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 PhiTM coprocessors are capable of delivering more performance and better energy efficiency than Intel[®] Xeon[®] processors for certain parallel applications¹. 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	What Intel Xeon Phi coprocessors bring to the table	2
2	N-body simulation: basic algorithm	3
3	Unoptimized N-body simulation in the C language	4
4	N-body simulation as a native Intel Xeon Phi application	7
5	Optimization: unit-stride vectorization	8
6	Optimization: accuracy control	11
7	Under the hood: assembly listing of automatically vectorized code	12
8	Conclusions	15

Colfax International (http://www.colfax-intl.com/) is a leading provider of innovative and expertly engineered workstations, servers, clusters, storage, and personal supercomputing solutions. Colfax International is uniquely positioned to offer the broadest spectrum of high performance computing solutions, all of them completely customizable to meet your needs - far beyond anything you can get from any other name brand. Ready-to-go Colfax HPC solutions deliver significant price/performance advantages, and increased IT agility, that accelerates your business and research outcomes. Colfax International's extensive customer base includes Fortune 1000 companies, educational institutions, and government agencies. Founded in 1987, Colfax International is based in Sunnyvale, California and is privately held.

¹Intel, 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 way2: "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 chassis³. 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 hyperthreading, 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.

²http://software.intel.com/sites/default/files/blog/337861/reindersxeonandxeonphi-v20121112a.pdf ³http://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):

$$\vec{F}_{i} = KC_{i} \sum_{j \neq i} C_{j} \frac{\vec{R}_{j} - \vec{R}_{i}}{|\vec{R}_{j} - \vec{R}_{i}|^{3}}.$$
(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 $G \approx 6.674 \cdot 10^{-11} \text{ m}^3 \text{ kg}^{-1} \text{ s}^{-2}$, 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 $k_e \approx$ $8.988 \cdot 10^9 \text{ N m}^2 \text{ C}^{-2}$ (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

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

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

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

and the coordinates at the end of the time step as

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

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

$$z_i(t + \Delta t) = z_i(t) + v_{z,i}(t + \Delta t)\Delta t.$$
(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 $O(N^2)$ 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 $O(N \log N)$. 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.

 NBOY
 Use
 North

3 UNOPTIMIZED N-BODY SIMULATION IN THE C LANGUAGE

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.

```
#include <math.h>
2
     #include <mkl_vsl.h>
     #include <omp.h>
3
     #include <stdio.h>
     struct ParticleType { float x, y, z, vx, vy, vz; };
     int main(const int argc, const char** argv) {
8
       const int nParticles = 30000, nSteps = 10; // Simulation parameters
9
       const float dt = 0.01f; // Particle propagation time step
       // Particle data stored as an array of structures
11
       ParticleType* particle = (ParticleType*) malloc(nParticles*sizeof(ParticleType));
14
        // 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
18
19
       printf("Propagating particles using %d threads...\n", omp_get_max_threads());
20
       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++) {</pre>
22
         const double tStart = omp_get_wtime(); // Start timing
24
25
         #pragma omp parallel for schedule(dynamic)
26
         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
28
           for (int j = 0; j < nParticles; j++) { // Vectorized loop over particles that exert force
29
             if (j != i) {
31
                // Newton's law of universal gravity calculation.
                const float dx = particle[j].x - particle[i].x;
32
                const float dy = particle[j].y - particle[i].y;
33
34
                const float dz = particle[j].z - particle[i].z;
               const float drSquared
                                         = dx * dx + dy * dy + dz * dz;
35
               const float drPowerN32 = 1.0f/(drSquared*sqrtf(drSquared));
36
37
                // Reduction to calculate the net force
38
               Fx += dx * drPowerN32; Fy += dy * drPowerN32; Fz += dz * drPowerN32;
39
40
             }
           }
42
           // Move particles in response to the gravitational force
           particle[i].vx += dt*Fx; particle[i].vy += dt*Fy; particle[i].vz += dt*Fz;
43
44
45
         for (int i = 0 ; i < nParticles; i++) { // Not much work, serial loop
           particle[i].x += particle[i].vx*dt;
46
           particle[i].y += particle[i].vy*dt;
particle[i].z += particle[i].vz*dt;
47
48
         }
49
50
51
         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)); }
53
54
         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);
     }
```

1

4

5 6

7

10

12 13

15

16 17

21

23

27

30

41

52

55

56

57 58

59

60

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 server⁴ 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 9: 0.123 seconds
Average rate for iterations 2 through 10: 8.178 +- 0.049 steps per second.
andrey@dublin$
```



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

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

⁴http://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
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

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

