2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

Optimizing Non-Contiguous Memory Access on Intel Xeon Phi Coprocessors

Mingfei Ma[1], Jinlong Hou[1], Jason Ye[1], Meena Arunachalam[2], Rafael Gutierrez[3]
Software and Service Group, Intel Corporation
[1]Shanghai, PRC, [2]Portland, USA, [3]Santa Clara, USA
{mingfei.ma, jinlong.hou, jason.y.ye, meena.arunachalam, rafael.gutierrez}@intel.com

***Abstract*— As an innovative design for high performance computing, Intel Xeon Phi coprocessor based on Intel Many Integrated Core (Intel MIC) architecture relies heavily on its SIMD (single instruction multiple data) unit. However, performance of non-contiguous memory access has become the memory wall towards efficient utilization of SIMD unit on Intel Xeon Phi coprocessors due to gather/scatter overhead. Existing vectorization techniques in the optimization of gather/scatter overhead have been focusing on extracting data parallelism from inter-loop and intra-loop in a decoupled means. In this paper, we propose a novel inter-intra-hybrid vectorization technique which further exploits SIMD efficiency. In this technique, we generate optimized SIMD code for loops requesting non-contiguous memory. Additional strategies are also presented to improve SIMD unit parallelism through data padding and redundant computation. To evaluate our technique, the two major functions from Sandia's miniMD benchmark, i.e., LJ force calculation and neighbor list build, are taken for experiments which show that our proposed method achieves a performance gain of 25%-40% compared with Intel compiler auto vectorized code and outperforms the existing methods. Our optimization method can be further applied to other highly parallel workloads with frequent non-contiguous memory access, which is very common in real-world scientific applications.**

Keywords—high performance computing; gather/scatter; vectorization technique; performance optimization

I. INTRODUCTION

Intel Xeon Phi coprocessor is based on the Intel Many Integrated Core (Intel MIC) architecture which comprises of up to 61 x86-based power efficient cores. Each core is composed of four components: an inorder dual-issue pipeline with 4-way simultaneous multi-threading (SMT), a 512-bit wide SIMD unit; 32 KB L1 data and instruction caches; and a 512 KB fully coherent L2 cache [1]. The SIMD unit provides data parallelism at a very fine grain, 16 single precision (or 8 double precision) floating point operations can be executed as a single vector operation [2].

Intel MIC provides a completely new 512-bit SIMD instruction set which has a larger vector length compared to prior vector architectures (MMX, SSE, and AVX). Besides, many new features have been introduced, such as *write-mask* which conditionally updates data in the destination vector register according to mask bits. All the SIMD instructions can be explicitly used through intrinsics [3] which serve as a programming interface for code developers.

Intel Xeon Phi coprocessor is designed to satisfy the growing needs of High Performance Computing workloads which are highly parallel and it is crucial to effectively utilize the SIMD unit in respect of performance optimization. However, fully exploiting data level parallelism inherent in scientific workloads can be challenging due to the fact that compiler is unaware of field knowledge and probably incapable of generating optimal vectorized code. For example, Molecular Dynamics (MD) simulation codes such as miniMD, LAMMPS, NAMD and Amber are highly parallel but heavily depend on non-contiguous memory access via gather/scatter operations that are the key performance bottleneck on Intel Xeon Phi coprocessor. In this paper, we propose a novel vectorization technique which improves gather/scatter performance via hand coding with intrinsics based on the latest Sandia's miniMD benchmark.

The contributions of this paper are:

- We propose a novel inter-intra-hybrid vectorization technique which efficiently generates optimized SIMD code for loops that heavily depend on non-contiguous memory access.
- We present practical strategies of improving SIMD unit parallelism through data padding and redundant computation.
- We implement our technique and compare it with existing vectorization techniques of alleviating gather/scatter overhead on Intel Xeon Phi coprocessor, i.e., inter-loop and intra-loop vectorization. In addition, we provide an in-depth performance analysis of advantages and disadvantages for these two methods.

Experiments show that the proposed method outperforms the existing methods and achieves a performance gain of 25%-40% compared with Intel compiler auto vectorized code.

The remainder of this paper is organized as follows: Section II provides a survey of related work; Section III provides a brief introduction of miniMD and identifies performance bottleneck of Intel compiler auto vectorized code; Section IV describes the existing vectorization techniques as well as our proposed optimization method; Section V provides

978-1-4799-8937-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HPCC-CSS-ICESS.2015.182

performance results of the vectorization techniques; Section VI concludes this paper.

