Parallel Computing 85 (2019) 190–203

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

pointerchain: Tracing pointers to their roots – A case study in molecular dynamics simulations

Millad Ghane[a], Sunita Chandrasekaran[c,*], Margaret S. Cheung[a,b,d]

[a] Department of Computer Science, University of Houston, TX, United States
[b] Department of Physics, University of Houston, TX, United States
[c] Department of Computer & Information Sciences (CIS), University of Delaware, DE, United States
[d] Center for Theoretical Biological Physics (CTBP), Rice University, Houston, TX, United States

ARTICLE INFO

Article history:
Received 16 September 2018
Revised 12 March 2019
Accepted 20 April 2019
Available online 22 April 2019

Keywords:
Directives
Portability
Scientific computing
Molecular dynamics
Performance
Parallel computing
Heterogeneous system
GPU
Accelerators

ABSTRACT

As scientific frameworks become sophisticated, so do their data structures. A data structure typically includes pointers and arrays to other structures in order to preserve application's state. In order to ensure data consistency from a scientific application on a modern high performance computing (HPC) architecture, the management of such pointers on the host and the device, has become complicated in terms of memory allocations because they occupy separate memory spaces. It becomes so severe that one must go through a *chain of pointers* to extract the effective address. In this paper, we propose to reduce the need of excessive data transfer by introducing the idea of `pointerchain`, a directive that replaces the pointer chains with their corresponding effective address inside the parallel region of a code. Based on our analysis, `pointerchain` leads to a 39% and 38% reduction in the amount of generated codes and the total executed instructions, respectively.

With `pointerchain`, we have parallelized CoMD, a Molecular Dynamics (MD) proxy application on heterogeneous HPC architectures while maintaining a single portable codebase. This portable codebase utilizes OpenACC, an emerging directive-based programming model, to address the need of memory allocations from three computational kernels in CoMD. Two of the three embarrassingly parallel kernels highly benefit from OpenACC and perform better than the hand-written CUDA counterparts. The third kernel performed 61% of peak performance of its CUDA counterpart. The three kernels are common modules in any MD simulations. Our findings provides useful insights into parallelizing legacy MD software across heterogeneous platforms.

1. Introduction

Heterogeneous computing systems comprise multiple and separate levels of memory spaces; thus, they require a developer to explicitly issue data transfer from one memory space to another with software application programming interfaces (APIs). In a system composed of a host processor and an accelerator (referred to as device in this paper), the host processor cannot directly access the data on the device and vice versa. For such systems, the data are copied back and forth between the host and the device with an explicit request from the host. This issue has become particularly severe for supercomputers as the number of devices connected to one node increases. For an example, the Titan supercomputer from ORNL has only one NVIDIA K20 GPU per node, while this number for the latest supercomputer, Summit from ORNL, is six NVIDIA Volta GPUs [1]. Supercomputers with different device families will continue exacerbating this issue [2].

Heterogeneous computing systems pose a challenge to the community of scientific computing. As a scientific framework becomes sophisticated, so does its data structures. A data structure typically includes pointers (or dynamic arrays) that point to *primitive data types* or to other user-defined data types. As a result, transfer of the data structure from the host to the other devices mandates not only the transfer of the main data structure but also its nested data structures, a process known as the *deep copy*. The tracking of pointers that represent the main data structure on the host from its counterpart on the device further complicates the maintenance of the data structure. Although this complicated process of deep copy avoids a major change in the source codes, it imposes unnecessary data transfers. In some cases, a *selective deep copy* is sufficient when only a subset of the fields of the data structure on the device is of interest [3]; however, even though the data motion

* Corresponding author.
E-mail addresses: mghane2@uh.edu (M. Ghane), schandra@udel.edu (S. Chandrasekaran), mscheung@central.uh.edu (M.S. Cheung).

https://doi.org/10.1016/j.parco.2019.04.007
0167-8191/

decreases proportionally, the burden to maintain data consistency among the host and other devices still exists.

In this study, we address the shortcoming of data transfer between the host and a device by extracting the effective address of a final pointer in a chain of pointers as the source for our data transfer. Utilizing the effective address also leads to a reduction of the generated assembly code by replacing the pointer chain with a single pointer. This single pointer will suffice for the correct execution of the kernel on both the host and the device. As a result, we have improved the performance of parallel regions by reducing an unnecessary deep copy of the data structure between the host and the device. We have developed the `pointerchain` directive to provide these useful features to a developer to transfer data, which eliminates the need for a complete implementation of deep copy in a compiler and runtime library.

We have demonstrated the merit of `pointerchain` by improving the efficiency and portability of scientific applications, molecular dynamics (MD) simulations, on a heterogeneous computing system. MD is an essential tool for investigating the dynamics and properties of small molecules at the nano-scale. It simulates the physical movements of atoms and molecules with a Hamiltonian of N-body interactions. Over the past three decades, we have witnessed the evolution of MD simulations as a computational microscope that has provided a unique framework for understanding the molecular underpinning of cellular biology [4], which applies to a large number of real-world examples [5–11]. Currently, major MD packages, such as AMBER [12], LAMMPS [13], GROMACS [14], and NAMD [15], use low-level approaches, like CUDA [16] and OpenCL [17], to utilize GPUs to their benefits for both code execution and data transfer. They are not, however, equipped for the dire challenge in next-generation exascale computing in which the demand of parallelism [18] is achieved by the integration of a wide variety of accelerators, such as GPUs [19] and many integrated cores (MIC) co-processors [20,21], into the high-performance computational nodes.

Legacy MD codes, which use low-level methods like CUDA and OpenCL, require a steep learning curve, which is not an ideal scenario for scientists. Therefore, scientists have been exploring the utilization of high-level approaches like domain-specific and script-based languages [22–25] in MD simulations. Such approaches, however, demand significant code change, which is not feasible in many cases. In such cases, the software has reached a maturity level that the incorporation of other languages with various levels of complexity is not a trivial task and may present unintended consequences. To overcome this dilemma and address above-mentioned concerns, one can utilize directive-based programming models [26–28]. These programming models, for instance, OpenMP [29] and OpenACC [30], provide facilities to express parallelism with less code intervention. Scientists have first hands-on experience in parallelizing their potential code regions by inserting simple *directive constructs* into the source codes. To that end, there have been many attempts recently to incorporate directive-based models into MD simulation frameworks [31–33] and other disciplines in science [28,34–36]. Relying on directives helps developers deal with a single code base instead of one for every upcoming architecture [37–40], and thus increasing the application's portability opportunities.

In this paper, we have chosen OpenACC as the targeting model for realizing the `pointerchain` directive to reduce the burden of data transfer in proxy code for molecular dynamics simulations across HPC in scientific applications. Ratified in 2011, OpenACC is a standard parallel programming model designed to simplify the development process of scientific applications for heterogeneous architectures [38,41]. The success of our approach provides far-reaching impacts on modernizing legacy MD codes ready for the exascale computing.

Following are the contributions of our research discussed in this paper:

- We create a directive called `pointerchain` (Section 3) to simplify the pointer management in scientific applications (Section 2) thus reducing the lines of code required.
- We apply our proposed directive on an MD proxy application, CoMD [42]. We will also discuss the effect of `pointerchain` on the source code and code generation process for CoMD in the results section (Section 6).
- We propose guidelines for parallelization of CoMD that can also be applied to other legacy MD source codes, discussed in Section 4.
- Finally, we investigate the performance of CoMD implemented with OpenACC in terms of scalability, speedup, and floating-point operations per second (Section 6).

The remainder of this paper is structured as follows: Section 2 describes the programmatic gap in current directive-based programming models that handle multiple pointers. Section 3 describes our proposed directive to fill the gap mentioned in Section 2. In Section 4, we provide a case study that utilizes our proposed directive to parallelize a scientific application. Sections 5 and 6 describe our evaluation system and the results of our conducted experiments. Related works are discussed in Section 7. And finally, we conclude our paper in Section 8.

2. Motivation: The programmatic feature gap

Modern HPC platforms comprise two separate memory spaces: the *host* memory space and the *device* memory space. A memory allocation in one does not guarantee an allocation in the other. Such an approach demands a complete replication of any data structure in both spaces to guarantee data consistency. However, data structures become more complicated as they retain complex states of the application. Throughout this paper, we opt for the C/C++ languages as our main programming languages in developing scientific applications.

Fig. 1 shows a typical case of the design of a data structure for scientific applications. The arrows represent pointers. The number next to each structure shows the physical address of an object in the main memory. Here, the main data structure is the `simulation` structure. Each object of this structure has member pointers to other structures, like the `atoms` structure. The `atoms` structure also has a pointer to another `traits` structure, and so on. As a result, to access the elements of the `positions` array from the `simulation` object we would have to dereference the following chain of pointers: `simulation->atoms->traits->positions`. Every arrow from this chain goes through a dereference process to extract the effective address of the final pointer. We call this chain of accesses to reach the final pointer (in this case, `positions`) a *pointer chain*. Since every pointer chain eventually resolves to a memory address, we propose the extraction of the effective address and replace it with the chain in the code.

