = v_{y,\,i}(t) + \Delta t \sum_{j\neq i} \frac{(y_j - y_i)}{\left[(x_j-x_i)^2 + (y_j-y_i)^2 + (z_j-z_i)^2\right]^{\frac{3}{2}}}, \tag{3}$$

$$v_{z,\,i}(t+\Delta t) = v_{z,\,i}(t) + \Delta t \sum_{j\neq i} \frac{(z_j - z_i)}{\left[(x_j-x_i)^2 + (y_j-y_i)^2 + (z_j-z_i)^2\right]^{\frac{3}{2}}}, \tag{4}$$

and the coordinates at the end of the time step as

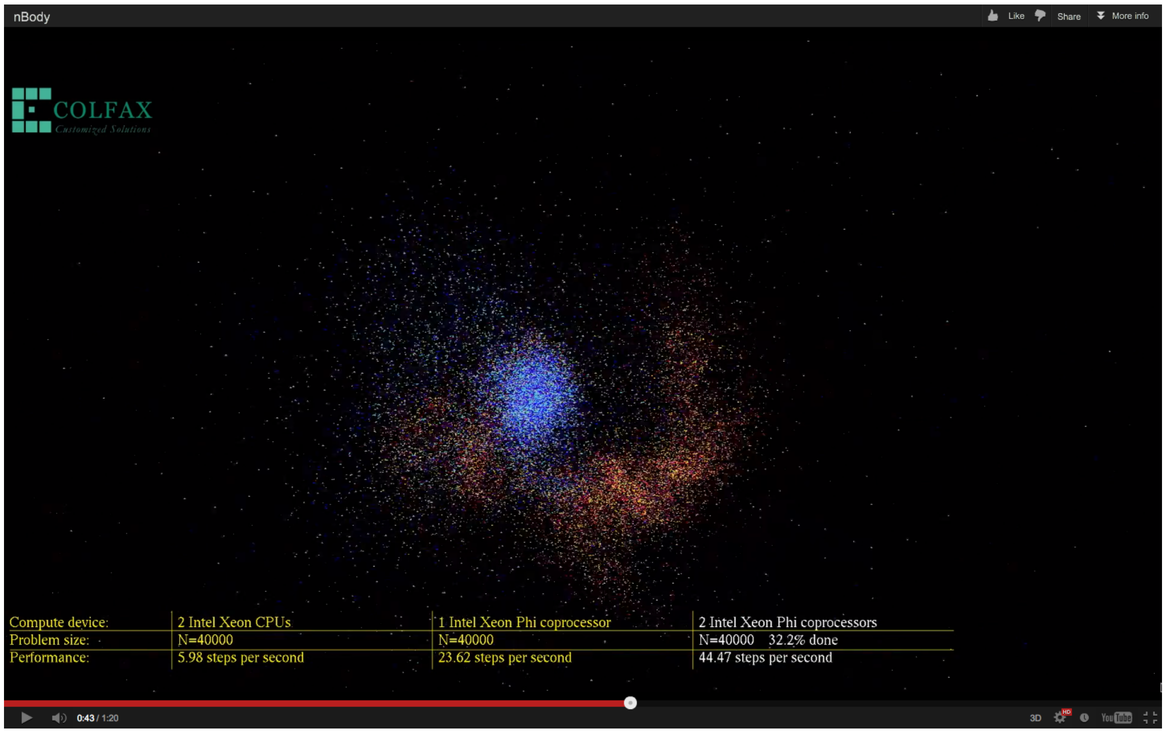
$$x_i(t+\Delta t) = x_i(t) + v_{x,\,i}(t+\Delta t)\Delta t, \tag{5}$$

$$y_i(t+\Delta t) = y_i(t) + v_{y,\,i}(t+\Delta t)\Delta t, \tag{6}$$

$$z_i(t+\Delta t) = z_i(t) + v_{z,\,i}(t+\Delta t)\Delta t. \tag{7}$$

The quantities shown here are the particle coordinates and the components of particle velocities, i.e., the position vector $\vec{R}_i \equiv (x_i, y_i, z_i)$ and the velocity vector $\dot{\vec{R}}_i = \vec{V}_i \equiv (v_{x,\,i}, v_{y,\,i}, v_{z,\,i})$.

Algorithm (2)–(7) may be familiar to some readers from applications other than astrophysics or biophysics: this algorithm is used as a test workload in some software development kits. Note that this algorithm is not particularly efficient nor accurate enough for practical large-scale simulations. This algorithm has a computational complexity of $O\left(N^2\right)$ and employs the forward Euler scheme, which is only first-order accurate in time, and unstable for large time steps. In practical scientific applications, the calculation of gravitational force on each particle is usually performed with tree algorithms, such as the Barnes-Hut method. Tree algorithms can reduce the computational complexity to $O\left(N \log N\right)$. In order to solve the equations of motion, explicit or implicit Runge-Kutta methods may be used instead of the forward Euler scheme. These methods can improve the accuracy to fourth order in time and greatly expand the algorithm stability range.



Compute device:	2 Intel Xeon CPUs	1 Intel Xeon Phi coprocessor	2 Intel Xeon Phi coprocessors
Problem size:	N=40000	N=40000	N=40000 32.2% done
Performance:	5.98 steps per second	23.62 steps per second	44.47 steps per second

Figure 1: Snapshot of the N-body simulation visualization. The full video is available on Colfax's YouTube channel at the following URL: http://www.youtube.com/embed/KxaSEcmkGTo?rel=0.

In this paper, do not discuss advanced algorithms, so as not to distract the reader from the main focus of this study: the exploration of performance optimization on Intel Xeon Phi coprocessors. However, the code design and optimization methods discussed in this paper also apply to advanced algorithms used in practice. For instance, in the Barnes-Hut method, the space is split into zones, and in order to calculate the force on particle i, distant zones are approximated as a single massive particle, while interactions in the local zone are calculated using the $O(N^2)$ algorithm discussed here.

3 Unoptimized N-body simulation in the C language

We implemented algorithm (2)–(7) as a code in the C language. The code `nbody-1.c` is shown in Listing 1. This listing is full in the sense that it compiles, runs, and produces the benchmark information. However, it uses a trivial setup for initial condition, has no output facilities, does not perform visualization, and is limited to only one shared-memory compute device. These limitations are necessary to fit the code on one page and make it easily readable.

The code is parallelized in shared memory using the OpenMP framework. The outer `for`-loop over index `i` calculates the force acting on each particle and changes the particle momentum in response to that force. The iteration space of this loop is distributed across multiple threads by `#pragma omp`.