II. Related Work

Improving the efficiency of vector processing unit is one of the crucial aspects in performance optimization on SIMD architectures, especially Intel Xeon Phi coprocessor which employs a vector length as long as 512-bit. Many approaches have been proposed to handle the difficulties of non-contiguous memory access on SIMD architectures, such as interleaved data access [4], non-aligned and irregular data access [5], indirect data access [6], etc. These approaches focus on auto generation of vectorized code and bring inspiration to compiler optimization. Even so, perfectly exploiting the data parallelism inherent in scientific applications is extremely difficult for compiler due to the various possibilities of data access pattern which compiler is unaware of. Wu et al. [7] proposed an integrated SIMDization framework that pursues the vectorization of both inter-loop and intra-loop simultaneously but it is not feasible for non-contiguous memory access. Pennycook et al. [8] describes vectorization techniques that address the gather/scatter overhead through coding with platform specific intrinsics and reports a significant performance gain over scalar code on Intel Xeon processor as well as Intel Xeon Phi coprocessor. However, their proposed techniques extract data parallelism from inter-loop and intra-loop in a decoupled means and there still exist alternatives for further performance improvement.

III. Motivating Example

Molecular Dynamics (MD) is a computational simulation of physical movements of atoms and molecules in the context of N-body simulation. The atoms and molecules are allowed to interact for a period of time, giving a view of the motion of the atoms. Molecular Dynamics has become one of the most powerful computational tools and has been widely used in the simulation of a huge variety of systems both in and out of thermodynamic equilibrium [9]. MD simulations are highly parallel computations and have been developed to run efficiently on various hardware architectures. Optimization of MD simulation is quite a popular topic and a lot of research focuses on rewriting the code with CUDA or OpenCL so as to take advantage of the growing computing power of Graphics Processing Unit (GPU) [12]-[15]. Other research aims at acceleration on SIMD architecture such as Intel Xeon Phi coprocessor [8],[16],[17].

miniMD is a simplified version of LAMMPS package from Sandia Labs with the purpose of optimization research [10], [11]. miniMD benchmark includes two major components: LJ force calculation and neighbor list build, which account for approximately 55% and 36% of total execution time on Knights Corner (KNC) Intel Xeon Phi coprocessor.

A. LJ Force Calculation

LJ force calculation (Algorithm 1) uses Newton's equation to evaluate forces between all atoms. The algorithm makes use of a pre-computed neighbor list so as to save trouble of calculating the distance between all atom pairs and only neighbors with a distance shorter than the cut-off distance *cutforcesq* contribute to the accumulated result.

Algorithm 1 LJ force calculation

```
1:  for i = 0; i < num of atoms; i++ do
2:      xi = pos[i][0]
3:      yi = pos[i][1]
4:      zi = pos[i][2]
5:      for k = 0; k < num of i's neighbors do
6:          j = neighbor_list[k]
7:          delx = xi – pos[j][0]
8:          dely = yi – pos[j][1]
9:          delz = zi – pos[j][2]
10:         rsq = (delx * delx) + (dely * dely) + (delz * delz)
11:         if rsq ≤ cutforcesq then
12:             sr2 = 1.0 / rsq
13:             sr6 = sr2 * sr2 * sr2 * sigma6
14:             force = 48.0 * sr6 * (sr6 - 0.5) * sr2 * epsilon
15:             f[i][0] += force * delx
16:             f[i][1] += force * dely
17:             f[i][2] += force * delz
18:         end if
19:     end for
20: end for
```

Intel compiler is able to auto vectorize the inner loop of LJ force calculation efficiently. For double precision, Intel Xeon Phi coprocessor works on 8 neighbors at a time, as shown in Fig. 1. Atoms are stored in the format of AoS which needs to be reconstructed into SoA so as to achieve 100% SIMD efficiency. Lines 7-10 calculate the distance between the target atom and its neighbors in 3 dimensions. *delx* is calculated for the 8 neighbors at one instruction and position *x* of the 8 neighbors are packed into one vector register. The non-contiguous memory accesses of atom positions result in a gather operation which introduces significant inefficiency. Similarly, Lines 15-17 accumulate the LJ force in 3 dimensions and the SoA to AoS conversion leads to scatter operation.