```
#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++) {</pre>
   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 \star dx + dy \star dy + dz \star 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);
}
```

1 2

3

4

5 6

7

8

9

10

11

12 13

14 15

16 17

18 19

20

21

22

23

24 25

26

27

28

29 30 31

32

33

34

35

36 37

38 39 40

41 42

43

44

45

46 47

52 53 54

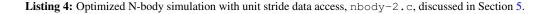
55

56

57 58

59

60



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 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.

₽	Analysis Target 💧 Analy	/sis Type 🔛 Co	llection Log	📋 Summary	👶 Bottom-up	💑 Top-down Tree	🛃 Tasks
0	Elapsed Time: ^① 30	5.808s					
	CPU Time: ③	7568.	441s				
	Instructions Retired:	2,647,294,000	0,000				
	CPI Rate: The CPI may be too high. T metrics to identify what is	his could be cause	8.145 d by issues suc	n as memory stal	ls, instruction star	ation, branch mispred	iction or lon
	Paused Time: 3		0s				
0	Top Hotspots 📄				h - h h d' h' h' h'	t l ll lk l l	
0	This section lists the most act Function	tive functions in you	ur application. (Optimizing these	notspot functions	typically results in imp	roving over
0		-	ur application. (Optimizing these	notspot functions	typically results in imp	oroving over
0	Function	CPU Time	ur application. (Optimizing these	notspot functions	typically results in imp	proving over
0	Function main\$omp\$parallel_for@27	CPU Time	ur application. (Optimizing these	notspot functions	typically results in imp	proving over
•	Function main\$omp\$parallel_for@27 svml_invsqrtf16	CPU Time 3877.500s 3187.349s	ur application. (Optimizing these	notspot functions	typically results in imp	proving over
•	Function main\$omp\$parallel_for@27 svml_invsqrtf16 do_softirq	CPU Time 3877.500s 3187.349s 284.923s	ur application. (Optimizing these	notspot functions	typically results in imp	proving over

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 Doerner⁵.

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

7 UNDER THE HOOD: ASSEMBLY LISTING OF AUTOMATICALLY VECTORIZED CODE

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.

7	UNDER THE HOOD: ASSEMBLY LISTING OF AUTOMATICALLY VECTORIZED CODE
/	UNDER THE HOOD. ASSEMBLI LISTING OF AUTOMATICALLI VECTORIZED CODE

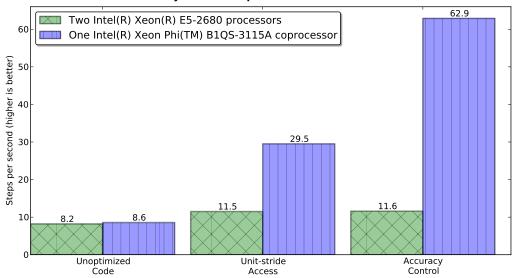
Source Source Context Context <thcontext< th=""> <thcontext< th=""> <thcon< th=""><th>4</th><th>🕀 Analysis Target 🔬 Analysis Type</th><th>Bill Col</th><th>lection Log</th><th>it s</th><th>ummary</th><th>Botton</th><th>n-up 🔩 Top-down Tree 🛃 Tasks 🔒 nbody</th><th>c 💥</th><th></th></thcon<></thcontext<></thcontext<>	4	🕀 Analysis Target 🔬 Analysis Type	Bill Col	lection Log	it s	ummary	Botton	n-up 🔩 Top-down Tree 🛃 Tasks 🔒 nbody	c 💥	
Source OP* Inductions R. Or Control Line Source Op* Inductions R. Op* Control Line Source Op* Inductions R. Op* Inductions R. Op Control Line Source Op* Inductions R. Op Control Line Source Op Source Sourc							Dotton			97777
0 // Propagate particles 0 0 // Propagate particles 0 2 00000 0.00000 0 2 00000 0.00000 0 0.00000 10000000 4 for (int i teg = 1) (fig in the plane) 0.00000 0.00000 0.00000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000000 0.000000 0.0000000	-			_		Code Loc	Sour	Ascembly	СРЦ 🛠	Instructions
2 0x1	20		Cr W	instructions (t						maraccions
3 0	21									
4 6 6 0.403109 34 ViolatingsAid (Vii, Vii, 1, 4), Viol. Viol	22	double rate = 0.0, dRate = 0.0; // Be				0x4031d2	31	vprefetch0 0x140(%r14)		
5 cont double tStart = ong get vite 0.40310 34 vprefetch 0.006(vr14, vrs1, 4) 0.055 40,000.00 7 program ong parallel for scheduler 0.027 10,000.00 0.40310 34 vprefetch 0.006(vr14, vrs1, 4) 0.055 40,000.00 9 for (int 1 = 6, j < nparticles	23	<pre>const int skipSteps = 1; // Skip firs</pre>								
6 0	24	<pre>for (int step = 1; step <= nSteps; ste</pre>				0x4031d9	34	vloadunpackld (%r14,%rsi,4), %k0, %zmm30	0.900s	290,000,000
7 Program amp parallel for schedule(s) 0.0225 0.000000 9 fisk Fx = 0.01; fisk fy = 0.01; 0.039 0.000000 0 0.000000 0 0.0000000 0.0000000 0.0000000 0.00000	25	<pre>const double tStart = omp_get_wtime</pre>				0x4031e0	34	<pre>vprefetch1 0x200(%r14,%rsi,4)</pre>		
8 Gr (Int i = 0; i < nParticles: i++	26								1.091s	330,000,000
9 Float Fx = 0.61; float Fy = 0.61; 0.4095 0.000000 1 for (int j = 0; j = nParticles; j. 0.4275 0.000000 31 // Mexton's Law of nutversal grav 0.0000000 32 // Mexton's Law of nutversal grav 0.0000000 5 const float d' = p.(j) = p.(j); 2.4555 7.50.000000 6 const float d' = p.(j) = p.(j); 2.4555 7.50.000000 0 const float d' = p.(j) = p.(j); 2.4555 7.50.0000000 0 const float d''spund = a'd'd 5.9915 2.80.0000000 0 const float d''spund = a'd'd 5.9915 2.80.000000 0 // Reduction to calculate the net 6.10.000000 0.403223 1 meds2319 rum float strand, N.sh. Yum31 1.856 65.000000 1 F.* e. d'* d'rewert02: F.* e'd' 6.118 3.10.000000 0.403224 1 remd23195 rum float strand, N.sh. Yum31 1.856 65.000000 1 F.* e'd * d'rewert02: F.* e'd' 6.10.00000 rum float strand, N.sh. Yum31 1.856 65.000000 1 F.* e'd * d'rewert02: F.	27									