The inner `for`-loop over index `j` calculates the force acting on particle `i`. This loop is not parallelized across multiple threads, however, it uses parallelism on another level. At the default optimization level, the compiler will attempt to automatically vectorize this loop in order to use SIMD parallelism. The opportunity for SIMD (single instruction, multiple data) parallelism is obvious here: for every set of `x`, `y` and `z`, the calculation of variables `dx`, `dy`, `dz`, `drSquared` and `drPowerN32` can be performed independently using the same sequence of operations. The compiler indeed vectorizes the `j`-loop in `nbody-1.c`, which can be verified by using the compiler argument `-vec-report3`.

```c
#include <math.h>
#include <mkl_vsl.h>
#include <omp.h>
#include <stdio.h>

struct ParticleType { float x, y, z, vx, vy, vz; };

int main(const int argc, const char** argv) {
  const int nParticles = 30000, nSteps = 10; // Simulation parameters
  const float dt = 0.01f; // Particle propagation time step
  // Particle data stored as an array of structures
  ParticleType* particle = (ParticleType*) malloc(nParticles*sizeof(ParticleType));

  // Initialize particles
  VSLStreamStatePtr rnStream;  vslNewStream( &rnStream, VSL_BRNG_MT19937, 1 );
  vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, 6*nParticles, (float*)particle, -1.0f, 1.0f);

  // Propagate particles
  printf("Propagating particles using %d threads...\n", omp_get_max_threads());
  double rate = 0.0, dRate = 0.0; // Benchmarking data
  const int skipSteps = 1; // Skip first iteration is warm-up on Xeon Phi coprocessor
  for (int step = 1; step <= nSteps; step++) {
    const double tStart = omp_get_wtime(); // Start timing

    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < nParticles; i++) { // Parallel loop over particles that experience force
      float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f; // Components of force on particle i

      for (int j = 0; j < nParticles; j++) { // Vectorized loop over particles that exert force
        if (j != i) {
          // Newton's law of universal gravity calculation.
          const float dx = particle[j].x - particle[i].x;
          const float dy = particle[j].y - particle[i].y;
          const float dz = particle[j].z - particle[i].z;
          const float drSquared    = dx*dx + dy*dy + dz*dz;
          const float drPowerN32   = 1.0f/(drSquared*sqrtf(drSquared));

          // Reduction to calculate the net force
          Fx += dx * drPowerN32;  Fy += dy * drPowerN32;  Fz += dz * drPowerN32;
        }
      }
      // Move particles in response to the gravitational force
      particle[i].vx += dt*Fx; particle[i].vy += dt*Fy; particle[i].vz += dt*Fz;
    }
    for (int i = 0 ; i < nParticles; i++) { // Not much work, serial loop
      particle[i].x  += particle[i].vx*dt;
      particle[i].y  += particle[i].vy*dt;
      particle[i].z  += particle[i].vz*dt;
    }

    const double tEnd = omp_get_wtime(); // End timing
    if (step > skipSteps) // Collect statistics
      { rate  += 1.0/(tEnd - tStart); dRate += 1.0/((tEnd - tStart)*(tEnd-tStart)); }
    printf("Step %d: %.3f seconds\n", step, (tEnd-tStart));
  }
  rate/=(double)(nSteps-skipSteps); dRate=sqrt(dRate/(double)(nSteps-skipSteps)-rate*rate);
  printf("Average rate for iterations %d through %d: %.3f +- %.3f steps per second.\n",
         skipSteps+1, nSteps, rate, dRate);
  free(particle);
}
```

Listing 1: Unoptimized N-body simulation, `nbody-1.c`, discussed in Section 4.

We must also point out the data structure chosen for the code. Each particle is represented by an object of derived type `ParticleType`. This type is a structure containing the coordinates of the particle and its velocity components. This kind of data structure is not the best representation for optimization purposes, as we will see below. However, we demonstrate the results with this data layout because we have seen similar configurations in real-world codes written with the object-oriented paradigm in mind.

The procedure for compiling and running this code using the Intel software development tools is shown in Listing 2. We use the compiler argument `-std=c99` in order to enable some abbreviated C syntax, `-openmp` to enable pragma-directed parallelization in OpenMP, and `-mkl` to link the Intel Math Kernel Library (Intel MKL) for random number generation. This listing also demonstrates the performance result. The performance is measured on the Colfax CX2265i-XP5 server[4] with two eight-core Intel Xeon E5-2680 (Sandy Bridge) processors clocked at 2.7 GHz (3.5 GHz Turbo frequency). We set the number of particles $N = 30000$ for benchmarks in this paper. The performance metric for $N = 30000$ is $T = 0.122$ seconds per step (8.2 steps per second).

```
andrey@dublin$ icc -std=c99 -openmp -mkl -o nbody-1 nbody-1.c
andrey@dublin$ ./nbody-1
Propagating particles using 32 threads...
Step 1: 0.136 seconds
Step 2: 0.124 seconds
Step 3: 0.122 seconds
Step 4: 0.122 seconds
Step 5: 0.122 seconds
Step 6: 0.122 seconds
Step 7: 0.122 seconds
Step 8: 0.122 seconds
Step 9: 0.122 seconds
Step 10: 0.123 seconds
Average rate for iterations 2 through 10: 8.178 +- 0.049 steps per second.
andrey@dublin$
```

Listing 2: Compiling and running `nbody-1.c` (Listing 1) on the host system.

In order to estimate the efficiency of this code, we compare the number of arithmetic operations to the number of CPU clock cycles elapsed in the course of one simulation step. The number of arithmetic operations per iteration of the `j`-loop can be estimated by counting the arithmetic operations in the loop body, and this number is no less than $A = 15$ (counting multiply-add as two operations). At least one of these operations is transcendental: the square root. The number of CPU clock cycles issued by two E5-2680 processors per second is no greater than $C = 2 \times 8 \times 3.5 \cdot 10^9 = 5.6 \cdot 10^{10}$. Here 8 is the number of physical cores per processor, and $3.5 \cdot 10^9$ Hz is the Turbo clock frequency of the processor. We use the upper limit of the clock frequency in order to estimate the maximum number of clock cycles. Therefore, our conservative estimate for the minimum efficiency (the number of operations per cycle) is

$$\text{OPC} \gtrsim \frac{A \cdot N^2}{C \cdot T} = \frac{15 \cdot (30000)^2}{5.6 \cdot 10^{10} \cdot 0.122} = 1.98. \tag{8}$$

There are two reasons why this number greater than 1, even though one of the operations is a long-latency transcendental instruction. First, Intel Xeon processors are able to issue up to two instructions per cycle thanks to their deep instruction pipeline. Second, they use SIMD instructions in order to perform some arithmetic operations on multiple numbers at once. The latter is possible due to the automatic vectorization functionality of the Intel C compiler.