Currently, there are two primary approaches to address pointer chains. The first approach is the deep copy that requires excessive data transfer between the host and the device, as mentioned in the Introduction. The second approach is the utilization of Unified Virtual Memory (UVM) on Nvidia devices. UVM provides a single coherent memory image to all processors (CPUs and GPUs) in the system, which is accessible through a common address space [43]. It eliminates the necessity of explicit data movement by applications. Although it is an effortless approach for developers, it has several drawbacks: (1) It is supported only by Nvidia devices but not by Xeon Phis, AMD GPUs, and FPGAs, among others; (2) It is not a *performance-friendly* approach due to its arbitrary memory

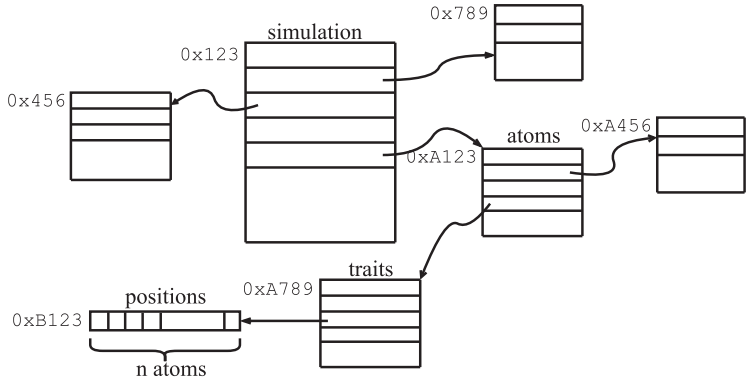

Fig. 1. An example of a pointer chain: an illustration of a data structure and its children. To reach the `position` array, the processor must dereference a chain of pointers to extract the effective address.

transfers that could happen randomly. The consistency protocol in UVM depends on the underlying device driver that traces memory page-faults on both host and device memories. Whenever a page fault occurs on the device, the CUDA driver fetches the most up-to-date version of the page from the main memory and provides it to the GPU. Similar steps are taken when a page-fault happens on the host.

Although deep copy and UVM address the data consistency, they impose different performance overheads on the application. In many cases, we are looking for a somewhat intermediate approach; while we are not interested in making a whole object and all of its nested children objects accessible on the device (like UVM), we aim at transferring only a subset of the data structures to the device without imposing deep copy's overhead. Our proposed approach, **`pointerchain`**, is meant to be a minimal approach that borrows the beneficial traits of the above-mentioned approaches. **`pointerchain`** is a directive-based approach that provides selective accesses to data fields of a structure while offering a less error-prone implementation.

```
1  typedef struct {
2    ...
3    // position, momenta, and force in 3D space
4    double *positions[3];
5  } Traits;
6  typedef struct {
7    ...
8    // position, momenta, and force in 3D space
9    Traits *traits;
10 } Atoms;
11 typedef struct {
12   ...
13   // atom data (positions, momenta, ...)
14   Atoms* atoms;
15 } Simulation;
16 Simulation *simulation;
17 for(int i=0;i<nAtoms;i++) {
18   simulation->atoms->traits->positions[i][0] = ...;
19   simulation->atoms->traits->positions[i][1] = ...;
20   simulation->atoms->traits->positions[i][2] = ...;
21 }
```

Fig. 2. A sample code and its data structures based on Fig. 1. In the commercial softwares, due to the code maintainability purposes, the `positions` arrays are accessed as shown below. The goal is to improve the software readability for future references.

3. Proposed directive and clauses

3.1. Proposed directive: `pointerchain`

As a compiler reaches a pointer chain in the source code, it generates a set of machine instructions to dereference the pointer and correctly extract the effective address of the chain for both the host and the device. Dereferencing each intermediate pointer in the chain, however, is the equivalent of a memory load operation, which is a high-cost operation. As the pointer chain lengthens with a growing number of intermediate pointers, the program performs excessive memory load operations to extract the effective address. This extraction process impedes performance, especially when the process happens within a loop (for instance a `for` loop). To alleviate the implications of the extraction process, we propose to perform the extraction process before the computation region begins, and then reuse the extracted address within the region afterwards.

We demonstrate the idea of extracting process from a pointer chain using the configuration in Fig. 1. Fig. 2 shows an implementation of this configuration in the C++ code. In this configuration, we replace the pointer chain of `simulation->atoms->traits->positions` with the corresponding effective address of `positions` (in this case `0xB123`). This pointer then is used for data transfer operations to and from the accelerator and also with the computational regions. It bypasses the transmission of redundant structures (in this case, `simulation`, `atoms`, and `traits`) to the accelerator that, in any case, will remain intact on the accelerator. The code executed on the device will modify none of these objects. Moreover, it keeps the accelerator busy performing "useful" work rather than spending time on extracting effective addresses.

The targeted pointers are allocated either dynamically (`malloc` API in C or `new` in C++) or statically (e.g., '`double arr[128];`' at compile time). Since **`pointerchain`** is utilizing the effective address of a chain, the allocation strategy does not affect how **`pointerchain`** works.

Utilizing the effective addresses as a replacement to a pointer chain, however, demands code modifications on both the data transfer clauses and the kernel codes. To address these concerns, we propose a set of directives that minimally change the source code for announcing the pointer chains and for specifying the regions that benefit from pointer chain replacements. The justification for having an **`end`** keyword in **`pointerchain`** is that our implementation does not rely on a sophisticated compiler (as we will discuss in Section 3.2) to recognize the beginning and the end of complex statements (e.g., the `for` loops and the compound block statements). Our motivation behind utilizing a script rather than a compiler was to minimize the prototyping process and to implement our proof-of-concept approach by avoiding the steep learning curve of the compiler design. The steps mentioned in the Section 3.2 can also be supported with a modern compiler.

Our proposed directive – **pointerchain** directive – accepts two constructs: **declare** and **region**. Developers use **declare** construct to announce the pointer chains in their code. The syntax in C/C++ is as following:

#pragma pointerchain declare(*variable [,variable]...*)

where *variable* is defined as below:

variable := name{type[: ***qualifier*** **]}** where:

- **name**: the pointer chain
- **type**: the data type of the last member of the chain
- **qualifier**: an optional parameter that is either **restrictconst** or **restrict**. These will make the underlying variable decorated with **__restrict const** and **__restrict** in C/C++, respectively. These qualifiers provide hints to the compiler to optimize the code with regard to the effects of pointer aliasing.

The following lines show how to use **begin** and **end** clauses with **region** construct after marking the pointer chains in the source code with the **declare** clause. The pointer chains that were previous declared in the current scope are the subject for transformation in subsequent regions.

```
#pragma pointerchain region begin
<... computation or data movement...>
#pragma pointerchain region end
```

Our two proposed clauses (**declare** and **region**) provide developers with the flexibility of reusing multiple variables in multiple regions; however, there exists a condensed version of **pointerchain** that performs the declaration and replacement process at the same time. The condensed version of **pointerchain** replaces the declared pointer chain with its effective address in the scope of the targeted region. It is placed on the **region** clauses. An example of a simplified version, enclosing a computation or data movement region, is shown below:

```
#pragma pointerchain region begin declare
(variable [,variable]...)
<... computation or data movement...>
#pragma pointerchain region end
```

When our kernels (regions) have a few variables, the condensed version is a favorable choice in comparison to the **declare/region** pair. It leads to a clean, high-quality code; however, utilizing the pair combination helps with code readability, reduces code complexity, and expedites porting process to the OpenACC programming model. Having utilized modern compilers, such compilers are able to incorporate the condensed version of **pointerchain** with the OpenACC and OpenMP directives directly, as shown below for the OpenACC case.

```
#pragma acc parallel pointerchain (variable [,variable]...)
<... computations...>
```

Our proposed directive, **pointerchain**, is a language- and programming-model-agnostic directive. In this paper, for implementation purposes, **pointerchain** is developed in C/C++ and OpenACC. One can utilize it for Fortran language or target the OpenMP programming model. We show a sample code about how to use our proposed directive in Fig. 2.

In Fig. 2, the defined structures follow the illustration described in Fig. 1. Lines 1–22 show the defined data structures we used. Our computational kernel, lines 26–30, initializes the position of every atom in 3D space in the system. These lines represent a normal, formal `for-loop` that has the potential to be parallelized by directive-based programming models. Fig. 3 shows an example of how to parallelize a `for` loop by exploiting **pointerchain** with that `for` loop on lines 17–21 of Fig. 2. At first, we declared the pointer chain (line 2), then utilized the **region** clause for data transfer (Lines 4–6), and finally, utilized the **region** clause to parallelize the `for` loop (lines 9–16). No modification to the `for` loop is required in comparison to Fig. 2. Fig. 3 also shows how the **pointerchain** directives are transformed to a set of conformed C/C++ statements. A local pointer is assigned to the chain of pointers, then the local pointer is utilized for both data transfer and kernel execution. Despite its simplicity as shown in Fig. 3, **pointerchain** provides certain flexibility to the system. For instance, if we target only a multicore device, we easily ignore the **pointerchain** directives in the code. Furthermore, if developers perform this task manually, it will reduce the readability of the code.

3.2. Implementation strategy

To simplify the prototyping process, we have developed a Python script that performs a *source-to-source* transformation of the source codes annotated with the **pointerchain** directives. Our transformation script searches for all source files in the current folder and finds those annotated with the **pointerchain** directives. They are then transformed to their equivalent code.

Here is the overview of the transformation process. Upon encountering a **declare** clause, for each variable, a *local variable* with the specified **type** is created and initialized to the effective address of our targeted pointer chain (variable **name**). If **qualifiers** are set for a chain, they will also be appended. Any occurrences of pointer chains in between **region begin** and **region end** clauses are replaced with their counterpart local pointers announced before by **declare** clauses in the same functional unit.

Scalar variables (i.e. `simulation->atoms->N`) are treated differently in **pointerchain**. We start by defining a local temporary variable to store the latest value of the scalar variable. Then all occurrences of the scalar pointer chain within the region are replaced with the local variable. Finally, after exiting the region, the scalar pointer chain variable is updated with the latest value in the local variable.

Introducing new local pointers to the code has some unwelcome implications on the memory (stack) usage. They are translated into a memory space on the call stack of the calling function. We have alleviated this burden by reusing the local variables that were extracted from the pointer chain instead of reusing pointer chains over and over again. This is especially beneficial when we target GPU devices. We have investigated the implications of the **pointerchain** from several perspectives including code generation, performance, and memory (stack) layout, and, compared the results with UVM. We will discuss our findings regarding the imposed overheads by **pointerchain** in Section 6.

4. Case study: CoMD

The Co-Design Center for Particle Applications (COPA) [44], a part of Exascale Computing Project (ECP), has established a set of proxy applications for real-world applications [45] that are either too complex or too large for code development. The goal of these proxy applications is for vendors to understand the application and its workload characteristics and for application developers to understand the hardware. The tools and software developers need them for expanding libraries, compiler and programming models as well.