0 0										
Image: start (int j = 0; j = nBarticles; j = 0.2276 0.000000 0.042206 35 yprefect00 BoB(v13, vrs1, 4) 0.0373 1000000 3 // Nexton's Law of mutwersal grav 0.00120 0.042226 37 0.001224 0.0012233 0.001223 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.0012233 0.001		TLOAT $FX = 0.0T$; TLOAT $FY = 0.0T$;	0.4095	90,000,000						
2 if (j = 1) ⁴ 0 0.9735 100,000,000 3 // Nextors has of universal grave 0 0.9735 100,000,000 4 const flast dr = p.xi[] - p.xi[] 26555 1750,000,000 0 0.04032210 % unips (kr10, krs1,4) 0.0185 50,000,000 6 const flast dr = p.xi[] - p.xi[] 28555 1060,000,000 0 0.0403223 37 Vrfmad2321ps % trand, % unnol, % unn		for (int i = A: i < pParticles: i	0.4276	80.000.000					1.0275	230,000,000
3 J J Output Superior <	32		0.4275	80,000,000					0.9735	190.000.000
4 Const flast dx = p.x[] - p.x[] = 2.655: 1.750.000.000 0	33									
5 Const T(at $4' = p, v_1(1) = p, v_1(1) = 2.845; p. 700.000.00 0$	34		6.855s	1.750.000.000						
6 const float dz = p.r[j] - p.r[j]: 2855 1.060.000.00 0.403223 37 vmdzd21ge tzmel, kzmel, well, kzmel 1 1.164 460.000.00 8 const float dr%serd92 = 1.0f(r) 7.218 3.410.000.00 0.403223 37 vmdzd21ge tzmel, kzmel, well, kzmel 1 0.3356 61.000.00 0 // Reduction to calculate the net 1 4.35 0.600.000 0.403223 37 vmdzd21ge tzmel, kzmel, kell, kzmel 1 0.3356 60.000.00 2 j r + e d. efrbaerH32: Fy = dy 6.188 3.160.000.00 0.403224 38 vmules tzmal, tzmal, kmel 1 1.3256 65.000.00 2 j r + e d. efrbaerH32: Fy = dy 6.188 3.160.000.00 0.403224 38 vmules tzmal, tzmal, kmel 8.40, tzmal 5 1.0356 50.000.00 2 j r + e d. efrbaerH32: Fy = dy .318 10.000.00 0.403224 18 vmules tzmal, tzmal 9, kel, tzmal 5 0.0356 50.000.00 6 j r + e d. efrbaerH32: Fy = dy .318 10.000.00 0.403224 13 vmels tzmal, tzmal 9, kel, tzmal 5 0.500.000.00 6 j r + e d. efrbaerH32: Fy = dy .418 0.40327 72 vmels tzmal, tzmal 9, kel, tzmal 5 0.500.000.00 6 r + e d. efrbaerH32: Fy = dy	35									
7 Const float drSungerd = drSung \$ 59915 2430.000.000 0	36								1.164s	460,000,000
9 // Reduction to calculate the net 0 403243 11 cep trax, trai 1 1 Fx += dx * drPowerN32; Fy += dy 6.4185 3.160.000.000 0 403242 38 vsrag12; systam31, %am30, %ab, %am31 1.3255 650.000.00 3 j 0 403242 38 vsrag12; systam31, %am30, %ab, %am31 1.0355 500.000.00 4 // Move particles in response to 0.3185 10.000.000 0 403226 31 110.000310p since. 412 10.005 50.000.00 6 p.x.[1] == p.vx[1]*dt; p.y [1] = 0.3185 10.000.000 0x403264 41 vfmadd231ps %am3, %am3, %ab, %am23 1.0355 560.000.00 0x403276 31 10.005210p since. 412 0.000.00 0x403276 11 0x403276 11 0x40327 10 0x40328 11 0x40328 10 0x40328 10 0x40328 11 0x40328 10 0x40328 10 0x40328 10	37		5.991s	2,830,000,000		0x403239	31			
0 // Reduction to calculate the net	38	const float drPowerN32 = 1.0f/(7.218s	3,410,000,000		0x40323d	37	vfmadd231ps %zmm2, %zmm2, %k0, %zmm31	0.936s	610,000,000
1 Fx += dx * drPowerN32; Fy += dy 6.418s 3.160.000.000 3)	39					0x403243	31	cmp %rax, %rsi		
2)) 0×03252 38 vmu(ps kamal), kman0, kk0, kamale 1036s 500,000.00 4 // Move particles in response to p.v(i) 0.318s 10,000.00 0×043254 41 vfmadd321ps kama0, kama0, kk0, kama26 10,34s 500,000.00 6) p.v(i) + edt*fx; p.v(i) + 0.318s 10,000.00 0×043254 41 vfmadd321ps kama0, kama0, kk0, kama26 0.91s 560,000.00 6) j.i < mParticles; i+	40					0x403246	38	vrsqrt23ps %zmm31, %k0, %zmm30		
3) 0x403258 41 Vfmodd231ps %zml, %zml30, %k0, %zml4 1.018s 520,000.00 5 p.vx[i] += dt*fx; p.vy[i] 0.318s 10,000.00 0x403258 41. Vfmodd231ps %zml, %zml30, %k0, %zml2 1.018s 520,000.00 6 j p.x [i] += p.x [i] += p.y [i] += 0x403254 41. Vfmodd31ps %zml2, %zml30, %k0, %zml2 0.991s 560,000.00 7 for (int i = 0; i < nParticles; i++	41		6.418s	3,160,000,000	ľ					
4 // Move particles in response to spin spin spin spin spin spin spin spin	42									
5 p.vx[i] += dt+Fx; p.vy[i] 0.3185 10.000.000 6) for (int i = 0; i < nParticles; i++	43									
6) 0x40326a 1 1b 0x40327a 29 vpermf32x4 90xee, %zmm25, %k0, %zmm28 0x40327a 1b 0x40327a 29 vpermf32x4 90xee, %zmm25, %k0, %zmm28 0x40327a 1b 0x40327a 29 vpermf32x4 90xee, %zmm25, %k0, %zmm28 0x40327a 1b 0x40327a 29 vpermf32x4 90xee, %zmm25, %k0, %zmm28 0x40327a 1b 0x40326a 31 0x40326a 32 0x40326a 31 0x40326a 31 0x40326a 32 0x40326a 31 0x40326a 32 0x40336a <td></td> <td></td> <td>0.710</td> <td>10.000.000</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td>			0.710	10.000.000						