[4] http://www.colfax-intl.com/xeonphi/CX2265i-XP5.html

4 N-body simulation as a native Intel Xeon Phi application

Intel Xeon Phi coprocessors run a Linux operating system and can execute native applications. Native applications for Intel Xeon Phi coprocessors are not binary compatible with Intel Xeon processors. Native execution is not the only method to employ the coprocessor. Offload models and hybrid MPI jobs may be more suitable for more complex applications. However, because our interest here is the evaluation of performance and optimization methods, we restrict the discussion to native execution.

In order to produce an executable for the coprocessor, the code must be compiled with the flag `-mmic`. After compilation, in order to run the application on the Intel Xeon Phi coprocessor, the executable must be copied to the coprocessor file system. Sometimes, additional libraries must be copied to the coprocessor or NFS-shared with it. In the case of the N-body simulation `nbody-1.c`, some Intel MKL and Intel OpenMP library files must be transferred to the coprocessor. We do not show the file transfer process, because it is trivially performed using the secure copy Linux tool `scp`.

The procedure for compiling and running the N-body simulation on the Intel Xeon Phi coprocessor is shown in Listing 3. Note that it is possible to simplify and shorten the procedure (for example, using the tool `micnativeloadex`); however, we use a longer procedure in order to demonstrate the Linux functionality of Intel Xeon Phi coprocessors. Indeed, we transfer the file using the secure copy tool `scp`, log in to the coprocessor system using `ssh`, and run commands on the coprocessor from the shell.

```
andrey@dublin$ icc -std=c99 -openmp -mkl -mmic -o nbody-1-mic nbody-1.c
andrey@dublin$ scp nbody-1-mic mic0:~/
nbody-1-mic                                       100%  217KB 217.2KB/s   00:00
andrey@dublin$ ssh mic0
andrey@dublin-mic$ ./nbody-1-mic
Propagating particles using 228 threads...
Step 1: 0.349 seconds
Step 2: 0.117 seconds
Step 3: 0.116 seconds
Step 4: 0.117 seconds
Step 5: 0.117 seconds
Step 6: 0.117 seconds
Step 7: 0.117 seconds
Step 8: 0.117 seconds
Step 9: 0.116 seconds
Step 10: 0.117 seconds
Average rate for iterations 2 through 10: 8.568 +- 0.021 steps per second.
andrey@dublin-mic$ exit
andrey@dublin$
```

Listing 3: Compiling and running `nbody-1.c` (Listing 1) on the Intel Xeon Phi coprocessor.

Benchmark shows that we gain some performance by simply recompiling the code and running it on the coprocessor: the simulation takes $T = 0.117$ seconds per step (8.6 steps per second). However, the performance gain is only of order 5%. Before discussing why the speedup using the simple recompilation is not higher, we estimate the number of instructions per cycle achieved on the coprocessor.

We were using an engineering sample of the Intel Xeon Phi coprocessor, SKU B1QS-3115A. It has 57 active cores clocked at 1.1 GHz. They issue a total of $57 \times 1.1 \cdot 10^9 = 6.3 \cdot 10^{10}$ instructions per second. Therefore, the number of arithmetic operations per cycle achieved on the coprocessor is

$$\text{OPC} \gtrsim \frac{A \cdot N^2}{C \cdot T} = \frac{15 \cdot (30000)^2}{6.3 \cdot 10^{10} \cdot 0.117} = 1.22. \tag{9}$$

The coprocessor uses SIMD parallelism as well. However, this application is less efficient on the coprocessor than on the host processor, especially considering that SIMD vector registers on Intel Xeon Phi coprocessors are 512 bits wide, compared to the 256-bit AVX registers on Xeon E5 processors. The reason that we not achieve significantly better performance on the Intel Xeon Phi coprocessor is discussed in Section 5.

5 Optimization: unit-stride vectorization

The key to unlocking the performance of Intel Xeon Phi coprocessors is exploiting thread parallelism in combination with data parallelism. We recompiled a thread-parallel N-body simulation for Intel Xeon Phi coprocessors. However, the performance gain was unimpressive, even though vectorization was automatically implemented by the compiler. Because the thread parallel portion of the application is trivial (it involves no synchronization), we must inspect what we are doing with the data parallel portion of the application.

A close inspection of the inner `j`-loop in Listing 1 shows that the data access pattern is lacking a very important property. This property is unit-stride data access. Indeed, consider using the quantity `particle[j].x`. Suppose that a processor core or a coprocessor core is trying to perform a data-parallel computation for iterations `j=0` through `j=15`. In order to apply a vector instruction to the operation `dx = particle[j].x - particle[i].x`, the core must load `particle[0].x`, `particle[1].x`, ..., `particle[15].x` into a vector register, which is a 512-byte contiguous region of the register file. However, the loaded data are not contiguous in the application memory space! The distance between `particle[0].x` and `particle[1].x` is equal to `sizeof(ParticleType)=24` bytes. This data access pattern is inefficient. In each memory access, the core loads not one byte, but a whole cache line, which is 64 bytes wide. In order to load the x-coordinate for iterations `j=0` and `j=1`, the core must either load the same cache line twice, or load two different cache lines. This incurs unnecessary instructions, cache misses, and breaks down the streamlined data access pattern. A much more convenient data access pattern is *unit-stride* access, where scalars from consecutive iterations are contiguous in memory. This allows the core to load a whole 64 bytes of data from memory (or cache) into the vector register in a single instruction. This corresponds to loading `particle[0].x` through `particle[15].x` in just one memory access.

In order to have unit-stride access in the N-body simulation, the data storage structure must be redesigned. As a rule of thumb, programmers should use structures of arrays instead of array of structures in order to achieve unit-stride access. This often is in conflict with the interests of abstraction or object-oriented programming. However, failure to provide a good data layout to a multi-core or a many-core parallel processor may result in a very significant performance loss.

Listing 4 shows an optimized implementation of the N-body simulation. In this new code, we store all x-, y- and z-components of particle coordinates and velocities as arrays, and we bundle the six arrays together as a structure of type `ParticleSystemType`. In the inner `j`-loop, data access pattern now has unit stride. Indeed, `p.x[0]` and `p.x[1]` are contiguous in memory. If data alignment is correct, the coprocessor core can load `p.x[0]` through `p.x[15]` into the vector register with a single memory access. Unit-stride data layout also simplifies the task of pre-fetching particle data from memory into caches. Even if the alignment of data does place `p.x[0]` at the beginning of the cache line, the compiler will implement a code path that peels off a few iterations at the beginning of the loop, and still uses aligned memory accesses in the bulk of the iteration space. The peeled off iterations are executed with scalar instructions (i.e., without vectorization).

