Analysis of Molecular Dynamics (MD_OPENMP) on Intel® Many Integrated Core Architecture

Abhishek Das¹, Deepti Malav², Shraddha Desai³, Shweta Das⁴, Nisha Kurkure⁵, Goldi Misra⁶, Prasad Wadlakondawar⁷

HPCS Group, Centre for Development of Advanced Computing, Agriculture College Campus, District Industries Centre, Shivajinagar, Pune 411005-India

abhishekd@cdac.in, dmalav@cdac.in, shraddhav@cdac.in, shwetad@cdac.in, nishak@cdac.in, goldi@cdac.in, prasadw@cdac.in

Abstract. This paper assesses the behavior of Molecular Dynamics (MD) simulation program by using OpenMP on the x86-based Intel® Xeon Phi™ coprocessor. The coprocessor is used to offload highly parallel, compute intensive tasks from the host CPU and examine the ability of this new hardware architecture to offload the target workloads with minimal developer efforts. The applicability of SEP (Sampling Enabling Platform) and Intel(R) VTune™ Amplifier XE performance analyzer for different hardware matrices has also being tested.

Keywords

Intel® Xeon Phi™ coprocessor; MPI; OpenMP; MD;
Sampling Enabling Platform; native compilation; Offload;
Intel® VTune™ Amplifier XE

1 Introduction

Intel(R) Xeon Phi™ coprocessors are based on Intel® Many Integrated Core (Intel MIC) architecture. It consists of large number of specialized x86 cores. Existing CPU based programs and applications can be very easily migrated with relative ease in programming skills; it is inherently supported by latest generation of Intel compilers and tools. MD is a parallel code and very much suitable to assessing the capabilities of many core architecture.

1.1 Molecular Dynamics (MD)

The Molecular Dynamics (MD) is a computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating their equations of motion and used to study the time dependent behavior of a molecular system. This simulation technique is used to investigate the structure, dynamics and thermodynamics of biological molecules. It provides detailed information on the fluctuations and conformational changes of proteins and nucleic acids. The replicated data strategy, in which every node holds ‘coordinate and force data’ on the entire system and parallelization is achieved by algorithmic decomposition of the force calculation.

Intel® Xeon Phi™ coprocessors

Intel® Xeon Phi™ coprocessors have coherent cache hierarchy and x86 instruction set which shows the architecture compatibility with CPU. The Intel® Xeon Phi™ Software Development Vehicle (formerly known as “Knights Ferry” has 32 x86 in-order processor cores running at 900MHz (up to 1.2GHz). The first generation production version of Intel® Xeon Phi™ coprocessor (codename: “Knights Corner” uses a 22nm build process to pack a huge 50+ x86 in-order processor cores onto the die. Intel® Many Integrated Core Architecture is based on the x86 ISA, extended with 64-bit addressing, and 512-bit wide SIMD vector instructions and registers.

-8-
Programming Approaches on Intel® Xeon Phi™ coprocessors:

- Offloading: For heterogeneous compilation, the programmer needs to add pragmas and directives to the code in such a way that enables the highly parallel compute intensive section to run on the Intel® Xeon Phi™ coprocessor. The offload directives are available for C, C++ and FORTRAN. The Compiler generates a binary when compiled with “offload-build” compiler flag. When the program executes, it causes two compilations, the host compiler generates code that sends input data to the target and receives results from the target. The target compiler generates code that receives input data from the host, performs the computation and then returns results to the host. The target environment may supply its own sets of standard libraries and these would be available to be called from offloaded code with no need to use special syntax or runtime features.

- Native compilation: Standalone application is to be built which will be directly executed on the Intel® Xeon Phi™ coprocessor. This is also known as implicit compilation as programmer need not change single line of source code. “-mmic” compiler flag is used while compilation. Additionally, programmer needs to set environment variable “MIC_LD_LIBRARY_PATH” to point to own target library. To execute natively compiled code on the Intel® Xeon Phi™ coprocessor, end users need to copy the executable and the required data, libraries directly onto the Intel® Xeon Phi™ coprocessor.