The branch at Line 11-18 is handled by vector masking. Intel Xeon Phi coprocessor has 16-bit mask registers that control which entries of the 16 32-bit elements are accessed during a computation, for double precision only the lower 8 bits of the mask register are used. To be specific, a comparison at Line 11 is used to generate the mask bits and the computation at Lines 12-14 is executed for all neighbors, but only neighbors that passes the test are written back.

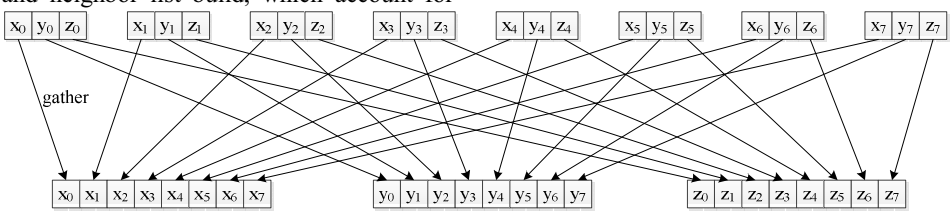

Fig. 1. Intel Compiler auto vectorized code

B. Neighbor List Build

Neighbor list build (Algorithm 2) identifies neighbors of each atom. The list is updated at a given time stamp and used for LJ force calculation.

Algorithm 2 Neighbor List Build

```
1:  for i = 0; i < num of atoms; i++  do
2:     xi = pos[i][0]
3:     yi = pos[i][1]
4:     zi = pos[i][2]
5:     for all k = 0; k < num of potential neighbors do
6:        j = potential_neighbor[k]
7:        delx = xi – pos[j][0]
8:        dely = yi – pos[j][1]
9:        delz = zi – pos[j][2]
10:       rsq = (delx * delx) + (dely * dely) + (delz *delz)
11:       if rsq ≤ cutneighsq  then
12:          neighbor[i][numneigh[i]] = j
13:          numneigh[i]++
14:       end if
15:    end for
16: end for
```

Lines 7-10 of the loop are similar to LJ force calculation, the distance of target atom and potential neighbor is computed. The AoS to SoA conversion introduces gather operation which is the only performance inefficiency in neighbor list build. Lines 12-13 append the qualified neighbor whose distance to the target atom is closer than *cutneighsq* to the list and are handled by packed store. Intel Xeon Phi coprocessor provides pack/unpack instructions to handle the case where the memory data has to be compressed or expanded as they are written to memory or read from memory into a register. Mask bits generated at Line 11 is used to dictate which entries of the vector register data to be selected to fill the 64-byte form of the compressed memory data.

C. Gather/Scatter Overhead

Intel Xeon Phi coprocessor supports gather/scatter instructions to read/write sparse data in memory in or out of the packed vector registers, thus simplifying code generation for non-contiguous memory accesses. Using gather/scatter instructions are much more expensive than vector load/store because each gather/scatter instruction accesses only one cache line at a time. To fully populate all elements of the entire vector register, 16 cache line accesses are requested for single precision at most (8 for double precision).

Gather/scatter instructions in Molecular Dynamics simulation result from AoS/SoA conversion and the requested elements for one vector register are not on the same cache line which means every gather/scatter hits the worst case.

IV. Optimization

In this section, we firstly show several data alignment and padding skills in our optimization work and then discuss various vectorization techniques to achieve higher SIMD efficiency on Intel Xeon Phi coprocessor.

A. Data Padding and Redundant Computation

Vector unit of Intel Xeon Phi coprocessor reads and writes the data cache at a cache-line granularity of 512-bit through a dedicated bus. Unaligned data access causes expensive cache line split up and therefore harms performance. The default data structure of an atom in miniMD has 3 doubles, corresponding to positions in 3 dimensions. We pad an extra double to each atom and make the atom array aligned with 512-bit.

Another performance hazard is the non-divisible loop count. Wide SIMD units such as KNC cannot be effectively utilized simply exposing complier to Intel MIC architecture. For example, using a SIMD unit of vector length *VL*, a scalar code that executes a loop of *N* iterations will execute *floor*(*N*/*VL*) full vector iterations followed by *N mod VL* scalar remainder iterations. Only if *N* is sufficiently larger than *VL*, the remainder iterations can still take up a significant portion of the total execution time. Loop count in miniMD depends on the cut-off distance R_c, larger R_c introduces more neighbors. In general, the remainder iterations bring 5-10% performance inefficiency.