Listing 5 demonstrates the performance results on the host processors and on the Intel Xeon Phi coprocessor. This time, the performance on the host is 11.5 steps per second, and on the coprocessor 29.5 steps per second. Compared to the unoptimized version, we achieved 40% greater performance on the host and 140% greater performance on the coprocessor. Furthermore, the coprocessor version performs 2.3x faster than the host version. This is a very significant speedup, considering that two Intel Xeon processors have roughly the same power envelope as a single Intel Xeon Phi coprocessor.

We can tap even more performance from the coprocessor by relaxing the requirements of arithmetic accuracy. Section 6 describes this optimization.

```c
#include <math.h>
#include <mkl_vsl.h>
#include <omp.h>
#include <stdio.h>

struct ParticleSystemType { float *x, *y, *z, *vx, *vy, *vz; };

int main(const int argc, const char** argv) {
  const int nParticles = 30000, nSteps = 10; // Simulation parameters
  const float dt = 0.01f; // Particle propagation time step
  ParticleSystemType p; // Particle system stored as a structure of arrays
  float *buf = (float*) malloc(6*nParticles*sizeof(float)); // Malloc all data
  p.x  = buf+0*nParticles; p.y  = buf+1*nParticles; p.z  = buf+2*nParticles;
  p.vx = buf+3*nParticles; p.vy = buf+4*nParticles; p.vz = buf+5*nParticles;

  // Initialize particles
  VSLStreamStatePtr rnStream;  vslNewStream( &rnStream, VSL_BRNG_MT19937, 1 );
  vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, 6*nParticles, buf, -1.0f, 1.0f);

  // Propagate particles
  printf("Propagating particles using %d threads...\n", omp_get_max_threads());
  double rate = 0.0, dRate = 0.0; // Benchmarking data
  const int skipSteps = 1; // Skip first iteration is warm-up on Xeon Phi coprocessor
  for (int step = 1; step <= nSteps; step++) {
    const double tStart = omp_get_wtime(); // Start timing

    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < nParticles; i++) { // Parallel loop over particles that experience force
      float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f; // Components of force on particle i

      for (int j = 0; j < nParticles; j++) { // Vectorized loop over particles that exert force
        if (j != i) {
          // Newton's law of universal gravity calculation.
          const float dx = p.x[j] - p.x[i];
          const float dy = p.y[j] - p.y[i];
          const float dz = p.z[j] - p.z[i];
          const float drSquared    = dx*dx + dy*dy + dz*dz;
          const float drPowerN32   = 1.0f/(drSquared*sqrtf(drSquared));

          // Reduction to calculate the net force
          Fx += dx * drPowerN32;  Fy += dy * drPowerN32;  Fz += dz * drPowerN32;
        }
      }
      // Move particles in response to the gravitational force
      p.vx[i] += dt*Fx;        p.vy[i] += dt*Fy;        p.vz[i] += dt*Fz;
    }
    for (int i = 0; i < nParticles; i++) { // Not much work, serial loop
      p.x [i] += p.vx[i]*dt; p.y [i] += p.vy[i]*dt; p.z [i] += p.vz[i]*dt;
    }

    const double tEnd = omp_get_wtime(); // End timing
    if (step > skipSteps) // Collect statistics
      { rate  += 1.0/(tEnd - tStart); dRate += 1.0/((tEnd - tStart)*(tEnd-tStart)); }
    printf("Step %d: %.3f seconds\n", step, (tEnd-tStart));
  }
  rate/=(double)(nSteps-skipSteps); dRate=sqrt(dRate/(double)(nSteps-skipSteps)-rate*rate);
  printf("Average rate for iterations %d through %d: %.3f +- %.3f steps per second.\n",
         skipSteps+1, nSteps, rate, dRate);
  free(buf);
}
```

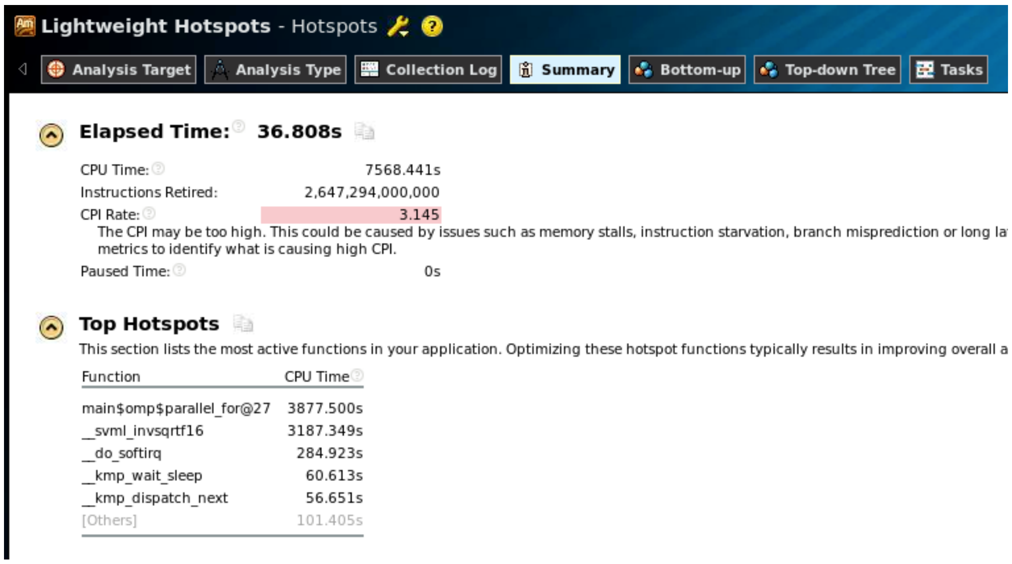
Listing 4: Optimized N-body simulation with unit stride data access, `nbody-2.c`, discussed in Section 5.

```
andrey@dublin$ # First, compile and run the optimized simulation on host processors
andrey@dublin$ icpc -openmp -mkl -o nbody-2 nbody-2.c
andrey@dublin$ ./nbody-2
Propagating particles using 32 threads...
Step 1: 0.099 seconds
Step 2: 0.090 seconds
Step 3: 0.086 seconds
Step 4: 0.090 seconds
Step 5: 0.086 seconds
Step 6: 0.087 seconds
Step 7: 0.086 seconds
Step 8: 0.086 seconds
Step 9: 0.086 seconds
Step 10: 0.088 seconds
Average rate for iterations 2 through 10: 11.473 +- 0.211 steps per second.
andrey@dublin$
andrey@dublin$ # Now compile and run a native version for Xeon Phi coprocessors
andrey@dublin$ icc -std=c99 -openmp -mkl -mmic -o nbody-2-mic nbody-2.c
andrey@dublin$ scp nbody-2-mic mic0:~/
nbody-1-mic                                         100%  217KB 217.2KB/s   00:00
andrey@dublin$ ssh mic0
andrey@dublin-mic$ ./nbody-2-mic
Propagating particles using 228 threads...
Step 1: 0.262 seconds
Step 2: 0.034 seconds
Step 3: 0.034 seconds
Step 4: 0.034 seconds
Step 5: 0.034 seconds
Step 6: 0.034 seconds
Step 7: 0.034 seconds
Step 8: 0.034 seconds
Step 9: 0.034 seconds
Step 10: 0.034 seconds
Average rate for iterations 2 through 10: 29.502 +- 0.260 steps per second.
andrey@dublin-mic$ exit
andrey@dublin$
```

Listing 5: Compiling and running `nbody-2.c` (Listing 4) on the host system and on the Intel Xeon Phi coprocessor.

