Parallelization of Push-based System for Molecular Simulation Data Analysis with GPU

Iliiazbek Akhmedov, Yi-Cheng Tu †, Vladimir Grupcev, Joseph Fogarty, Sagar Pandit

***Abstract*—Modern simulation systems generate big amount of data, which consequently has to be analyzed in a timely fashion. Traditional database management systems follow principle of pulling the needed data, processing it, and then returning the results. This approach is then optimized by means of caching, storing in different structures, or doing some sacrifices on precision of the results to make it faster. When it comes to the point of doing various queries that require analysis of the whole data, this design has the following disadvantages: considerable overhead on random I/O while reading from the simulation output files and low throughput of the data that consequently results in long latency, and, if there was any indexing to optimize selections, overhead of storing those becomes too big, too.**

There is a new approach to this problem presented in the previous paper – Push-based System for Molecular Simulation Data Analysis for processing network of queries proposed in the previous paper and its primary steps are: i) it uses traditional scan-based I/O framework to load the data from files to the main memory and then ii) the data is pushed through a network of queries which consequently filter the data and collect all the needed information which increases efficiency and data throughput. It has a considerable advantage in analysis of molecular simulation data, because it normally involves all the data sets to be processed by the queries.

In this paper, we propose improved version of Push-based System for Molecular Simulation Data Analysis. Its major difference with the previous design is usage of GPU for the actual processing part of the data flow. Using the same scan-based I/O framework the data is pushed through the network of queries which are processed by GPU, and due to the nature of science simulation data, this gives a big advantage for processing it faster and easier. In the old approach there were some custom data structures such as quad-tree for calculation of histograms to make the processing faster and those involved loss of data and some expectations from the data nature, too. In the new approach due to high performance of GPU processing and its nature, custom data structures were not even needed much though it didn't bear any loss in precision and performance.

***Index Terms*—Push-based system, molecular simulation, scientific databases, spatial distance histogram, GPU, parallel processing, CUDA.**

I. Introduction

In various sciences simulation systems take big place and often times they may be the clue for results. One of such sciences, which is primarily related to this paper, is physics. In this case, from computer engineering point of view, simulations have the following flow:

1) initial physical properties are given to simulation software as arguments which are interpreted and provided in a language defined by the simulation software
2) then it runs for given amount of time that can generally last from milliseconds to months or even more
3) finally, the file generated during the simulation is analyzed to extract useful information

The purpose of this paper is primarily focused on the third step, which basically involves the entire simulation data to be processed by the analysis software.

Big data processing is becoming one of the key issues with the amount of data being generated by modern systems. Eventually, this data is required to be processed on the fly, thus, analysis software should be able to handle massive data in a very short period of time. When working with huge volume of data, there are non-trivial issues arise. For example, it can take days and weeks to analyze big enough data sets, because the data cannot be simply loaded into memory, since it can get up to terabytes, thus, it has extra overhead because of widely used random access disk I/O framework to read from disk chunks by chunks. Besides it, analysis of the data can get even more complicated, since going through certain parts of the data once would not be enough, which leads to low throughput and efficiency of loaded data. One such example, the data analysis approach has polynomial complexity just for reading the data in order to come up with result, and since the data can't simply be saved in memory, it raises the overhead of disk I/O, too. Pull-based architectures in data processing engines are inefficient, since having a set of specific queries, in order to compute them all, it is needed to fetch data, filter it, and apply needed formulas. It is inefficient, since for every query the same chunk of data needs to be pulled into the memory at least the number of queries times or more in case of more complex queries.

As it has already been mentioned in the previous paper [1], one of the modern issues of analyzing massive data on the fly is social networks. . "In order for a system to be able to perform analytical examination of the data produced in such streaming media, the system should have the capability of fast data access. The reason, the millions of data records(tweets) produced every second. Moreover, these tweets may have different geographical origin, introducing different languages and forms and often times containing unsolicited messages, errors, malicious content, etc. Therefore, some low level data uniformity and cleaning on top of the data access and man-

† Author to whom all correspondence should be sent.

Iliiazbek Akhmedov, Vladimir Grupcev, and Yi-Cheng Tu are with the Department of Computer Science and Engineering, University of South Florida, 4202 E. Fowler Ave., ENB 118, Tampa, FL 33620, U.S.A. Emails: akhmedovi@mail.usf.edu, ytu@cse.usf.edu

Joseph Fogarty and Sagar Pandit are with the Department of Physics, University of South Florida, 4202 E. Fowler Ave., ISA 2019, Tampa, FL 33620, U.S.A. Emails: jcfogart@mail.usf.edu, pandit@usf.edu

agement issues should be considered and possibly incorporated in the process of analytical investigation in order to achieve relevant result. ” [1]–[5]

The primary focus and problem in this paper is scientific data analysis. Particles simulation is one of the most popular methods of analyzing certain chemical reactions, physical processes, or other behavior of different materials. Molecular simulations (Molecular Dynamics) are applied in different fields and represent a method of analyzing physical movements of particles, atoms, and molecules in a fixed space with a given period of time, apparently with a possibility of giving initial state for each item that is involved in the process and can affect the system. This system is an N-body simulation. The number of atoms in simulations vary in hundred of thousands, particularly, we may observe two simulation systems of a collagen fiber structure and dipalmitoylphosphatidylcholine (DPPC) bi-layer lipid system consisting of 890,000 and 402,400 atoms respectively on Figure 1. Simulation data represents number of records of physical properties such as mass, charge, velocity, coordinates, and forces for each item aggregated as frames, where each frame represents a snapshot of time, placed with a fixed time interval which may also vary depending on the simulation itself and simulation precision requirement. ”Quantities measured during the simulations are analyzed to test the theoretical model [6], [7]. In short, the MS is proven and powerful tool for understanding the inner-workings of a biological system, by supplying a model description of the biophysical and biochemical processes that are being unfold at a nanoscopic scale.” [1]