7 for (int i = 0; i < nParticles; i+ Block 42: 8 p.x [i] += p.vs[i]*t; p.y [i] ++ 0x403277 29 vprmf32x4 \$loxee, %zm25, %k6, %zm2 0x403277 9 > Highlighted 0 row(s): v Selectel 1 row(s): u Selectel 1 row(s): u 0 // Propagate particles (m > Code Loc Source (PL*K Instructions. 0 // Propagate particles (m > Code Loc Source (PL*K Instructions. 0x403267 21 movq 0x763(%rsp), %r14 0.2059 40.000.000 0x403aed 31 movq 0x763(%rsp), %r14 0.2059 40.000.000 0x403aed 31 movq 0x763(%rsp), %r14 0.0295 40.000.000 0x403aed 34 Vloadupactd (Wrcx,%r14,4), %k0, %zm30 1.073 25.000.000 0x403aed 35 vprefetch0 8x80(%rcx,%r14,4) 0.0295 60.000.000 0x403aed 35 vprefetch0 8x80(%rcx,%r14,4) 0.0295 60.000.000 0x403aed 35 vprefetch0 8x80(%rc1,%r14,4) 0.0295 60.000.000 0x403b28			0.318s	10,000,000				· · · · · · · · · · · · · · · · · · ·	0.991s	560,000,000
8 p.x [1] += p.vx[1]		-				0x40326a	31			
9) 0						0×402270	20			
Highlighted 0 row(s): Selected 1 row(s): Selected 1 row(s): C Instructions R C Instructions R Selected 1 row(s): C Instructions R Out. Source CP* Instructions R Code Loc Source CPU* Instructions R Out. Source CP* Instructions R Code Loc Source CPU* Instructions R Out. Source CP* Instructions R Code Loc Source Code Loc Source CPU* Instructions R Out. Source Code Loc Source Source Code Loc Source Assembly CPU* Instructions R Ox403ab1 Source Source <th< td=""><td>49</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></th<>	49									
C Image: Source C Image: Source CPL* Instructions R Code Loc Source CPL* Instructions R 0 // Propagate particles Image: Source CPL* Instructions R Assembly CPL* Instructions R 1 Iprint("Propagate particles using \$\mathbf{w}\$ Image: Source Code Loc Source Source CPL* Instructions R 2 double rate = 0.6, dRate = 0.6; // Be Image: Source					~			· · · · · · · · · · · · · · · · · · ·	:	
0 // Propagate particles 0x403ae4 31 movg 0x7c8(krsp), %r14 1 printf("Propagating particles using % 0x403ae4 31 movg 0x7c8(krsp), %r14 2 double rate = 0.0, dfAate = 0.0; // Re 0x403aec 31 movg 0x7c8(krsp), %r14 3 const int skipSteps = 1; // Skip firs 0x403aec 31 movg 0x7c8(krsp), %r14 0x20s 4 for (int step = 1; step == n5teps; st 0x403aec 31 movg 0x7c8(krsp), %r14 0x20s 5 const double tStart = omp.get wtime 0x403afd 34 vytoreftch0 0x80(krcx, %r14,4) 0x60s 6 for (int i = 0; i < nParticles; i + 0.009s 0 0x403ab2 34 vypreftch1 0x200(krcx, %r14,4) 0.027s 0x00.000 0x403ab2 34 vypreftch1 0x200(krc1, %r14,4) 0.127s 0x00.000 0x403ab2 0x403ab2 35 vypreftch0 0x80(krc1, %r14,4) 0.127s 0x00.000 0x403ab2 5 vypreftch1 0x200(krc1, %r14,4) 0.127s 0x00.000 0x403ab2 35 vypreftch0 0x80(kr15, %r14,4) 0.127s 0x00.000 1 for (int i = 0; i < nParticles; i 0.427s 80.000.000			<		ĵ		(<		-111
0 // Propagate particles 0x403ae4 31 movg 0x7c8(krsp), %r14 1 printf("Propagating particles using % 0x403ae4 31 movg 0x7c8(krsp), %r14 2 double rate = 0.0, dfAate = 0.0; // Re 0x403aec 31 movg 0x7c8(krsp), %r14 3 const int skipSteps = 1; // Skip firs 0x403aec 31 movg 0x7c8(krsp), %r14 0x20s 4 for (int step = 1; step == n5teps; st 0x403aec 31 movg 0x7c8(krsp), %r14 0x20s 5 const double tStart = omp.get wtime 0x403afd 34 vytoreftch0 0x80(krcx, %r14,4) 0x60s 6 for (int i = 0; i < nParticles; i + 0.009s	Sou	Source	CP 🍲	Instructions P	^	Code Loc	Sour	Accembly	CPU 🛠	Instructions
1 printf("Propagating particles using % 0x403aec 31 movq 0x7d0(%rsp), %r15 0x403aec 30 0x403aec 31 movq 0x7d0(%rsp), %r15 0x403aec 30 0x403aec 31 word 0x7d0(%rsp), %r15 0x403aec 10.73s 250,000.00 0x403aec 34 vtoadunpackld (%rcx,%r14,4) 0.029s 40,000.00 0x403ab1a 35 vyprefetch1 0x200(%r1,%r14,4) 0.027s 60,000.00 0x403ab1a 35 vyprefetch1 0x200(%r1,%r14,4) 0.127s 60,000.00 0x403ab2a 35 vyprefetch1 0x200(%r1,%r14,4) 0.127s 60,000.00 0x403ab2 35 vyprefetch1 0x200(%r1,%r14,4) 0.029s 40,000.00 0x403ab2a 36 vyprefetch1 0x200(%r15,%r14,4) 0.029s 40,000.00 0x403ab2 36 vyprefetch1 0x200(%r15,%r14,4) 0.029s 40,000.00 0x403ab2a 36 vyprefetch1 0x200(%r15,%r14,4) 0.029s 40,000.00 0x403ab2a 36 vyprefetch1 0x200(%r15,%r14,4) </td <td></td> <td></td> <td>01</td> <td>instructions real</td> <td></td> <td></td> <td></td> <td></td> <td>0.0</td> <td>monacciono</td>			01	instructions real					0.0	monacciono
2 double rate = 0.0, dRate = 0.0; // Be 910ck 95: 3 const int skipSteps = 1; // Skip firs 0.0000 4 for (int step = 1; steps = nsteps; step 0.403afd 34 vloadunpackld (%rcx,%r14,4), %k0, %zmm30 1.073s 250,000.00 5 const double tStart = omp get_wtime 0.403afd 34 vloadunpackld (%rcx,%r14,4), %k0, %zmm30 1.064s 320,000.00 7 #pragma omp parallel for schedule(d) 0.027s 10,000.000 0.403afd 34 vloadunpackld 0x40(%rcx,%r14,4) 0.020s 0.000.00 0x403abd 34 vprefetch0 8x80(%rc1,%r14,4) 0.127s 260,000.00 0x403bba 34 vprefetch0 8x80(%rc1,%r14,4) 0.127s 260,000.00 0x403bba 35 vprefetch0 8x80(%rc1,%r14,4) 0.127s 260,000.00 0x403bba 36 vprefetch0 8x80(%r1,%r14,4) 0.127s 260,000.00 0x403bba 36 vprefetch0 8x80(%r15,%r14,4) 0.109s 40,000.00 0x403bba 36 vprefetch0 8x80(%r15,%r14,4) 0.109s 40,000.00 0x403bba 36 vprefetch0 8x80(%r15,%r14,4) 0.109s 50,000.00	20								-	
3 const int skipSteps = 1; // Skip firs 0x403af4 34 vicadunpackld (%rcx,%r14,4), %k0, %zmm30 1.073s 250,000,00 4 for (int step = 1) step <= n5teps; st	22					0x105dcc	51			
4 for (int step = 1; step <= nSteps; st	23					0x403af4	34		1.073s	250.000.000
5 const double tStart = omp_get_wtime 0	24									