Intel Sampling Enabling Platform (SEP)

The Intel® Performance Tuning Utility (Intel® PTU) is a cross-platform performance analysis tool set. Alongside, with such traditional features as identifying the relevant modules and functions of the application, tracking call sequences, identifying performance-critical source code. Intel PTU has new, more powerful capabilities of data collection, analysis, and visualization. For experienced users, Intel PTU offers the processor hardware event counters for in-depth analysis of the memory system performance and architectural tuning. It associates performance issues with the source code. If you do not have symbol sources for an analyzed application, Intel PTU represents data with basic block granularity and provides a graph of the function execution flow (control flow graph) to navigate the disassembly.

Intel(r) VTune™Amplifier XE Analyzer

Intel® VTune™ Amplifier XE is a powerful threading and performance optimization tool for C/C++, .NET, and FORTRAN developers who need to understand an application’s serial and parallel behavior to improve performance and scalability. The powerful performance profiling tool removes the guesswork and analyzes performance behavior of applications, providing quick access to scaling information for faster and improved decision making. Intel VTune Amplifier is used to fine-tune for optimal performance, ensuring cores are fully exploited and new processor capabilities are supported to the fullest.

2 Experiment in Detail

The MD code used in the experiment is a FORTRAN90 program, using OpenMP for parallel execution. It implements a simple molecular dynamics simulation, using the velocity Verlet time integration scheme in which particles interact with a central pair potential. The number of particles used in the simulation is 1000. The code is executed on multi-core architecture and Intel Many Integrated Core coprocessor. Below is the complete specification of hardware, software platform and the detailed behavior of the test cases used.

Tab.1. Complete Hardware Specification of HPC System

<table>
<thead>
<tr>
<th>Sr No.</th>
<th>Cluster Parameter</th>
<th>Host Node Configuration</th>
<th>Intel® Xeon Phi™ Software Development Vehicle (formerly known as “Knights Ferry”)</th>
<th>Intel® Xeon Phi™ coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU</td>
<td>Intel Xeon@ 3.33GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RAM</td>
<td>24GB</td>
<td>2GB</td>
<td>2GB</td>
</tr>
<tr>
<td>3</td>
<td>Frequency</td>
<td>3.33 GHz</td>
<td>1.2 GHz</td>
<td>1 GHz</td>
</tr>
<tr>
<td>4</td>
<td>Cores</td>
<td>12</td>
<td>32</td>
<td>&gt;50</td>
</tr>
<tr>
<td>5</td>
<td>OS</td>
<td>Red Hat EL6 Kernel 2.6.32</td>
<td>Micro OS 2.6.34-ge08b6b4</td>
<td>Micro OS 2.6.34-g38bc7c2</td>
</tr>
<tr>
<td>6</td>
<td>Threads /core</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Tab. 2. Complete Software Specification of HPC System

<table>
<thead>
<tr>
<th>Sr No.</th>
<th>Software Name</th>
<th>Software version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MD</td>
<td>4.5.4</td>
</tr>
<tr>
<td>2</td>
<td>Intel® C++ Compiler</td>
<td>13.0.0</td>
</tr>
<tr>
<td>3</td>
<td>Sampling Enabling Platform (SEP)</td>
<td>3.3</td>
</tr>
<tr>
<td>4</td>
<td>Performance Intel® VTune™ Amplifier XE</td>
<td>2013XE</td>
</tr>
</tbody>
</table>