Scientist gives the properties to simulation software (for example, Gromacs), runs the simulations, and finally get the output file. The output file must then be analyzed to produce certain results which may help him come up with certain consensus on original theoretical model that resulted in the molecular simulation system [6]. Gromacs is simulation software tool that helps scientist to run the actual simulation. It is a molecular dynamics package primarily designed for biomolecular systems such as proteins and lipids. [8]. Besides the fact that it helps to generate the output files for the simulations, apparently it also helps to analyze the data itself, but the original problem is that it is not as optimized as it can be in order to analyze the data. Gromacs follows approach of pull-based design, which means that for any given query (e.g. total mass or total charge, which are very similar type of 1-body queries without sophisticated selection) it will pull data separately and generate addition overhead wasting disk read I/O in order to come up with the result just for a single query. As it has been proposed in the previous paper, in order to remedy such issues, the push-based design does exactly the opposite, where instead of loading the data on demand for each query, the queries are batched into a network, then the entire dataset is loaded chunk by chunk pushing it through the network which has its internal relationships and dependencies amonth the queries. In this case, since scientific simulation data is run once with specific physical properties, it is never modified, thus, on continuation, it will only append, which means that the processing can also be run on appended frames. [1] This type of approach has already been revised by other systems [9]–[11] as of reusing loaded data through queries produced [12]–[14].

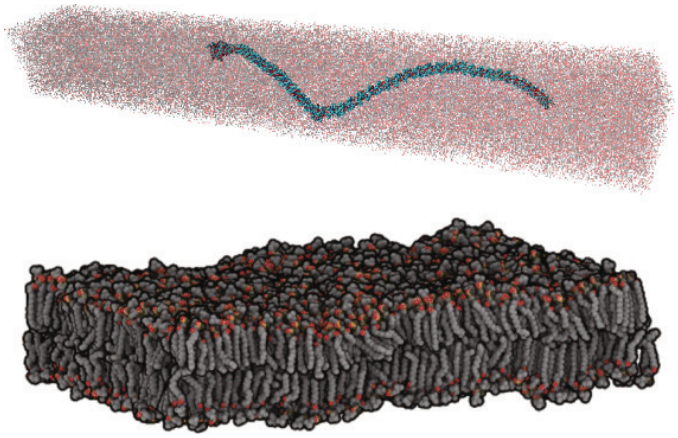

Fig. 1: Snapshots of two MS systems: a collagen fiber structure with 890,000 atoms (top) and a dipalmitoylphosphatidylcholine (DPPC) bi-layer lipid system with 402,400 atoms (bottom) [15]

In this paper, we are incorporating parallelization into the processing part of the proposed design by means of CUDA programming language for GPU. Since storage of the simulation data is very expensive, it might come to the point of analyzing the data on the fly (meaning running the simulation and analyzing it at the same time in a streaming manner), which leads to a problem of optimizing the processing part of the design proposed in the previous paper, because time spent on generating data should be tried to conceal the time spent on the processing part by means of overlapping or simply running it in a quick manner.

A. Problem Statement

Simulation software systems, in general, follow the same methodology of running and storing simulation data. The simulation software system examples are: Gromacs [16], VMD [17], MDAnalysis [18], Wordom [19], MD-TRACKS [20], SimulaidOne [21], Charmm [22]. In the type of simulations brought up as examples above, the flow of the data is the following. Once the simulation is run, the output files are contained as trajectory files with descriptors (they contain information about space dimensions, number of atoms and frames, etc.) that can be easily transposed into simple flat files containing the physical properties atom by atom, frame by frame, which are consequently read and processed by the proposed push-based system. Since we have certain amount of queries needed to be run on given simulation data, generating high I/O traffic followed by design of pull-based system is not considerable. The approach proposed in the previous paper is very good in terms of performance in comparison with the original pull-based system [1]. The problem is still that some of the queries processed by those means are still improvable, especially taking into consideration the fact that in the used previous works for calculation of 2-body functions, which take the biggest time for processing [23], [24], we might have some error bounds, which might be unacceptable

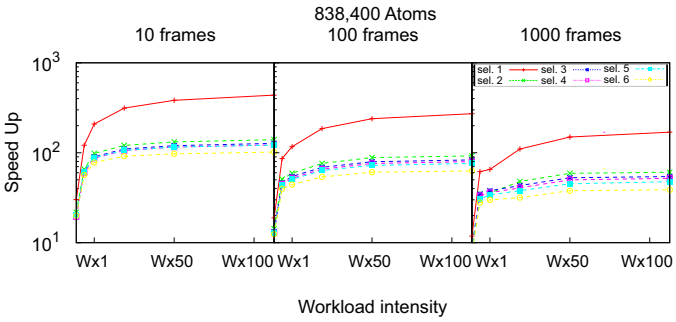

Fig. 2: Speed up over different levels of atom selection for 838,400 atoms [1]

for certain simulation analysis where sacrifice on loss of data is unbearable. Besides it, although we might have huge performance boost on specific data structure as density map, we still have certain expectation from data nature (such as its uniformity), thus, it makes sense to have desire to simplify the processing and still having pure computation based on any values, considering that the queries should be available to be pre-programmed by user in a separate module of the code.

B. Our approach with improvement

Reading the data from files is a huge overhead, thus, speedup of push-based system in general is significant over pull-based system in our given case. It's been demonstrated in the previous paper with different amount of atoms, frames, and workloads. Since we have to load the data every time there is a different query versus push-based system loads a chunk of data once, it is quite noticeable how the framework contributes to efficiency of analysis of simulation data. You can observe some estimation samples in Figure 2. Even though the memory loading and pushing steps are the key features of the proposed design, nevertheless, having general knowledge about the network of queries, in this paper, we will try to focus on optimizing the actual execution of them.

Some of the queries may be very slow due to their nature on a sequential type of computation, especially, 2-body functions. Since the nature of the data that comes with simulation is basically physical properties of atoms, there is a lot of computation that involves independent primitive mathematical operations, which makes it a perfect problem for parallelism.