CoMD [42] is a proxy application of classical molecular dynamics simulations, which represents a significant fraction of the workload that the DOE is facing [46,47]. It computes short-range forces between each pair of atoms whose distance is within a cutoff range. It does not include long-range and electrostatic forces

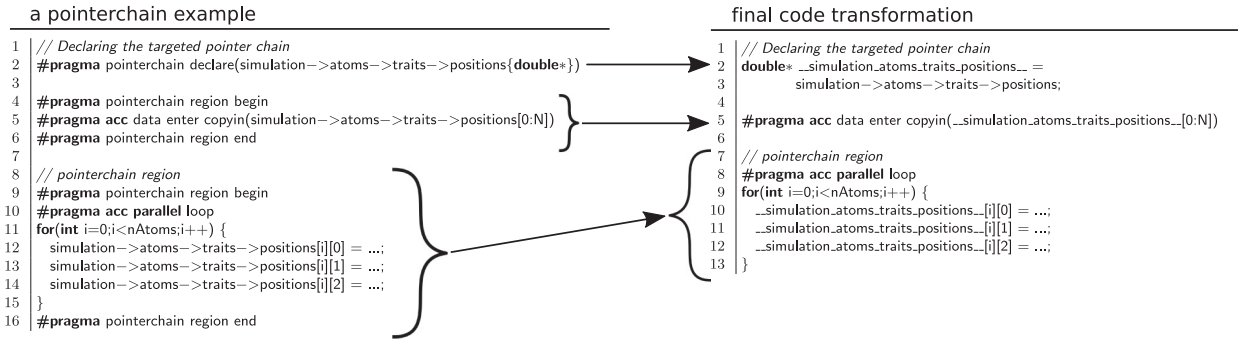

Fig. 3. An example on how to use `pointerchain` directive for data transfer and kernel execution. Required code transformation is also shown in this figure.

inherently. The evaluated forces are used to update atoms characteristics (position, velocity, force, and momenta) via numerical integration [48].

Computations in CoMD are divided into three main kernels for each time step: force computation, advancing position, and advancing velocity. The latter two kernels are considered as embarrassingly parallel (EP) kernels since their associated computations are performed on each atom independently. The velocity of an atom is updated according to the exerted force on that atom, and the position of an atom is updated according to its updated velocity. The most time-consuming phase, however, is the force computation phase.

Computing the forces that atoms exert on each other follows the equations of Newton's Laws of Motions, which is based on the distance between every pair of atoms; however, searching for neighbors of all atoms requires an $O(N^2)$ computation complexity, which is utterly inefficient. To overcome such an issue, CoMD exploits the link-cell method. It partitions the system space by a rectangular decomposition method in such a way that size of each cell exceeds the cutoff range in every dimension. This way, neighbors could be extracted from the cell-containing atom and the 26 neighboring cells around that cell. Through using link-cells, the computational complexity decreases to $O(27 \times N)$, which essentially is linear.

Algorithm 1 describes the CoMD phases. It follows the Verlet algorithm [49] in MD simulations. In each time step, velocity is advanced at an interval of one half time-step, and the position is updated for the full time-step based on the computed velocities. With the updated velocity and position values, we update the forces for all atoms. Later, velocities are updated for the remainder of the time step to reflect the advances for one full time-step.

Algorithm 1 MD timesteps in Verlet algorithm.

Input: *sim*: simulation object
Input: *nSteps*: total number of time steps to advance
Input: *dt*: amount of time to advance simulation
Output: New state of the system after *nSteps*.

```
1: function TIMESTEP(sim, nSteps, dt)
2:     for i ← 1 to nSteps do
3:         ADVANCEVELOCITY(sim, 0.5*dt)
4:         ADVANCEPOSITION(sim, dt)
5:         REDISTRIBUTEATOMS(sim)
6:         COMPUTEFORCE(sim)
7:         ADVANCEVELOCITY(sim, 0.5*dt)
8:     end for
9:     KINETICENERGY(sim)
10: end function
```

Updating the position of atoms leads to the migration of atoms among neighbor cells and, in many cases, among neighbor processors. After position updates, link-cells are required to be updated locally (intra node/processor) and globally (inter nodes/processors) in each time step as well. This process is guaranteed to be done in the REDISTRIBUTEATOMS function of Algorithm 1.

Force calculations in the Verlet algorithm are derived from the gradient of the chosen potential function. A well-known interatomic potential function that governs relation of atoms and is extensively used in MD simulations is Lennard–Jones (LJ) [50]. CoMD supports an implementation of LJ to represent force interaction between atoms in a system. The LJ force function will be called inside the `ComputeForce` kernel in Verlet algorithm (Algorithm 1). Moreover, CoMD also supports another potential function known as the Embedded Atom Model (EAM), which is widely used in metallic simulations. In this paper, due to its simplicity in design and the fact that it is widely used in protein-folding applications, we will be focusing on the LJ potential function.

4.1. Reference implementations

CoMD was originally implemented in C language and it uses the OpenMP programming model to exploit the intra-node parallelism and MPI [51] to distribute work among nodes [42]. Cicotti et al. [52] have investigated the effect of exploiting a multithreading library (e.g. *pthreads*) instead of using the OpenMP and MPI approach. In addition to the OpenMP and MPI implementations, a CUDA-based implementation was also developed in C++ language [47]. These reference versions include all of the three main kernels: force computation, advancing velocity, and advancing position of atoms. Developers used CUDA to be able to fully exploit the capacity of the GPUs. As a result, the data layout of the application was significantly changed in order to tap into the rich capacity of the GPUs. Naturally, this puts a large burden on the developers and, the code cannot be used on any other platforms other than NVIDIA GPUs. Both OpenMP and CUDA implementations were optimized to utilize the full capacity of the underlying hardware. In our paper we focus on the optimizations beneficial to the OpenACC implementation.

4.2. Parallelizing CoMD with OpenACC

This subsection is dedicated to the discussion of porting CoMD to a heterogeneous system using the OpenACC programming model. We started with the OpenMP code version for this porting process instead of the serial code. This may not be the best approach because in most cases the OpenMP codes are well-tuned and optimized for shared memory platform but not for heterogeneous systems, especially the codes that have used OpenMP 3.x.

As the first step, we profiled the code and discovered that the force computation (*line 6* in Algorithm 1) was the most time-consuming portion of the code. Consequently, it urges us to port

force computations to the device requiring the transfer of both the computational kernel and its data (the data that the kernel is working on) to the device. If we accelerate only the force computation kernel, however we need to transfer data back and forth to and from the device for each time step, which will lead to dramatic performance degradation. That is, it imposes two data transfers (between host and device) for each time step. As a result, this pushes us to parallelize other steps (*line 3, 4, and 7*) too. Hence, data transfers can be performed before (*line 2*) and after (*line 8*) the main loop.

The REDISTRIBUTEATOMS step (*line 5*) guarantees data consistency among different MPI [51] *ranks*. Since MPI functions are allowed to be called only within the host, the data have to be transferred back to the host for synchronization purposes among the ranks. After performing synchronization, the updated data are transferred from the host to the device. The synchronization process is done on every time step to maintain data consistency. Consequently, two data transfers are performed in this step between the host and the device, and, since no remarkable computations were performed in this step, no parallelization was required for this step.

Based on our analysis, the parallelization of the three above-mentioned kernels (`ComputeForce`, `AdvancePosition`, and `AdvanceVelocity`) contributes the most towards the performance of our application because they are the most time-consuming computational kernels. Although the latter two kernels may seem insignificant due to their smaller execution time, they will progressively affect the wall clock time of the application in the long run. Thus, the focus of our study is applying performance optimization on these three kernels. Our measurements reveal that our OpenACC implementation was able to reach the same occupancy level as that of the CUDA implementation. Force computation, however, is more complex and requires more attention with respect to its optimization opportunities; however, we can safely use the OpenACC version of `ComputeForce` and `AdvancePosition` kernels with their CUDA counterparts with no performance loss.

Considering the difficulties in dealing with pointers, we have four options to parallelize CoMD: (1) UVM, (2) deep copy, (3) significant code changes to transfer data structures manually, and (4) **`pointerchain`**. *Step 1* in our proposed steps represents the UVM approach and, as elaborated in Section 2, it has several disadvantages. Deep copy is not yet fully implemented in many compilers. The third option, significant code changes performed manually, is not a favorable approach for developers, and it contradicts the philosophy of OpenACC. That brings us to our fourth and the last option, **`pointerchain`**. Annotating CoMD's source codes with **`pointerchain`** directive helps us to easily port CoMD to OpenACC and it also helps us apply the different optimizations listed in the Table 1. Please refer to the Supplementary Material (Section A) for detailed description of each step.

Table 1 provides a brief description of the ten steps taken in this paper to parallelize CoMD. Fig. 4 shows the order in which we took the steps. These steps also provide a roadmap for parallelization of any other scientific applications using OpenACC. The *pointerchain* column shows whether our proposed novel directive has been used for a step or not. Without the **`pointerchain`** directive, the source code needs to undergo numerous modifications. Such modifications are error-prone and cumbersome for developers.

We would like to add that to the best of our knowledge, *pointerchain* is the ideal candidate for applications that heavily utilize multiple nested data structures. Particularly, it is the innermost data structure that benefits from parallelization the most. Nested data structures is a very common approach in MD and other scientific domains to maintain the simulation state of the application.

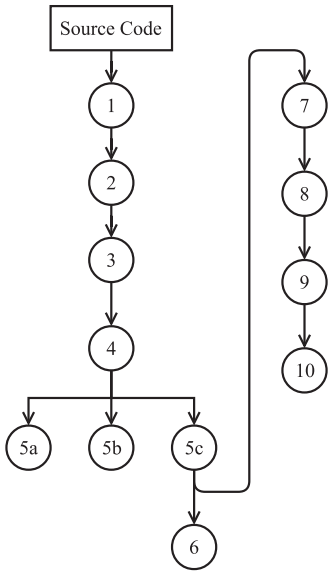

Fig. 4. Relationship among steps.