66 0.0009 7 #pragma omp parallel for schedule(dr 0.027s 10.000,000 8 for (int i = 0; i < nParticles; i+ 0.009s 0.009s 0.009s 0.0009 0	25									
8 for (int i = 0; i < nParticles; i+	26					0x403b0b	34	vprefetch0 0x80(%rcx,%r14,4)	0.009s	0
9 float Fx = 0.0f; float Fy = 0.0f; 0.409s 90,000,000 0	27	#pragma omp parallel for schedule(d	0.027s	10,000,000		0x403b13	35		1.273s	260,000,000
0 0.403b28 35 vprefetch0 0x80(%rdi,%r14,4) 0.009 2 if (j != i) { 0.4278 80,000,000 0x403b28 35 vprefetch0 0x80(%rdi,%r14,4) 0.009 280,000,000 2 if (j != i) { 0x403b28 37 vmulps %zmnl, %zmnl, %k0, %zmnl1 0.009 280,000,000 2 if (j != i) { 0x403b28 36 vprefetch0 0x80(%r15,%r14,4) 0.1095 40,000,00 0x403b45 36 vprefetch0 0x80(%r15,%r14,4) 0.1095 40,000,00 0x403b45 36 vprefetch0 0x80(%r15,%r14,4) 0.955 580,000,00 0x403b45 37 vfmadd231ps %zmm0, %k0, %zmm31 0.9565 700,000,00 0x403b57 37 vfmadd231ps %zmm31, %k0, %zmm30 1.3274 90,000,00 0x403b56 38 vmulps %zmm31, %k0, %zmm31 0.9664 570,000,00 0x4	28	for (int i = 0; i < nParticles; i++	0.009s	0		0x403b1a	35	<pre>vprefetch1 0x200(%rdi,%r14,4)</pre>	0.127s	60,000,000
1 for (int j = 0; j < nParticles; j	29	float $Fx = 0.0f$; float $Fy = 0.0f$;	0.409s	90,000,000		0x403b22	34	vsubps %zmm27, %zmm30, %k0, %zmm0	1.127s	250,000,000
2 if (j = i) (0.403b36 36 vprefetch1 0x200(%r15,%r14,4) 0.109s 40,000,00 3 // Newton's law of universal grav. 0.403b36 36 vprefetch1 0x200(%r15,%r14,4) 0.109s 40,000,00 44 const float dx = p.x[j] - p.x[i]; 6.855s 1,750,000,000 0x403b45 36 vsubrps (%r15,%r14,4) %rmm38, %k0, %zmm2 1.409s 500,000,00 55 const float dz = p.z[j] - p.z[i]; 2.855s 1,060,000,000 0x403b45 31 add \$60x10, %r14 0.936s 700,000,000 7 const float drPowerN32 = 1.0fr(r 5.991s 2,830,000,000 0x403b53 31 add \$60x10, %r14 0.936s 700,000,000 0.403b56 38 vrmad231ps %zmm3, %zmm2, %zmm3, %zmm31 0.936s 700,000,000 0.403b56 38 vmulps %zmm30, %k0, %zmm31 1.207s 490,000,00 0.403b56 38 vmulps %zmm30, %k0, %zmm31 1.205s 50,000,000 0.403b76 4 vrmad231ps %zmm1, %zmm30, %k0, %zmm25 1.145s 51,000,000 0.403b76 41 vfmadd231ps %zmm1, %zmm30, %k0, %zmm25 1.145s 560,000,000 0	30					0x403b28	35	<pre>vprefetch0 0x80(%rdi,%r14,4)</pre>		
3 // Newton's law of universal grav 0	31		0.427s	80,000,000						
4 const float dx = p.x[j] - p.x[j]; 6.855s 1.750.000.000 0x403b45 36 vprefetch0 0x80(%r15,%r14,4) 0 5 const float dx = p.x[j] - p.y[j]; 2.645s 700.000,000 0x403b4d 37 vfmadd231ps %zmm0, %zmm0, %zmm0, %zmm31 0.955s 580.000.00 6 const float dx = p.x[j] - p.z[j]; 2.855s 1.060.000,000 0x403b53 1a dd \$0x10, %r14 0.935s 700.000.00 7 const float drPowerN32 = 1.0f/(7.21s 3.410.000,000 0x403b56 31 cmp %rdx, %r14 0.935s 700,000,00 0 // Reduction to calculate the met 6.418s 3.160,000,000 0x403b56 38 vmulps %zmm30, %zmm30, %z0, %zmm31 0.964s 570,000,00 2 } - - - - 0x403b56 38 vmulps %zmm30, %zm30, %z0, %zmm31 0.964s 570,000,00 0x403b72 1 vfmadd231ps %zmm0, %zmm30, %k0, %zmm32 0.964s 570,000,00 0x403b72 1 vfmadd231ps %zmm0, %zm30, %k0, %zmm25 1.145s 510,000,00 2 } - - - - - 0x403b72 1 vfm	32							• • • • • • • •		
5 const float dy = p.y[j] - p.y[j]; 2.645s 700,000,000 6 const float dz = p.z[j] - p.z[j]; 2.855s 1.060,000,000 7 const float drSquared = dx4dx 5.991s 2.830,000,000 8 const float drPowerN32 = 1.6f/(7.218s 3.410,000,000 9 // const float drPowerN32; Fy + dy 6.418s 3.160,000,000 0 // Reduction to calculate the net	33							· · · · · · · · · · · · · · · · · · ·	1.409s	500,000,000
6 const float dz = p.z[j] - p.z[j]; 2.855 1,060,000,000 7 const float drSquared = dx*dx; 5.9915 2,830,000,000 8 const float drPowerN32 = 1.0fr(r 7.2185 3,410,000,000 0 // Reduction to calculate the net 0.403b53 31 add \$60x10, %r14 0.9365 1 Fx += dx * drPowerN32; Fy += dy 6.4185 3,160,000,000 38 vmulps %zmm31, %k0, %zmm30 1.3275 3 } 6.4185 3,160,000,000 0x403b56 38 vmulps %zmm31, %k0, %zmm30 0.9664 570,000,00 0x403b72 J 0x403b72 41 vfmadd231ps %zmm3, %k0, %zmm30, %k0, %zmm25 1.1455 500,000,00 0x403b72 J 0x403b72 41 vfmadd231ps %zmm31, %zmm30, %k0, %zmm25 1.0455 580,000,00 0x403b78 1 vfmadd231ps %zmm31, %zmm30, %k0, %zmm24 0.9825 560,000,00 0x403b78 1 vfmadd231ps %zmm31, %zmm30, %k0, %zmm24 0.9825 560,000,00 0x403b78 1 vfmadd231ps %zmm31, %zmm30, %k0, %zmm24 0.9825 560,000,00 0x403b78	34									
77 const float drSquared = dx*dx 5.991s 2,830,000,000 0x403b57 37 vfmadd231ps %zmm2, %k0, %zmm31 0.936s 700,000,00 8 const float drPowerN32 = 1.0fr/(7.218s 3,410,000,000 0x403b50 31 cmp %rdx, %r14 0 0x403b50 1227s 490,000,00 9 // Reduction to calculate the net 0x403b60 38 vrsqrt23ps %zmm30, %k0, %zmm30 1.327s 490,000,00 1 Fx += dx * drPowerN32; Fy += dy 6.418s 3,160,000,000 0x403b66 38 vmulps %zmm30, %k0, %zmm30 0.964s 570,000,00 2 } 0x403b72 41 vfmadd231ps %zmm30, %k0, %zmm30 0.964s 570,000,00 0x403b72 11 vfmadd231ps %zmm30, %k0, %zmm30, %k0, %zmm25 1.055s 580,000,00 0x403b72 11 vfmadd231ps %zmm30, %zmm30, %k0, %zmm26 1.055s 560,000,00 0x403b72 11 vfmadd231ps %zmm1, %zmm30, %k0, %zmm26 1.055s 560,000,00 0x403b72 11 vfmadd231ps %zmm1, %zmm30, %k0, %zmm26 1.055s 560,000,00 0x403b73 1 vfmadd231ps %zmm2, %zmm30, %k0, %zmm26 1.05	35					- 2			0.955s	580,000,000