Although the original idea of push-based system for molecular simulation data analysis was focused on optimizing throughput and usage of loaded data, there were some extra approaches specifically for 2-body functions. For example, Density Map for Spacial Distance Histogram was used in order to avoid additional memory allocation and latency reduction with proven error bounds. SDH is a quite intensive and computational problem, especially with increasing amount of atoms.

We believe that having this nature of computational problems, the proposed GPU improved version of push-based system will significantly change in terms of performance by incorporating parallelism with CUDA.

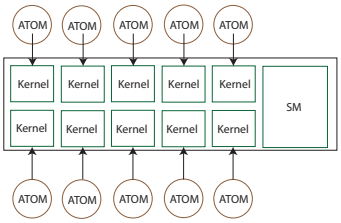

Fig. 3: Example of atom mapping with kernels

C. Contribution and roadmap of the paper

The proposed improved version of push-based system for molecular simulation data analysis, we believe, gives an opportunity for scientists to run their analysis even faster now. Taking an advantage of GPU devices nature and incorporating parallelism and streaming for different types of queries, we have come up with a good speed up over sequential processing, which already had a good speed up in terms of performance in comparison with original pull-based approach. Since in this paper we don't expect certain form of data, we have come up with a tool that generates mock data for any number of atoms and frames, which can be used for performance tests that were done consequently, too. This mock data has exactly the same information that would come with Gromacs simulation files, and it has exactly the same format as in the previous paper [1] right before loading it to memory. This improved version has been developed on Amazon Elastic GPU [25] nodes that consequently can be replicated and scaled up becoming more of a streaming distributed processing engine (which can be very effective on costs, since it is always easy to scale them up, shut them down and spin them back up), as well as on a simple desktop computer with a GPU device. In general, major contribution of this paper is the framework of push-based system with an incorporated parallelism based on CUDA programming language. Besides it, there was developed a tool which generates mockup data of the same format as of the real MS simulations and represent a python script with arguments given as number of frames and atoms.

The structure of the framework which will be described later is developed in such a way that one can easily add new queries based on their complexity or modify existing ones adding appropriate selections and filters.

II. Related Work

Push-based system for MS data analysis is primarily proposing an idea of pushing chunks of data through a network of queries. Essentially, this has also been proposed, as mentioned already, by other frameworks [9]–[11]. In other words, the major point is to use common data loaded into memory for execution of concurrent queries, which is supposed to eliminate I/O overhead predominantly. The architecture and explanation of DSMS (Data Stream Management System) versus DBMS is well explained in one of the lectures of Morgan and Claypool [26]. It is very important to capture the differences and specialties of DSMS in network architecture, stream models and windows, scheduling, load balancing, approximation, data

expiration, etc, because of the fact that data flow design is the key that required all of those changes to common designs.

Since dealing with data is very delicate in this kind of engines, there are couple of features and requirements that this architecture is applicable to, in order to avoid inappropriate understanding or application of this approach in systems that were covered in the lectures [26]. The data is volatile and not persistent anymore because of obvious sizes of data. Since we are pulling the data as little times as possible taking into consideration that MS data is only appended and not modified, the access is sequential rather than random. There are concepts of framing and windowing, and memory is limited in streaming engine, while in DBMS secondary storage is considered to be unlimited. Obviously, because of all of these features or requirements to data, the streaming engine frameworks have some limitations like going back into the history of data, since it is volatile. In fact, this has also been tried by Borealis Streaming Engine [27]. On top of the fact that it is a streaming engine, as it was built based on Medusa [28] and Aurora [29], it proposes distributed processing at this point having some features to handle errors and look back in the history. In our case, it is primarily a single data source, since it is based on the simulation, though in the future the simulation software systems might introduce distributed computing, too. It is also important to mention such projects as PeriScope [30], SciDB [31], and other [32], [33], since they target scientific data, which is a lot more different, than usual industry customer oriented data.

This version of Push-based System of MS data analysis is primarily about taking an advantage of GPU processing of the data. Hadoop is a well-known and well-used framework by many companies nowadays [34]. Its purpose is an ability to store and serve large scale data across clusters in a timely fashion. Since usage of GPU of massive data (not only related to graphics processing) is a relatively new concept, there is also an improved version of Hadoop MapReduce Framework with GPU [35]. Having an ability to scale it up in a distributed computing environment is probably one of the best approaches, but again it might be a bigger overhead in terms of costs, since keeping all nodes up and still being able to solve specific problem sets with hardware stack of a lower capacity raise a will to explore more. Having a specific software for simulation like Gromacs, apparently there is also a possibility of taking an advantage of GPUs. For example, Gromacs allows to optimize simulation and query tools by adding configurations based on GPU device located on the processing computer. Unfortunately, even with this feature, Gromacs doesn't follow push-based approach which means that improvement with GPU with pull data per query makes it negligible.

Gromacs generate output files with physical properties description for each atom in each frame. Modern CPUs do great job in terms of caching and computing, but obviously in this particular problem massive monolithic computation is needed, thus, GPU devices are perfect candidates for such a problem following SIMD type of computation. To be more exact, for example, a single kernel in a GPU device could be dedicated to a single atom (this relationship might change depending on the query, but this is to understand the scale and relationship of multiprocessors), and since we have scientific data of MS, the data is appended, which means that we will only move forward and not take an advantage of caching on chunks high level (though we might take an advantage of GPU caching features particularly in processing phase) having multiple workloads per chunk. Just to make it more clear, workloads in this particular case represent transactions based on number of clients. For example, if we were to serve it all in a streaming fashion, 100 clients might demand concurrent queries with different slight selection properties and expect their results, each workload might be dedicated to each client if not aggregated. Thus, the performance of GPU is certainly increasing based on workload too, which might benefit considerably.

III. Network of Queries

Having generated big amount of data in large files, now scientists need to get useful information out of it by means of analysis utilities. Often times, the queries that need to be run over the data look like selection and some kind of accumulation, if it is not more complicated. To be more exact, in this section we will try to summarize common set of queries we will be improving.