Test-Case-I: Offload Compilation

- Offloading to the Intel® Xeon Phi™ coprocessor (target) is done by inserting offload directives in C and Fortran, while the parallelism is controlled by using OpenMP directives. The directive “#pragma offload target (mic)” in C and “!dir$ offload target(mic)” in Fortran gives an indication to the compiler that the next statement, function or set of statements are going to be offloaded onto the Intel(r) Xeon Phi(tm) coprocessor. Since the host CPU and the target do not share physical or virtual memory in hardware, programmer needs to decide variables to be offloaded and copied back between the two. This can be done by using in and out offload-parameters. Specify the length (not the size in bytes) for offloading the pointer, as compilers understand the data type which automatically copies elements to the coprocessor.
- Parallelism for Intel® Xeon Phi™ coprocessors can be achieved in a similar way as with OpenMP. In this heterogeneous programming, setting of the number of threads to be spawned on the Intel(r) Xeon Phi(tm) coprocessor should be done. This can be done by setting environment variable MIC_OMP_NUM_THREADS.
- Using profiler a time consuming portion of the MD code has been identified, which is a function (compute). The offloaded function is the compute function which computes the forces and energies, with given positions, masses, and velocities and also computes the displacement vector between two particles. The function (compute) appears two times in the code, first it is called individually while second time it is called inside a loop. The first approach was to offload the function only, as explained in equation (1). Due to this, execution results into several switching between the host and target, as offloaded function is present within a loop i.e. multiple invocations of offload region.

```fortran
!dir$ offload target(mic)
call compute(np, nd, pos, vel, mass, force, potential, kinetic)
....
do step = 1, step_num
!dir$ offload target(mic)
call compute(np, nd, pos, vel, mass, force, potential, kinetic)
if ( step == step_print) then
    ....
end if
call update ( np, nd, pos, vel, force, acc, mass, dt )
end do                  (1)
```

- In order to avoid switching between host and target, offloaded the complete loop onto the Intel Xeon Phi. Nested parallelism is enabled within the offloaded region; to avoid the execution by single thread. Equation (2) is the pseudo code of the changed approach.

```fortran
!dir$ offload target(mic)
call compute(np, nd, pos, vel, mass, force, potential, kinetic)
....
!dir$ offload target(mic)
do step = 1, step_num
call compute(np, nd, pos, vel, mass, force, potential, kinetic)
if ( step == step_print ) then
    ....
end if
call update(np, nd, pos, vel, force, acc, mass, dt )
end do                  (2)
```
• Hence with minimal changes to the code, able to port the code to the coprocessor. The following Tab.3 and Fig.1 show the performance for offload execution observed with different versions of MPSS (MIC Platform Software Stack) on Intel® Xeon Phi™ Software Development Vehicle (formerly known as “Knights Ferry”) and Intel® Xeon Phi™ coprocessor.

The analysis was done by running tests on different number of cores.

Fig.1. Off-load MD

<table>
<thead>
<tr>
<th>Threads</th>
<th>Intel® Xeon Phi™ Software Development Vehicle (formerly known as “Knights Ferry”)</th>
<th>Intel® Xeon Phi™ coprocessor Coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alpha6</td>
<td>Alpha7</td>
</tr>
<tr>
<td>1</td>
<td>905</td>
<td>628</td>
</tr>
<tr>
<td>4</td>
<td>226</td>
<td>169</td>
</tr>
<tr>
<td>8</td>
<td>113</td>
<td>85</td>
</tr>
<tr>
<td>16</td>
<td>57</td>
<td>43</td>
</tr>
<tr>
<td>32</td>
<td>39</td>
<td>33</td>
</tr>
<tr>
<td>64</td>
<td>30</td>
<td>27</td>
</tr>
<tr>
<td>96</td>
<td>31</td>
<td>25</td>
</tr>
<tr>
<td>124</td>
<td>35</td>
<td>22</td>
</tr>
<tr>
<td>128</td>
<td>177</td>
<td>18</td>
</tr>
<tr>
<td>164</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>200</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Tab.3. OFFLOAD MD-CODE**

3 Test-Case-II: Native Compilation

• Invoke the compiler with “--mmic” rather than “--offload-build” to generate purely “native” code. Observe the difference in performance from offload code.
The following Tab.4 and Fig.2 shows the performance for native compilation with Intel(r) Xeon Phi(tm) coprocessor observed with different versions of MPSS on INTEL® XEON PHİ™ SOFTWARE DEVELOPMENT VEHICLE (FORMERLY KNOWN AS “KNIGHTS FERRY”) and INTEL® XEON PHİ™ COPROCESSOR.