The non-divisible loop can be handled with several methods such as redundant computation and masked vectorization. In this work, we pad invalid neighbors which are atoms placed at infinity and always fail the cut-off check to the neighbor list so as to make loop count the nearest multiples of *VL*.

B. Vectorization Techniques

Modern SIMD architecture employs wide vector length and low power and low complexity processor design. For programmers, fully exploiting data parallelism to efficiently utilize the SIMD unit is one of the most important considerations. In this section, we first implement and analyze the existing vectorization techniques addressing the gather/scatter overhead, i.e., inter-loop, intra-loop vectorization and then propose inter-intra-hybrid vectorization method. These techniques are applicable for both LJ force calculation and neighbor list build since the overall procedure of the two loops are quite similar.

1) Inter-Loop Vectorization: Inter-loop vectorization extracts data parallelism across iterations. This approach follows the same methodology as compiler and usually is the first to come to mind for programmers. For double precision, every 8 neighbors are worked in a batch, but rather than expensive gather/scatter instructions, atom sets are read/written in the format of AoS taking the advantage of vector load/store. The AoS to SoA conversion is handled by swizzle/permute intrinsics which perform data element re-arrangement, as shown in Fig. 2. The rest of the loop executes exactly the same as Intel compiler auto vectorized code.

Although gather/scatter instructions handle data conversion together with data transfer and thus are much easier to use, the performance is limited due to cache line split. Inter-loop vectorization manipulates data transfer and data conversion separately but in a much more efficient way. Data transfer takes the advangtage of spatial locality of atom elements and data conversion is handled in vector registers rather than memory.

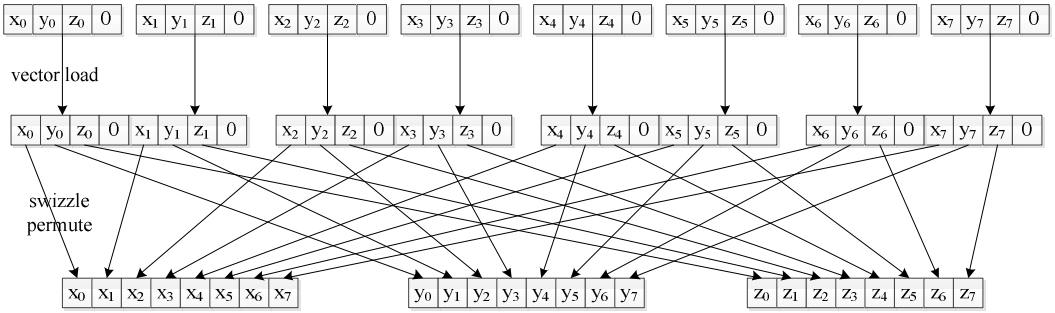

Fig. 2. Inter-loop vectorization

2) Intra-Loop Vectorization: Intra-loop vectorization extracts data parallelism within iteration and largely depends on data structure of the scalar loop which is usually overlooked by programmers. In the case of miniMD, intra-loop vectorization makes use of atom data structure {x, y, z, 0}. Atoms are manipulated in the format of AoS and no swizzle or permute instructions are needed as no data conversion is required, as shown in Fig. 3. On 512-bit SIMD unit such as Intel Xeon Phi coprocessor, distance *rsq* is calculated with vector add/mul and swizzle instructions. On 128-bit and 256-bit SIMD units, it is possible to calculate *rsq* with a single dot product instruction.

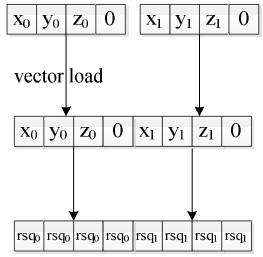

Fig. 3. Intra-loop vectorization

In intra-loop vectorization, every 2 neighbors are worked in a batch to fully populate the 512-bit vector register for double precision and no more data conversion is needed. Although SIMD architecture favors SoA essentially, manipulating data in the format of AoS can be beneficial since the trouble of AoS to SoA conversion when loading and storing are both saved. However, Naïve intra-loop vectorization cannot guarantee a better performance than Intel compiler auto vectorized code. Despite of the simpler data flow, 3/4 of the SIMD vector length is wasted after *rsq* is computed, which is a significant inefficiency of the SIMD unit.