In previous work, there has already been developed a network of queries widely used by scientists for MS systems. In Table I you may observe the common set of queries that has been developed. Basically, these are the queries that need to be improved with our new approach of parallelism. Just to mention, n, r_i, m_i, c_i and q_i denote number of particles, coordinates (vector form), mass, charge, and number of electrons of a particle i.

In general, with the given queries we have divided them into two categories: one body functions and multiple body functions. The first ones apparently have linear complexity of $O(n)$, while the other ones have bigger complexity.

One body functions such as sum of masses, center of mass, or simple counting with selection are of a complexity $O(n)$, thus, in order to get the results for them going through every atom (essentially over the entire data) only once is enough, and taking into consideration the fact that we are going with push-based system, it is done at the same time for all of them.

Multiple body functions such as SDH and RDF (Spacial Distance Histogram and Radial Distribution function) require computation of distances pairwise across all the particles. This is generally combination of two across N data: $C\binom{N}{2}$. These are the most expensive queries, thus, it is important to focus on their implementation taking advantage of GPU nature and its processing power. SDH is a very expensive computation, and we have used some of the existing work of ours [36] to incorporate in this design.

The might be some dependencies and preliminary computation possibilities. For example, some queries need total mass, and we could compute it before we push it further, but this can be easily done by user based on the complexity and dependency. It is explained in more details in further sections.

Function Name	Equation/Description
Moment of Inertia	$I = \sum_{i=1}^{n} m_i r_i$
Moment of Inertia on z axis	$I_z = \sum_{i=1}^{n} m_i r_{zi}$
Sum of masses	$M = \sum_{i=1}^{n} m_i$
Center of mass	$CoM = \frac{I}{M}$
Radius of Gyration	$RG = \sqrt{\frac{I_z}{M}}$
Dipole Moment	$D = \sum_{i=1}^{n} q_i r_i$
Dipole Histogram	$D_z = \sum_{i=1}^{n} \frac{D}{z}$
Electron Density	$ED = \frac{\sum_{i=1}^{n}(e_i - q_i)}{dz \cdot x \cdot y}$
Heat Capacity	$HC = \frac{3000 \cdot \sqrt{T} \cdot boltz}{2 \cdot \sqrt{T} - n \cdot df \cdot VarT}$
Isothermal Compressibility	$I = \frac{VarV}{V_{avg} \cdot boltz \cdot T \cdot PresFac}$
Mean Square Displacement	$msd = \langle (r_{t+\Delta_t} - r_t)^2 \rangle$
Diffusion Constant	$D_t = \frac{6 \cdot msd(t)}{t}$
Velocity Autocorrelation	$V_{acor} = \langle (V_{t+\Delta_t} \cdot V_t) \rangle$
Force Autocorrelation	$F_{acor} = \langle (F_{t+\Delta_t} \cdot F_t) \rangle$
Density Function	Histogram of atom counts
SDH	Histogram of all distances
RDF	$rdf(r) = \frac{SDH(r)}{4 \cdot \pi \cdot r^2 \cdot \sigma_r \cdot \rho}$

TABLE I: Popular analytical queries in MS [1]

IV. The system design

The proposed design has a pretty straightforward structure which is going to be explained in this section. Shortly, the system starts with original input files (or data) and gets consumed in a sequential manner by simple disk IO framework into random access memory by chunks, then, chunk gets uploaded to GPU device and processed. At some point, the results that need to be shown to the user are transported from global memory of GPU (where they initially get accumulated) to the main memory, and then might get written as well to an output file. General data flow map is represented in Figure 4.

A. Data input

Since the size of the data in the problem is huge, organization of the data flow is crucial. As mentioned before, original data is retrieved from simulation software output files and in our case it is Gromacs Software's trajectory files[1]. For a simulation, there can be several files of different format and usually they are also compressed without any loss of information. Tool for decompressing those files to a flat format for each frame and atom has been developed in the previous work [1].

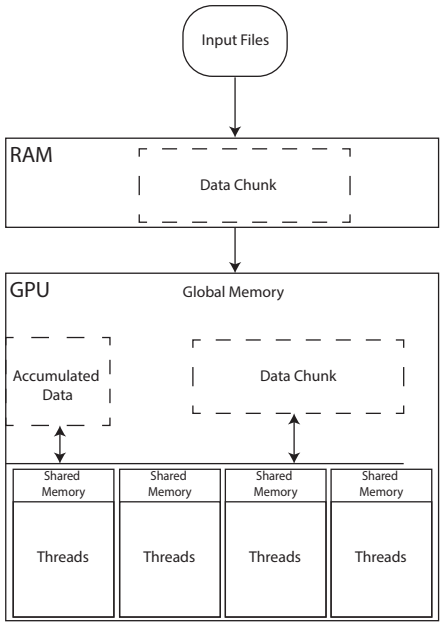

Fig. 4: Data flow in the improved design of push-based system for MS analysis

In order to experiment with different input cases, the input files have been generated with a tool that we have built in Python language. Thus, we are able to generate mock data files with any number of frames and atoms, and the values are random and can be adjusted with needs of the experiment (for example, for cases when we want to analyze bucket widths in a limited space, the coordinates are generated in range of $0-1$, etc.). As it has already been mentioned earlier, we don't expect the data to be distributed with a certain behavior, the importance of having this kind of tool to generate mock data is having a variety of random data sets that could reveal any issues related to values provided to the program, on top of that, having the actual file written and then being read has been a part of the experiments to have more realistic use case. In the future, the same script for generating the mock data could be used for regression and performance tests in a distributed implementation of this design as a single source.

B. Data flow architecture

MS data roughly can be represented as a list of atoms with physical properties aggregated as frames. Thus, we consider a single frame as a chunk which will be loaded into the memory[2] and pushed through our network of query. In this case, it is very convenient, because there are almost no dependencies between frames, rather than list of atoms in the same frame, which potentially again could create additional disk I/O overhead.