For instance, other real-world and proxy applications like miniMD [53], miniAMR [53], miniFE [53], GROMACS [14], and LAMMPS [31] have nested data structures in their source code similar to CoMD. We chose CoMD due to its similarity to MD simulation applications.

Real-world simulation applications, such as GROMACS and LAMMPS, are very time-consuming. For instance, a two-microsecond simulation will take weeks and months to finish. Consequently, applying *pointerchain* to the real-world applications and investigating its effects will require quite a long time before arriving at meaningful results. Hence, we use a proxy code, CoMD, to demonstrate the applicability of our approach and showcasing promising results. There is definitely potential to apply our method to real-world applications in the near future or as part of our next publication.

5. Evaluation

We used three HPC clusters for our experiments in this paper. The BigRed II [54], housed at Indiana University (IU), contains Cray XK7 with 1020 compute nodes, where their GPU-accelerated nodes have one AMD Opteron 16-core Interlagos x86-64 CPU, 32 GB of RAM, and an NVIDIA Tesla K20 GPU accelerator. NVIDIA's K20 has a peak single-precision FLOP rate of 3.52 TFLOPS and memory bandwidth of 208 GB/s and is equipped with 5 GB of GDDR5 memory. We used PGI compiler and CUDA toolkit with versions 17.7 and 7.5.17, respectively.

The UHPC [55] cluster, located at the University of Houston, hosts compute nodes of type HPE Apollo XL190r Gen9. They are equipped with a dual Intel Xeon Processor E5-2660 v3 (10 cores) running at 2.6 GHz with 128 GB of memory. An NVIDIA Tesla K80 GPU with 12 GB of GDDR5 is connected through PCI-Express Gen3 to compute nodes, which is capable of transferring 15.75 GB/s between main memory and the GPU. PGI compiler version 17.5 and CUDA Toolkit 8.0 was used on this cluster to build our codes.

The NVIDIA [56] cluster hosts nodes with dual socket 16-core Haswell E5-2698 v3 at 2.30GHz and 256 GB of RAM. Four NVIDIA Pascal P100 GPUs are connected to this node through a PCI-Express bus. NVIDIA's P100 leads to a peak performance of 10.5 TFLOPS and memory bandwidth of 160 GB/s. P100's memory system is a 16 GB GDDR5 memory. We used ICC 17.0 to compile the OpenMP optimized code for Intel architectures. To compile CUDA and OpenACC

Table 1
Overview of all steps that were applied to CoMD. The *pc* column designates whether `pointerchain` was applied at that step or not.

S.	Title	*pc*	Description
1	Kernel parallelization	×	Relying on the UVM for data transfer. Annotating the potential kernels with `#pragma acc kernels`.
2	Efficient data transfer	✓	Disabling UVM and specifying manual data transfer between host and device. We started using `pointerchain` from this step forward. `#pragma acc kernels` for parallelization.
3	Manual parallelization	✓	Utilizing `#pragma acc parallel` on kernels instead of `#pragma acc kernels`. Designating `gang` and `vector` levels on multi-level loops.
4	Loop collapsing	✓	Collapsing tightly nested loops into one and generating one bigger, flat loop.
5a	Improving data locality (dummy field)	✓	Adding a dummy field to make data layout cache-friendly.
5b	Improving data locality (data reuse)	✓	Improving the locality of the innermost loops by employing local variables in the outermost loops.
5c	Improving data locality (layout modif.)	✓	Modifying layout as described in details in the Supplementary Material.
6	Pinned memory effect	✓	Enabling *pinned memory allocations* instead of regular pageable allocations.
7	Parameters of parallelism	✓	Setting `gang` and `vector` parameters for parallel regions.
8	Control resources at compile time	✓	Manually setting an upper limit on the number of registers assigned to a `vector` at compilation time.
9	Unrolling fixed sized loops	✓	Unrolling one of the time consuming loops with fixed iteration count.
10	Rearranging computations	✓	Applying some code modifications to eliminate unnecessary computations.

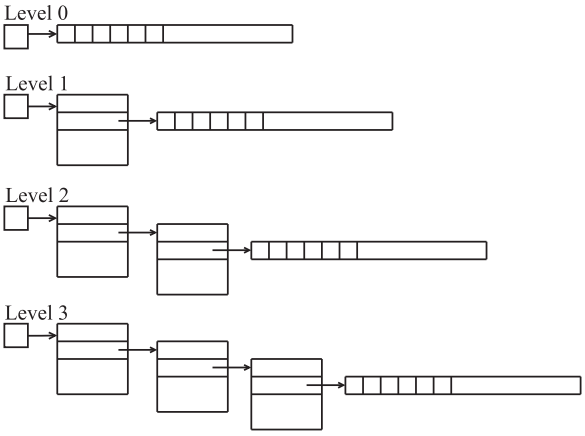

Fig. 5. Synthetic structures.

codes, we used CUDA Toolkit 9.0.176 and PGI 17.10, respectively. A number of PSG nodes are also equipped with NVIDIA Volta V100. They have 16 GB of memory with bandwidth of 600 GB/s and the peak theoretical performance of 14 TFLOPS in single-precision mode.

6. Results

We show results from our experiments in this section. First, we use a set of synthetic codes to discuss the overhead imposed by `pointerchain`. We show that elongating pointer chains adversely affects the performance. Secondly, we present the performance results of parallelizing and accelerating CoMD using OpenACC. The performance implications of `pointerchain` in each step is included in our measurements.

6.1. `pointerchain` overhead

We investigated the effect of our proposed directive, `pointerchain`, on a set of synthetic codes. These synthetic codes reveal that elongating pointer chains affects the *code generation* on both the host and the device and the *stack memory usage of the application*. Fig. 5 depicts the synthetic structures in our experiments. At Level 0, we start with no chains (i.e., introducing an extra pointer to hold current pointer) and then increase the pointer levels one by one. At Level 1, a data structure has a pointer to the main data array. At Level 2, we add another intermediate level to the chain. Such a transformation adds an extra pointer-dereferencing process to extract the effective address. At Level 3, we increase the chain size by adding another intermediate layer to reach the final array. At this level, we dereference three pointers to reach the final address and eventually extract the effective address. Each synthetic structure was implemented in a simple C program for both UVM and `pointerchain`. The following discussions show the overhead that `pointerchain` imposes on the applications in comparison to UVM.

Table 2
The effect of `pointerchain` on code generation for different architectures (x86-64 and NVIDIA K80) on the UHPC cluster [55]. Numbers in the *ptc (Δ)* columns show the extra lines/instructions imposed by the `pointerchain` method with respect to their UVM counterparts. All the results are median values of 20 runs, and they belong to the synthetic benchmarks.

	Source Code		Assembly			
	C Source Code		Device (PTX)		Host (X86-64)	
Level	UVM	ptc (Δ)	UVM	ptc (Δ)	UVM	ptc (Δ)
0	11	+3	55	0	2510	+24
1	18	+3	53	−6	2620	+42
2	24	+3	72	−25	2740	−2
3	30	+3	77	−30	2827	−10
	Instructions at the execution time					
	Device		Host (user level)		Host (kernel level)	
Level	UVM	ptc (Δ)	UVM	ptc (Δ)	UVM	ptc (Δ)
0	6952	0.00	142967	−28	1342851	+1925
1	11060	−1896	142167	−1	1347762	−2788
2	14852	−5688	148755	+12	1529134	−3706
3	14457	−5293	148755	+12	1527692	+2596

6.1.1. Code generation

The `pointerchain` directive affects the process of code generation in three aspects: (a) total lines of C source code, (b) total lines of assembly code generated for both the host and the device, and, (c) total number of instructions executed by the application at run time. Such codes generated by `pointerchain` positively impact the address dereferencing process on the device, as discussed in Section 3. The generated codes are free of instructions that are required to extract the effective addresses. We show how a few extra lines of `pointerchain` – three lines in our case – leads to dramatic reductions in the number of generated and executed instructions, especially on the device.

Table 2 shows the values measured for the above-mentioned metrics. We tabulated results as we stepped through each of the four levels. Each row represents a level as shown in Fig. 5, and the columns represent different methods, UVM and **pointerchain**. The results for **pointerchain**, shown as **ptc (Δ)**, represent additional lines that **pointerchain** imposes on the source code with respect to the one that utilizes UVM.

Let us look into the metrics and our results in detail:

a) Total number of modified C source code: As a classic metric to measure code complexity quantitatively, we counted *lines of code (LOC)* using the *cloc* tool [57]. This shows the efforts taken by developers to add directives to the code. With UVM, for example, Level 2 took 24 lines of code for implementation. The addition of **pointerchain** increased it by only 3 lines – one line for **declare** directive and two lines for **region begin** and **region end** directives. This metric represents the amount of effort required to implement the code with a particular approach: UVM or **pointerchain**. It estimates the productivity and maintainability of the approach.

b) Total number of assembly code generated for both the host and the device: For the host code, we relied upon the output assembly code from the PGI compiler to count the LOC of files with *cloc*. For the device code, we generated PTX files with the `-keepptx` flag at the compile time with the PGI compiler, then counted their LOC. They are pseudo-assembly files used in NVIDIA's CUDA programming environment. The compiler translates these files into a final binary file for execution on the device. For example, for Level 2, the LOC of PTX-generated code for UVM was 72; however, adding **pointerchain** reduced LOC by 25 (34% reduction). Similarly, for the host, the LOC of UVM was 2740 and utilizing **pointerchain** reduced this by 2 LOC. This metric shows how utilizing **pointerchain** affects code generation. These numbers are interesting for compiler developers who might adapt our approach in their compiler.

c) Total number of instructions executed at execution time: We have measured the total executed instructions on the device and the host (at user and kernel levels). The `nvprof` tool from NVIDIA (with `-m inst_executed` option) counts instructions on the device. For the host, the counting of instructions was performed by the `perf` tool from Linux. Table 2 reveals how **pointerchain** reduces device code as we dropped the required chain of instructions to extract effective address. For instance, at Level 2, the total number of executed instructions for UVM is 14,852, whereas **pointerchain** utilization reduces it by 5688 instructions (38% reduction). On the host, **pointerchain** led to a reduction of 3706 instructions, in comparison to its UVM equivalent. The device-side code definitely benefited from **pointerchain** by a large margin.