6 Optimization: accuracy control

The N-body simulation shown in Listing 4 is bottlenecked by the reciprocal square root transcendental function. When the code is compiled as shown in Listing 5, the compiler invokes an Intel Short Vector Math Library (Intel SVML) implementation of this function. It is easy to verify this using the Intel VTune amplifier. We performed the Lightweight hotspots analysis of the N-body simulation on the coprocessor. The simulation was running it for 1000 steps instead of 10 in order to reduce the impact of the initialization overhead on the cumulative event counts. The result is shown in Figure 2, where the summary panel indicates that almost a half of the CPU time is spent inside the function `__svml_invsqrtf16`.



Elapsed Time: 36.808s

CPU Time:	7568.441s
Instructions Retired:	2,647,294,000,000
CPI Rate:	3.145

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long la
metrics to identify what is causing high CPI.

Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall a

Function	CPU Time
main$omp$parallel_for@27	3877.500s
__svml_invsqrtf16	3187.349s
__do_softirq	284.923s
__kmp_wait_sleep	60.613s
__kmp_dispatch_next	56.651s
[Others]	101.405s

Figure 2: Profiling results on the coprocessor of the code shown in Listing 4 with 1000 iterations.

For arithmetically intensive codes involving transcendental arithmetics such as this N-body simulation, accuracy control may have a significant effect on the performance of Intel Xeon Phi coprocessors. The Intel IMCI instruction set contains special implementations for certain transcendental functions. These transcendental vector instructions can be used if some of the IEEE 754 floating point arithmetics requirements can be safely ignored in the application. This results in a performance boost compared to the IEEE 754-compliant implementations of transcendental functions in the Intel SVML.

In the case of the N-body simulation, only one additional step one is required in order to yield additional performance with non-IEEE 754 compliant arithmetics. That step is adding the compiler argument `-fimf-domain-exclusion=8`. This argument declares that we are not interested in accurately processing denormal numbers. Denormal floating point numbers represent values around zero that result in underflow with normal numbers. In this special class of numbers, one or more leading bits of the mantissa are zero. The result of this optimization is seen in Listing 6. This time, the host system performs 11.6 steps per second, but the coprocessor performs 62.9 steps per second, which is 5.4x as fast.

More information about accuracy control on Intel Xeon Phi coprocessors can be found in the online publication "Advanced Optimizations for Intel MIC Architecture, Low Precision Optimizations" by Wendy Doerner[5].

[5]http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture-low-precision-optimizations

```
andrey@dublin$ # First, compile and run the optimized simulation on host processors
andrey@dublin$ icpc -openmp -mkl -fimf-domain-exclusion=8 -o nb-2a nbody-2.c
andrey@dublin$ ./nb-2a
Propagating particles using 32 threads...
Step 1: 0.104 seconds
Step 2: 0.097 seconds
Step 3: 0.085 seconds
Step 4: 0.085 seconds
Step 5: 0.085 seconds
Step 6: 0.085 seconds
Step 7: 0.085 seconds
Step 8: 0.085 seconds
Step 9: 0.085 seconds
Step 10: 0.085 seconds
Average rate for iterations 2 through 10: 11.604 +- 0.446 steps per second.
andrey@dublin$
andrey@dublin$ # Now compile and run a native version for Xeon Phi coprocessors
andrey@dublin$ icpc -openmp -mkl -fimf-domain-exclusion=8 -mmic -o nb-2a-mic nbody-2.c
andrey@dublin$ scp nb-2a-mic mic0:~/
nb-2-a-mic                                          100%  217KB 217.2KB/s   00:00
andrey@dublin$ ssh mic0
andrey@dublin-mic$ ./nb-2a-mic
Propagating particles using 228 threads...
Step 1: 0.254 seconds
Step 2: 0.016 seconds
Step 3: 0.016 seconds
Step 4: 0.016 seconds
Step 5: 0.016 seconds
Step 6: 0.016 seconds
Step 7: 0.016 seconds
Step 8: 0.016 seconds
Step 9: 0.016 seconds
Step 10: 0.016 seconds
Average rate for iterations 2 through 10: 62.944 +- 0.963 steps per second.
andrey@dublin-mic$ exit
```

Listing 6: Compiling and running `nbody-2.c` (Listing 4) on the host system and on the Intel Xeon Phi coprocessor with relaxed floating point arithmetics accuracy.

7 Under the hood: assembly listing of automatically vectorized code

In this section we will discuss the assembly listing produced by the compiler for the N-body simulation. This discussion demonstrates the optimization methods used by Intel compilers. The knowledge of this "under-the-hood" activity may be helpful for designing performance-critical applications in high-level programming languages. We consider the assembly for the native Intel Xeon Phi application compiled from the code shown Listing 4 with arithmetic precision relaxed as described in Section 6.

It is possible to produce a file with the complete assembly listing by compiling the code with the argument `-S`. It is usually difficult to figure out from the assembly listing alone which lines correspond to the performance critical part of the code. This is because the compiler usually produces multiple versions of some loops. These multiple versions handle different problem sizes `N` and different runtime memory alignment situations. For example, the compiler may produce a scalar version of the loop, which is executed if the problem size is too small for vectorization. The compiler may also be prepared peel off several iterations at the beginning or end of the loop if the loop count is not a multiple of the vector register size, or if the memory address of the first data element is not aligned on a proper boundary. However, with the help of the Intel VTune Amplifier XE tool, it is possible to measure the number of CPU clock cycles spent executing each line of code. This information reveals the code paths that are taken at runtime.