[1]The file types, their structure, and their decompression has been well explained in the previous paper in section "Data orgranization in main memory"

[2]Memory in this design might be interpreted differently, since we have a separate device memory on GPUs, but it will be explained better later

Having defined the chunk, we load it from flat format file to RAM. A single atom data structure has all the attributes such as charge, mass, velocity, type of atom, coordinates, and so on. In other words, we have an array of atoms, and most of the attributes (in case of data structure, it is a class field), are simply primitive floats or doubles depending on precision requirements. Next, the same chunk of data with the same structure is loaded onto GPU global memory. GPU devices, in general, have about 4-5 types of memory, and global memory is the biggest and slowest. Consequently, depending on algorithm or query we load parts of data chunk to another types of memory called shared memory. It depends on the actual query, because the same part of chunk can be loaded and unloaded in regards with shared memory several times, especially with SDH.

C. Network of queries organization

Since we assume that our framework should be easily used by a scientist doing MS analysis, it is important to keep queries module as a separate piece of code to make it easily extendable. In order to understand how queries module is organized and the reason for it, we need to understand how CUDA and GPU work.

GPU execution. From hardware standpoint, it is important to understand that every GPU device consists of SM (streaming multiprocessors), and each SM has thousands of registers, which can be partitioned among execution threads. Having this in mind, since this is a pure parallel processing power, the goal is to increase hardware throughput and keep highest utilization to exploit every register given. CUDA is an amazing tool that tries to help developers map this hardware capabilities with the actual application layer. In CUDA we have grids, blocks, and threads, where grid is 3 dimensional unit where each cell is a block, and the same relationship applies between blocks and threads[3]. Originally, this mapping was perfect for solving image processing problems or other graphics computation, since dimensions and such division (e.g. by pixels to threads) is very inherent. As a language feature, in CUDA we have to define such a function called 'kernel' function. Essentially, it is the function where we are supposed to map every execution thread with data and perform the actual execution, this approach is widely known as SIMD[4]. As a convenience we have separated one-body functions with multiple-body functions into separate kernels.

One-body queries kernel. Commonly used CUDA array reduction [38] makes the implementation of all the one-body queries obvious. Figure 5 represents general idea of the data mapping, where each cell of the vector can be associated with a thread of execution in the kernel. Having an array of atom data structures, we reduce the array accumulating all the needed data from each atom's attributes. We don't expect much performance improvement from coalesced memory access, since the data structure for atom can eliminate any gain of coalesced memory access, though for specific types of queries

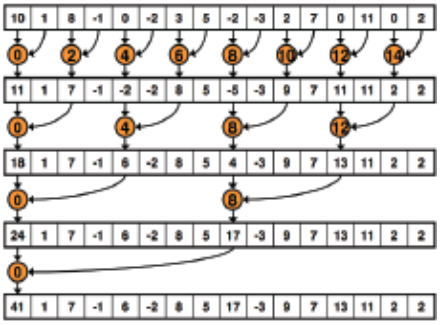

Fig. 5: Array reduction example [38]

it could be possible to aggregate certain attributes into merged arrays and take an advantage of it. In this case, the solution should be simple in order to keep it easily extendable for other simple one-body queries for a scientist to program. We fetch all the atoms from GPU global memory, since we use references to atoms once. In this case, allocating shared memory and copying it over would complicate the implementation and possibly create additional overhead for one time reference usage. Finally, memory transportation and data consumption are implemented for user, thus, he only needs to extend the queries module which consequently can be turned into plugins model.

Two-body queries kernel. SDH (or RDF) is the algorithm that requires special treatment in our design. The point, as mentioned earlier, is that this function requires more than one iteration of the given set of atoms, but rather all combinations of pairs in a given frame, thus, it is not linear complexity. An implementation of this type of query might be very non-intuitive because of the nature of GPUs. Another issue is that since to take an advantage on this type of computation, before writing an implementation for it, a user must have a good understanding of GPU architecture which makes the system less user-friendly. At this point, it is more important to achieve good performance improvement results and then take care of ease of system extensions. As we already mentioned, GPU threads execute in parallel and consume certain amount of work followed by best try of all hardware capacities utilization, but, unfortunately, it is also important to take care of equal workload for every thread independently, so, that utilization is kept high. For this reason, we've come up with a solution that should increase performance of the computation which will be described later.

Architecture overview. This section is to emphasize general architecture of the system and summarize the work flow as well as the data flow. In total, we have flat files with ready flat values for each atom aggregated in frames coming into the program as data input using common disk I/O framework of an operating system. Then, once we have loaded a frame (which is considered to be a chunk of the entire data set) onto RAM of our machine, we copy over the entire frame onto the GPU device global memory. It is important that, at this point, we never write any values to simulation files or any other data related to simulation, thus, we don't have to care about any synchronization of incoming data. After this, we have two

[3]This explanation doesn't include memory setup in GPU devices, but it is covered in further sections

[4]SIMD - Single Instruction Multiple Data

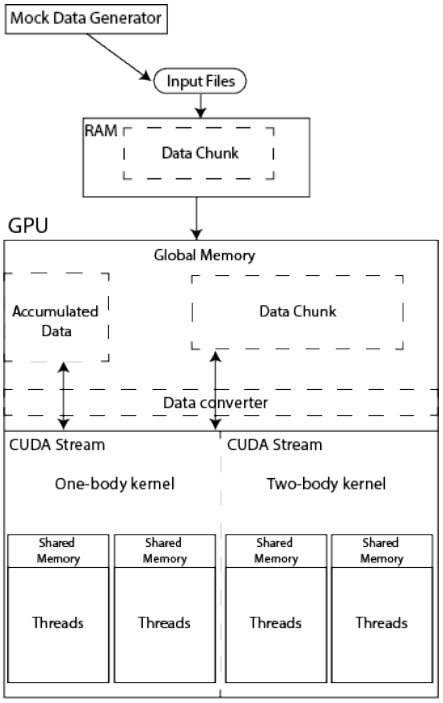

Fig. 6: Overall system architecture

predefined kernel functions (which were explained before), where first kernel stands for one-body queries, and the second one is for two-body query SDH. Just for architecture clearance and possible future improvements, two kernels are executed in two different streams[5] (You can see them on Figure6). In each kernel we apply certain heuristics to improve memory access using such architecture features of GPU as shared memory and blocking of kernel threads.