6.1.2. Stack usage

pointerchain affects the stack memory on the host by introducing extra local variables to the source code. Local variables in C/C++ translate to an address in the stack memory section of a program. As a result, **pointerchain** directly impacts the stack usage on the host. Introducing more local variables to the code eventually increases stack memory usage of a program. We have measured the peak level of stack usage for the synthetic applications with Valgrind [58]. It tracks the stack usage through the execution time of a program and records a *snapshot* of them. Then, we extract the maximum value from those snapshots. Fig. 6 shows the results for the stack usage in the presence of **pointerchain**. We considered any extra stack allocation on top of UVM's peak value as our stack overhead. The overhead for all four levels is less than 6% (773Bytes). This shows how **pointerchain** leads to low implications on the source code.

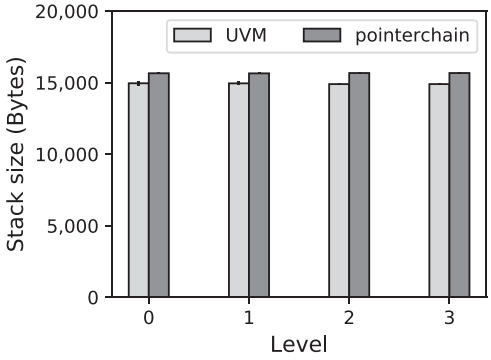

Fig. 6. Stack memory usage with respect to different levels of pointer chains. The results are average of 20 runs and confidence interval of 95%.

6.2. Porting CoMD: performance implications

We ported CoMD to heterogeneous systems using OpenACC and applied the optimization steps, as mentioned in Table 1. We discuss the influence of each step on the final outcome.

6.2.1. Measurement methodology

We relied on the NVIDIA's `nvprof` profiler for device measurement. It provided a minimum, a maximum, and an average execution time, a driver/runtime API calls, and a memory operation for each GPU kernel. It is a handy tool for those who tune an application to achieve maximum performance of GPUs. All simulations were executed with single precisions.

6.2.2. Model preparation

To extract optimal values for `gang` and `vector` parameters, we traversed through a parameter search space for them. We also investigated the effect of manually choosing number of registers at compile time over the performance. Through the rest of this paper, we used the extracted optimal values for `gang`, `vector`, and register count parameters. Please refer to the Supplementary Material for detailed discussion on characterizing the above-mentioned parameters.

6.3. Speedup for each parallelization step

To observe the accumulated effect on the final result, our modifications in each step were implemented on top of its preceding steps unless noted. Please refer to Fig. 4 for the causal effect between each consecutive step.

Fig. 7 illustrates the impact of each step on our program by showing changes in the execution time of the three kernels. We included the results from the CUDA and OpenMP versions. The OpenMP version was compiled with both Intel[1] and PGI[2] compiler, shown as OMP-ICC and OMP-PGI, respectively. Besides targeting OpenACC for NVIDIA GPUs, we also retargeted our OpenACC code for *multicore* systems (ACC-MC in the figures). We did not modify a single line of code when retargeting our code to multicore systems with OpenACC. We changed only the target device from NVIDIA Tesla to multicore at compilation time. Results are shown for both small (bottom) and large (top) data sizes; they are normalized with respect to CUDA.

Enabling UVM on the memory-intensive kernels impedes performance in the first few steps. The reduction in execution time is in several orders of magnitude while proceeding from *Step 1 to 2*. The same trend was observed from *Step 2 to 3* for all three kernels.

[1] Intel Compiler flags: `-Ofast -O3 -xHost -qopenmp`.
[2] PGI Compiler flags: `-mp -fast -O3 -Mipa=fast`.

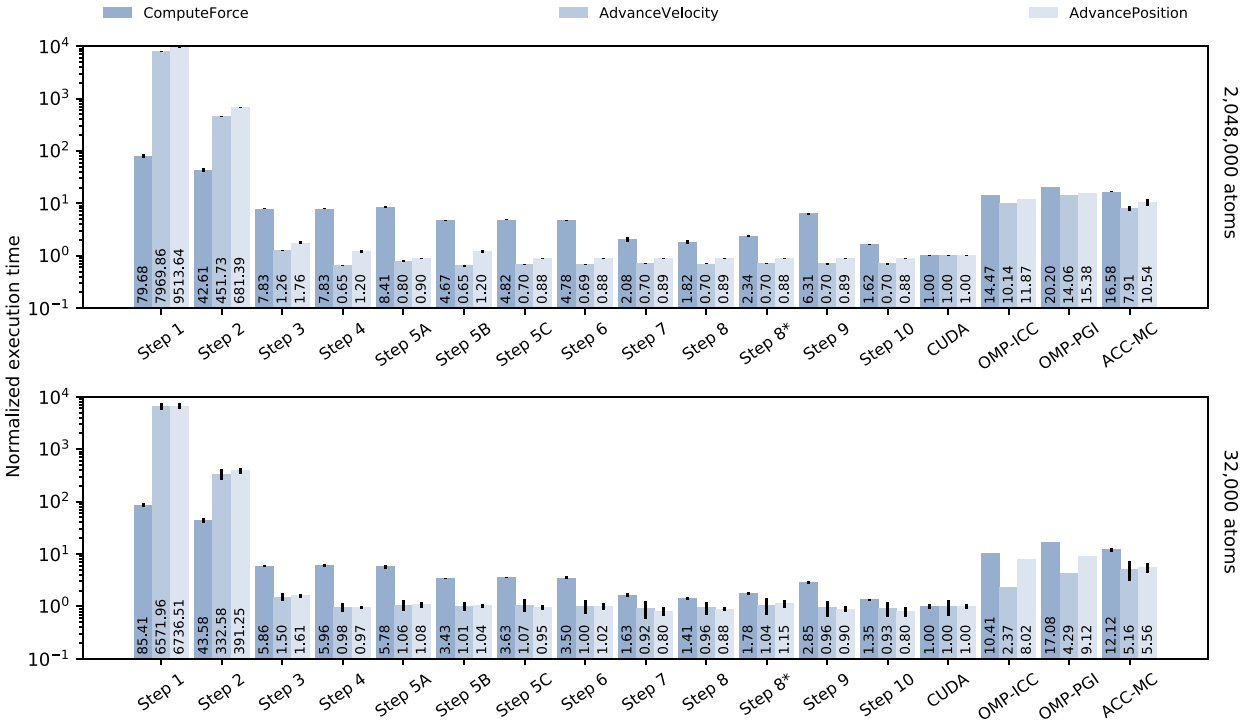

Fig. 7. Normalized execution time after applying all optimization steps and run on NVIDIA P100. After applying all 10 steps on the OpenACC code, we were able to reach 61%, 129%, and 112% of performance of the CUDA kernels for `ComputeForce`, `AdvancePosition`, and `AdvanceVelocity`, respectively. Step 8* is similar to Step 8 but it exploits suggested register count by CUDA Occupancy Calculator for full occupancy of warps in NVIDIA GPUs. Results are normalized with respect to *CUDA*.

Due to developers' insight on data layout and parallelism opportunities, the impact of proposed changes in these steps is significant in comparison to the compiler's insights.

The next significant reduction in execution time happens when data-locality improves by reusing variables (from *Step 5A* to *Step 5B* and *Step 5C*). Such an improvement is due to the reduction in the physical memory accesses by caching such accesses with local variables. To compute exerted force on Atom A, we looped through all atoms in the vicinity and computed the force between them. Therefore, instead of redundantly loading Atom A from memory for each loop iteration, we have loaded it once before the inner loop and reused it within the loop as many times as possible.

Step 7 marks the next substantial reduction in the execution time for our compute-intensive kernel. At *Step 7*, we set the `gang` and `vector` parameters to their optimal values from Supplemental Materials (Section B) and (collected measurements for each kernel. Manually setting these parameters enables the scheduler to issue extra gangs on the device and keep the resources busy at all time (in comparison to the choices by compiler).

Inefficient utilization of resources leads to performance loss. When kernels use registers optimally, we see 16% performance gain from *Step 7 to 8*. Increasing the number of utilized registers for all kernels is not beneficial to the performance. In a sense, *kernels with different traits require different considerations*. As our experiments reveal how memory-intensive kernels do not benefit from a large number of registers, it is better to limit the register count for such kernels. On the other hand, compute-intensive kernels benefit highly from a large number of registers since they minimize the access of global memory for temporary variables.

Elimination of redundant *reduction* operations, as described in *Step 10*, boosted the performance and helped our implementation to reach performance to that of CUDA's. Rearrangement of computations and elimination of unnecessary redundant operations has definitely led to performance gain.

We have discussed ten optimization steps in this paper that for our proxy application, CoMD, boosted the *ComputeForce* kernel's performance by 61–74% in comparison to its counterpart written in CUDA. Although OpenACC did not reach CUDA's efficiency, it got close to its performance with a very small code modification footprint. Additionally, our OpenACC code is portable to another architecture without needing to change any portion of the code; however, a CUDA-based application needs to be updated or revisited every time when the architecture is upgraded, thus affecting maintenance of the code base. The memory-intensive kernels are performing better than their counterparts written in CUDA, as noted from *Step 7* for both small and large data sizes. This improvement is probably due to scheduler-friendly instruction generation by the PGI compiler.