Figure 3 shows two screenshots from the Vtune Amplifier XE interface. These screenshots contain the most computationally intensive part of the code, which is the body of the `j`-loop. The listing shown in Figure 3 reveals a number of capabilities of the Intel C compiler:

a) There are two distinct blocks of assembly (blocks 41 and 95 in the screenshots) in which the code spends roughly equal amounts of time. These two blocks are produced in order to deal with the branch "`if (j != i)`" in the `j`-loop. One block executes vectorized calculation from the beginning of the loop to some point close to the condition `(j == i)`. The other block executes the remainder of the loop. This allows the compiler to avoid checking for the branch condition in every iteration, and to employ vector instructions.

b) In order to handle the "`if (j != i)`" condition, and also in order to enable arbitrary values of `N`, the compiler implements peeling. Peeling is the separation of the first or last few iterations at the beginning or the end of the loop. This procedure is performed in order to (a) work with a loop with the number of iterations equal to a multiple of the vector register capacity, and (b) work with data arrays beginning at a properly aligned memory address.

c) The compiler inserts prefetch instructions, e.g., code location `0x4031e0` uses `vprefetch1`. These instructions request the transfer of data from memory into caches, or from a higher-level cache to a lower-level cache. The prefetch distance, i.e., how many loop iterations in advance the prefetch must be issued, is calculated by the compiler automatically.

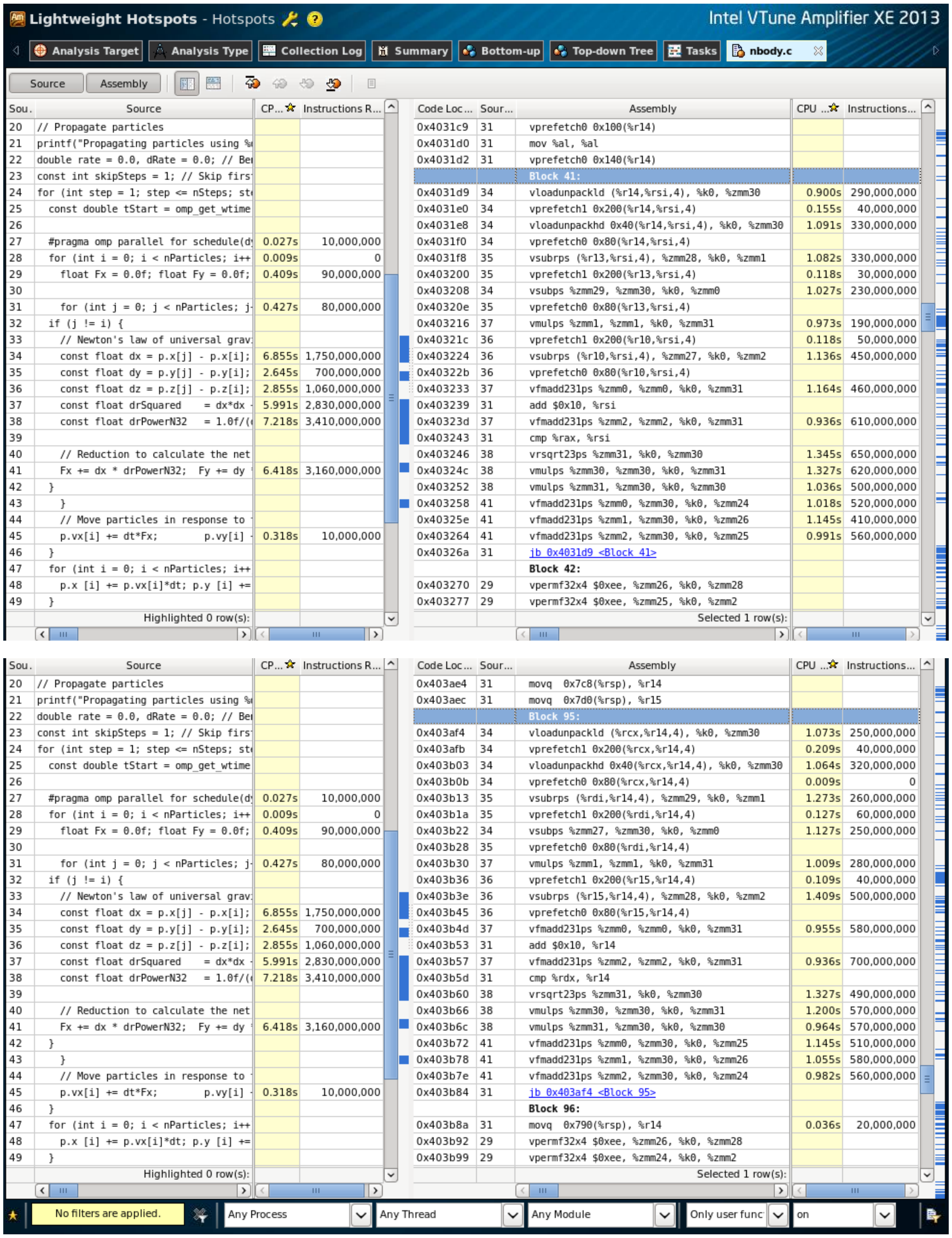
d) Vector instructions are used by the compiler in the vectorized part of the code. For example, code location `0x403208` shows a SIMD subtraction instruction from the Intel IMCI instruction set `vsubps`.

e) The compiler is capable of aggregating arithmetic expressions in order to use optimized vector instructions. For example, code location `0x403233` shows a fused multiply-add instruction `vfmadd231ps` corresponding to the code line 37. Another example is code location `0x403246`, where instruction `vrsqrt23ps` is used. That instruction calls the SIMD calculation of the reciprocal square root of its argument. Note that after that instruction, two multiplication instructions `vmulps` are issued to multiply . They calculate the value of `drPowerN32 = 1.0f/(drSquared*sqrtf(drSquared));`. The compiler replaced three instructions: square root, multiplication and reciprocal, with three more efficient operations: reciprocal square root and two multiplications.

f) Reduction is implemented by the compiler in an intelligent manner. Code locations `0x403258`, `0x40325e` and `0x403264` show that the compiler accumulates the data for the reduction of variables `Fx`, `Fy` and `Fz` in vector registers, and these vector registers are reduced into scalars at the end of the loop calculations. Reduction to scalars is not shown in these screen shots.

All of the above optimizations indicate that the Intel compiler makes complex and effective decisions in the process of automatic vectorization and optimization of calculations for MIC architecture. In practice, the performance of the automatically vectorized code is very impressive. The applications for Xeon processors and for MIC architecture compiled from a common source code do reach the limits of the capabilities of both platforms. This fact is manifested in much better performance of the N-body calculation on Intel Xeon Phi coprocessors compared to the host system.