GPU grid and block sizes. As it has already been mentioned, utilization of hardware on GPU device is important, because it is the key to performance improvement, and the higher utilization is, the bigger improvement is. The approach we came up with is 1 execution thread per atom. For CUDA version 2.0 maximum number of threads per block is 1024. Blocks are 3 dimensional entities in a 3 dimensional grid. The less abstract unit is, the less overhead we have in communication between them, thus, we try to utilize smallest units more. For example, for 1000,000 atoms, we would allocate 1024 threads per block as a static hard coded value, since we usually have always more than 1024 atoms in simulations. One dimension of grid can take up to 64,000 blocks, thus, if N is bigger than $1024 * 64,000$, than the rest would be allocated to $grid.y$ and $grid.z$ respectively. We have not considered numbers bigger than GPU can allocate, which could be explored in future work. It is important to understand that even though there are abstract entities of grids and blocks, it is not real hardware capacities, but they can affect scheduling and performance consequently. Decision to go with atom to thread execution is because it is pretty convenient to keep high

[5]In CUDA stream is another abstract layer of organizing execution threads. They are not guaranteed to be executed at the same time or after each other, GPU scheduling architecture is completely different from CPU having a warp to be single scheduling unit at maximum of 32 threads at a time

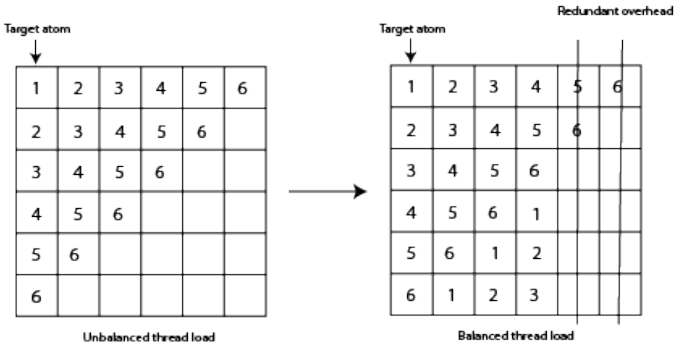

Fig. 7: SDH thread loading

utilization of hardware and still not lose idea if we are about to incorporate more functions, which makes the system more user-friendly, though later on as a plugin we left some room to add data transformer between RAM and GPU global memory, because there are a lot of custom ways to aggregate data and take an advantage of coalesced memory access.

D. Computation work flow

Once the data is loaded into global memory of GPU, we have two kernels for two types of queries setup for two different streams. Besides it, we allocate some memory in GPU global memory for results, too, and keep it as an object with fields to easily pull it back when it is ready, in our case all the data sizes are static before the kernels are executed, which follows nature of GPU devices.

1) One-body kernel: Every thread of execution in one-body query writes into shared memory, before accumulating the data into global memory result in order to avoid massive read and write blocking conflicts which will eliminate any kind of performance advantages. Once a block is finished, first thread of each block will accumulate data from shared memory into global synchronizing the results, every write operation to shared and global memories is atomic (e.g. generic CUDA function atomicAdd). We run the first stream with one-body queries and wait till it gets finished. The reason is that we expect certain behavior from shared memory from each of the streams, thus, just to make sure that they don't overlap, we run them separately. While the computation is running, there is a possibility to use a third stream to load another chunk of data, too. First stream and kernel execute simple vector reduction type of computation accumulating all the needed data and keep it in the GPU.

2) Two-body kernel: Once the first kernel is finished we run the two-body kernel with its own stream. As it has already been mentioned, the biggest workload is in this kernel, because it has higher complexity. In this particular case it is $O(n^2)$. GPU streaming multiprocessors are very powerful for straightforward primitive independent calculations, and each core is much weaker than a moden CPU which has a whole bunch of L caches with some prediction behavior systems for code divergence. As you can see on Figure 7, unbalanced thread load assumes for GPU kernel to check every time if the target atom can interact with other atom and if it has to

wait till other threads finish their jobs. Thus, we had to come up with a strategy of making equally balanced thread jobs and distribute them among the threads leaving atom to thread mapping as is.

For SDH we the following strategy

1. Find current index (for target atom)
2. Find range of atoms it has to interact with in an equally loaded fashion
3. Load the range of atoms into shared memory
4. Iterate through the range of atoms from shared memory and atomically write the distances counts into buckets pre-allocated in share memory
5. Synchronize with other threads in the block (not in the entire grid)
6. If it is the first thread in block, add all bucket counts from shared memory into global memory

V. Experiments Results

The framework has been developed in a programming language CUDA, and the sequential version for benchmark [1] has been originally developed C++. Experiments were run on operating systems which didn't have any additional workload besides usual installation, and the operating systems particularly were Amazon Linux, which doesn't have a lot of modifications except for some additional features for management with Amazon EC-2 Console. We have generated mock data sets with different variations of atoms, frames and ran them with different workloads comparing sequential version of Push-based system with improved version written in CUDA.