Further investigation of the generated PTX code in either CUDA or OpenACC version reveals why `AdvancePosition` and `AdvanceVelocity` kernels perform better when implemented with OpenACC in comparison to CUDA. In the CUDA version, each thread is only responsible for one single iteration of the loop targeted for parallelism; however, in the OpenACC version, each thread is responsible for multiple iterations of a loop. This shows how parallelization granularity affects performance for different kernel types. The memory-intensive kernels do not benefit from a fine-grained approach since they inherently benefit from the spatial locality of data used in the consequent iterations. Moreover, unnecessarily oversubscribing the CUDA scheduler adversely affects the performance in case of memory-intensive kernels. This is not, however, the case for the compute-intensive kernels.

6.4. Floating-point operations per seconds

We measured the floating-point operations per second (FLOPS) of our kernels under study and compared it with the CUDA implementation for one GPU. FLOPS is one of the most common metrics

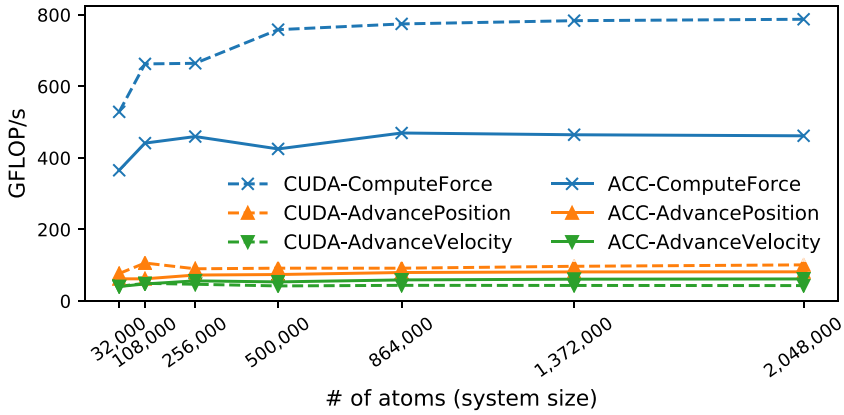

Fig. 8. Giga floating-point operations per second (GFLOP/s). In case of `ComputeForce` kernel, despite comparable speedups with respect to CUDA, the number of floating-points operations that OpenACC implementation executes is behind CUDA's performance; however, other kernels are performing close to CUDA's performance. OpenACC implementation of `AdvanceVelocity` performs better that its CUDA counterpart. Measurements are performed on Nvidia's P100 from PSG. Higher is better.

in scientific domain used to measure the performance of the underlying systems, particularly in MD domain.

There is an increasing gap between the implementations of *ComputeForce* kernel and a decreasing gap for memory-intensive kernels in Fig. 8. For the latter ones, the difference is negligible and in case of *AdvanceVelocity*, the OpenACC version is performing better than CUDA. The case for *ComputeForce* kernel is different, however. As it becomes complicated for the OpenACC compiler to apply necessary optimization techniques on that kernel, the performance gap between the OpenACC and CUDA implementations increases. When developers take advantage of the interoperability feature of OpenACC to run CUDA kernels within OpenACC code, they are allowed to manually tune the bottleneck kernels that do not necessarily benefit from the compiler-generated code; however, this will adversely affect the portability of OpenACC codes.

Fig. 8 shows how the OpenACC version maintains the computation sustainability of the floating-point operations as the number of atoms increases. Similar to the CUDA implementation, the OpenACC implementation does not lose performance as system size increases exponentially.

6.5. Scalability with data size

We investigated the scalability of our OpenACC implementation with respect to varying system sizes. We changed the system size from 32,000 to 2,048,000 atoms and measured the per-atom execution time for five implementations; OpenACC-GPU (*acc-GPU*), OpenACC-Multicore (*acc-MC*), CUDA, Open MP-ICC (*OMP-icc*), OpenMP-PGI (*OMP-pgi*). The results are depicted in Fig. 9 for NVIDIA's PSG cluster. Interestingly, our OpenACC implementation scales with the system size without any performance loss. As discussed in the last section, we experienced better performance with OpenACC than using CUDA for memory-intensive kernels.

Another interesting observation is that there is no significant gap between OpenACC-Multicore and its OpenMP counterparts. In some cases, OpenACC performs better than the Intel optimized OpenMP version for Haswell processors on the PSG platform. In comparison to the generated code for OpenMP by the PGI compiler (OMP-PGI), OpenACC code performs better in the case of the `ComputeForce` kernel.

6.6. Scalability measured at different architectures

Upcoming new architectures have a positive impact on the scalability of systems. Fig. 10 shows the scalability of different architectures; BigRed's K20, UHPC's K80, and PSG's P100 and V100. The gap between CUDA and OpenACC implementations narrows as the underlying architecture evolves positively. Results in this section are based on the utilization of one single GPU. Fig. 10 also shows the speedup of each kernel and each programming model. As we expected, there is no significant improvement between K20 and K80 architectures since both of them are based on the same architecture (Kepler); however, as architectures improve, we observed a boost in performance. The five-fold improvement in memory performance from K80 to P100 is credited with the speedup in memory-intensive kernels.[3]

With respect to the processing power of modern GPUs, as architectures improve progressively, so does their performance. The Kepler architecture performs 6 TFLOPS,[4] while performance of the new generation of GPU processors, which are based on the Volta architecture, has doubled (14 TFLOPS). One can observe how a two-fold improvement in floating-point operations per second has led to an increase of one order of magnitude in performance of a compute-bound kernel, particularly the OpenACC version of `ComputeForce`. In the 500,000-atom case, one is able to observe more than 25X speedup with respect to K20 for the OpenACC version.

6.7. Scalability with multiple GPUs

We have investigated the scalability of our OpenACC implementation for more than one GPU. NVIDIA's Pascal P100 has four GPUs inside the PCI card. For each GPU, an MPI process is initiated and that process takes control of a single GPU. All processes communicate through the MPI library to distribute workload among themselves. The original implementation of CoMD (OpenMP and CUDA) supports MPI. Our contributions in this paper are focused merely on the parallelization within a node. For inter-node parallelization, we rely on the workload distribution provided by the original OpenMP version with MPI.

Results, depicted in Fig. 11, show speedups with respect to 100 timesteps of CoMD with different system sizes. The *ComputeForce* kernel shows promising results for both system sizes. The OpenACC implementation scales better in comparison to its CUDA implementation; The other two memory-intensive kernels do not benefit from multi-GPU scalability of OpenACC code due to the fact that they spend most of their time waiting for memory. Consequently, they do not benefit from the extra computational resources in

[3] From a GDDR5 memory with 240 GB/s in bandwidth in K80 to a HBM2 memory with 732 GB/s in P100.

[4] tera floating-point operations per second.

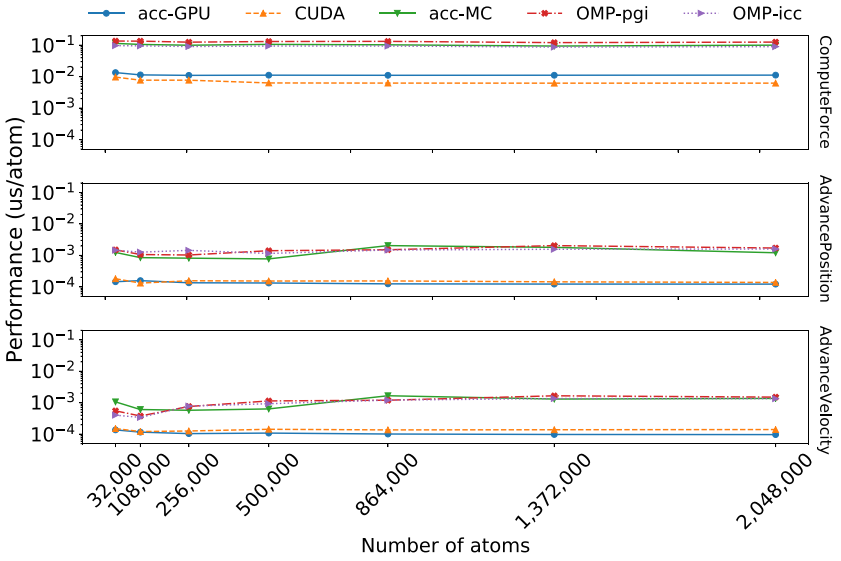

Fig. 9. Scalability with different data sizes with *one* GPU of NVIDIA P100. One can observe that performance is not lost when data size is increased. OpenACC-Multicore performs better in comparison to OpenMP counterparts. Measurements are performed on Nvidia's P100 from PSG.

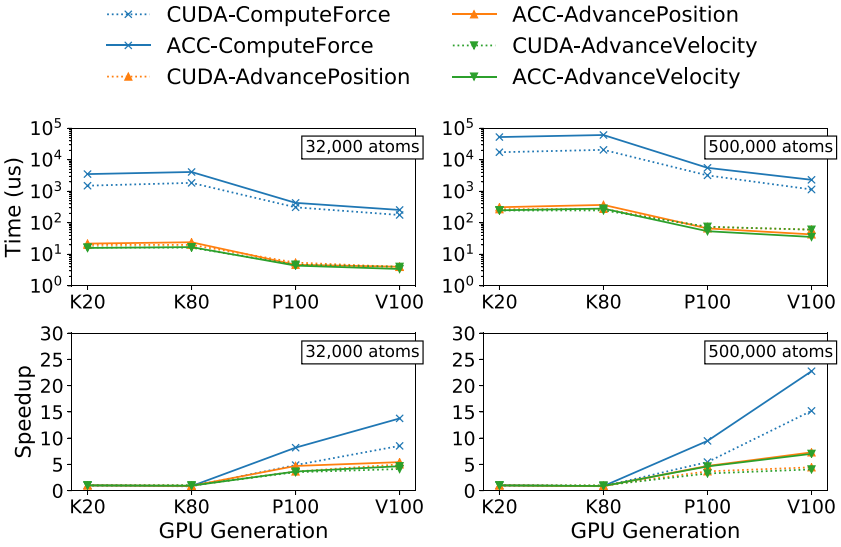

Fig. 10. Scalability with different architectures while utilizing *one single GPU*. With new architectures, performance is improving by shortening time. For time results, a lower reading is better; for speedup results, a higher reading is better.