However, it is the responsibility of the programmer to produce a high-level language code that the compiler can optimize. This involves the design of data layout that allows for unit-stride access. Data locality in space and time is generally also helpful and can be improved by the use of tiling or cache-oblivious recursive algorithms. Additionally, we find that the programmer must be careful with the types and precision of variables, constants and functions. For example, the usage of `sqrtf()` instead of `sqrt()` is very important in the N-body simulation. The function `sqrtf()` calls a highly optimized single precision square root function. Likewise, constants in single precision should be marked with the letter "`f`" (for example, "`1.0f`"), otherwise they will be treated as double precision numbers. Variables of different types should not be mixed in arithmetic expressions, because type casts can be expensive.



Sou.	Source	CP...	Instructions R...
20	// Propagate particles		
21	printf("Propagating particles using %		
22	double rate = 0.0, dRate = 0.0; // Be		
23	const int skipSteps = 1; // Skip firs		
24	for (int step = 1; step <= nSteps; st		
25	const double tStart = omp_get_wtime		
26			
27	#pragma omp parallel for schedule(d	0.027s	10,000,000
28	for (int i = 0; i < nParticles; i++	0.009s	0
29	float Fx = 0.0f; float Fy = 0.0f;	0.409s	90,000,000
30			
31	for (int j = 0; j < nParticles; j	0.427s	80,000,000
32	if (j != i) {		
33	// Newton's law of universal grav		
34	const float dx = p.x[j] - p.x[i];	6.855s	1,750,000,000
35	const float dy = p.y[j] - p.y[i];	2.645s	700,000,000
36	const float dz = p.z[j] - p.z[i];	2.855s	1,060,000,000
37	const float drSquared = dx*dx	5.991s	2,830,000,000
38	const float drPowerN32 = 1.0f/(	7.218s	3,410,000,000
39			
40	// Reduction to calculate the net		
41	Fx += dx * drPowerN32; Fy += dy	6.418s	3,160,000,000
42	}		
43	}		
44	// Move particles in response to		
45	p.vx[i] += dt*Fx; p.vy[i]	0.318s	10,000,000
46	}		
47	for (int i = 0; i < nParticles; i++		
48	p.x [i] += p.vx[i]*dt; p.y [i] +=		
49	}		
	Highlighted 0 row(s):		

Code Loc...	Sour...	Assembly	CPU ...	Instructions...
0x4031c9	31	vprefetch0 0x100(%r14)		
0x4031d0	31	mov %al, %al		
0x4031d2	31	vprefetch0 0x140(%r14)		
		Block 41:		
0x4031d9	34	vloadunpackld (%r14,%rsi,4), %k0, %zmm30	0.900s	290,000,000
0x4031e0	34	vprefetch1 0x200(%r14,%rsi,4)	0.155s	40,000,000
0x4031e8	34	vloadunpackhd 0x40(%r14,%rsi,4), %k0, %zmm30	1.091s	330,000,000
0x4031f0	34	vprefetch0 0x80(%r14,%rsi,4)		
0x4031f8	35	vsubrps (%r13,%rsi,4), %zmm28, %k0, %zmm1	1.082s	330,000,000
0x403200	35	vprefetch1 0x200(%r13,%rsi,4)	0.118s	30,000,000
0x403208	34	vsubps %zmm29, %zmm30, %k0, %zmm0	1.027s	230,000,000
0x40320e	35	vprefetch0 0x80(%r13,%rsi,4)		
0x403216	37	vmulps %zmm1, %zmm1, %k0, %zmm31	0.973s	190,000,000
0x40321c	36	vprefetch1 0x200(%r10,%rsi,4)	0.118s	50,000,000
0x403224	36	vsubrps (%r10,%rsi,4), %zmm27, %k0, %zmm2	1.136s	450,000,000
0x40322b	36	vprefetch0 0x80(%r10,%rsi,4)		
0x403233	37	vfmadd231ps %zmm0, %zmm0, %k0, %zmm31	1.164s	460,000,000
0x403239	31	add $0x10, %rsi		
0x40323d	37	vfmadd231ps %zmm2, %zmm2, %k0, %zmm31	0.936s	610,000,000
0x403243	31	cmp %rax, %rsi		
0x403246	38	vrsqrt23ps %zmm31, %k0, %zmm30	1.345s	650,000,000
0x40324c	38	vmulps %zmm30, %zmm30, %k0, %zmm31	1.327s	620,000,000
0x403252	38	vmulps %zmm31, %zmm30, %k0, %zmm30	1.036s	500,000,000
0x403258	41	vfmadd231ps %zmm0, %zmm30, %k0, %zmm24	1.018s	520,000,000
0x40325e	41	vfmadd231ps %zmm1, %zmm30, %k0, %zmm26	1.145s	410,000,000
0x403264	41	vfmadd231ps %zmm2, %zmm30, %k0, %zmm25	0.991s	560,000,000
0x40326a	31	jb 0x4031d9 <Block 41>		
		Block 42:		
0x403270	29	vpermf32x4 $0xee, %zmm26, %k0, %zmm28		
0x403277	29	vpermf32x4 $0xee, %zmm25, %k0, %zmm2		
		Selected 1 row(s):		

Sou.	Source	CP...	Instructions R...
20	// Propagate particles		
21	printf("Propagating particles using %		
22	double rate = 0.0, dRate = 0.0; // Be		
23	const int skipSteps = 1; // Skip firs		
24	for (int step = 1; step <= nSteps; st		
25	const double tStart = omp_get_wtime		
26			
27	#pragma omp parallel for schedule(d	0.027s	10,000,000
28	for (int i = 0; i < nParticles; i++	0.009s	0
29	float Fx = 0.0f; float Fy = 0.0f;	0.409s	90,000,000
30			
31	for (int j = 0; j < nParticles; j	0.427s	80,000,000
32	if (j != i) {		
33	// Newton's law of universal grav		
34	const float dx = p.x[j] - p.x[i];	6.855s	1,750,000,000
35	const float dy = p.y[j] - p.y[i];	2.645s	700,000,000
36	const float dz = p.z[j] - p.z[i];	2.855s	1,060,000,000
37	const float drSquared = dx*dx	5.991s	2,830,000,000
38	const float drPowerN32 = 1.0f/(	7.218s	3,410,000,000
39			
40	// Reduction to calculate the net		
41	Fx += dx * drPowerN32; Fy += dy	6.418s	3,160,000,000
42	}		
43	}		
44	// Move particles in response to		
45	p.vx[i] += dt*Fx; p.vy[i]	0.318s	10,000,000
46	}		
47	for (int i = 0; i < nParticles; i++		
48	p.x [i] += p.vx[i]*dt; p.y [i] +=		
49	}		
	Highlighted 0 row(s):		