3) Inter-Intra-Hybrid Vectorization: Inter-intra-hybrid vectorization extracts data parallelism across and within iterations simultaneously. Similar to intra-loop vectorization, data is manipulated in the format of AoS but every 8 neighbors are worked in a batch, which is the same as inter-loop vectorization. Adjacent atom sets are loaded into one vector register and 4 four vector registers are needed. After distance *rsq* is calculated, $\{rsq_0, rsq_1, \ldots, rsq_7\}$ from the 8 neighbors are packed into one vector register so as to achieve 100% SIMD efficiency in the following computations, as shown in Fig. 4, which differs from intra-loop vectorization.

Inter-intra-hybrid vectorization has the advantages of both inter-loop vectorization and intra-loop vectorization: (*i*) Data transfer is handled by vector load/store which is a much more efficient way than gather/scatter on SIMD architecture. (*ii*)Atom sets are directly manipulated in the format of AoS and thus save the trouble of data conversion when loading/storing. (*iii*) high efficiency of SIMD unit is guaranteed by data packing/unpacking at specific sections of computation.

Extrating data parallelism of both inter-loop and intra-loop can be difficult since this vectorization technique highly relies on data flow of scalar code and therefore requires specific field knowledge. However, inter-intra-hybrid vectorization provides an alternative sulotion which further extend SIMD unit efficiency and should be carefully considered in the process of performance optimization.

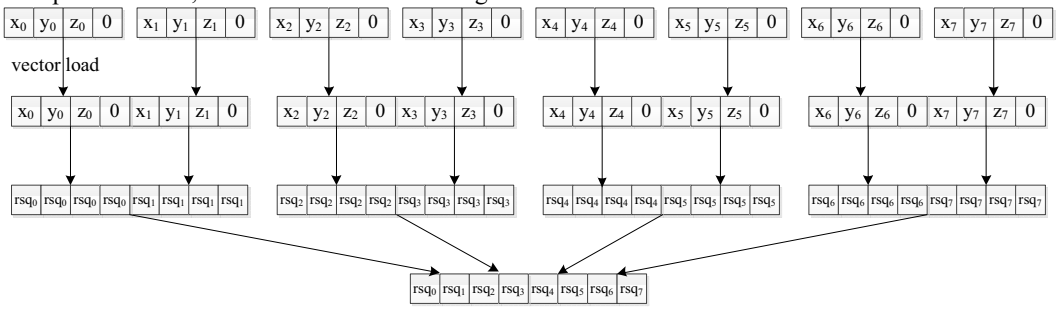

Fig. 4. Inter-intra-loop vectorization

V. RESULTS

A. miniMD Benchmark

To evaluate the performance of the different vectorization techniques, the latest code from Sandia Labs is used and the results of both LJ force calculation and neighbor list build are presented. Double precision is used and the other simulation parameters are fixed as follows: size = 60, timesteps = 100. The experiment is repeated by 100 times and the average execution time of the two major functions is calculated. The relative performance of inter-loop, intra-loop and inter-intra-hybrid in comparison of Intel compiler auto vectorized code are presented.

B. System Configuration

The detailed information on the configuration of Intel Xeon Phi coprocessor used in the performance study of proposed vectorization techniques is provided in Table I.

TABLE I. TARGET SYSTEM CONFIGURATIONS

System Parameters	Intel Xeon Phi coprocessor 7120P
Cores	61
Threads	244
Frequency	1.238GHz
Power	300W
Memory Capacity	16GB
Memory Technology	GDDR5
Memory Frequency	2.75GHz
Memory Channels	16
Memory Bandwidth	352GB/s
SIMD vector length	512 bits

C. Performance Results

Table II shows the performance comparison for 1) Default Intel compiler auto vectorized code; 2) inter-loop vectorization; 3) intra-loop vectorization; 4) inter-intra-hybrid vectorization.

TABLE II. PERFORMANCE RESULTS

Components	Auto vectorized	Inter -loop	Intra-loop	Inter-intra-hybrid
LJ force calculation	1.00x	1.20x	0.66x	1.25x
Neighbor list build	1.00x	1.35x	0.74x	1.40x

In general, inter-intra-hybrid vectorization achieves 25%-40% performance gain compared with Intel compiler auto vectorized code and outperforms the existing vectorization techniques. All the three vectorization techniques handle data transfer by vector read/write rather less efficient gather/scatter instructions. Furthermore, inter-loop vectorization uses swizzle and permute intrinsics to cope with AoS to SoA conversion and achieves 100% SIMD efficiency. Intra-loop vectorization manipulated atom sets in the format of AoS and saves the trouble of data conversion. However, it is even slower than the default Intel auto vectorized code due to the fact that 3/4 of the SIMD vector length is wasted after distance *rsq* is computed. Inter-intra-hybrid vectorization addresses this SIMD unit inefficiency by exploiting data parallelism across and within iterations simultaneously, therefore achieves the best performance.