Hardware stack. There are primarily several parts that are really important in this hardware stack. Since this is primarily big data processing, we care about processing power and memory. RAM on the node that we were running has 16GB. The server allocated from Amazon has the following CPU parameters:

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	8
On-line CPU(s) list	0-7
Thread(s) per core	2
Core(s) per socket	4
CPU family	6
Model	45
Model name	Intel Xeon E5-2670 2.60GHz
Stepping	7
CPU MHz	2593.814
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	20480K
NUMA node0 CPU(s)	0-7

The same CPU has been used for sequential as well as for improved GPU version during experiments with benchmarks. Just to mentioned, CPU in improved version is used in-between memory loading from file and uploading it to GPU, printing the results to the user, which is not big load. Consequently, we also have GPU device with the following parameters:

Name	Grid 34C K520
Memory	4095Mib
Total Nvidia CUDA Cores	1536
Memory Size	8 GB GDDR5 (4 GB/GPU)
Total GPUs	2 GK104 GPUs

Data sets. As it has already been mentioned, we have developed a small python script that generated data sets that are used for the experiments. It accepts two numbers as arguments (number of atoms and frames respectively) and generate a file with appropriate number of atoms/frames and format. We have generated number of mock data sets in range of $1K - 8M$ atoms with $10, 100, 1000$ frames each. For each data set we run workloads with the values: $1, 5, 25, 50, 100$. Number of atoms and frames are obviously up to the scientist and depend on simulations, but workloads are a bit trickier. First of all, they give us a bit better understanding of the average value, which might vary on different circumstances related to OS, secondly, once this system is integrated into a distributed system, we can imagine that each frame could be processed by a separate node keeping the values inside of the appropriate memory (global or RAM) and generate responses based on the clients selections requests. This possibility will be elaborated in more details in Future Work section.

SDH implementation. Even though implementation of SDH can be different, in the previous paper the algorithm in the previous paper was based on quad tree data structure with data proximity approach based on data uniformity [1], but in our experiments we follow special CUDA based approach which perfectly fits the nature of GPU devices with the given memory hierarchy, and the number of computations are equal to brute force, consequently, there is no expectation on data uniformity, thus, the problem solution complexity is still the same with the worst case of $O(N^2)$, but processed in parallel by GPU cores. More improvements can be done for acceleration of this approach, but in the system we have not gone too far with data converter module, because initial framework has to be simple and user friendly.

A. Results

Results are represented, in general, on Figures 8, 9 10. You can instantly notice that speedup for 1000 atoms is not very impressive, especially for 10 frames, and the reason for that is nature of GPU and memory transfer. There is a small overhead we have to consider before computing part: in order to let GPU process the data, we have to transfer the data to Global Memory, and upon completion it is needed to load the results back to RAM to show it to the user. In this particular case, for small workload and frames, it is even slower than the sequential processing. Afterwards, the bigger workload the more convincing speedup gets, unfortunately, it doesn't get more than 4 times for 1000 atoms. Fortunately, scientists usually work with much larger data sets, thus, this doesn't really represent general speedup for average use case.

10000 atoms. As you can see, speedup bumps up to almost 30 with 10000 atoms (Figure 9). Once the overhead on

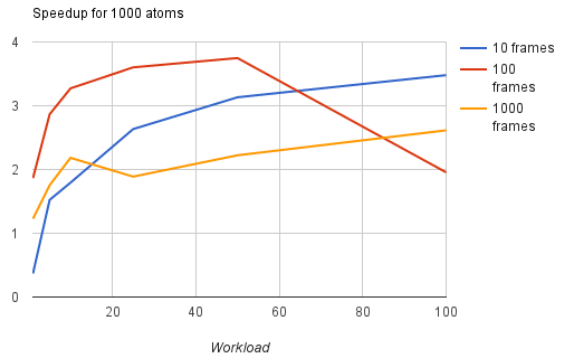

Fig. 8: Speedup for 1000 atoms

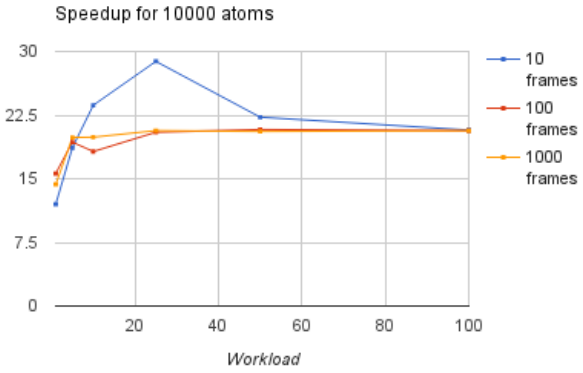

Fig. 9: Speedup for 10000 atoms

memory transfer is covered by computation speedup, it keeps growing. It is important to pay attention to the growth of speedup over growth of workload. It is not that significant, because even though CPUs are much slower, they still use smart caching and internal multi-core processing, which is pretty fast, too.

Fortunately, scientific data is much bigger than above examples. You can see average speedup for bigger number of atoms on Figure 10. The speedup gets upto 70 times which could save huge amount of time for scientists. These results are not the best, as it has been already mentioned earlier, since we have not modified general structure of memory being transferred to Global Memory not taking advantage of coalesced memory access to make it easily extendable.

It is noticable that growth of the speedup is not stable and with certain number of atoms it drops dramatically. It is because we've used static approach of computing grid dimensions. Besides it, in the future, it can be expanded into a dynamic version that will automatically adjust the grid dimensions choosing least overhead that it has to go with.

VI. Conclusions and Future Work

The ultimate goal of this work was to get an improved version of Push-based system for molecular simulation data analysis. As you could see from result on charts (Figure 8, 10), the proposed improved version of the system could save

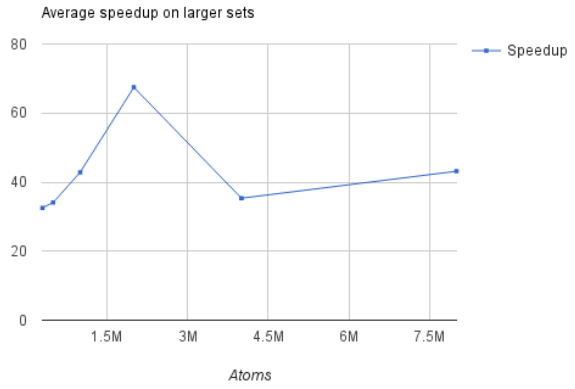

Fig. 10: Speedup for larger number of atoms

a lot of time for scientists which is up to 60-70 times less of time. The results have been gathered with a single snapshot of the framework tracked in control version system which can be improved with means of data converter module and dynamic grid dimensions computation module. In the future, this system has all potentials to be expanded into a distributed system, where frames could be aggregated by nodes and processed in parallel.