88 const float drPowerN32 = 1.0f/(7.218s 3.410.000.000 0 V403b50 31 cmp %rdx, %r14 0 0 99 - - 0 V/Reduction to calculate the net 0 0 - 0 0 - 0 0 - 0 0 - 0 - 0 - 0 0 - 0 0 - 0 0 - 0 0 - 0 0 0 0 - 0	36				Ξ				0.036	700 000 01
99 // Reduction to calculate the net 0	37								0.936s	/00,000,000
0 // Reduction to calculate the net 0 0 0.403b66 38 vmulps %zmm30, %zmm30, %k0, %zmm31 1.200s 570,000,00 1 Fx += dx * drPowerN32; Fy += dy 6.418s 3,160,000,000 0.403b66 38 vmulps %zmm30, %zmm30, %k0, %zmm30 0.9644 570,000,00 2 } 0 0.403b72 41 vfmadd231ps %zmm30, %k0, %zmm26 1.145s 510,000,00 3 > 0 0.403b72 41 vfmadd231ps %zmm30, %k0, %zmm26 1.055s 560,000,00 4 // Move particles in response to 0 0.318s 10,000,000 0x403b72 41 vfmadd231ps %zmm30, %k0, %zmm26 0.982s 560,000,00 0x403b7a 1 b0.9403af4 <block 95-<="" td=""> 0 0x403b78 41 vfmadd231ps %zmm30, %k0, %zmm26 0.982s 560,000,00 0x403b7a 1 b0.9403af4 <block 95-<="" td=""> 0</block></block>	38 39	constituat drPowerN32 = 1.0T/(7.2185	3,410,000,000					1 3 2 7 -	490.000.000
1 Fx += dx * drPowerN32; Fy += dy 6.418s 3,160,000,000 0 9 0.403b6c 38 vmulps %zmm31, %zmm30, %k0, %zmm30, %k0, %zmm25 1.145s 510,000,000 2 } vfmadd231ps %zmm31, %zmm30, %k0, %zmm25 1.145s 510,000,000 3 } vfmadd231ps %zmm1, %zmm30, %k0, %zmm26 1.055s 580,000,000 4 // Move particles in response to 0.403b72 41 vfmadd231ps %zmm1, %zmm30, %k0, %zmm26 1.055s 580,000,000 6 > 0.318s 10.000,000 0.403b7e 41 vfmadd231ps %zmm1, %zmm30, %k0, %zmm24 0.928 560,000,000 0.403b7 0.318s 10.000,000 0.403b78 31 ib 0.4033644 ~481ock 95> 0.40356 20,000,000 0.403b8 0.011 i = 0; i < nParticles; i++		// Reduction to calculate the not								
22 }	40 41		6.418-	3,160,000,000						
3 } 0x403b78 41 vfmadd231ps %zmm1, %zmm30, %k0, %zmm26 1.055s 580,000,00 4 // Move particles in response to 5 p.vx[i] += dt*Fx; p.vy[i] 0.318s 10,000,00 41 vfmadd231ps %zmm2, %zmm30, %k0, %zmm24 0.982s 560,000,00 0 0.318s 10,000,000 0x403b7e 41 vfmadd231ps %zmm2, %zmm30, %k0, %zmm24 0.982s 560,000,00 0 0x403b7e 41 vfmadd231ps %zmm2, %zmm30, %k0, %zmm24 0.982s 560,000,00 0x403b8a 11 b0x403b7e 41 vfmadd231ps %zmm2, %zmm30, %k0, %zmm24 0.982s 560,000,00 0x403b8a 11 b0x403b7e 41 vfmadd231ps %zmm30, %k0, %zmm24 0.982s 560,000,00 0x403b8a 11 b0x403b7e 41 vfmadd231ps %zmm30, %k0, %zmm26 0.0403b8 0x403b82 10x403b92 29 vpermf32x4 \$0xee, %zmm26, %k0, %zmm26 0.036s 20,000,00 0x403b92 29 vpermf32x4 \$0xee, %zmm26, %k0, %zmm26 0x403b92 29 vpermf32x4 \$0xee, %zmm24, %k0, %zmm26 0x403b92 20 0x403b92 0 0x403b92 0x403b92 0x403b92 0x403b92	41 42		0.4105	5,100,000,000						
4 // Move particles in response to 0	43									
5 p.vx[i] += dt*Fx; p.vy[i] 0.318s 10.000.000 6 } Block 96: 0 7 for (int i = 0; i < nParticles; i++	43							· · · · · · · · · · · · · · · · · · ·		
6 } Block 96: 7 for (int i = 0; i < nParticles; i++	45		0.3185	10.000.000					0.5015	
7 for (int i = 0; i < nParticles; i++										
8 p.x [i] += p.vx[i]*dt; p.y [i] += 0 0x403b92 29 vpermf32x4 \$8xee, \$zmm26, \$k0, \$zmm26, \$k0, \$zmm26 0x403b92 9 } Highlighted 0 row(s): V V Selected 1 row(s): V						0x403b8a	31		0.036s	20,000,000
9 } Highlighted 0 row(s): 0x403b9 29 vpermf32x4 \$0xee, %zmm24, %k0, %zmm2 Selected 1 row(s): 0x403b9 29 Selected 1 row(s): 0x403b9 Selecte	40									
Highlighted 0 row(s): Selected 1 row(s):										
	47	}				0,403099	23			
	47 48				~	0,403099	2.5		5	

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.



N-body simulation performance with N=30000

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.

INCOMPLETE WITHOUT ADDENDUM

Addendum to the white paper "Test-driving Intel[®] Xeon Phi[™] coprocessors with a basic N-body simulation"

Andrey Vladimirov Stanford University and Vadim Karpusenko Colfax International

May 8, 2013

Abstract

This addendum addresses an optimization that was not included in the original paper. Using the compiler argument -xAVX or -xhost, the performance of the simulation on the host can be accelerated by a factor of 1.7x.

Colfax International (http://www.colfax-intl.com/) is a leading provider of innovative and expertly engineered workstations, servers, clusters, storage, and personal supercomputing solutions. Colfax International is uniquely positioned to offer the broadest spectrum of high performance computing solutions, all of them completely customizable to meet your needs - far beyond anything you can get from any other name brand. Ready-to-go Colfax HPC solutions deliver significant price/performance advantages, and increased IT agility, that accelerates your business and research outcomes. Colfax International's extensive customer base includes Fortune 1000 companies, educational institutions, and government agencies. Founded in 1987, Colfax International is based in Sunnyvale, California and is privately held.