VI. CONCLUSIONS

Driven by the increasing prevalence of SIMD architecture in High Performance Computing, we propose a novel vectorization technique that addresses the gather/scatter overhead on Intel Xeon Phi coprocessor. We analyze the existing vectorization techniques that focus on the optimization of non-contiguous memory access on SIMD architecture, i.e. inter-loop, intra-loop vectorization and provide an in-depth analysis of the advantages and disadvantages. We use the two major functions of miniMD benchmark for evaluation whose results show that our proposed method achieves a performance gain of 25%-40% compared with Intel compiler auto vectorized code and outperforms the existing methods on Intel Xeon Phi coprocessor.

The inter-intra-hybrid vectorization technique is also applicable for other vector length such as 128-bit and 256-bit. Furthermore, besides MD simulation, we believe that the proposed vectorization technique can be applied to other classes of workload that heavily rely on non-contiguous memory access.

ACKNOWLEDGMENT

The authors would like to thank Ashish Jha for his help in providing performance data of inter-loop vectorization in miniMD benchmark.

REFERENCES

[1] Intel Corporation, "Intel® Xeon Phi™ Coprocessor System Software Developers Guide," March 2014, https://software.intel.com/sites/default/files/managed/09/07/xeon-phi-coprocessor-system-software-developers-guide.pdf.

[2] Intel Corporation, "Intel® Xeon Phi™ Coprocessor Vector Microarchitecture," May 2013, https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture.

[3] Intel Corporation, "Overview: Reference for Intrinsics Supporting Intel® Initial Many Core Instructions", https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-35876E20-882E-4067-88FA-EC110B5A3D9F.htm

[4] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," in Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06, pages 132–143, 2006.

[5] H. Chang and W. Sung, "Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware," in Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '08, pages 167–176, 2008.

[6] S. Kim and H. Han, "Efficient SIMD Code Generation for Irregular Kernels," in Proceedings of the Symposium on Principles and Practice of Parallel Programming, New Orleans, LA, 25-29 February 2012, pp. 55–64.

[7] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao, "An integrated simdization framework using virtual vectors," in Proceedings of the 19th

annual International Conference on Supercomputing, ICS '05, pages 169–178, 2005.

[8] S. Pennycook, C. Hughes, M. Smelyanskiy and S. Jarvis, "Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and 1ntel® Xeon Phi Coprocessors," Parallel & Distributed Processing (IPDPS), IEEE 27th International Symposium, Boston, MA., May 2013, pp.1085-1097.

[9] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units," Journal of Computer Physics, vol. 227, no. 10, pp. 5342–5359, 2008.

[10] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, Improving Performance via Mini-applications, Technical Report, SAND2009-5574, 2009.

[11] S. Plimpton et al., "LAMMPS Molecular Dynamics Simulator,"http://lammps.sandia.gov/, May 2011.

[12] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, TX., 2008, pp. 1–9.

[13] C. R. Trott, "LAMMPScuda - A New GPU-Accelerated Molecular Dynamics Simulations Package and its Application to Ion-Conducting Glasses," Ph.D. dissertation, Ilmenau University of Technology, 2011.

[14] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," J. Comp. Chem. 28 (2007) 2618–2640.

[15] J. Yang, Y. Wang, Y. Chen, "GPU accelerated molecular dynamics simulation of thermal conductivities," J. Chem. Phys. 221 (2007) 799–804.

[16] A. Harode, A. Gupta, B. Mathew, and N. Rai, "Optimization of Molecular Dynamics Application for Intel Xeon Phi Coprossor," High Performance Computing and Applications, Bhubaneswar, 2014, pp. 1-6.

[17] F. Wende, T. Steinke, F. Cordes, "Concurrent Kernel Execution on Xeon Phi within Parallel Heterogeneous Workloads," Euro-Par 2014 Parallel Processing, Lecture Notes in Computer Science, Volume 8632, 2014, pp 788-799.