References

[1] Vladimir Grupcev, Yicheng Tu *et. al.*, "Push-based system for molecular simulation data analysis." 2015.

[2] D. H. et al., "Big data: The future of biocuration," *Nature*, vol. 455, pp. 47–50, 2008.

[3] B. Huberman, "Sociology of science: Big data deserve a bigger audience," *Nature*, vol. 482, p. 308, 2012.

[4] D. Centola, "The spread of behavior in an online social network experiment," *Science*, vol. 329, pp. 1194–1197, 2010.

[5] J. Bollen, H. Mao, and X.-J. Zeng, "Twitter mood predicts the stock market," *Journal of Computational Science*, vol. 2, pp. 1–8, 2011.

[6] Daan Frenkel *et. al.*, *Understanding Molecular Simulation: From Algorithms to Applications*, 2nd ed. Academic Press, Inc., 2001, vol. 1.

[7] David Landau *et. al.*, *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press, 2005.

[8] Gromacs group, "GROMACS - Online Reference." [Online]. Available: http://gromacs.org/

[9] Subi Arumugam and Alin Dobra and Christopher Jermaine and Niketan Pansare and Luis Perez, "The datapath system: A data-centric analytical procesing engine for large data warehouses," *SIGMOD*, vol. 1, pp. 519–530, 2010.

[10] Goetz Graefe, "Volcano - an extensible and parallel query evaluation system," *TKDE*, vol. 6, pp. 120–135, 1994.

[11] Stavros Harizopoulos and Vladislav Shkapenyuk and Anastassia Ailamaki, "Qpipe: A simultaneously pipelined relational query engine," *SIGMOD*, pp. 383–394, 2005.

[12] Gorge Candea and Neoklis Polyzotis and Radek Vingralek, "A scalable, predictable join operator fo highly concurrent data warehouses," *VLDB*, pp. 277–288, 2009.

[13] P Unterbrunner and G Giannikis and G Alonso and D Fauser and D Kossmann, "Predictable performance for unpredictable workloads," *VLDB*, vol. 2, pp. 706–717, 2009.

[14] Marcin Zukowski and Sandor Heman and Niels Nes and Peter Boncz, "Cooperative scans: Dynamic bandwidth sharing in dbms," *VLDB*, pp. 723–734, 2007.

[15] Vladimir Grupcev, Yicheng Tu, Meryem Berrada *et. al.*, "Dcms: A data analytics and management system for molecular simulation," 2014.

[16] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, March 2008.

[17] W. Humphrey, A. Dalke, and K. Schulten, "Vmd: visual molecular dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33–38, 1996.

[18] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein, "Mdanalysis: A toolkit for the analysis of molecular dynamics simulations," *Journal of Computational Chemistry*.

[19] M. Seeber, M. Cecchini, F. Rao, G. Settanni, and A. Caflisch, "Wordom: a program for efficient analysis of molecular dynamics simulations," *Bioinformatics*, vol. 31, pp. 2658–2668, 2010.

[20] T. Verstraelen, M. V. Houteghem, V. V. Speybroeck, and M. Waroquier, "Md-tracks: a productive solution for the advanced analysis of molecular dynamics and monte carlo simulations," *Journal of Chemical Information and Modeling*, vol. 48, pp. 2414–2424, 2008.

[21] M. Mezei, "Simulaid: a simulation facilitator and analysis program," *Journal of Computational Chemistry*, vol. 23, pp. 2625–2627, 2007.

[22] B. R. Brooks *et. al.*, "Charmm: the biomolecular simulation program," *Journal of Computational Chemistry*, vol. 30, pp. 1545–1614, 2009.

[23] Yicheng Tu *et. al.*, "Computing distance histograms efficiently in scientific databases," in *ICDE*, 2009.

[24] Anand Kumar *et. al.*, "Distance histogram computation based on spatiotemporal uniformity in scientific data." in *EDBT*, March 2012.

[25]

[26] L. Golab and T. Ozsu, *Data Stream Management*. Morgan And Claypool, 2010.

[27] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska *et. al.*, "The design of the borealis stream processing engine," 2005.

[28] B. L. *et. al.*. Moo-Ryong Ra, "Medusa: A Programming Framework for Crowd-Sensing Applications," 2012.

[29] Apache, "Aurora Framework." [Online]. Available: http://aurora.apache.org/

[30] J. M. Patel, "The Role of Declarative Querying in Bioinformatics," *OMICS: A Journal of Integrative Biology*, vol. 7, no. 1, pp. 89–91, 2003.

[31] P. Cudre-Mauroux *et. al.*, "A demonstration of scidb: A science-oriented dbms," *VLDB*, vol. 2, pp. 1534–1537, 2009.

[32] A. S. Szalay, J. Gray, A. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg, "The SDSS Skyserver: Public Access to the Sloan Digital Sky Server Data," in *Proceedings of International Conference on Management of Data (SIGMOD)*, 2002, pp. 570–581.

[33] M. Arya, W. F. Cody, C. Faloutsos, J. Richardson, and A. Toya, "QBISM: Extending a DBMS to Support 3D Medical Images," in *ICDE*, 1994, pp. 314–325.

[34] S. R. R. C. Konstantin Shvachko, Hairong Kuang, "The Hadoop Distributed File System," 2010.

[35] A. S. *et. al.*. RadhaKishan Yadav, Robin Singh Bhadoria, "Gpu-accelerated large scale analytics using mapreduce model," 2015.

[36]

[37] J. Orenstein, "Multidimensional tries used for associative searching," *Information Processing Letters*, vol. 14, no. 4, 1982.

[38]

[39] Vladimir Grupcev *et. al.*, "Approximate algorithms for computing spatial distance histograms with accuracy guarantees." *TKDE*, 2012.