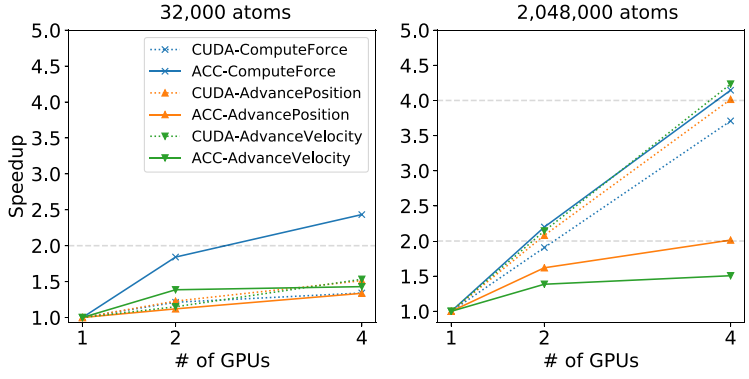

Fig. 11. Scalability of implementations on NVIDIA P100. The `ComputeForce` kernel is performing linearly, and its performance is close to its CUDA counterpart.

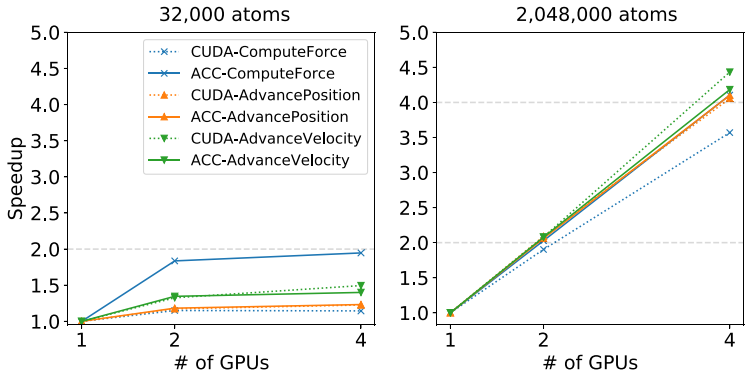

Fig. 12. Scalability of implementations on NVIDIA V100. For 2,048,000-atom system, OpenACC and CUDA scale linearly with number of GPUs. In case of `ComputeForce`, OpenACC shows more scalable performance in comparison to CUDA. `AdvanceVelocity` kernel display a super-linear performance for CUDA.

comparison to our compute-intensive kernel. Such a conclusion, however, is not true for their CUDA counterparts, and they show linear speedup for 2,0480,000 atoms.

Fig. 12 displays results for V100. Similar to its predecessor Pascal P100, Volta V100 also has four GPUs inside the PCI card. All the algorithms show linear (or super-linear) scalability when our system size is large. The scalability of our implementation is comparable to the CUDA's, and in the case of the `ComputeForce` kernel, OpenACC performs better. When our system size is not large enough, OpenACC's scalability of the `ComputeForce` kernel is 59% and 70% better for two and four GPUs, respectively. In the case of the other two kernels, CUDA and OpenACC's scalability are similar.

Fig. 11 shows the super-linear scalability for the three kernels with 2,048,000 atoms. The OpenACC's *ComputeForce* kernel is super-linear due to the utilization of a cut-off range within the algorithm, which leads to skip atoms in far distance. Skipping such atoms leads to skipping some iterations of the main loop, which in turn helps the kernels to skip unnecessary computations and reach super-linearity. On the other hand, the efficient cache utilization of CUDA's *AdvancePosition* and *AdvanceVelocity* kernels has led to a super-linear speedup in performance. Fig. 12 depicts similar results on V100 architecture. Due to improvements in cache performance of V100 architecture in comparison to P100,[5] the two CUDA kernels that were underutilized on P100 show linear performance. Figs. 11 and 12 show how CoMD shows sub-linear speedups for 32,000 atoms for all three kernels due to high overhead of workload distribution. When our system size is small, CoMD does not benefit from the multi-device distribution. However, as we increase the system size, we notice an explicit improvement in the speedup of the kernels.

6.8. Effects on the source code

OpenACC does not impose a significant impact on source code size and maintenance; thus, it retains the integrity of a complex scientific application. Similar to OpenMP, developers are not required to write excessive lines of code to maintain the state of the application and accelerators. As a result, we exploited *lines of code (LOC)* to quantitatively measure the code complexity. The measurement was performed with the *cloc* [57] tool. Table 3 presents the results for the LOC for each step. We used reference implementation of CoMD (the OpenMP version) as the starting point for our porting process to OpenACC. The LOC column shows that the total extra lines of code required to implement that step with respect to OpenMP implementation as the base version. The third column (%) shows the percentage with respect to the base version. CUDA implementation doubles the code size in comparison to the OpenMP version; however, for OpenACC, LOC is less than 8%. Results in Table 3 include the LOC from *Step 2 to 10* with extra `pointerchain` lines. In some transitions from one step to the next (e.g., *Step 7 to 8*), there is no difference in LOC. That is, we changed only the compilation flags, which naturally does not count towards the LOC count.

Table 3
Effect of the OpenACC adaption on the source code – lines of code (LOC) column shows extra line required to implement this step with respect to the OpenMP implementation as the base version. The third column (%) shows the increase with respect to the base version.

Step	LOC	%	Step	LOC	%
OpenMP	3025	–	Step 5C	+165	5.45
Step 1	+2	0.07	Step 6	+163	5.39
Step 2	+99	3.27	Step 7	+198	6.55
Step 3	+103	3.4	Step 8	+198	6.55
Step 4	+109	3.6	Step 9	+187	6.18
Step 5A	+109	3.6	Step 10	+215	7.11
Step 5B	+125	4.13	CUDA	+4745	**1.57X**

7. Related work

The other directive-based programming model focusing on targeting GPUs is OpenMP [29]. In its early stages, OpenMP 3.1 primarily supported only shared memory processors. With processors becoming increasingly heterogeneous, OpenMP has extended its support for such systems. OpenMP 4.5 has also introduced routines to *associate/disassociate* device pointers with their counterparts on the host, which is essential for deep copy implementation. Similarly, OpenACC has also introduced *attach/detach* to their API to assist developers in assigning correct pointers on the host. Despite having such functionalities, deep copy has not been fully implemented by the directive-based programming models.

Recent efforts in C++ performance portability libraries, such as RAJA [59], Kokkos [60], StarPU [61], and SkePU [62] provide facilities that make the applications less susceptible to underlying hardware changes. They allow developers to write applications such that applications can be recompiled, with minimal code changes, for different devices. Both libraries rely heavily on the C++ abstractions to address portability. In Kokkos, switching between Arrays of structures (AoS) and Structures of Arrays (SoA) data layout is as easy as changing a template parameter in the source code. Similarly, SkePU ensures the data consistency with the utilization of the concept of Smart Pointers in the code, without relying on a complete replication of data between CPUs and GPUs. Nevertheless,

[5] The L2 cache size has increased from 4 MB in the P100 to 6 MB in V100.

utilizing such libraries requires major modifications and changes to be applied to the source codes to support then full features that they provide.

Other efforts for CPU-specific approaches, like Cilk [63] and TBB [64] from Intel, employ abstract yet low-level programming interfaces to parallelize the code with tasks and threads on multicore architectures only. Other library-based approaches and APIs have also been introduced for GPUs that simplify GPU programming for developers; for instance, Thrust [65] and ArrayFire [66]. Although the aforementioned library-based approaches are convenient, they are not flexible for general-purpose development and need, to some extent, major modifications to current existing codes.

8. Conclusion

This contribution proposes a novel high-level directive, *pointerchain*, to reduce the burden of data transfer in a scientific application that executes on the HPC systems. We have employed a source-to-source transformation script to translate the `pointerchain` directive to conformed statements in C/C++ language. We observed that using the `pointerchain` directive leads to 36% reduction in both generated and executed instructions on the GPU devices. We evaluated our directive using CoMD, an MD proxy application. By exploiting OpenACC directives on the CoMD code, OpenACC code outperforms CUDA on two out of three kernels while it achieves 61% of the CUDA performance the third kernel. We showed a linear scalability with growing system sizes with OpenACC. We provided a step-by-step approach readily applicable to any other application. As part of our near future work we will extend our implementation to support multi-node execution.

Acknowledgments

This material is based upon work supported by the NSF Grant Nos. 1531814, 1412532 and DOE Grant No. DE-SC0016501. This research was also supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. We are also very grateful to NVIDIA for providing us access to their PSG cluster and thankful to the OpenACC technical team especially Mat Colgrove and Pat Brooks.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.parco.2019.04.007.