Code Loc...	Sour...	Assembly	CPU ...	Instructions...
0x403ae4	31	movq 0x7c8(%rsp), %r14		
0x403aec	31	movq 0x7d0(%rsp), %r15		
		Block 95:		
0x403af4	34	vloadunpackld (%rcx,%r14,4), %k0, %zmm30	1.073s	250,000,000
0x403afb	34	vprefetch1 0x200(%rcx,%r14,4)	0.209s	40,000,000
0x403b03	34	vloadunpackhd 0x40(%rcx,%r14,4), %k0, %zmm30	1.064s	320,000,000
0x403b0b	34	vprefetch0 0x80(%rcx,%r14,4)	0.009s	0
0x403b13	35	vsubrps (%rdi,%r14,4), %zmm29, %k0, %zmm1	1.273s	260,000,000
0x403b1a	35	vprefetch1 0x200(%rdi,%r14,4)	0.127s	60,000,000
0x403b22	34	vsubps %zmm27, %zmm30, %k0, %zmm0	1.127s	250,000,000
0x403b28	35	vprefetch0 0x80(%rdi,%r14,4)		
0x403b30	37	vmulps %zmm1, %zmm1, %k0, %zmm31	1.009s	280,000,000
0x403b36	36	vprefetch1 0x200(%r15,%r14,4)	0.109s	40,000,000
0x403b3e	36	vsubrps (%r15,%r14,4), %zmm28, %k0, %zmm2	1.409s	500,000,000
0x403b45	36	vprefetch0 0x80(%r15,%r14,4)		
0x403b4d	37	vfmadd231ps %zmm0, %zmm0, %k0, %zmm31	0.955s	580,000,000
0x403b53	31	add $0x10, %r14		
0x403b57	37	vfmadd231ps %zmm2, %zmm2, %k0, %zmm31	0.936s	700,000,000
0x403b5d	31	cmp %rdx, %r14		
0x403b60	38	vrsqrt23ps %zmm31, %k0, %zmm30	1.327s	490,000,000
0x403b66	38	vmulps %zmm30, %zmm30, %k0, %zmm31	1.200s	570,000,000
0x403b6c	38	vmulps %zmm31, %zmm30, %k0, %zmm30	0.964s	570,000,000
0x403b72	41	vfmadd231ps %zmm0, %zmm30, %k0, %zmm25	1.145s	510,000,000
0x403b78	41	vfmadd231ps %zmm1, %zmm30, %k0, %zmm26	1.055s	580,000,000
0x403b7e	41	vfmadd231ps %zmm2, %zmm30, %k0, %zmm24	0.982s	560,000,000
0x403b84	31	jb 0x403af4 <Block 95>		
		Block 96:		
0x403b8a	31	movq 0x790(%rsp), %r14	0.036s	20,000,000
0x403b92	29	vpermf32x4 $0xee, %zmm26, %k0, %zmm28		
0x403b99	29	vpermf32x4 $0xee, %zmm24, %k0, %zmm2		
		Selected 1 row(s):		

No filters are applied. Any Process Any Thread Any Module Only user func on

Figure 3: Assembly listing of code nbody-2.c with relaxed arithmetics compiled for native execution on Intel Xeon Phi coprocessors (see Section 6 for details of the code and Section 7 for a discussion of the assembly listing).

8 Conclusions

Figure 4 summarizes the performance results for the three cases considered in this publication:

1) Unoptimized parallel code with non-unit stride data access,
2) Optimized parallel code with unit-stride data access, and
3) Optimized parallel code with unit-stride data access and relaxed accuracy of arithmetics.

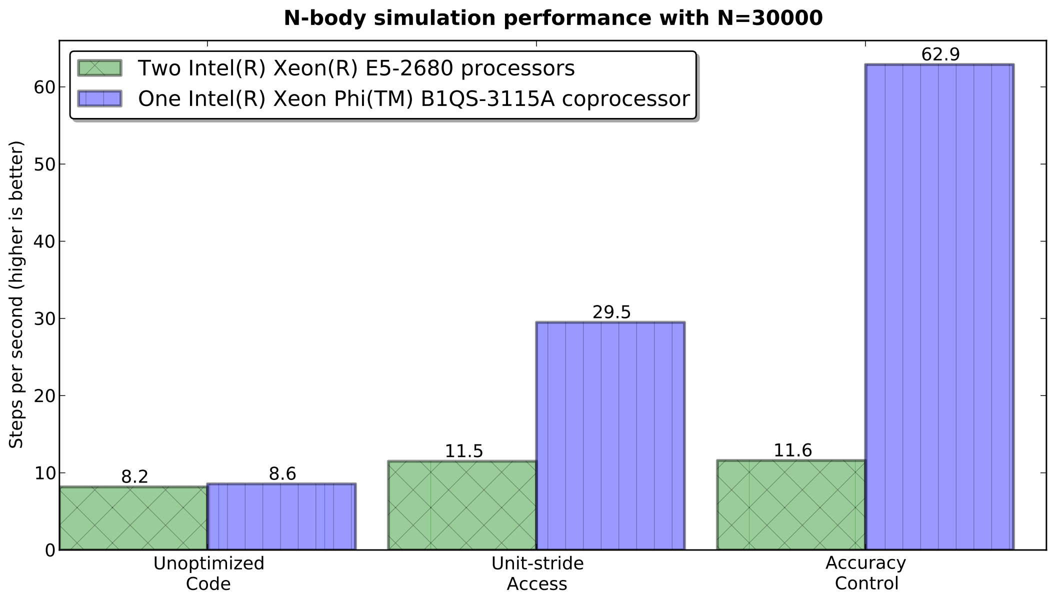


Figure 4: Summary of performance measurements of the N-body simulation for cases discussed in Sections 4, 5 and 6.

Parallelism is an essential requirement for good performance on Intel Xeon Phi coprocessors. Task parallelism must be exploited simultaneously with vector parallelism. Vector instructions can be efficiently implemented by the compiler as long as the data structure and memory access patterns are favorable. This includes data locality and unit-stride data access in vectorized loops.

The Intel C++ compiler can implement multiple execution paths in order to handle various data alignment situations in automatically vectorized codes. It is also able to deal with reduction and low-granularity branches in vectorized loops. In the case considered in this paper, no additional code or compiler arguments are necessary in order to enable efficient automatic vectorization. These capabilities of the compiler make it possible to develop highly efficient applications without low-level programming.

The N-body simulation in the form presented here is a good learning case for programming models and optimization for Intel Xeon Phi coprocessors. In addition to native execution demonstrated here, the offload model can also be used. The offload model allows to easily scale this calculation across multiple coprocessors. One picture is worth a thousand words. View the demonstration of the N-body simulation on multiple coprocessors on Colfax's YouTube channel (see also Figure 1).

Acknowledgements

AV is grateful to Troy A. Porter, Senior Research Scientist at Stanford University, for a lot of constructive comments on this paper.