Addendum

For the N-body simulation reported in the original publication¹, we used Intel Xeon E5-2680 processors, which support the AVX instruction set. This instruction set improves some of the SSE instructions by using 256-bit vector registers instead of 128-bit registers. However, in the original version of the paper, we did not use the argument -xhost or -xAVX, and, as a result, the host code was compiled with SSE instructions instead of AVX.

In this addendum, we report improved results obtained by using the compiler arguments -xhost and -03. The latter has a minor effect on performance. However, -xhost improves the speed of the host application by a factor of 1.7x. Correspondingly, the ratio of the performance of an Intel Xeon Phi coprocessor to the performance of two Intel Xeon E5 processors is 3.7x.

Figure 1 shows the performance metrics of the simulation on the host system and on the coprocessor. Compared to the original version of the paper, these benchmarks use a more recent SKU of the coprocessor, B1QS-5110P, and a more recent compiler, Intel C++ Composer XE version 13.1.1.163. The last set of bars reflects the performance with the argument -xhost.

Figure 2 reveals the the effect of the argument -xhost on the result of compilation. The top panel shows the analysis in the Intel VTune Parallel Amplifier XE of the code compiled without -xhost. According to the assembly listing (shown in the right-hand side of the figure), the reciprocal square root is calculated using the instruction rsqrtps, which is a SIMD instruction of the SSE instruction set. In contrast, the bottom panel shows the analysis of the code compiled with -xhost. Instead of rsqrtps, the AVX instruction vrsqrtps is used there.

Additional information on the argument -xcode can be found in the Intel C++ Compiler XE 13.1 User and Reference Guides² and in an earlier Colfax Research publication "Auto-Vectorization with the Intel Compilers: is Your Code Ready for Sandy Bridge and Knights Corner?"³.

Thanks to Georg Hager⁴ for spotting the missing argument $-xAVX^5$!

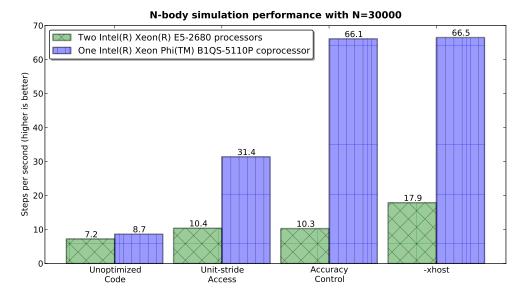


Figure 1: Summary of performance measurements of the N-body simulation.

¹http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx

²http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm ³http://research.colfaxinternational.com/post/2012/03/12/AVX.aspx

⁵http://goparallel.sourceforge.net/test-driving-intel-r-xeon-phitm-coprocessors-with-a-basic-n-body-simulation/

⁴http://blogs.fau.de/hager/

Welcon	ne rooge 🗷					letel \/Ture	o Amalif	er XE 2013
	Analysis Target Analysis Type 📰 Collection Log 🕅 Summary 🚱 Bottom-up 🔹 Top-dow	n Tree 🔣 Tasks and	d Frames 🚯 nboo	ly.c 🔀		inter v run		ET XE 2015
Sou	rce Assembly 🔢 🔛 🧔 🖘 🍫 🧕 🗉	Hardware Event	Hardware Event				Hardwara	. Hardware 🔿
			naiuware Evenu					
Sour Line	Source	CPU_CLK_UNH THREAD	INST_RETIRED. ANY	Addres	Sour Line	Assembly	CPU_CL THREAD	INST_RETI ANY
22	double rate = 0.0 , dRate = 0.0 ; // Benchmarking data			0x40209	e 31	add \$0x4, %rsi	5,338,00	7,272,000
23	const int skipSteps = 1; // Skip first iteration is warm-up on Xeon Phi coprocessor			0x4020a	2 37	mulps %xmm13, %xmm10	46,000,0	30,000,000
24	<pre>for (int step = 1; step <= nSteps; step++) {</pre>			0x4020a	6 36	subps %xmm2, %xmm15	3,880,00	4,702,000
25	const double tStart = omp get wtime(); // Start timing			0x4020a	a 37	mulps %xmm14, %xmm8	6,328,00	. 7,738,000
26				0x4020a	e 37	movaps %xmm15, %xmm9	5,362,00	7,692,000
27	<pre>#pragma omp parallel for schedule(dynamic)</pre>	16,000,000	2,000,00	0x4020b	2 31	cmp %rax, %rsi	22,000,0	2,000,000
28	for (int i = 0; i < nParticles; i++) { // Parallel loop over particles that experience	52,000,000	12,000,00	0x4020b	5 37	mulps %xmm15, %xmm9	4,016,00	. 4,720,000 💶
29	float Fx = θ . θ f; float Fy = θ . θ f; float Fz = θ . θ f; // Components of force on partic	116,000,000	120,000,00	0x4020b	9 37	addps %xmm8, %xmm10	7,028,00	. 7,660,000
30				0x4020b	d 37	addps %xmm9, %xmm10	5,486,00	. 6,084,000
31	for (int $j = 0$; $j < nParticles$; $j \leftrightarrow 1$ { // Vectorized loop over particles that exert	20,154,000,000	28,082,000,00	0x4020c	1 38	rsgrtps %xmm10, %xmm12	912,000,	. 1,258,000
32	if (j != i) {			0x4020c	5 38	mulps %xmm12, %xmm10	8,724,00	. 9,736,000
33	// Newton's law of universal gravity calculation.		-	🗖 0x4020c	9 38	mulps %xmm12, %xmm10	15,754,0	. 13,632,00
34	const float $dx = p.x[j] - p.x[i];$	15,934,000,000	20,466,000,00	0x4020c	d 38	mulps %xmm7, %xmm12	25,046,0	. 17,988,00
35	const float dy = p.y[j] - p.y[i];	16,600,000,000	22,086,000,00	0x4020d	1 38	subpsx 0xf407(%rip), %xr	m 50,000,0	. 28,000,000
36	const float dz = p.z[j] - p.z[i];	7,334,000,000	8,814,000,00	0x4020d	9 38	mulps %xmm10, %xmm12	17,420,0	. 10,692,00
37	const float drSquared = dx*dx + dy*dy + dz*dz;	79,838,000,000	96,864,000,00	0x4020d	d 38	movaps %xmm12, %xmm11	44,612,0	. 24,590,00