References

[1] ORNL's Summit, 2018, (https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/summit-faqs/), Accessed: 2018-04-10.
[2] D. Unat, Trends in data locality abstractions for HPC systems, IEEE Trans. Parallel Distrib. Syst. 28 (10) (2017) 3007–3020.
[3] OpenACC Standard Committee, 2016, (Technical Report TR-16-1), Accessed: 2017-12-03.
[4] J.R. Perilla, et al., Molecular dynamics simulations of large macromolecular complexes, Curr. Opin. Struct. Biol. 31 (2015) 64–74.
[5] G. Giupponi, M. Harvey, G.D. Fabritiis, The impact of accelerator processors for high-throughput molecular modeling and simulation, Drug Discov. Today 13 (23) (2008) 1052–1058.
[6] H. Zhang, et al., HIV-1 Capsid function is regulated by dynamics: quantitative atomic-Resolution insights by integrating magic-Angle-Spinning NMR, QM/MM, and MD, J. Am. Chem. Soc. 138 (42) (2016) 14066–14075.
[7] J.E. Stone, D.J. Hardy, I.S. Ufimtsev, K. Schulten, GPU-accelerated molecular modeling coming of age, J. Mol. Graphics Modell. 29 (2) (2010) 116–125.
[8] R. Friedman, K. Boye, K. Flatmark, Molecular modelling and simulations in cancer research, Biochimica et Biophysica Acta (BBA) - Rev.Cancer 1836 (1) (2013) 1–14.
[9] H. Zhao, A. Caflisch, Molecular dynamics in drug design, Eur. J. Med. Chem. 91 (2015) 4–14.
[10] M. Feig, I. Yu, P.-h. Wang, G. Nawrocki, Y. Sugita, Crowding in cellular environments at an atomistic level from computer simulations, J. Phys. Chem. B 121 (34) (2017) 8009–8025.
[11] A. Singharoy, C. Chipot, Methodology for the simulation of molecular motors at different scales, J. Phys. Chem. B 121 (15) (2017) 3502–3514.
[12] D.A. Pearlman, et al., AMBER, A package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules, Comput. Phys. Commun. 91 (1–3) (1995) 1–41.
[13] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys. 117 (1) (1995) 1–19.
[14] E. Lindahl, B. Hess, D. Van Der Spoel, GROMACS 3.0: a package for molecular simulation and trajectory analysis, J. Mol. Model. 7 (8) (2001) 306–317.
[15] J.C. Phillips, et al., Scalable molecular dynamics with NAMD, J. Comput. Chem. 26 (16) (2005) 1781–1802.
[16] NVIDIA Corporation, CUDA C Programming Guide, 2018.
[17] The Khronos Group Inc, The OpenCL specification (2008).
[18] S. Páll, et al., Tackling exascale software challenges in molecular dynamics simulations with gromacs, in: Solving Software Challenges for Exascale, Springer International Publishing, Cham, 2015, pp. 3–27.
[19] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, D. Glasco, GPUs and the future of parallel computing, IEEE Micro 31 (5) (2011) 7–17.
[20] R. Lucas, et al., DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges, Technical Report, US-DOE Office of Science, United States, 2014.
[21] J. Vetter, et al., Advanced Scientific Computing Research Exascale Requirements Review, Technical Report, Argonne National Lab. (ANL), Argonne, IL, United States, 2017.
[22] J. Anderson, A. Keys, C. Phillips, T. Dac Nguyen, S. Glotzer, Hoomd-blue, general-purpose many-body dynamics on the gpu, APS Meeting Abstracts, 2010.
[23] A. Gupta, S. Chempath, M.J. Sanborn, L.A. Clark, R.Q. Snurr, Object-oriented programming paradigms for molecular modeling, Mol. Simul. 29 (1) (2003) 29–46.
[24] K. Refson, Moldy: a portable molecular dynamics simulation program for serial and parallel computers, Comput. Phys. Commun. 126 (3) (2000) 310–329.
[25] W.R. Saunders, J. Grant, E.H. Mller, A domain specific language for performance portable molecular dynamics algorithms, Comput. Phys. Commun. 224 (2018) 119–135.
[26] S. Wienke, C. Terboven, J.C. Beyer, M.S. Müller, A pattern-based comparison of OpenACC and OpenMP for accelerator computing, in: Euro-Par Parallel Processing, Springer International Publishing, 2014, pp. 812–823.
[27] S. Lee, J.S. Vetter, Early evaluation of directive-based gpu programming models for productive exascale computing, in: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for, IEEE, 2012, pp. 1–11.
[28] M.G. Lopez, et al., Evaluation of directive-based performance portable programming models, Int. J. Signal Imaging Syst. Eng. (2017).
[29] OpenMP Language Committee, OpenMP Application Programming Interface, Version 3.1, 2011, (http://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf).
[30] OpenACC Language Committee, OpenACC Application Programming Interface, Version 2.6, 2017, (https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf).
[31] W.M. Brown, J.-M.Y. Carrillo, N. Gavhane, F.M. Thakkar, S.J. Plimpton, Optimizing legacy molecular dynamics software with directive-based offload, Comput. Phys. Commun. 195 (2015) 95–101.
[32] K.B. Tarmyshov, F. Mller-Plathe, Parallelizing a molecular dynamics algorithm on a multiprocessor workstation using OpenMp, J. Chem. Inf.Model. 45 (6) (2005) 1943–1952.
[33] H.M. Aktulga, et al., Optimizing the performance of reactive molecular dynamics simulations for many-core architectures, Int. J. High Perform. Comput. Appl. (IJHPCA) (2018).
[34] B.P. Pickering, C.W. Jackson, T.R. Scogland, W.-C. Feng, C.J. Roy, Directive-based GPU programming for computational fluid dynamics, Comput. Fluids 114 (2015) 242–253.
[35] O. Hernandez, W. Ding, B. Chapman, C. Kartsaklis, R. Sankaran, R. Graham, Experiences with High-level Programming Directives for Porting Applications to Gpus, in: Facing the Multicore-Challenge II, Springer, 2012, pp. 96–107.
[36] K. Puri, V. Singh, S. Frankel, Evaluation of a directive-based GPU programming approach for high-order unstructured mesh computational fluid dynamics, in: Proceedings of the Platform for Advanced Scientific Computing Conference, in: PASC '17, ACM, New York, NY, USA, 2017, pp. 4:1–4:9.
[37] L.G. Szafaryn, et al., Trellis: portability across architectures with a high-level framework, J. Parallel Distrib. Comput. 73 (10) (2013) 1400–1413.
[38] J.A. Herdman, et al., Achieving portability and performance through openacc, in: First Workshop on Accelerator Programming using Directives, 2014, pp. 19–26.
[39] M. Ghane, S. Chandrasekaran, M.S. Cheung, Gecko: Hierarchical Distributed View of Heterogeneous Shared Memory Architectures, in: Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, ACM, Washington, DC, USA, 2019, pp. 21–30.
[40] M. Ghane, S. Chandrasekaran, R. Searles, M.S. Cheung, O. Hernandez, Path forward for softwarization to tackle evolving hardware, in: Proceedings of SPIE - The International Society for Optical Engineering, 10652, 2018.

[41] S. Wienke, P. Springer, C. Terboven, D. an Mey, OpenACC – First experiences with real-world applications, in: Euro-Par Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 859–870.
[42] CoMD Proxy Application, 2018, (https://github.com/ECP-copa/CoMD), Accessed: 2018-04-02.
[43] R. Landaverde, T. Zhang, A.K. Coskun, M. Herbordt, An investigation of unified memory access performance in cuda, in: IEEE High Performance Extreme Computing Conference (HPEC), 2014, pp. 1–6.
[44] COPA: Codesign Center for Particle Applications, 2018, (*Exascale Computing Project (ECP)*).
[45] I. Karlin, A. Bhatele, J. Keasler, B.L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, et al., Exploring traditional and emerging parallel programming models using a proxy application, in: IPDPS, 2013, pp. 919–932.
[46] O. Villa, D.R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, et al., Scaling the power wall: a path to exascale, in: High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for, IEEE, 2014, pp. 830–841.
[47] J. Mohd-Yusof, N. Sakharnykh, Optimizing CoMD: a molecular dynamics proxy application study, in: GPU Technology Conference (GTC), 2014.
[48] O. Pearce, et al., Enabling work migration in CoMD to study dynamic load imbalance solutions, in: Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, in: PMBS '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 98–107.
[49] L. Verlet, Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules, Phys. Rev. 159 (1967) 98–103.
[50] J.E. Jones, On the determination of molecular fields. II. From the equation of state of a gas, in: Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 106, The Royal Society, 1924, pp. 463–477.
[51] MPI Forum, MPI: A Message-Passing Interface Standard. Version 2.2, 2009, Available at: http://www.mpi-forum.org (Dec. 2009).
[52] P. Cicotti, S.M. Mniszewski, L. Carrington, An evaluation of threaded models for a classical md proxy application, in: Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014, IEEE, 2014, pp. 41–48.
[53] 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, R.W. Numrich, Improving Performance via Mini-applications, Technical Report SAND2009-5574, Sandia National Laboratories, 2009.
[54] Big Red II at Indiana University, 2017, (https://kb.iu.edu/d/bcqt), Accessed: 2017-12-03.
[55] UHPC, 2017, (https://uhpc-mri.uh.edu/), Accessed: 2017-12-03.
[56] NVIDIA PSG, 2017, (http://psgcluster.nvidia.com/trac), Accessed: 2017-12-03.
[57] cloc, (https://github.com/AlDanial/cloc), Accessed: 2018-04-10.
[58] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI '07, ACM, New York, NY, USA, 2007, pp. 89–100.
[59] R.D. Hornung, J.A. Keasler, The RAJA Poratability Layer: Overview and Status, Technical Report, Lawrence Livermore National Laboratory (LLNL-TR-661403), 2014.
[60] H. Carter Edwards and Christian R. Trott and Daniel Sunderland, Kokkos: enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing
[61] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: aunified platform for task scheduling on heterogeneous multicore architectures, in: Concurrency and Computation: Practice and Experience, 23, 2011, pp. 187–198. Special Issue: Euro-Par 2009
[62] J. Enmyren, C.W. Kessler, Skepu: a multi-backend skeleton programming library for multi-gpu systems, in: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications, in: HLPP '10, ACM, New York, NY, USA, 2010, pp. 5–14.
[63] R.D. Blumofe, et al., Cilk: an efficient multithreaded runtime system, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, in: PPOPP '95, ACM, New York, NY, USA, 1995, pp. 207–216.
[64] ReindersJames, Intel Threading Building Blocks, 1st, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.
[65] NVidia, Thrust, 2018, (https://developer.nvidia.com/thrust).
[66] P. Yalamanchili, et al., ArrayFire - a high performance software library for parallel computing with an easy-to-use API, 2015.

Further reading

L.G. Szafaryn, T. Gamblin, B.R. de Supinski, K. Skadron, Experiences with achieving portability across heterogeneous architectures, in: Proceedings of WOLFHPC, in Conjunction with ICS, Tucson, 2011.