**Tab.4. Native MD Code**

<table>
<thead>
<tr>
<th>Threads</th>
<th>INTEL® XEON PHİ™ SOFTWARE DEVELOPMENT VEHICLE (FORMERLY KNOWN AS “KNIGHTS FERRY”) Coprocessor</th>
<th>INTEL® XEON PHİ™ COPROCESSOR Coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPSS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha6</td>
<td>898</td>
<td>410</td>
</tr>
<tr>
<td>Alpha7</td>
<td>439</td>
<td></td>
</tr>
<tr>
<td>Alpha8-update</td>
<td>411</td>
<td></td>
</tr>
<tr>
<td>Alpha2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>898</td>
<td>410</td>
</tr>
<tr>
<td>4</td>
<td>224</td>
<td>107</td>
</tr>
<tr>
<td>8</td>
<td>112</td>
<td>53</td>
</tr>
<tr>
<td>16</td>
<td>56</td>
<td>27</td>
</tr>
<tr>
<td>32</td>
<td>28</td>
<td>13</td>
</tr>
<tr>
<td>64</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>96</td>
<td>18</td>
<td>10</td>
</tr>
<tr>
<td>124</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>128</td>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>164</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>200</td>
<td>-</td>
<td>27</td>
</tr>
</tbody>
</table>

**Fig.2. Native MD-code**

4 Test-Case-III: Thread Affinity

- This test has been done to observe that which threads affinity performs better on Intel® MIC architecture.
- To bind the “OpenMP threads” to physical processors, thread affinity is used. It can be set using environment variable KMP_AFFINITY for Native compilation and MIC_KMP_AFFINITY for Offloading.
- The three major types of high level thread affinity available are Scatter, Compact and Balanced. Every thread affinity has a different pattern of utilizing available cores.
- Since, one core can run four logical threads; maximum 124 logical threads can be used on INTEL® XEON PHİ™ SOFTWARE DEVELOPMENT VEHICLE (FORMERLY KNOWN AS “KNIGHTS FERRY”) keeping one core free for OS threads.
- Compact Affinity starts utilizing the core one by one progressively. It first fills all the 4 threads available on first core and move to the second core and so on. Good, in case adjacent OpenMP threads are sharing cache data.
• Scatter Affinity starts utilizing the cores in such a way that each upcoming thread go to the cores in a round robin fashion. So, all the OpenMP threads are evenly distributed among threads. Good, in case adjacent threads are not sharing cache data and want to exploit the memory bandwidth effectively.

• Balanced Affinity is now only available for the Xeon Phi, ignored by the CPU. This uses good aspects of above two affinities. It divides the thread on all available cores evenly, while preserving thread locality.

• Fig.3, Fig.4, Fig.5, shows the distribution of threads on the entire available cores of INTEL® XEON PHI™ COPROCESSOR card using Compact, Balanced and Scatter kind of thread affinity respectively.

![Fig.3. Compact Affinity](image)

![Fig.4. Balanced Affinity](image)

![Fig.5. Scatter Affinity](image)

• Fig.6. shows the performance of code for the available thread affinity for Native compilation where balanced affinity scales fine as compared to compact and scatter.

![Fig.6. NATIVE MD code](image)

### 5 Test-Case-IV: OBSERVATION THROUGH SEP

This test case explains MD Code analysis with SEP on Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors. This causes SEP to collect listed counts or events from the PMU (Performance Monitoring Unit) events on the Intel® Xeon Phi™ coprocessors and write to the output file *.tb6. Fig.7 and Fig.8 shows the data collected on INTEL® XEON PHI™ COPROCESSOR and Intel ® Xeon® processor CPU when it runs with different number of coprocessor and processor threads respectively. Important event counters computed are as follows:

- `CPU_CLK_UNHALTED`
- `EXEC_STAGE_CYCLES`
- `INSTRUCTIONS_EXECUTED`
- `MICROCODE_CYCLES`
- `HARDWARE_INTERRUPT`
- `DATA_READ_MISS`
- `DATA_WRITE_MISS`
- `FLOPS`
- `MEMORY_ACCESSSES_IN_BOTH_PIPES`
As a process of identifying hotspots, tried to determine the efficiency of hotspots with help of above identified events of SEP. Most events are counted per thread, while some events are per core. Here, in this case all SEP events are thread based. Usually hotspots are defined in terms of the CPU_CLK_UNHALTED event (aka “clock-tick”). The CPU_CLK_UNHALTED counter measures unhalted clock-ticks on a per thread basis. So for each tick of the CPU’s clock, the counter will count 2 ticks if Hyper-Threading is enabled, 1 tick if Hyper-Threading is disabled. There is no per-core clock-tick counter. Efficiency of the hotspot can be determined using one of the three methods: % Execution Stalled, Code Examination and Changes in Cycles per instruction (CPI). This paper explores CPI method.

CPU_CLK_UNHALTED divided by INSTRUCTIONS_EXECUTED is used to calculate CPI. Ideal CPI value is 0.25, while greater than 4 lies in poor range.

Observations from CPI: For MD code, CPI varies in the range of 4.72 to 3.12 on INTEL® XEON PHI™ COPROCESSOR. So, the program is executing almost 4.47 to 2.87 times slower than optimal value (0.25). On the other side, when MD code runs on the hostCPU, the value of CPI ranges from 5.22 to 4.81, which is very slow with respect to the coprocessor, about 4.97 to 4.56 times slower than optimal value.

A poor CPI ratio indicates one of the following:
- The selected code could probably be optimized to execute more efficiently on the processor.
- The selected code includes string instructions or micro-code.

CPI readings help in optimization for micro architecture. Poor CPI is a hint for a read through memory-related and branch-related events. CPI can vary widely depending on the application and platform. If code size stays constant, optimizations should focus on reducing CPI. If the code size changes for a binary, CPI will change. In general, it is better if CPI reduces as a result of optimizations.

MICROCODE_CYCLES: Microcode is a layer of hardware-level instructions or data structures involved in the implementation of higher level machine code instructions in many computers and other processors; it resides in special high-speed memory and translates machine instructions into sequences of detailed circuit-level operations.

DATA_READ_MISS event is due to Cache L1 of Many Core architecture.

Fig. 7 and 8 shows the decrease in MICROCODE_CYCLES, as number of threads increases for coprocessors and processors respectively. This decreasing pattern is uniform in the host CPU while fluctuate on the coprocessor.

Fig.7. INTEL® XEON PHI™ COPROCESSOR Observation

Fig.8. HOST Observation
6 Conclusion

Considering the ease of programmability and availability of tools like debuggers and profilers from the very familiar family of Intel compilers, it’s an exciting technology. As the future Generation supercomputing will be based on heterogeneous many-core architectures, it makes a great choice to adopt Intel MIC Architecture into both Hardware and Software ecosystem at the earliest. Expecting a far better performance to be achieved on this architecture with release of better production quality products

Best performance will be realized by optimizing the codes for the highly parallel and vector capabilities of the Intel MIC Architecture. Certainly, we benefit by being able to seamlessly port existing highly parallel codes on the Intel® Xeon Phi™ coprocessor that has with the constraints of limited memory on cards.

7 Future Work

- Will continue our current efforts and extend the activity to various other scientific compute intensive applications and same with Hybrid (MPI+OpenMP) programming paradigms.
- Analyzing and tuning applications in better way using Intel software tools.

References

- http://people.sc.fsu.edu/~jburkardt/f_src/md_openmp/md_openmp.html
- ym.edu.tw/infotable/class/Molecular_dynamics_simulations.pdf
- Cycle Accounting Analysis on Intel® Core™2 Processors
- http://software.intel.com/file/8486
- http://software.intel.com/file/14324/