38	const float drPowerN32 = 1.0f/(drSquared*sqrtf(drSquared));	343,898,000,000	220,358,000,00	0x4020e	1 38	mulps %xmm12, %xmm11	2,464,00	. 1,190,000
39				0x4020e	5 38	mulps %xmm11, %xmm12	57,586,0	. 30,716,00
40	// Reduction to calculate the net force			0x4020e	9 41	mulps %xmm12, %xmm13	54,272,0	. 36,770,00
41	Fx += dx * drPowerN32; Fy += dy * drPowerN32; Fz += dz * drPowerN32;	314,484,000,000	324,116,000,00	0x4020e	d 41	mulps %xmm12, %xmm14	62,080,0	. 70,962,00
42	}			0x4020f	41	mulps %xmm12, %xmm15	4,606,00	. 6,026,000
43	}			0x4020f	5 41	addps %xmm13, %xmm6	7,012,00	. 9,088,000
44	<pre>// Move particles in response to the gravitational force</pre>			0x4020f	9 41	addps %xmm14, %xmm5	16,390,0	. 21,982,00
45	p.vx[i] += dt*Fx; p.vy[i] += dt*Fy; p.vz[i] += dt*Fz;	718,000,000	110,000,00	0x4020f	i 41	addps %xmm15, %xmm3	12,382,0	. 15,954,00
46	}			0x40210	1 31	jb 0x40207f <block 21=""></block>		. 9,084,000
		343,898,000,000				Highlighted 76 row(s):		. 220,358,0 🗸
		< III	>			< III >		>

Without -xhost, the SSE instruction rsqrtps is used:

With -xhost, the AVX instruction vrsqrtps is used:

Welco	me r000ge r001ge 🕱								Ŧ
🌽 Ge	neral Exploration General Exploration viewpoint (<u>change</u>) ⑦				1 Also		Intel VTur	e Amplifi	er XE 2013
1	Analysis Target 🛕 Analysis Type 🧱 Collection Log 📓 Summary 🛃 Bottom-up 🚱 Top-dow	n Trop	d Framer Danka	, dare	c 💥				
~				Ju y.	L 🔊		11111110/2	1.11	/////
So	irce Assembly 🔢 🖽 🖓 🛷 🍫 🦉 🗉								
		Hardware Event	Hardware Event	•				Hardware	. Hardwar
Sour Line	Source	CPU_CLK_UNH THREAD	INST_RETIRED. ANY		Address	Sour Line	Assembly	CPU_CLK THREAD	INST_RE AÑY
22	double rate = 0.0, dRate = 0.0; // Benchmarking data				0x4020c1	31	add \$0x8, %r15	2,912,000,.	. 2,188,00 . 882,000,
23	const int skipSteps = 1; // Skip first iteration is warm-up on Xeon Phi coprocessor				0x4020c5	34	vsubps %ymm5, %ymm11, %y	1,202,000,.	. 882,000,
24	<pre>for (int step = 1; step <= nSteps; step++) {</pre>				0x4020c9	35	vsubps %ymm6, %ymm13, %y	3,754,000,.	
25	const double tStart = omp_get_wtime(); // Start timing				0x4020cd	37	vmulps %ymm0, %ymm0, %ym	2,638,000,.	
26					0x4020d1	37	vmulps %ymm2, %ymm2, %ym	346,000,0.	. 246,000,
27	<pre>#pragma omp parallel for schedule(dynamic)</pre>	18,000,000	4,000,00		0x4020d5	37	vmulps %ymm1, %ymm1, %ym	1,604,000,.	. 1,288,00 🔤
28	for (int i = 0; i < nParticles; i++) { // Parallel loop over particles that experien	26,000,000	12,000,00		0x4020d9	37	vaddps %ymm10, %ymm15, %	4,150,000,.	. 3,124,00 📃 🗖
29	float Fx = $0.0f$; float Fy = $0.0f$; float Fz = $0.0f$; // Components of force on parti	168,000,000	84,000,00		0x4020de	37	vaddps %ymm12, %ymm11, %	3,406,000,.	. 2,650,00
30					0x4020e3	38	vrsqrtps %ymm13, %ymm10	1,642,000,.	. 1,152,00
31	for (int j = 0; j < nParticles; j++) { // Vectorized loop over particles that exer	13,196,000,000	11,594,000,00		0x4020e8	38	vmulps %ymm10, %ymm13, %	14,176,00.	. 7,190,00
32	if (j != i) {				0x4020ed	38	vmulpsy θxf8cb(%rip), %	12,076,00.	. 4,510,00
33	<pre>// Newton's law of universal gravity calculation.</pre>		-		0x4020f5	38	vmulps %ymm10, %ymm14, %	2,818,000,.	. 4,510,00 . 1,326,00 . 6,682,00
34	<pre>const float dx = p.x[j] - p.x[i];</pre>	2,918,000,000	2,246,000,00	1	0x4020fa	38	vsubpsy 0xf89e(%rip), %	17,144,00.	. 6,682,00
35	<pre>const float dy = p.y[j] - p.y[i];</pre>	20,944,000,000	15,104,000,00		0x402102	38	vmulps %ymm12, %ymm11, %	11,524,00.	. 5,488,00
36	<pre>const float dz = p.z[j] - p.z[i];</pre>	15,094,000,000	11,090,000,00		0x402107	38	vmulps %ymm13, %ymm13, %	25,432,00.	. 14,200,0
37	const float drSquared = dx*dx + dy*dy + dz*dz;	24,554,000,000	18,812,000,00		0x40210c	38	vmulps %ymm14, %ymm13, %	26,414,00.	. 18,780,0
38	<pre>const float drPowerN32 = 1.0f/(drSquared*sqrtf(drSquared));</pre>	222,832,000,000	119,126,000,00		0x402111	41	vmulps %ymm15, %ymm2, %y	27,268,00.	. 22,338,0
39					0x402116	41	vmulps %ymm15, %ymm1, %y	29,332,00.	. 22,338,0 . 34,156,0
40	// Reduction to calculate the net force				0x40211b	41	vmulps %ymm15, %ymm0, %y	2,832,000,.	. 3,462,00
41	Fx += dx * drPowerN32; Fy += dy * drPowerN32; Fz += dz * drPowerN32;	155,226,000,000	161,302,000,00		0x402120	41	vaddps %ymm2, %ymm9, %ym	4,072,000,.	
42	}				0x402124		vaddps %ymm1, %ymm8, %ym	6,558,000,.	. 7,992,00
43	}				0x402128	41	vaddps %ymm0, %ymm7, %ym	7,240,000,.	. 7,992,00 . 8,034,00
44	<pre>// Move particles in response to the gravitational force</pre>				0x40212c	31	cmp %r12, %r15	3,582,000,.	. 3,440,00
45	p.vx[i] += dt*Fx; p.vy[i] += dt*Fy; p.vz[i] += dt*Fz;	630,000,000	88,000,00		0x40212f	31	<u>jb 0x40209b <block 16=""></block></u>		
46	}				0x402135		Block 17:		
		222,832,000,000		-			Highlighted 48 row(s):		. 119,126, 🖵
	< <u> </u>	<	>				< III >	<	

Figure 2: Performance analysis in the Intel VTune Parallel Amplifier tool reveals the instructions implemented by the Intel C++ compiler without and with the argument -xhost.